MODEL-BASED TESTING USING VISUAL CONTRACTS

Thesis Submitted for the degree of

Doctor of Philosophy

at the University of Leicester

by

Tamim Ahmed Khan

Department of Computer Sciences

University of Leicester, UK

2012

Abstract

Web services only expose interface level information, abstracting away implementation details. Testing is a time consuming and resource-intensive activity. Therefore, it is important to minimize the set of test cases executed without compromising quality. Since white-box testing techniques and traditional structural coverage criteria require access to code, we require a model-based approach for web service testing.

Testing relies on oracles to provide expected outcomes for test cases and, if implemented manually, they depend on testers' understanding of functional requirements to decide the correct response of the system on every given test case. As a result, they are costly in creation and maintenance and their quality depends on the correct interpretation of the requirements. Alternatively, if suitable specifications are available, oracles can be generated automatically at lower cost and with better quality. We propose to specify service operations as visual contracts with executable formal specifications as rules of a typed attributed graph transformation system. We associate operation signatures with these rules for providing test oracles.

We analyze dependencies and conflicts between visual contracts to develop a dependency graph. We propose model-based coverage criteria, considering this dependency graph, to assess the completeness of test suites. We also propose a mechanism to find out which of the potential dependencies and the conflicts were exercised by a given test case. While executing the tests, the model is simulated and coverage is recorded as well as measured against the criteria. The criteria are formalized and the dynamic detection of conflicts and dependencies is developed. This requires keeping track of occurrences and overlaps of pre- and post-conditions, their enabling and disabling, in successive model states, and interpreting these in terms of the static dependency graph.

Systems evolve over time and need retesting each time there is a change. In order to verify that the quality of the system is maintained, we use regression testing. Since regression test suites tend to be large, we isolate the affected part in the system only retesting affected parts by rerunning a selected subset of the total test suite. We analyze the test cases that were executed on both versions and propose a mechanism to transfer the coverage provided by these test cases. This information helps us to assess the completeness of the test suite on the new version without executing all of it.

Acknowledgements

In the name of Allah, the most merciful, the most beneficent

I am thankful to Almighty Allah that I landed here in Leicester and was in the able and kind hearted support of Prof Dr. Reiko Heckel. I was ready to do the hard work but was, like many other PhD students, wondering in a pointless manner. It was him who brought me to a proper direction and guided me throughout my degree. Few words or few sentences cannot do justice to what the Professor has done not only for me but all of the students who worked with him. I wish him all the best in his life and in his future pursuits. I would also like to thank Dr. Artur Boronat who was my second supervisor and Dr. Fer-Jan DeVres who was my course tutor for their kind help and support. I would like to thank Olga Runge who was extremely helpful and provided me a sound and prompt support whenever required.

When I came and joined computer sciences department, I noticed that the outfit was more towards theoretical aspects and less towards the application side. It was through a systematic support and guidance in the shape of regular Friday afternoon seminars, for which Dr. Alexander Kurz must be thanked besides many others, and short courses from all senior faculty members especially from Prof Dr. Jose Fiadeiro, Prof Dr. Rick Thomas, Prof Dr. Rajeev Raman, and Prof Dr. Thomas Erlebach that we were able to dope ourselves into theoretical domain. I would also like to thanks Prof Dr. Rick Thomas, Dr. Irek Ulidowski, Dr. Emilio Tuosto, Dr. Stephan Reiff-Marganiec, Dr. Alexander Kurz, Dr. Monika Solanki and Dr. Neil Walkinshaw for allowing to sit in their taught modules, besides courses from my first and second supervisors, and learn from them. I would like to express my gratitude to all of the rest of the staff members for their help and support in the shape of short courses, healthy discussions and sharing of information. I would also like to express my gratitude to help and support by Rodrigo Machado in understanding the basic concepts of category theory and graph transformation.

I would like to thank both of my parents who must be happily seeing me progress to completing my PhD from heavens and I would like to thank all of my family members especially Inaam Bhai, Fariha, Kashif and Kiran for their prayers and support. I express my gratitude to Khala and Khaloo. Last but not least, I am grateful to my wife, Madiha, for her continual support through thick and thin and I dedicate my thesis to my parents and to my boys - Talal and Qaseem.

Declaration of Authorship

I hereby declare that this submission is my own work and that is the result of work done during the period of registration. To the best of my knowledge, it contains no previously published material written by another person. None of this work has been submitted for another degree at the University of Leicester or any other University.

Part of this thesis appeared in the following conjoint publications, to each of which I have made substantial contributions:

- Tamim Ahmed Khan and Reiko Heckel, “A Methodology for Model-Based Regression Testing of Web Services", in Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques, TAIC-PART 09, IEEE Computer Society.

- Tamim Ahmed Khan and Reiko Heckel, “On Model-Based Regression Testing of Web-Services Using Dependency Analysis of Visual Contracts", in Proceedings of the Fundamental Approaches to Software Engineering - 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011, LNCS Vol. 6603, pp.341-355.

- Reiko Heckel, Tamim Ahmed Khan, Rodrigo Machado, “Towards test coverage criteria for visual contracts", In Proceedings of the Graph Transformation and Visual Modeling Techniques, GTVMT 11, Electronic Communications of the EASST, Vol. 41/2011

- Tamim Ahmed Khan, Olga Runge, Reiko Heckel, “Visual Contracts as Test Oracle in AGG 2.0", In Proceedings of the Graph Transformation and Visual Modeling Techniques, GTVMT 12, Electronic Communications of the EASST, Vol. 47/2012
- Tamim Ahmed Khan, Olga Runge, Reiko Heckel, “Testing Against Visual Contracts: Model-based Coverage", in Proceedings of the 6th International Conference on Graph Transformation, ICGT 12, University of Bremen, Germany 24 - 29 September, 2012, LNCS (to appear).

Contents

I Introduction and Preliminaries **1**

1 Introduction **2**

1.1 Motivation 2

1.2 Approach 5

1.2.1 Visual Contracts as Test Oracles 5

1.2.2 Model-based Coverage 6

1.2.3 Regression Testing 7

1.3 Contributions 8

1.4 Thesis Outline 9

2 Background **11**

2.1 Testing 11

2.2 Visual Contracts 12

2.3 Graph Transformation 12

2.3.1 Graphs, rules, and transformations 13

2.4 Running Example 16

2.5 Independence and Critical Pairs 18

2.6 Summary 21

II Model-Based Testing **22**

3 Typed Attributed Graph Transformation Systems with Rule Signatures **23**

3.1 Rule Signatures 23

3.2 Summary 30

4 Test Oracles Using Visual Contracts **31**

4.1 Challenges 32

4.1.1 Model as Oracle 33

4.1.2 Partiality of Visual Contracts 34

4.1.3 Failure Handling 34

4.1.4 Adaptation of Output Types 36

4.2 Using AGG as an Oracle 37

4.2.1 Model as Oracle 37

4.2.2 Partiality of Visual Contracts 39

4.2.3 Failure Handling 40

4.2.4 Adaptation of Output Types 42

4.3 Implementation 43

4.4 Application to the Running Example 46

4.5 Limitations of the Approach 48

4.6 Summary 49

5 Model-Based Coverage Criteria **50**

5.1 Dependency Graphs 50

5.2 Coverage Criteria 55

5.3 Summary 58

6 Dynamic Analysis of Dependencies and Conflicts **59**

6.1 Dynamic Analysis of Dependencies and Conflicts 60

6.2 Implementation 61
6.3 Application to the Running Example 68
6.4 Limitations of the Approach 70
6.5 Summary 71

7 Model Based Regression Testing 72

7.1 Evolution Scenarios 73
7.1.1 Scenario I 73
7.1.2 Evolution Scenarios II 74
7.2 Classification of Test Cases 74
7.3 Coverage Analysis 76
7.4 Application to the Running Example 79
7.4.1 Generation of Test Suite 79
7.4.2 Validation of Quality of Entire Test Suite 80
7.4.3 Classification of test cases into OB, RU, and RQ 83
7.4.4 Validation of the quality and required size of RQ 84
7.4.5 Coverage Analysis 84
7.5 Limitations of the Approach 85
7.6 Summary 86

8 Case Study: Bug Tracking System 87

8.1 Bug Tracking System 87
8.1.1 Service Implementation 87
8.1.2 Model Artifacts 89
8.2 Test Oracles 90
8.2.1 Setup and Execution 91
8.2.2 Results and Evaluation 93
8.2.3 Threats to Validity 98

8.3 Coverage Analysis 99
8.3.1 Setup and Execution 99
8.3.2 Results and Evaluation 101
8.3.3 Threats to Validity 104
8.4 Regression Testing 105
8.4.1 Setup and Execution 106
8.4.2 Results and Evaluation 110
8.4.3 Threats to Validity 111
8.5 Summary 112

III Comparison and Conclusion 113

9 Comparison to the State of the Art 114
9.1 Test Oracles 115
9.2 Model-based Coverage Criteria 118
9.3 Regression Testing 122
9.4 Summary 125

10 Conclusion and Outlook 127
10.1 Conclusion 127
10.2 Outlook 128
10.2.1 Test Case Analysis and Generation 129
10.2.2 Learning Visual Contracts 130
10.2.3 Rules Signature for Multi Objects 130
10.2.4 Application Conditions 131
10.2.5 Evaluation of Effectiveness of Proposal 131

A Visual Contracts: Bug Tracker Service 132

References **146**

List of Figures

1.1 Software testing process diagram adapted from [66] 3

2.1 E-Graph (Left) and condition for A-graph morphisms (Right) 14
2.2 Visual Contracts for Hotel Example 17
2.3 Type Graph 17
2.4 Rule to query information 18
2.5 Optional caption for list of figures 20

3.1 Optional caption for list of figures 25
3.2 An example transformation sequence 26
3.3 An example transformation sequence and an observation sequence . . 27
3.4 Start graph for test case example 29

4.1 Schematic diagram of proposed driver 32
4.2 Component diagram 33
4.3 Additional rule for finding all rooms 33
4.4 Optional caption for list of figures 37
4.5 Additional rule for finding all rooms 38
4.6 Start Graph 39
4.7 Additional rule for finding all rooms 47

5.1 Critical pairs and dependencies 51

5.2 Dependency Graph of $TAGTS$ representing hotel web service 54
5.3 Start graph 56
5.4 $Cov(DG, T)$ 57

6.1 Dependencies in a transformation sequence 60
6.2 Optional caption for list of figures 67
6.3 Method Coverage 68
6.4 Code Coverage with $cr + cd$ 69
6.5 Code Coverage with all paths 70

7.1 Optional caption for list of figures 74
7.2 Optional caption for list of figures 75
7.3 Fault seeding with L-Care 78
7.4 Start graph for version V2 80
7.5 Fault seeding with L-Care 82
7.6 Coverage analysis output for version V1 85
7.7 Coverage analysis output for version V2 86

8.1 Model level signatures 90
8.2 Optional caption for list of figures 91
8.3 Initial state of the model 93
8.4 Initial model state 94
8.5 Test Case Execution Time 97
8.6 Optional caption for list of figures 98
8.7 Optional caption for list of figures 100
8.8 Dependency graph for $\prec$ relation. 101
8.9 Dependency graph for $\nearrow$ relation. 102
8.10 Unreachable code example 103

8.11 Optional caption for list of figures 107

8.12 BugTracker, evolution to Version 2 108

8.13 BugTracker, evolution to Version 3 108

8.14 Fault seeding with L-Care 109

10.1 Test Case Generation Example 129

10.2 Test Case Generation Example 131

A.1 Visual Contract for Rule AddProject 132

A.2 Visual Contract for Rule AssignProject 132

A.3 Visual Contract for Rule DeleteProjectById 133

A.4 Visual Contract for Rule GetProjectByProjectTitle 133

A.5 Visual Contract for RuleScheme GetProjects 133

A.6 Visual Contract for GetProjectsForUsers 134

A.7 Visual Contract for Rule UpdateProject 134

A.8 Visual Contract for Rule AddBug 135

A.9 Visual Contract for Rule DeleteBug 135

A.10 Visual Contract for Rule DeleteBugsForProjectAndUsers 136

A.11 Visual Contract for RuleScheme GetAllBugs 136

A.12 Visual Contract for RuleScheme GetAllBugsForProject 137

A.13 Visual Contract for RuleScheme GetAllBugsForProjectAndUser . . . 137

A.14 Visual Contract for Rule UpdateBug 138

A.15 Visual Contract for AddUser 138

A.16 Visual Contract for RuleScheme DeleteUserById 138

A.17 Visual Contract for RuleScheme GetUsers 139

A.18 Visual Contract for Rule IsValidPassword 139

A.19 Visual Contract for Rule GetLogins 140

A.20 Visual Contract for Rule GetUserInfo 140

A.21 Visual Contract for Rule GetUserInfoForUserName 140

A.22 Visual Contract for Rule GetUserInfoForUserId 140

A.23 Visual Contract for Rule UpdateUser 141

A.24 Visual Contract for Rule AddIssue 141

A.25 Visual Contract for Rule DeleteIssues 142

A.26 Visual Contract for Rule DeleteIssuesForProjectAndUser 142

A.27 Visual Contract for RuleScheme GetAllIssues 143

A.28 Visual Contract for Rule UpdateIssue 143

A.29 Visual Contract for RuleScheme GetUsersForProject 144

A.30 Visual Contract for RuleScheme GetPossibleStatusses 144

A.31 Visual Contract for Rule UpdateXUserProject 145

Part I

Introduction and Preliminaries

Chapter 1

Introduction

1.1 Motivation

Software testing is a verification and validation approach in which the system under test (SUT) is executed with sample data such that the implemented behavior can be examined against a specification. The intention is to uncover the presence of defects in the SUT and to validate the implementation against the requirements. The testing process, as shown in Figure 1.1, consists of activities to design test cases, the preparation of test data, executing the SUT using test data, and validating the results [66].

In order to design test cases and prepare test data, artifacts related to specification or implementation are considered. Since testing is a time and cost intensive process, we need to select adequately many test cases such that they are likely to reveal faults [8]. There are several methods for assessing the completeness of test suites of which coverage analysis is one particular method [44]. Once test cases are selected, inputs that are generated to execute these test cases and expected results are computed. The expected outcomes are used to provide test oracles which are an essential part of the software testing process [70]. The SUT is executed with test data and the actual outcomes are validated against the expected results in the last two activities shown in Figure 1.1.

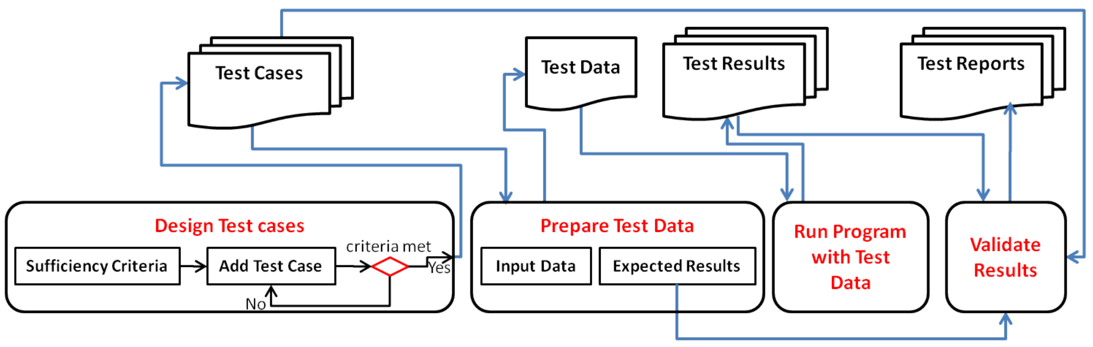


Figure 1.1: Software testing process diagram adapted from [66]

Service-oriented systems pose a variety of challenges to client-side testing due to lack of control over the service implementation [15]. A user's view of a web service is through provided interfaces, which abstract away the implementation details and prevent the use of white-box testing methods [15]. Therefore, we require a model-based approach to test oracles for web services.

Web services testing incurs additional overheads in terms of service invocation costs and resource consumption through network traffic [57]. Therefore, we need to assess the completeness of a given set of test cases. There are several methods for assessing the completeness of test suites, e.g., structural coverage criteria, fault-based approaches. Coverage provides a metric to see what portions of a software artefact, with respect to a given test requirement, the test cases in a test suite are able to exercise [3]. Traditionally, a test requirement is defined in terms of code elements such as statements, branches, data definitions, data usages, etc. Coverage criteria are used to enforce test requirements [3]. Since traditional structural coverage criteria require access to code, we need a model-based alternative for web services and component-based systems.

Software systems evolve as a consequence of repairing faults or adapting the software to a different system/environment. Evolution may also be in response to addition or modification of functionality. These maintenance activities are an integral part of the software development process contributing significant overheads as identified in [66]. Maintenance starts as soon as a software system is developed and delivered to the production environment. Large portion of the total software cost (up to 85 - 90%) is devoted to this activity [25]. Often, changes made to a system are local, arising from fixing bugs or specific additions or changes to the functionality. Rerunning the entire test set in such cases is wasteful. Instead, we would like to be able to identify the parts of the system that were affected by the changes and select only those test cases for rerun which test functionality that could have been affected.

When verifying that the quality of a web service under test has not regressed during maintenance, we are again faced with the problem of non-availability of code. Therefore, we require a model-based regression testing approach for web services, providing a test suite reduction mechanism based on analyzing the impact of the change. For the two versions of the SUT before and after the maintenance, there are test cases which are executable on both versions, but are not selected for retesting since they do not exercise the affected part of the system. However, we need to consider them to determine the coverage for the evolved version. We therefore require a mechanism to transfer coverage information to the next version.

In summary, the thesis addresses the problem as follows:

> *Web services hide implementation details preventing the use of white-box testing methods. We propose a model-based approach to support oracles, coverage analysis, and regression testing. We specify web service operations by visual contracts, i.e., pre- and post-conditions formalized as rules in a graph transformation system. This provides us with an executable model as oracle and allows analysis of conflicts and dependencies for assessing coverage and regression.*

1.2 Approach

We propose a model-based approach to test oracles, coverage criteria and regression testing of web services. Analysis of visual contracts formally specified as transformation rules in a typed attributed graph transformation system (TAGTS) has been used for interface specification [31], model-based testing [33], and test case generation [29] previously. We have used visual contracts for executing models to propose test oracles. Specifications are analyzed for model-based testing considering additional semantic information e.g., resource description framework (RDF) in [49], semantic web descriptions (WSDL-S) in [65] and specifications using business process execution language (BPEL) in [36]. The gathered information is either represented as control-flow graph or as extended finite state machine. Our treatment to the problem is different in the sense that we make use of visual contracts for model-based coverage criteria. We consider dataflow information and represent it as a dependency graph. Next, we provide a high-level introduction to our approach.

1.2.1 Visual Contracts as Test Oracles

Software testing relies on test oracles to predict expected test results [11]. Oracles can be implemented manually relying on testers' understanding of functional requirements to decide the correct response of the system on every given test case. As a result, they are costly in creation and maintenance and their quality depends on the correct interpretation of the requirements. Alternatively, if suitable specifications are available, oracles can be generated automatically at lower cost and with better quality [6].

Visual contracts specified as rules in a typed attributed graph transformation system (TAGTS) are directly executable and therefore suitable for generating automated oracles. However, the gap in abstraction between service implementation and visual models poses a number of challenges in implementing this basic idea. This includes the requirement to convert operation invocations into rule applications, passing and converting parameters and interpreting replies. This conversion is also required because model and implementation signatures may differ, with the implementation requiring extra parameters, providing additional results, or using different types, especially for collections.

Since the model is only concerned with functional aspects, we also have to filter out technical failures of the implementation, such as problems with the server or transport layer, distinguishing them from logical failures corresponding to non-applicable rules due to violation of preconditions. However, with visual contracts providing a partial specification only, even the functional aspect may be under-specified. As a further challenge, different web service implementations may report success and failure differently. Adapter and test driver both need to be flexible enough to accommodate different styles of error handling and reporting, allowing for a degree of customization. We analyze and overcome these challenges and propose model-based test oracles for web service testing in Chapter 4.

1.2.2 Model-based Coverage

Model-based testing relies on model-level artifacts to derive test cases and assess coverage [69]. Considering the classification of approaches into black-box and white-box, model-based testing is a special case of black-box testing [16] in which the system is tested considering its externally visible behavior. Test cases and coverage criteria are therefore derived from the model of the system. *White-Box* strategies, on the other hand, access implementation details.

Complete testing of software systems is not possible due to the infinite number of values that can be considered as inputs [3]. Therefore, we require a mechanism to establish the completeness of test cases. Coverage provides us with a metric to see what portion of a software artefact, with respect to a given test requirement, the test suite was able to exercise [3].

Our approach to model-based testing relies on the execution of models to assess completeness of test suites. The coverage criteria are based on static dependency analysis and dynamic analysis of actual coverage. We represent service specifications as TAGTS to analyze dependencies and conflicts between operations. We are then able organize this information as a graph where data definition, usage, and deallocation by different operations can be visualized by edge labels. Once a graph is constructed, we propose model-based coverage criteria based on combinations of such labels. To assess coverage, we need information from the model's execution, since individual test cases may or may not observe a dependency or a conflict. Our approach of model-based coverage analysis is therefore a natural extension of the use of executable visual contracts as test oracles. In line with most approaches to testing, we assume that our models are correct, i.e., if a discrepancy occurs between model and implementation, the assumption is that the implementation is at fault. Moreover, for some parts of our method, visual contracts are required to be complete. In particular, the use as test oracle requires complete post-conditions.

1.2.3 Regression Testing

Regression testing verifies if systems under evolution retain their existing functionality [40, 41, 30, 28]. Regression testing may be broadly classified into two types: one in which the specification of the system does not change, called corrective regression testing, and another in which the specification may change, called progression regression testing [62]. The former arises from activities like corrective maintenance or refactoring, whereas the later is required where there is a modification, addition, or deletion of functionality.

If we use a full set of test cases to reevaluate every new version, especially when test suites are accumulated over time, this can be resource and cost intensive. Since our knowledge about the implementation is limited to the interface-level, we need to consider the available information for selecting the relevant subset of existing test cases to exercise the functionality affected by the changes. For this, we need information about the change and a mechanism for analyzing the impact of change. We analyze what has changed in the model, i.e., what operations are affected at the model-level and what what operations have been impacted at the implementation level. We use this information to isolate the affected part of the system to find out which of the test cases need to be executed again.

Following the classification in [48], a test case in a regression test suite can be *obsolete* if it is no longer applicable to the new version, *reusable*if it is still applicable and *required* if it tests functionality affected by the changes. We only execute the required test cases for the latest version and, in order to retain coverage, we need to transfer coverage due to reusable test cases that were not rerun.

1.3 Contributions

This thesis presents the following contributions that have been made while addressing the research objectives.

Rule Signatures: We introduce signatures for visual contracts and a way to link them to rules in a TAGTS. This allows to provide an executable model with an interface comparable to that of the system under test.

Test Oracles: We propose an approach to test oracles based on visual contracts formally represented by graph transformation rules. We make use of the Attributed Graph Grammar (AGG) [2] to provide executable specifications as test oracles.

Coverage Criteria: We introduce model-based coverage criteria by analyzing conflicts and dependencies between visual contracts. We propose a visual representation in the shape of a dependency graph, based on static information about potential dependencies and conflicts between operations. Additional information about the nature of conflicts and dependencies is provided by edge labels, which we use to define coverage criteria. A given set of test cases may or may not exercise these dependencies or conflicts at runtime. We propose a mechanism by which we can find out the completeness of our test cases using dynamic dependencies and conflicts analysis.

Regression Testing: Since there are additional costs involved [57] in case of web service testing, we cannot afford to rerun all of the existing test cases. Hence we propose a model-based approach for test suite reduction for regression testing of web services. We also propose to analyze coverage by test cases individually so that we can consider the coverage due to reusable test cases for the version after evolution.

1.4 Thesis Outline

The thesis is organized as follows. Following this introduction, the background knowledge deemed required for a general readership is discussed in Chapter 2. These two chapters form the first part of the thesis.

The second part contains six chapters pertaining to the main contributions submitted in this thesis. Chapter 3 provides our basis for associating rules with signatures whereas Chapter 4 explains our approach for using visual contracts to provide model-based test oracles. The theoretical basis for model-based coverage criteria are discussed in Chapter 5 where the dynamic dependency analysis is explained in Chapter 6. Our approach to model-based regression testing is discussed in Chapter 7. An evaluation of our approach using a case study is given in Chapter 8, which also contains a critical analysis of the presented results as well as of the threats to the validity of the evaluation.

The third part contains two chapters in which related work and conclusions are provided. Related work and a critical comparison are presented in Chapter 9 whereas conclusion and outlook are discussed in Chapter 10.

Chapter 2

Background

This chapter introduces existing concepts of software testing with emphasis on test oracles, model-based testing, coverage criteria and regression testing. We also explain basic graph transformation concepts and visual contracts and introduce our running example.

2.1 Testing

A Failure is the inability of a system under test (SUT) to perform a function as per requirements specifications, whereas a fault is the incorrect part of the source code which presents itself as a failure [1]. A Test Case is a set of inputs together with the expected outputs. A mechanism that provides knowledge about expected or "correct" outputs by means of an automated program, a model checker, a manual activity, a program specification, a table of examples, etc., is called test oracle [11]. A Test Run is an execution of a particular test case, whereas a Test Driver is a program that executes a test case on an SUT.

In model-based testing approaches, we consider specification artifacts for testing [21, 44]. Considering graph based structural testing techniques, coverage criteria can be defined on the basis of selection of different paths through the system. Having this information, appropriate inputs are selected so that these paths through the SUT could be exercised. Example coverage criteria include all $defs$ paths in which we select inputs such that we exercise all paths exercising all occurrences of def nodes in dataflow graph of our SUT etc.

2.2 Visual Contracts

Design by Contract *(DbC)* was introduced by Meyers in [53] for Eifel, where *Require* and *Ensure* are used to specify assertions consisting of pre- and post-conditions. This tells the developer what state of the system a particular method expects before and what it should leave after invocation. Several methods for specification of the pre- and post-conditions have been proposed including the Object Constraint Language (OCL), Z and VDM. Visual contracts provide a visual representation of pre- and post-conditions using a pair of graphs, such that the pre-condition represents the left-hand side and the post-condition the right-hand side of a graph transformation rule. The two graphs represent the system state in terms of object diagrams with attribute values. Our use of visual contracts specified as typed attributed graph transformation system (TAGTS) for analysis is motivated by examples demonstrated, e.g., in [51, 5, 23, 51, 34, 24]. We use AGG to execute system described as a TAGTS which allows rule application only if match is found. This becomes an important aspect of our analysis since we do not need to filter out infeasible paths from a given test set.

2.3 Graph Transformation

This section introduces existing concepts of graph transformation with emphasis on conflicts and dependency analysis. We also explain visual contracts and introduce our running example.

2.3.1 Graphs, rules, and transformations

In this chapter we recall the basic definitions regarding typed attributed graph transformation systems as presented in [22].

A graph is a set of nodes connected by edges. For example, a class diagram is a graph providing typing information for object diagrams, which can also be seen as graphs.

Definition *2.3.1 (Graph) A graph is a tuple (V, E, src, tgt) where V is a set of nodes (or vertices), E is a set of edges and $src, tgt : E \to V$ associate, respectively, a source and target node to each edge in E. Given graphs G_1 and G_2, a graph morphism is a pair (f_V, f_E) of total functions $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ such that sources and targets of edges are preserved, i.e., $f_E \circ src_1 = f_V \circ src_2$ and $f_E \circ tgt_1 = f_V \circ tgt_2$.*

An E-graph is additionally equipped with a set of data nodes, used to provide the values of node and edge attributes.

Definition *2.3.2 (E-Graph) An E-graph is a graph equipped with an additional set V_D of data nodes (or values) and special sets of edges E_{EA} (edge attributes) and E_{NA} (node attributes) connecting, respectively, edges in E and nodes in V to values in V_D. An E-graph corresponds to a diagram in the category $Sets$, as shown in Fig. 2.1(a). An E-graph morphism $f : EG_1 \to EG_2$ is a 5-tuple $(f_V, f_E, f_{V_D}, f_{E_{EA}}, f_{E_{NA}})$ of total functions between E-graph components such that all sources and targets are preserved.*

Intuitively, an attributed graph is an E-graph where V_D corresponds to elements of the carrier set of D sorted by S. This way the components D and S define the data values that can be attributed to nodes and edges in the E-graph part.

$$\begin{tikzcd}
 & E \arrow[rr, shift left, "source"] \arrow[rr, shift right, "target"'] & & V & \\
E_{EA} \arrow[ur, "source_{EA}"] \arrow[dr, "target_{EA}"'] & & & & E_{VA} \arrow[ul, "source_{EA}"'] \arrow[dl, "target_{EA}"] \\
 & \arrow[r, phantom] & V_D & &
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
D_s^1 \arrow[r, "f_{D,s}"] \arrow[d, hook] & D_s^2 \arrow[d, hook] \\
V_D^1 \arrow[r, "f_{G,V_D}"'] & V_D^2
\end{tikzcd}$$

(b)

Figure 2.1: E-Graph (Left) and condition for A-graph morphisms (Right)

Definition *2.3.3 (A-graph) An attributed graph (A-graph) is a tuple (EG, D, S) where EG is an E-graph, D is an algebra with signature $\Sigma = (S, OP)$ and $S_0 \subseteq S$ is a subset of sort names such that $\biguplus_{s \in S_0} D_s = V_D$. An A-graph morphism $f : AG^1 \rightarrow AG^2$ is a pair (f_{EG}, f_D) such that $f_{EG} : EG^1 \rightarrow EG^2$ is a E-graph morphism, $f_D : D^1 \rightarrow D^2$ is an algebra morphism and, for every sort name $s \in S_0^1$, the diagram shown in Fig. 2.1(b) commutes.*

We restrict the set of graphs under consideration by a type graph.

Definition *2.3.4 (TA-graph) An attributed type graph is an A-graph $ATG = (EG, Z)$ where Z is the final algebra of its signature. A typed attributed graph (TA-graph) over ATG is a pair (AG, t) where AG is an A-graph and $t : AG \rightarrow ATG$ is an A-graph morphism. Given two TA-graphs TAG_1, TAG_2 over ATG, a TA-graph morphism $f : TAG_1 \rightarrow TAG_2$ is an A-graph morphism $f : AG_1 \rightarrow AG_2$ such that $t_2 \circ f = t_1$.*

Let us denote by $X = (X_s)_{s \in S}$ a family of countable sets of variables, indexed by sorts $s \in S$. The sort s represents the type of all variables in X_s. We also write $x : s \in X$ for $x \in X_s$. Considering a fixed attributed type graph (ATG) with signature $\Sigma = (S, OP)$ and variables X, an ATG-typed graph rule (or production) is a span $L \xleftarrow{l} K \xrightarrow{r} R$ where l, r are monomorphisms, the algebra component of L, K, R is $T_\Sigma(X)$ (the term algebra of Σ with variables in X), and the algebra component of the rule morphisms

$l_D = r_D = id_{T_\Sigma(X)}$ is the identity on the term algebra [22]. This means that, in particular, the names of variables are preserved across the entire rule. The class of all rules over ATG with variables in X is denoted $Rules(ATG, X)$. We define typed attributed graph transformation system (TAGTS) as follows:

Definition *2.3.5 (TAGTS) A typed attributed graph transformation system (TAGTS) is a tuple (ATG, P, π) where ATG is an attributed typed graph, P is a set of rule names and $\pi : P \to Rules(ATG, X)$ maps rule names to ATG-typed graph rules.*

We define a single-step transformation and a transformation sequence as follows:

Definition *2.3.6 (Transformation) Given an ATG-typed graph G, an ATG-typed rule $L \xleftarrow{l} K \xrightarrow{r} R$ and a match (i.e. a ATG-typed graph morphism) $m : L \to G$, a* transformation step *from G to H using q (based on m) exists if and only if the diagram below can be constructed, where both squares are pushouts in* $\mathbf{AGraph_{ATG}}$ *such that G, C, H share the same algebra D and the algebra components l^*_D, r^*_D of morphisms l^*, r^* are identities on D. In this case the transformation step is denoted by $G \overset{q,m}{\Longrightarrow} H$. A transformation sequence is a sequence $G_0 \overset{r_1,m_1}{\Longrightarrow} G_1 \overset{r_2,m_2}{\Longrightarrow} \ldots \overset{r_n,m_n}{\Longrightarrow} G_n$ of transformation steps.*

$$\begin{tikzcd}
L \arrow[d, "m=d_L"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "d_K"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "m^*=d_R"] \\
G & C \arrow[l, "l^*"] \arrow[r, "r^*"'] & H
\end{tikzcd}$$

This ensures that data elements are preserved across transformation sequences, which allows their use as actual parameters within a global namespace.

2.4 Running Example

We introduce an example, which we also treat as a running example throughout the rest of the thesis. We consider a service for managing hotel guests. A registered guest can book a room subject to availability. There are no booking charges and the bill starts to accumulate once the room is occupied. Since credit card details are already with the hotel management, the bill is automatically deducted when the guest announces their intention to check out. The guest can check out successfully only when the bill is paid. The operation signatures for some of the operations are shown in Listing 8.1. We call the web service implementation "implementation" and TAGT specifying the visual contracts "model" throughout the rest of the document.

Listing 2.1: HotelImplementationInt

```
        ...
public interface IHotel {
        ...
    public String bookRoom(String name, int room_no);
    public String occupyRoom(String name, int room, int bill);
    public boolean updateBill(int bill_no, int amount);
    public boolean clearBill(int bill_no) throws Exception;
    public boolean checkout(String name, int room_no, int bill_no);
    public String viewData(int room_no);
        ...
}
```

We specify the operations as visual contracts which are discussed in [51, 31, 50]. We associate a signature with each visual contract where we distinguish input and output parameters. The signature is using the name of the visual contract and the variables in attribute expressions. Consider Figure 2.2, where the signature *bookRoom(r:int, n:String)* has parameters r and n which are also used in the contract *bookRoom* to represent the possible sets of values for *room* and *name* in an abstract manner.

The type graph for this service is shown in Fig. 2.4 using AGG [2] notation. The signature associated with the visual contract in the Figure 2.4 returns results of running a query.

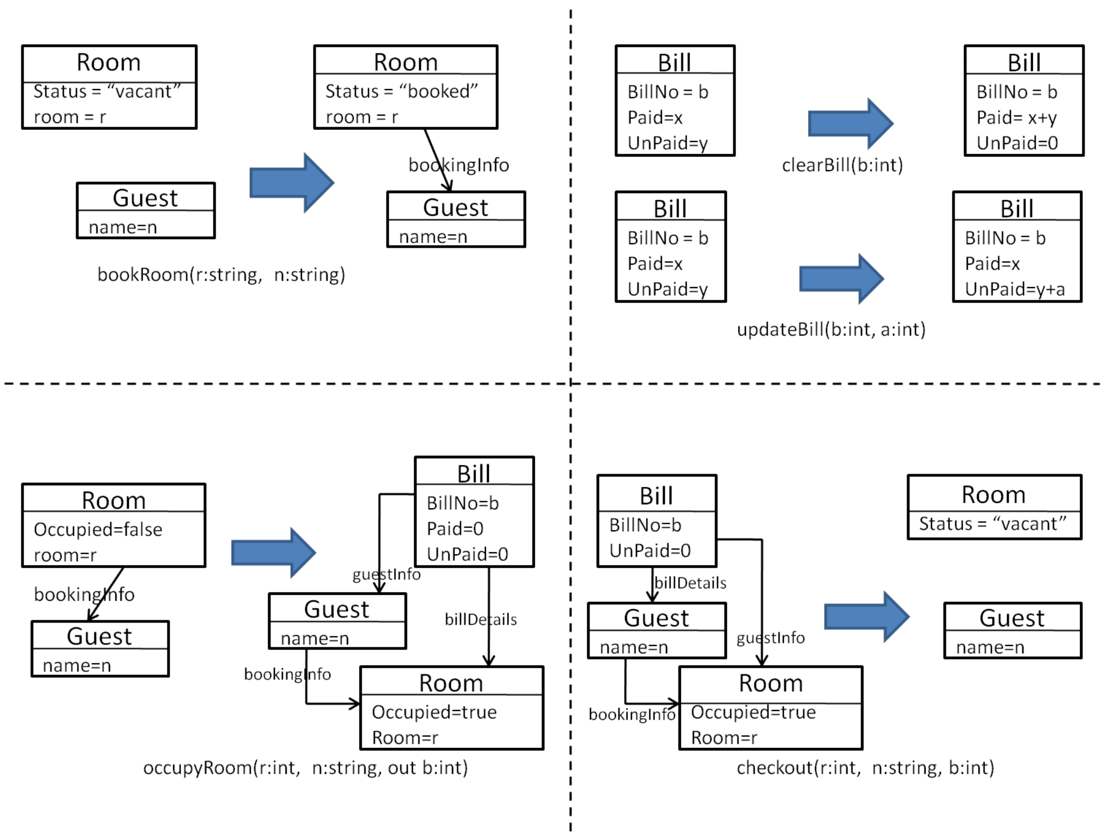


Figure 2.2: Visual Contracts for Hotel Example

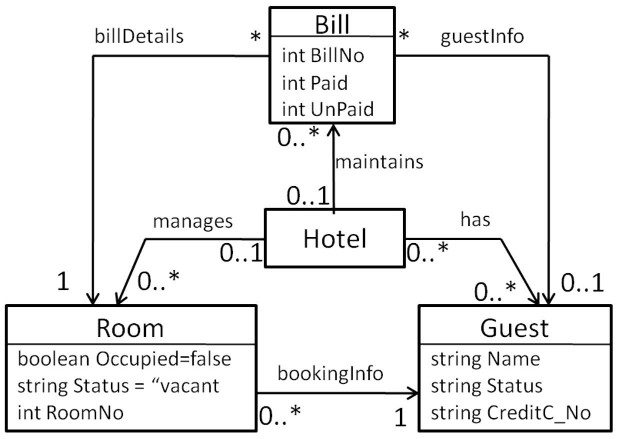


Figure 2.3: Type Graph

We use our running example as a vehicle to explain our contributions and a case study, introduced in Chapter 8, to evaluate our proposals. The system used as case study is adopted from a desktop bug tracking application available online such that the

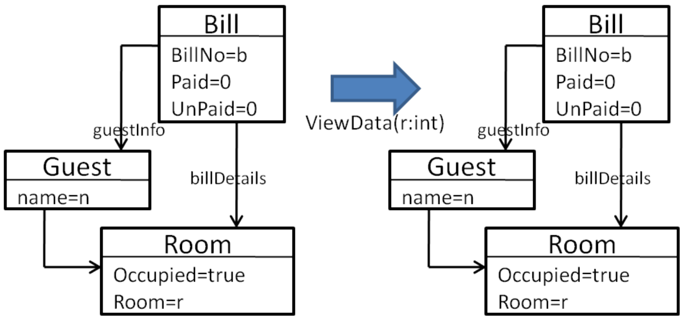


Figure 2.4: Rule to query information

GUI is replaced with service interface and is offering more than thirty operations. The system was selected as a case study since it was big enough to show the limitations especially in terms of scalability.

2.5 Independence and Critical Pairs

In software systems, there are situations where we need to invoke a particular operation to apply a certain other operation. Consider, for example, the running example presented in Figure 2.2 where we need to book a room using *bookRoom* before occupying a room using *occupyRoom*. There is a sequential dependence between operations *bookRoom* and *occupyRoom*. Instead, the operations *clearBill* and *viewData* do not depend on each other for application and therefore, they are sequentially independent.

Definition *2.5.1 (sequential independence) Consider a graph G, and two transformation steps* $G \overset{p_1,m_1}{\Longrightarrow} H$ *and* $H \overset{p_2,m_2}{\Longrightarrow} I$ *where* $p_1 = (l_1, r_1)$ *and* $p_2 = (l_2, r_2)$*. The transformations are said to be* sequential independent *iff there exists morphisms* $i : R_1 \to D_2$ *and*

$j : L_2 \to D_1$ such that $m_1' = r_1' \circ j$ and $m_2 = l_2' \circ i$, as shown below:

$$\begin{tikzcd}
L_1 \arrow[d, "m_1"'] & K_1 \arrow[l, tail, "l_1"'] \arrow[r, tail, "r_1"] \arrow[d, "k_1"] & R_1 \arrow[dr, "m_1'"'] \arrow[drrr, dotted, "i"] & & L_2 \arrow[dl, "m_2"] \arrow[dlll, dotted, "j"'] & K_2 \arrow[l, tail, "l_2"'] \arrow[r, tail, "r_2"] \arrow[d, "k_2"] & R_2 \arrow[d, "m_2'"] \\
G & D_1 \arrow[l, "l_1'"] \arrow[rr, "r_1'"'] & & H & & D_2 \arrow[ll, "l_2'"] \arrow[r, "r_2'"'] & I
\end{tikzcd}$$

If a transformation prevents the application of another, we have a conflict. Consider, the running example presented in Figure 2.2. The room occupied by a guest cannot be booked or occupied again before the guest checks out. If two transformation steps do not disable each other, we say they are parallel independent. Below, we define parallel independence.

***Definition** 2.5.2 (parallel independence) Consider a graph G, and two transformation steps $G \overset{p_1,m_1}{\Longrightarrow} H$ and $H \overset{p_2,m_2}{\Longrightarrow} I$ where $p_1 = (l_1, r_1)$ and $p_2 = (l_2, r_2)$. The transformations are said to be* parallel independent *iff there exists morphisms $i : L_1 \to D_2$ and $j : L_2 \to D_1$ such that $m_1' = r_1' \circ j$ and $m_2 = l_2' \circ i$, as shown below:*

$$\begin{tikzcd}
R_1 \arrow[d, "m_1'"'] & K_1 \arrow[l, tail, "r_1"'] \arrow[r, tail, "l_1"] \arrow[d, "k_1"] & L_1 \arrow[dr, "m_1"'] \arrow[drrr, dotted, "i"] & & L_2 \arrow[dl, "m_2"] \arrow[dlll, dotted, "j"'] & K_2 \arrow[l, tail, "l_2"'] \arrow[r, tail, "r_2"] \arrow[d, "k_2"] & R_2 \arrow[d, "m_2'"] \\
H_1 & D_1 \arrow[l, "r_1'"] \arrow[rr, "l_1'"'] & & G & & D_2 \arrow[ll, "l_2'"] \arrow[r, "r_2'"'] & H_2
\end{tikzcd}$$

A pair $P_1 \overset{p_1,m_1}{\Longleftarrow} K \overset{p_2,m_2}{\Longrightarrow} P_2$ of transformation steps is called a critical pair if it is parallel dependent and matches m_1 and m_2 are jointly surjective [22].

***Definition** 2.5.3 (critical pairs[47]) A critical pair is pair of transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ such that m_1 and m_2 are jointly surjective and:*

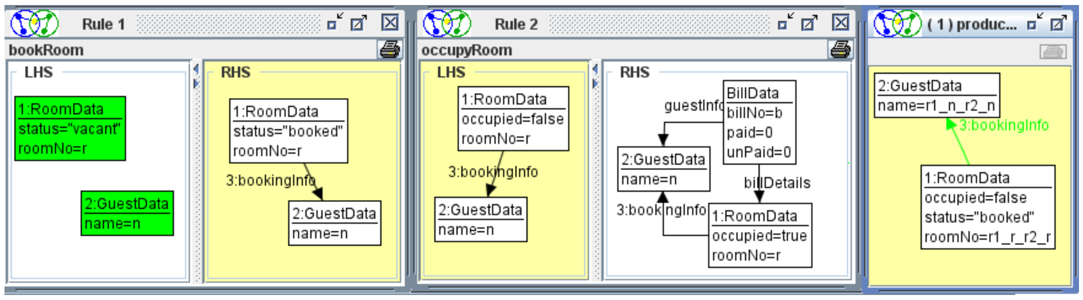
- *there's no $i : L_1 \to D_2$ with $l_2' \circ i = m_1$*

  or

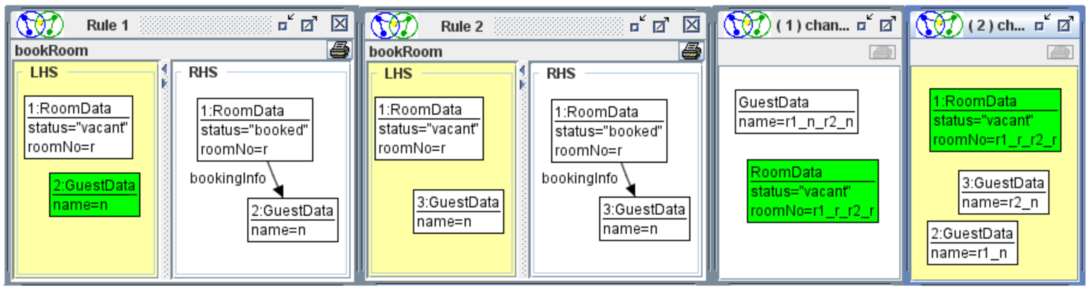
- *there's no* $j : L_2 \to D_1$ *with* $l'_2 \circ j = m_2$

The following types of conflicts can occur [38]:

1. Delete-Use: Application of rule p_1 deletes an object subsequently required for application of rule p_2.
2. Change-Use: Application of rule p_1 manipulates the attributes that are in the match of p_2.
3. Delete-Delete: Application of rules p_1 and p_2 delete an object subsequently required for application of the other rule.



(a) Produce-Use Dependency Example



(b) Change-Use-Attribute Conflict Example

Figure 2.5: Dependencies and Conflicts Example (a) and (b)

An example of conflicts and dependencies is shown in Fig. 2.5 where two rules *bookRoom* and *occupyRoom* are considered in Fig. 2.5(a) to illustrate an example of produce-use dependencies. The first rule produces an edge which is in the match of the second. The example shown in Fig. 2.5(b) illustrates the conflict where the application of *bookRoom* changes the *RoomData* variable *status* from *vacant* to *booked* and disables the second application of *bookRoom* which uses this attribute.

Change-Use conflicts are handled by deleting the edge between the node representing the attribute and the value in the data part of the E-graph, as shown by the schematic diagram in Figure 2.1, and creating another edge between the same node and another value in the algebra part of the graph. We therefore classify change-use, i.e., a conflict where the attribute value is changed as a delete-use conflict.

2.6 Summary

This chapter introduced the basic notation of software testing and coverage criteria. We also introduced basic graph transformation concepts together with visual contracts. These definitions are required for our theoretical developments and empirical results, which we discuss in the forthcoming chapters. Finally, we have introduced a running example that we use to explain our contributions.

Part II

Model-Based Testing

Chapter 3

Typed Attributed Graph Transformation Systems with Rule Signatures

This chapter provides the theoretical basis for the association between visual contracts and rules of a TAGTS. This is required since we specify service operations as visual contracts based on signatures, using formal parameters which map to the rules' local variables. We allow to associate more than one rule to a signature for specifying alternative outcomes of an operation. We introduce a notion of observation on transformation steps based on rule signatures to relate transformation sequences and test cases.

We first provide our definition of TAGTS with rule signatures, where we differentiate between input and output parameters.

3.1 Rule Signatures

Extending Definition 2.3.5 introducing TAGTS, we define TAGTS with rule signatures.

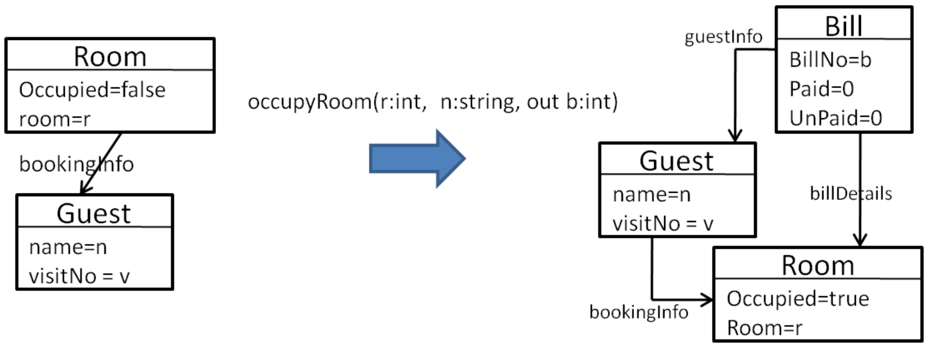
Definition *3.1.1 (TAGTS with rule signatures) A typed attributed graph transformation system with rule signatures is a tuple* $\mathcal{G} = (ATG, P, X, \pi, \sigma)$ *where*

- *ATG is an attributed type graph with set of node attributes* E_{NA}*,*
- *P is a countable set of rule names,*
- *X is an S-indexed family* $(X_s)_{s \in S}$ *of sets of variables,*
- $\pi : P \longrightarrow \mathcal{P}_{fin}(Rules(ATG, X))$ *assigns each rule name a finite set of rules* $L \xleftarrow{l} K \xrightarrow{r} R$ *over ATG, X,*
- $\sigma : P \to (\{\epsilon, out\} \times X)^*$ *assigns to each rule name* $p \in P$ *a list of formal inout and output parameters* $\sigma(p) = \bar{x} = (q_1 x_1 : s_1, \ldots, q_n x_n : s_n)$ *were* $q_i \in \{\epsilon, out\}$ *and* $x_i \in X_{s_i}$ *for* $1 \leq i \leq n$*. We write p's* rule signature $p(\bar{x})$ *and refer to the set of all rule signatures as* signature of $\mathcal{G}$*.*

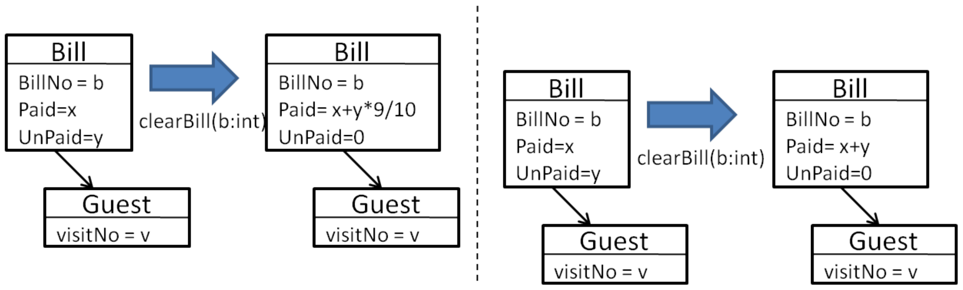
We distinguish inout and output parameters, the latter being indicated by *out* in front of the declaration. Normal parameters are both input and output, that is they are given in advance of the application, restrict the matching and are still valid after the rule has been applied. Output parameters are only assigned values during matching.

Since $L \xleftarrow{l} K \xrightarrow{r} R$ is attributed over $T_\Sigma(X)$, rule parameters $x_i \in X_{s_i}$ are from the set of variables used in attribute expressions. Hence, actual parameters will not refer to nodes or edges, but to attribute values in the graph. Since the algebra part of attributed graphs is preserved, actual parameters have a global name space across transformation sequences.

We can associate several rules to the same signature to represent alternative actions inside the same operation, chosen by different input values and the system's internal state. In our example, the hotel provides a 10% discount to its guests on every tenth visit. In order to describe this, two rules are required: One is applicable if the current visit of a particular guest is the tenth one, while the second is applicable otherwise. We consider the rules shown in Figure 2.2 where we amend *occupyRoom* to keep track of the guest's visits and introduce two rules *clearBill_1* and *clearBill_2* replacing rule *clearBill* shown in Figure 3.1



(a) Rule occupyRoom



(b) Rule clearBill_1 (c) Rule clearBill_2

Figure 3.1: Revised Hotel Example

***Example** 3.1.1 [TAGTS with rule signatures] For the system in Figure 2.2, the rule signatures shown below are based on data sorts $S = \{int, boolean, string\}$ with the usual operations. Output parameters are indicated by the prefix* out *in the declaration.*

- *bookRoom(n:string, out r:int)*
- *occupyRoom(r:int, n:string, out b:int)*
- *clearBill(b:int)*
- *checkout(r:int, n:string, b:int)*
- *updateBill(b:int, in:int)*

- *viewData(r:int)*

The rules *updateBill_1* and *updateBill_2* in Figure 3.1 are associated with the same signature, i.e., *updateBill(int:b)*. We can construct transformation sequences by successive rule applications starting from the initial graph shown in Figure 3.2.

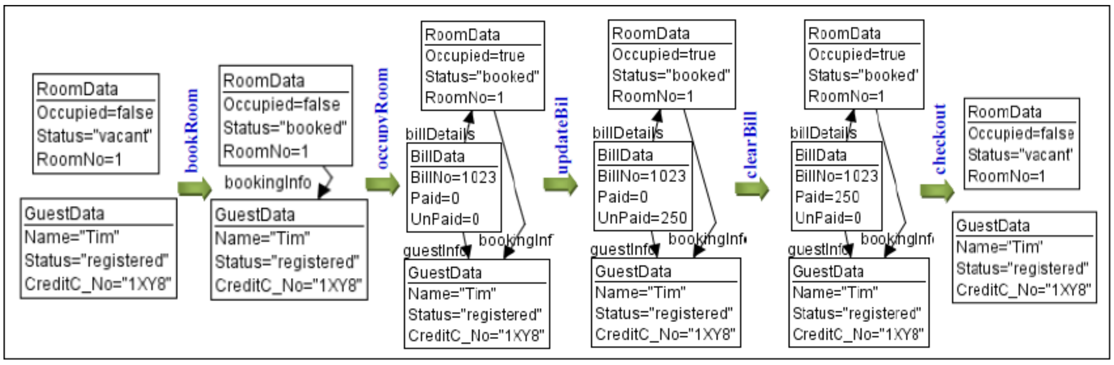


Figure 3.2: An example transformation sequence

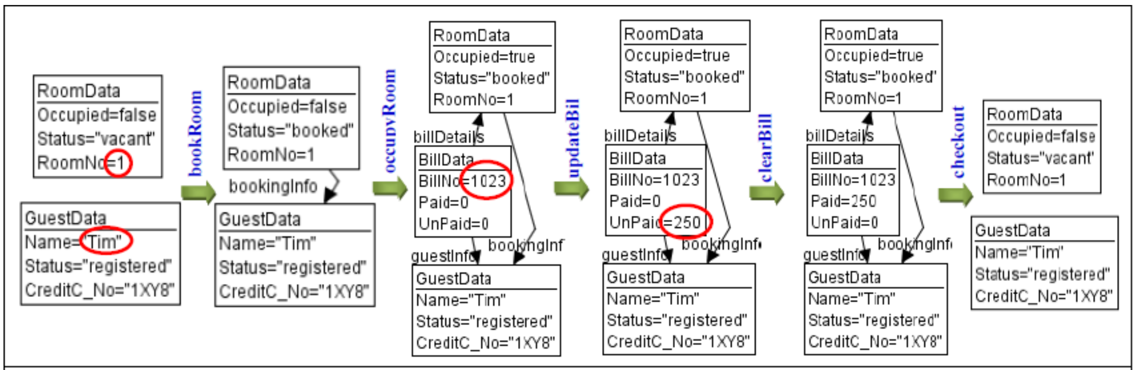
The purpose of signatures is to allow observations on transformations, including information about rules and their matches. Below we define the labels of a rule signature, then the observations associated with transformation sequences.

Definition *3.1.2 (labels) Given a rule $p : L \leftarrow K \rightarrow R$ with signature $p(q_1 x_1 : s_1, \ldots, q_n x_n : s_n)$ and a Σ-algebra D, we denote by $p(D)$ the set of all rule labels $p(a_1, \ldots, a_n)$ with $a_i \in D_{s_i}$. The label alphabet $L_{\mathcal{G},D}$ for a system $\mathcal{G}$ is defined as the union over all rule labels $\bigcup_{p \in P} p(D)$. If D and/or $\mathcal{G}$ are understood from the context, we write $L_{\mathcal{G}}$ or just L.*

The (usually infinite) alphabet of labels L consists of all possible instances of rule signatures, replacing their formal parameters by values from the algebra D. Labels in L may be interpreted as observations of transformation steps, where the instantiation is given by the algebra component of the matches. Let L^* denote the Kleene closure over the label alphabet, providing the set of all finite sequences of labels. The following definition describes an observational semantics of TAGTS via sequences of labels produced by its transformation sequences.

Definition *3.1.3 (observations from transformation sequences) Let* $G \overset{p,m}{\Longrightarrow} H$ *be a transformation step of a TAGTS* $\mathcal{G}$ *with algebra component D. The observation function* $obs : Tra(\mathcal{G}) \to L^*_{\mathcal{G}}$ *is defined on transformation steps by* $obs(G \overset{p,\,m}{\Longrightarrow} H) = p(a_1, \ldots, a_n)$ *if p's signature is* $p(q_1 x_1 : s_1, \ldots, q_n x_n : s_n)$ *with* $a_i = m(x_i)$*. The observation function freely extends to finite sequences of transformations, yielding sequences of labels.*

Consider the transformation sequence shown in Figure 3.2, the observation sequence considering $n =$ "Tim" and $r = 1$ is shown in Figure 3.3



bookRoom("Tim", 1); occupyRoom("Tim",1,1023); updateBill(1023,250); clearBill(1023);checkout("Tim",1,1023)

Figure 3.3: An example transformation sequence and an observation sequence

Rule names are instantiated with parameter values to represent the corresponding observation sequence in the lower part of Figure 3.3. The graphs shown represent sample states of a hotel with only one room and one registered guest.

A test case combines a graph defining its initial state with a sequence of invocations, i.e., rule names instantiated by constants or terms $T_\Sigma(Y)$ over a set of program variables

Y.

Definition *3.1.4 (invocation sequence, test case) Given a rule $p : L \leftarrow K \rightarrow R$ with signature $p(q_1 x_1 : s_1, \ldots, q_n x_n : s_n)$ with $q_i \in \{\epsilon, out\}$ and a sort-indexed family of variables $Y = (Y_s)_{s \in S}$, an invocation of p is of the form $p(t_1, \ldots, t_n)$ where $t_i \in T_\Sigma(Y)_{s_i}$ are terms with operations from Σ over the variables Y, such that $q_i = out$ implies that $t_i \in Y$, i.e., output parameters are instantiated by variables only.*

Given a sequence of m invocations, $s = p_1(t_{11}, \ldots, t_{1n_1}); \ldots ; p_m(t_{m1}, \ldots, t_{mn_m})$, for a prefix s_i of s we say that a variable is bound by s_i if it occurs in s_i as output parameter. An invocation sequence is ground *if for each invocation $p_1(t_{i1}, \ldots, t_{in_i})$, all variables occurring in inout parameters are bound by s_{i-1}.*

A test case is a pair $t = (G_0, s)$ of a graph G_0, called start graph, and a ground invocation sequence s.

Therefore, only input parameters are instantiated and output parameters as well as their subsequent occurrences are not initialized until the execution reaches the point where the parameter's value is computed.

Example *3.1.2 [test case] Considering the transformation sequence in Figure 3.2, a test case $t = (\mathrm{G}_0, s)$ is shown below where the graph G_0 is given in Figure 3.4.*

$$s = bookRoom(\text{“}Tim\text{”},\ 1); occupyRoom(\text{“}Tim\text{”},\ 1,\ bNo); updateBill(bNo,\ 250); \\ clearBill(bNo); checkout(\text{“}Tim\text{”},\ 1,\ bNo)$$

A test case, once executed, becomes an observation sequence as shown in the lower part of Figure 3.3.

Definition *3.1.5 (instantiation and run of test case) Given a test case $t = (G_0, p_1(t_{11}, \ldots, t_{1n_1}); \ldots ; p_m(t_{m1}, \ldots, t_{mn_m})$ with program variables in Y and*

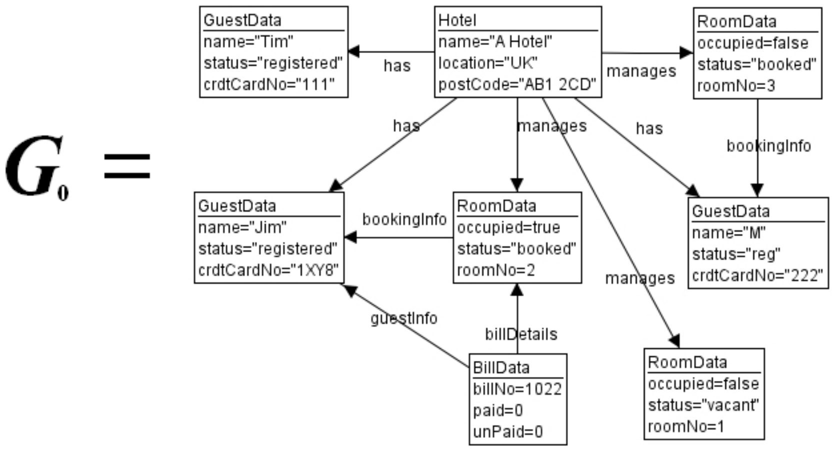


Figure 3.4: Start graph for test case example

sort-indexed assignment $a : Y \to D$, *the instantiation of s via a is given by* $a(t) = (G_0, p_1(\overline{a}(t_{11}), \ldots, \overline{a}(t_{1n_1})); \ldots; p_m(\overline{a}(t_{m1}), \ldots, \overline{a}(t_{mn_m})))$.

Given a transformation sequence $ts = (G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \overset{p_2,m_2}{\Longrightarrow} \ldots \overset{p_n,m_n}{\Longrightarrow} G_n)$ *of a TAGTS* $\mathcal{G}$ *and a test case t as above, ts is a run of t if there exists an assignment* $a : Y \to D$ *such that* $a(t) = obs(ts)$.

Example *3.1.3 [run of a test case] We obtain an observation sequence by instantiating invocations. Considering the invocation sequence given in Example 3.1.2 and assuming* $bNo = 1023$, *we get a run of a test case as shown below:*

$bookRoom(``Tim", 1); occupyRoom(``Tim", 1, 1023); updateBill(1023, 250);$
$clearBill(1023); checkout(``Tim", 1, 1023)$

The test run is the same as the observation sequence shown in the lower part of Figure 3.3.

Definition 3.1.5 is used to define under what conditions a transformation sequence can be seen as the result of executing a test. A test case contains the information about the initial state of system as a start graph. An observation sequence is achieved by applying the observation function, as defined in Definition 3.1.3, on a sequence of transformation steps.

The variables in a test case represent placeholders which are assigned values depending upon the state of the model. While executing the test case, we consider the first invocation in the test case and find the match of the corresponding rule in the start graph considering the parameter values in the attribute expressions. In case the match is found, we apply the match and find the comatch. The match and the comatch together provide the values for the abstract parameters in the invocation. For the next invocation, the match has to consider the values assigned to abstract parameters in previously executed invocation. The invocations in a test case become labels resulting in an observation sequence. We represent a test case as an observation sequence throughout the rest of the thesis where rule signatures are instantiated with all variables instead of just the input variables.

3.2 Summary

We have presented a way to associate signatures with rules in a TAGTS and defined a relation between observation sequences and test cases. We have provided an informal but operational explanation of these concepts. We use this formal basis in the following chapters for introducing our approaches to test oracles, dependency graphs, model-based coverage and regression testing.

Chapter 4

Test Oracles Using Visual Contracts

This chapter is dedicated to use of visual contracts as test oracles. We execute graph transformation rules using the application programming interface (*API*) of AGG [2]. We provide an adapter to AGG such that the model's functionality is comparable to the interface of the service to be tested. The system under test (SUT) is tested in a three staged process. The oracle is invoked for obtaining the expected result of the test. Then, the corresponding operation of the SUT is executed and finally, the two results are compared to see if there was any deviation. The adapter is required to translate invocations of services under test into rule applications, passing and converting parameters and interpreting replies.

There are two fundamental differences, at the conceptual level, between visual contracts as models and implementations. Firstly, models specify functional requirements and, secondly, they do not include error handling or reporting. Graph transformation rules can be executed using AGG [2], either through its graphical user interface or via an API. Using AGG to execute our model, we provide an adapter to present the model's functionality in a way that is comparable to the interface of the services to be tested. The test driver shown in Figure 4.1 implements a three step process. First, it invokes the oracle, obtaining the expected result of the test. Then, the corresponding operation of the system under test (SUT) is executed. The third step is to compare this response with the expected one and record any deviations.

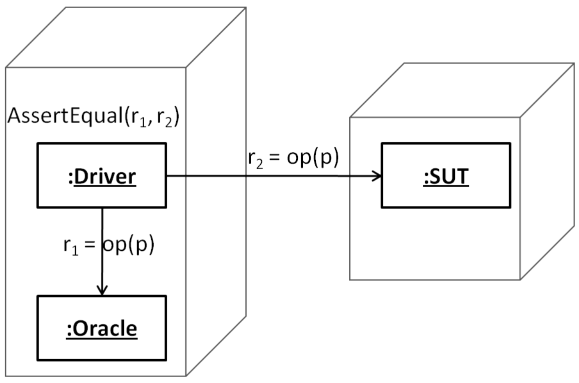


Figure 4.1: Schematic diagram of proposed driver

The adapter is required to translate invocations of services under test into rule applications, passing and converting parameters and interpreting replies. Conversion is required because model and implementation signatures may differ, with the implementation requiring extra parameters, providing additional results, or using different types. Since the model is only concerned with functional aspects, we also have to filter out technical failures of the implementation, such as problems with the server or transport layer, distinguishing them from logical failures corresponding to non-applicable rules due to violation of preconditions. However, with visual contracts providing a partial specification only, even the functional aspect may be under-specified. Moreover, different web services implementations may report success and failure differently. Adapter and test driver need to be flexible enough to accommodate different styles of error handling and reporting, for example, allowing for a degree of customization.

4.1 Challenges

In our approach, the oracle executing the TAGTS is invoked through an adapter to utilize the API of AGG 2.0 as shown in Figure 4.2. The driver invokes the oracle through the adapter and the system under test through the provided interface. The adapter is a generic component which uses the exposed API of AGG for executing the model. However, the comparison of the expected with the actual result raises a number of challenges, which we will discuss below.

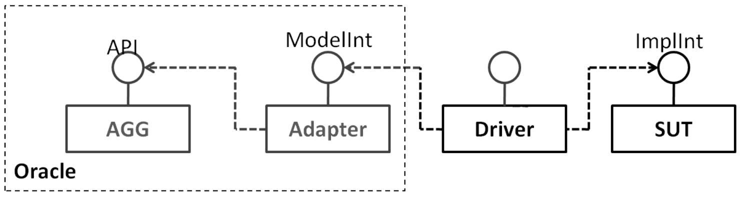


Figure 4.2: Component diagram

We also consider visual contracts which return result as a multi-object. AGG provides their implementation as rule schemes. We add one more rule to our hotel example we have taken as a running example, as shown in Figure 4.3.

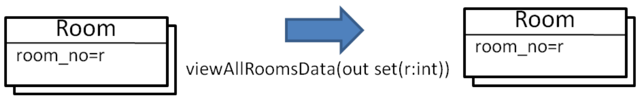


Figure 4.3: Additional rule for finding all rooms

The comparison between the expected and the actual output is performed using JUnit assert statements where required.

4.1.1 Model as Oracle

We represent visual contracts as production rules in a TAGTS using AGG. The tool does not provide a mechanism to associate a rule with a signature. The driver, however, requires to invoke the model by the same inputs used to invoke the implementation and needs to receive the computed outputs to compare the predicted and actual results. This is only possible if we can provide a mapping between the inputs and outputs of the model signature to the rule inputs and outputs, which is the role of the adapter in Figure 4.2.

4.1.2 Partiality of Visual Contracts

A visual contract may specify the intended behavior of the implementation only partially. Therefore, if the oracle predicts success and the implementation reports logical failure, this could either be due to an error in the implementation or the under-specified precondition of the contract. While comparing model and implementation responses, such a case should not be reported as a failed test, but as a warning, providing sufficient details so that the developer can decide the correct interpretation. Table 4.1 details all possible cases and will be more fully explained in the next subsection. The 5th row represents the situation where the oracle gives a response while the implementation reports a logical failure.

An under-specified postcondition would lead to a lack of synchronization between model and implementation state, because changes performed on the latter would not have to be matched by the former. Therefore, post-conditions are assumed to be complete.

4.1.3 Failure Handling

There are different ways in which failures can be reported to the test driver or client. We distinguish them by raising the following questions.

1. What is the origin of the failure?
2. How was the failure presented?
3. How is the failure interpreted?

A failure may have its origin either on the server or in communication. Server-side failure can be due to logical or technical reasons. Logical failures occur if the pre-condition of an operation is not satisfied, i.e., the application is invoked, but not executed correctly or not at all. Technical failures can be down to a variety of reasons, such as the database being off-line, server-side system failures, power fluctuations, hardware issues, etc. Communication failures result from loss of network access, congestion causing delays, etc.

A server-side failure presents itself as an exception, a fault message, or an application-specific error code, while a communication failure shows as an exception (timeout) on the client side.

We interpret these failure presentations as follows. If the client-side receives a logical failure, we expect a violation of the contract's pre-condition. If the client receives a technical failure, the oracle cannot provide a matching response since the model only covers the functional aspect. Similarly, if the client receives a communication failure, the comparison between expected and actual response is meaningless.

Table 4.1: Exception response table

Oracle response	**Result**	SUT **response**
r	$=$	r
r_1	$\neq_{post}$	r_2
non-applicable	$=$	logical-failure
non-applicable	undefined	technical failure
r	?	logical failure
r	undefined	technical failure
non-applicable	$\neq_{pre}$	r

We list the cases in Table 4.1, where r indicates equal responses from oracle and implementation in case of successful execution and r_1, r_2 represent successful responses that differ from each other. Equality "=" shows that the test was executed and successful whereas "undefined" means that, for all we know, the test case was not executed. If responses do not match due to non-applicability of a pre-condition in the model, this is represented by "$\neq_{pre}$", whereas if model and implementation produce different responses, this is represented by "$\neq_{post}$". The first and the third rows represent cases where both responses match, i.e., both show either successful execution or failure. The second row represents cases where the pre-condition was satisfied, yet the output was different. The forth and the sixth rows represent cases where the SUT experienced a technical failure and hence the test case was not executed. The fifth row represents cases where the oracle generated a response and the SUT returned a logical error. This case is reported to the developer as a possible failure that needs further investigation, as it may be due to a faulty implementation or a partially specified visual contracts. This is marked with “?" in the results column. Lastly, the seventh row represents cases where the oracle reported non-satisfaction of pre-condition, whereas the implementation generated a response.

4.1.4 Adaptation of Output Types

If implementation and model share the same signature, expected and actual output can be compared directly. However, there are cases where the implementation returns a result in the form of a complex type, such as a collection of objects. In such cases, the oracle returns its response as a set of nodes and the adapter needs to process this set into a suitable form for the driver to carry out the comparison.

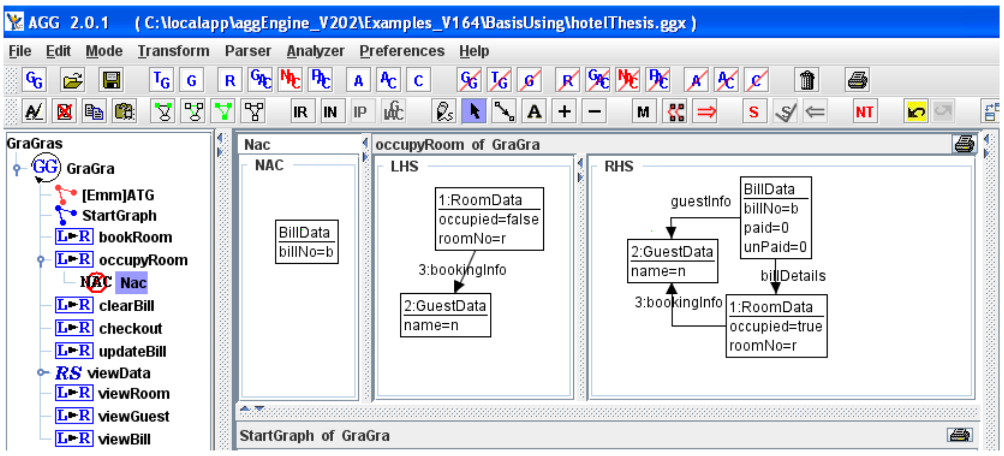
The implementation signatures can extend the model signatures by providing an additional response to indicate if the operation was successful. This response can be in the form of a numerical error code, a string, or a Boolean. One example from our running example is *bookRoom(. . .)*, where the success message of the implementation is a string “Room Booked...". The model instead reports successful execution of a rule by means of a Integer return (the room number booked) via the AGG API. We therefore not only need to maintain separate signatures, but also adapt the results to make them comparable.

4.2 Using AGG as an Oracle

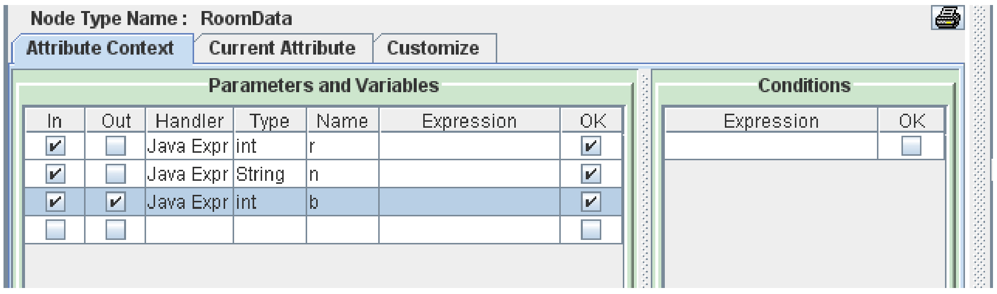
Having listed the challenges in using visual contracts as oracles, we discuss how we have used the AGG 2.0 [1] engine [2] and appropriate adapter and driver implementations to overcome these challenges.

4.2.1 Model as Oracle

In order to link model signatures with production rules in AGG, we map their formal parameters to the parameters and variables in the attribute context of the rules. An example is shown in Figure 4.4.



(a) Visual Contract as Rule in AGG



Node Type Name : RoomData

Attribute Context | Current Attribute | Customize

Parameters and Variables

In	Out	Handler	Type	Name	Expression	OK
✓	☐	Java Expr	int	r		✓
✓	☐	Java Expr	String	n		✓
✓	✓	Java Expr	int	b		✓
☐	☐					☐

Conditions

Expression	OK
	☐

(b) Attribute Context

Figure 4.4: Visual Contract as Rule and Attribute Context Example (a) and (b)

This enables the driver to use the same input parameters to invoke the model and the

[1]available at `http://user.cs.tu-berlin.de/~gragra/agg/`

implementation operations. This is enabled by the adapter, which wraps an invocation of the AGG API inside a call to an operation based on the model signature. Examples of such invocations are shown in Listing 8.2 (e.g., on line 11).

However, if the signatures involve multi-objects as shown in Figure 2.4, we make use of rule schemes where an amalgamated transformation [27] returns the set of nodes corresponding to the multi-object on the right-hand side of the rule. This is shown in Figure 4.5 where the start graph, as shown in Figure 4.7 contains four rooms and the right-hand side of the amalgamation shows all of them selected as nodes. Our rule schemes implement all-quantified operations on recurring graph patterns. The kernel rule is a common subrule of a set of multi-rules. It is matched only once, while multi-rules are matched as often as suitable matches are found. In AGG an amalgamated rule is constructed from all matches found for multi-rules that share the match of the kernel rule. The rule scheme for operation *viewAllRooms(...)* is shown in Figure 2.4. Our adapter is written such that we access the resulting set of nodes and process them to provide appropriate response as a structure containing the data values.

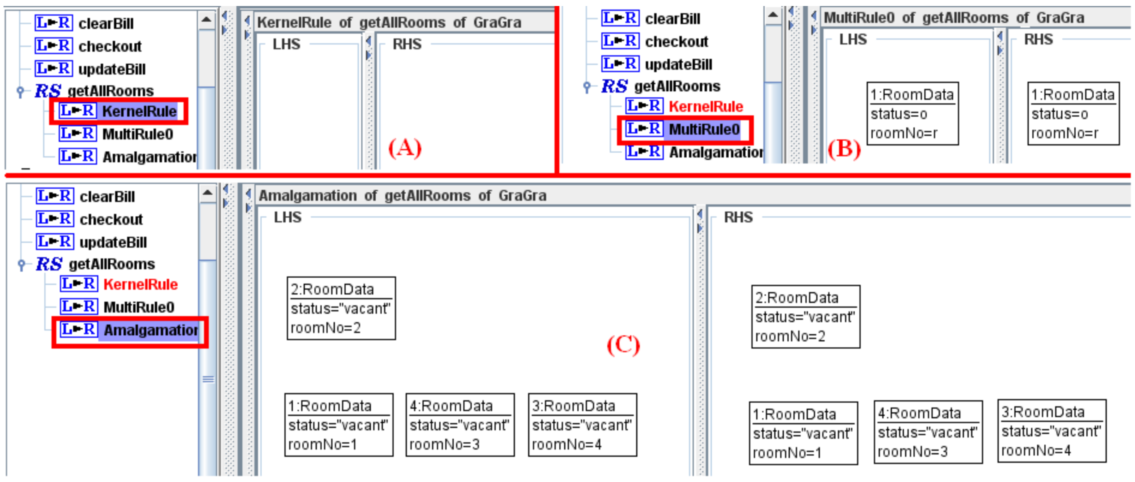

clearBill
checkout
updateBill
getAllRooms
KernelRule
MultiRule0
Amalgamation
KernelRule of getAllRooms of GraGra
LHS
RHS
(A)
MultiRule0 of getAllRooms of GraGra
1:RoomData status=o roomNo=r
(B)
Amalgamation of getAllRooms of GraGra
2:RoomData status="vacant" roomNo=2
1:RoomData status="vacant" roomNo=1
4:RoomData status="vacant" roomNo=3
3:RoomData status="vacant" roomNo=4
(C)

Figure 4.5: Additional rule for finding all rooms

In the example of Figure 2.4, the kernel rule of the rule scheme for operation *viewAllRooms(...)* is the empty rule. That means the kernel rule (and so the rule scheme) is applicable to every graph, including the empty graph. In AGG, the multi-rule of the rule scheme *viewAllRooms* identifies a room. When the multi-rule is matched to all rooms, we get as many room nodes in the left- and right-hand side of the amalgamated rule. Our adapter provides access to the list of these nodes and processes them to return the data for comparison to the driver.

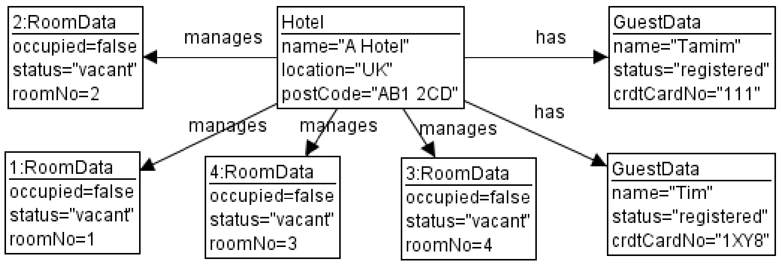


Figure 4.6: Start Graph

At the start of testing, we assume that the start graph of the model and the initial state of the implementation are in sync. Typically, neither of them have any non-mandatory data. The synchronization of model and implementation states is maintained by their co-evolution as long as the results match, assuming that post-conditions of visual contracts are completely specified. We consider a start state of the system, as shown in Figure 4.7, for the model as well the implementation.

4.2.2 Partiality of Visual Contracts

It remains to deal with under-specified preconditions. The adapter allows to observe applicability of rules and their generated output. If the intended behavior is specified only partially, the model may generate a response while the implementation returns a logical failure. In this case, we provide developers with both responses and the stack trace detailing the reasons for the mismatch. The developers can choose to ignore the results and subsequent test executions using annotation *@Ignore* if a detailed analysis reveals that the response is due to a partially specified visual contract.

Listing 4.1: JUnit Test:AddProject

```
SN    Test Case                 Oracle          SUT                          Result
==    =========                 ======          ===                          ======
1     bookRoom("Tim", 1)           1            Room Booked ...                =
2     bookRoom("Tim", 1)       not-applicable   Already Booked...              =
  ...
```

Based on the information in a log file, we can also compile a summary report where all detailed test reports are condensed as shown in Listing 4.1, referring to the test cases in Listing 8.2. The first row represents a successful execution of model and implementation. The second row represents logical failure responses from model and implementation. Hence, the system passes both tests.

4.2.3 Failure Handling

Our discussion in subsection 4.1.3 revealed that comparison between the responses of oracle and implementation is possible only in a subset of cases. In case of a technical server or communication failure, there is either no response or a timeout on the implementation side. Since the oracle only covers functional aspects, in this case comparison is not possible. In order to avoid further processing, we check that there is no timeout and use an assert statement *assertNotNull(. . .)* before comparing the oracle's with the implementation's output, as shown in Listing 8.2 (e.g., line 6 and line 18).

If the client receives a logical failure from the server, we check the response of the oracle. If the oracle returns *true*, our custom assertion *assertBothSucceededOrBothFailed(. . .)* evaluates to *false* since there is a mismatch between the two responses. However, if the oracle also reports a logical failure, our custom assertion evaluates to *true*. Listing 8.2, line 20, demonstrates the usage of the assertion. We have already booked a room (line 5 and line 12) and are trying to book it again (line 17 and line 18) with the same credentials. Since the oracle's response is *false* and the implementation returns “Already booked", our custom assertion returns *true*.

It is important to point out that we do not perform literal comparison between the expected and the actual result since the implementation can report a failure in a variety of ways. Examples include error messages, e.g., "error occurred..." or "Error code: 9876, check documentation", etc. Therefore, our custom assertions are general enough to be able to compare a variety of different failure representations coming from the implementation with the oracle's response. The composition of *assertBothSucceededOrBothFailed(. . .)* is such that it has two parts, *assertBothSucceeded(. . .)* to compare the successful cases and *assertBothFailed(. . .)* to compare the failure cases. The second part deals with cases where the oracle gives a Boolean response. We allow the developer to define how the implementation reports failure, based on a set of predefined alternatives.

Listing 4.2: JUnit Test:bookRoomTests

```
...
  log.info(" - Test Case run: " + this.getClass()+ ": IMPL. RESP. -\n");
  BookRoom bookRoom43 = new BookRoom();
  bookRoom43.setName(name); bookRoom43.setRoom_no(room_no);
  myAssert.successMessage = result;
  HotelServiceExampleStub.BookRoomResponse bRes = stub.bookRoom(bookRoom43);
  assertNotNull(bRes);
  String res1 = bRes.get_return();
...
  aggEngine agg = new aggEngine();
  ArrayList<String> list = new ArrayList<String>();
  list.add("\"" + name + "\"");
  list.add(Integer.toString(room_no));
  boolean res2 = agg.aggResult("C:\\hotelThesis.ggx", "bookRoom", list);
  myAssert.assertBothSucceededOrBothFailed((Object)res2,(Object)res1);
...
  BookRoom bookRoom44 = new BookRoom();
  bookRoom44.setName(name); bookRoom44.setRoom_no(room_no);
  myAssert.customFaultCode = "already booked";
  HotelServiceExampleStub.BookRoomResponse bRes2=stub.bookRoom(bookRoom44);
```

```
...
boolean res4 = agg.applyRule("bookRoom", list);
 try {
     assertNotNull(res4);
     assertFalse(res4);
     myAssert.assertBothSucceededOrBothFailed((Object)res4,(Object)res3);
 }
 catch (AssertionFailedError e1) {
      System.out.println("------  TEST FAILURE ------ \n");
      System.out.println("------  STACK TRACE ------- ");
      e1.printStackTrace();
      throw e1;
 }

...
```

We are left with cases where both implementation and oracle have reported successful invocation. The next subsection discusses the different cases for comparing output values.

4.2.4 Adaptation of Output Types

If the signatures of model and implementation are the same, the output value is passed to the driver through the variable *result*. The standard *assertEquals(...)* can deal with this case.

If the implementation signature is an extension of the model signature, we proceed as follows. The oracle informs the driver that the pre-condition was satisfied or not. The custom assertions allow us to compare the oracle's response with the implementation's, recording the latter. This allows the driver to know how a particular implementation responds in success cases. This is demonstrated in Listing 8.2 where we first check, by using *assertNotNull*, if the service invocation was successful and by using *assertTrue* if the rule application was successful. The driver then uses a custom assertion *assertBothSucceededOrBothFailed(...)* by initializing the expected result string to see if the results were compatible (lines 4 to 16).

Listing 4.3: JUnit Test:viewAllRooms

```
   ...
1.   HotelServiceExampleStub.ViewAllRoomsDataResponse vRes=stub.viewAllRoomsData();
   ...
2.   List<List<String>> roomsList = new ArrayList<List<String>>();
3.   for (int i = 0; i < result.length; i++) { ... }
   ...
4.   boolean res2 = agg.aggEngineGetAll("C:\\localapp\\hotelThesis.ggx", "↩
     viewAllRooms");
5.   List<List<String>> roomList = new ArrayList<List<String>>();
6.   roomList = agg.nodeStruct;
7.   assertNotNull(res2); assertTrue(res2);
8.   myAssert.assertSetEquivalent(roomsList, roomList);
   ...
```

If the implementation returns execution results in terms of a complex type, we access the data in the response object and the resulting set of nodes from a collection named *nodeStruct* in the oracle and use custom assertion *assertSetEquivalent(...)* to compare the two sets of values. Listing 4.3 presents an example. We receive the multi-object from the implementation (Listing 4.3, line 1) and extract the result in the form of a list. We invoke our oracle and access the result as another list (Listing 4.3, line 6) and test by using our custom assertion, *assertSetEquivalent(...)* (line 8), if the two responses were the same.

4.3 Implementation

We have implemented an adapter which makes use of the API exposed by AGG. The role of the adapter is to receive information about the name of the rule and the parameter values from the driver. The adapter invokes the rule by instantiating the variables in the attribute context for a possible rule application. Listing 4.4 explains how the input variables are instantiated.

Listing 4.4: Setting input parameters

```
...
VarTuple vars = (VarTuple) r.getAttrContext().getVariables();
for (int i=0; i<vars.getNumberOfEntries(); i++) {
    VarMember var = (VarMember) vars.getMemberAt(i);
    if (var.isInputParameter()) {
    if (!var.isSet()) {
        inputPars.add(value);
        var.setInputParameter(true);
        var.setExprAsText(value); varset = varset + value + ", ";
    }
    else {
...
```

Once the input parameters are instantiated, the next step is to find a match consistent with the variable binding. The system needs to find out if the selected operation corresponds to a rule or a rule scheme. The program execution follows two different paths depending upon the outcome. The process of finding out if the selected rule was a rule scheme is shown in Listing 4.5.

Listing 4.5: Executing Step considering rule scheme

```
...
Rule r = gragra.getRule(rulename);
if (r instanceof RuleScheme) {
    RuleScheme rs = (RuleScheme) r;
    if (doStepOfAmalgamatedRule(rs, gragra.getGraph(), gragra.↩
        getMorphismCompletionStrategy()))
        System.out.println("RuleScheme, Name: " + rs.getName()+"  was applied");
}
...
```

If the rule considered for application is a rule scheme, the API call finds and applies the match differently and the result is also obtained from the system in a different way. In case of a rule scheme, we apply match considering the rule scheme. Listing 4.6 shows how the match is found and how the result is obtained from the rule scheme whereas the Listing 4.7 (line-1) shows how a match is found in case for a normal rule.

Listing 4.6: Rule scheme application

```
...
AmalgamatedRule amalgamRule = rs.getAmalgamatedRule(g, s);
if (amalgamRule != null)
{
    Match match = amalgamRule.getMatch();
    try {
        Morphism co = StaticStep.execute(match);
        HashSet<Node> nodeList = co.getOriginal().getNodesSet();
        Node node = null;
        Iterator<Node> iterator = nodeList.iterator();
...
```

In case the rule is not a rule scheme, the adapter finds a match which considers the values passed to all the input variables in the attribute context of the rule. In case, a valid match is not found, the rule is not applicable and the adapter returns a Boolean response “false” to the driver indicating that the rule application considering the given set of input data was not successful.

In case of a successful match, the transformation step is performed and the value assignment to the output parameter is found out and shared with the driver. This is in addition to the result information in the shape of a Boolean showing that the transformation was possible by the adapter.

The driver is responsible for a comparison between the response returned by the implementation and the model. Listing 4.7 shows the steps involved in this process.

Listing 4.7: Rule Application

```
...
Match match = gratra.createMatch(r);
gragra.setGraTraOptions(this.strategy);
match.setCompletionStrategy(this.strategy, true);
while (match.nextCompletion())
{
    if (match.isValid())
    {
        try {
            Morphism co = StaticStep.execute(match);
            didTransformation = true;
            outPar = getOutputParameters(r, outPar);
            for (int i = 0; i<outPar.size(); i++) {
                VarMember p = outPar.get(i);
                String result = getValueOfOutputParameter(p, r, (↩
OrdinaryMorphism) co);
...
```

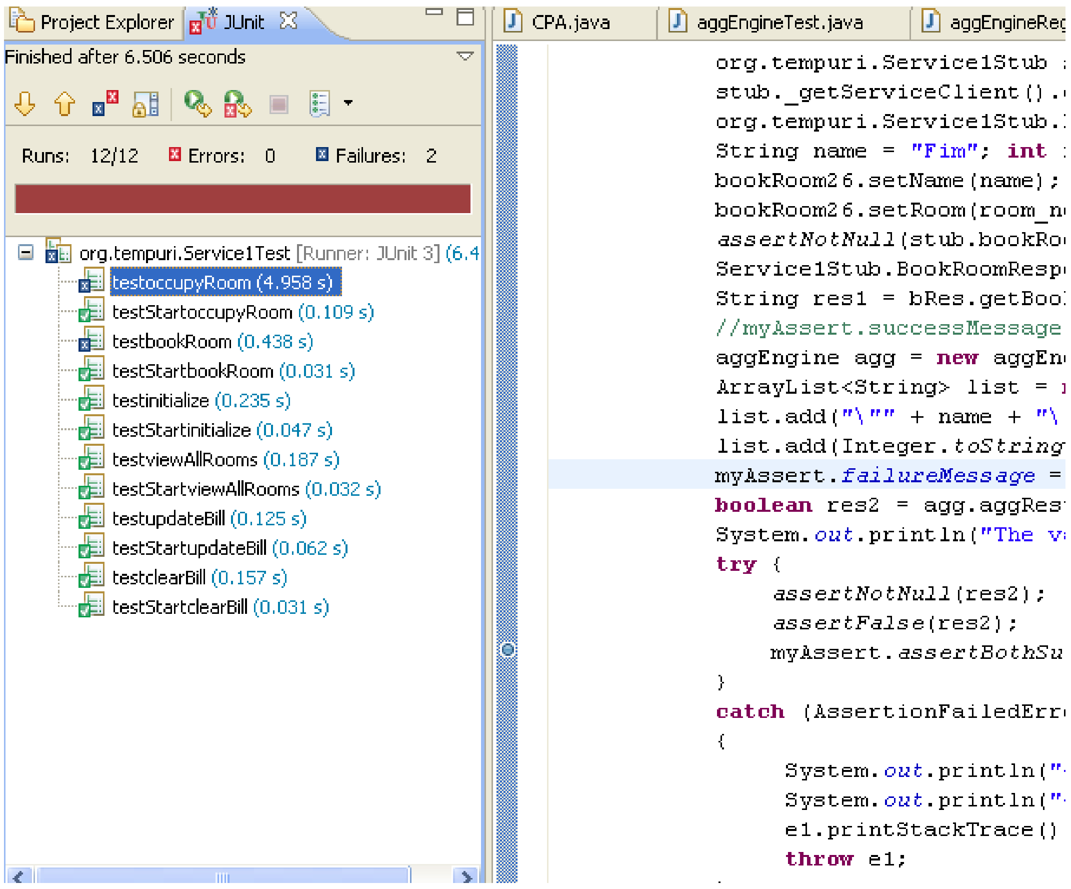
The difference in extracting the return values is also presented in Listing 4.6 and Listing 4.7 where the former returns the result as a node set which needs to be treated to extract the required information whereas the API is used for extracting the return value in the later case. We have also implemented custom assertions to support a variety of success as well as failure response as discussed in the preceding section. These custom assertions also allow us to compare sets of values from the model and the implementation.

4.4 Application to the Running Example

We implemented our hotel example as a web service written in C# using MS Visual Studio which is a software development tool by Microsoft[2]. We developed unit test cases using *JUnit* and tested the application as shown in Listing 8.2 and Listing 4.3.

[2]`http://www.microsoft.com/visualstudio/en-gb`

Our initial result was reporting failures in *bookRoom* and *occupyRoom* where a mismatch was found between the server and the model responses. Referring to Table 4.1, both were example of row 2, where there are responses different from each other.



```
Project Explorer | JUnit                    CPA.java | aggEngineTest.java | aggEngineReg
Finished after 6.506 seconds

Runs: 12/12    Errors: 0    Failures: 2

org.tempuri.Service1Test [Runner: JUnit 3] (6.4
    testoccupyRoom (4.958 s)
    testStartoccupyRoom (0.109 s)
    testbookRoom (0.438 s)
    testStartbookRoom (0.031 s)
    testinitialize (0.235 s)
    testStartinitialize (0.047 s)
    testviewAllRooms (0.187 s)
    testStartviewAllRooms (0.032 s)
    testupdateBill (0.125 s)
    testStartupdateBill (0.062 s)
    testclearBill (0.157 s)
    testStartclearBill (0.031 s)
```

```
org.tempuri.Service1Stub :
stub._getServiceClient().
org.tempuri.Service1Stub.
String name = "Fim"; int :
bookRoom26.setName(name);
bookRoom26.setRoom(room_n
assertNotNull(stub.bookRo
Service1Stub.BookRoomResp
String res1 = bRes.getBoo
//myAssert.successMessage
aggEngine agg = new aggEn
ArrayList<String> list = 
list.add("\"" + name + "\
list.add(Integer.toString
myAssert.failureMessage =
boolean res2 = agg.aggRes
System.out.println("The v
try {
     assertNotNull(res2);
     assertFalse(res2);
     myAssert.assertBothSu
}
catch (AssertionFailedErr
{
      System.out.println("
      System.out.println("
      e1.printStackTrace()
      throw e1;
```

Figure 4.7: Additional rule for finding all rooms

The structure of the unit test cases is already presented and discussed in detail for Listing 8.2 and Listing 4.3. We first invoke the service considering a particular operation and then we execute the model, using AGG, considering the corresponding rule, passing the parameters using our adapter. We then use our custom assertions to find out if the two responses could be considered as a success or failure. We implemented 12 test cases for the hotel example presented in Figure 2.2.

4.5 Limitations of the Approach

Our oracle consists of models in AGG which does not support directly the association of rule signatures to productions rules. We have proposed a way to associate signatures but, currently, it is a manual process. An automation would provide better support for our proposal since it would eliminate the possibility of a mismatch between the rule and the signature. Another weakness is that we can only use basic data types as signature inputs while associating the rule signatures with production rules. AGG allows basic data types supported by Java but the programmatic solutions to consider complex data types as inputs can also be implemented. The adapter is designed such that it receives information about the name of the rule and the parameter values from the driver and it also takes care of the difference of the rule signatures between the model and the implementation.

In order to make the approach applicable in practice, a solution is required to bridge the gap in abstraction level and scope between model-based oracle and implementation automatically. That means for the adapter implementing the model interface to be generated automatically. Adopting the techniques described in Chapter 4, this is straightforward except for contracts containing multi-objects which are realized in AGG by means of amalgamated rules and for which the extraction of input and output parameters has to be implemented manually. More generally, this would require integrating into AGG and its API the concept of rule signatures defined over the attribute context of a rule. The test driver on the other hand, with its customizable assertions, is required to accommodate variations in the use of exceptionts and implementation-specific error messages, which can vary widely based on the conventions of the application at hand. Automation here is meaningful only if these conventions could be formalized and made part of the model.

4.6 Summary

In this chapter, we have used high level visual specifications for oracle development. Our proposal can equally be adopted to test component-based systems since the idea is based on executing visual contracts using AGG.

In Chapter 6, we make use of the oracle for finding out if a conflict or a dependency between two rule applications is exercised at runtime.

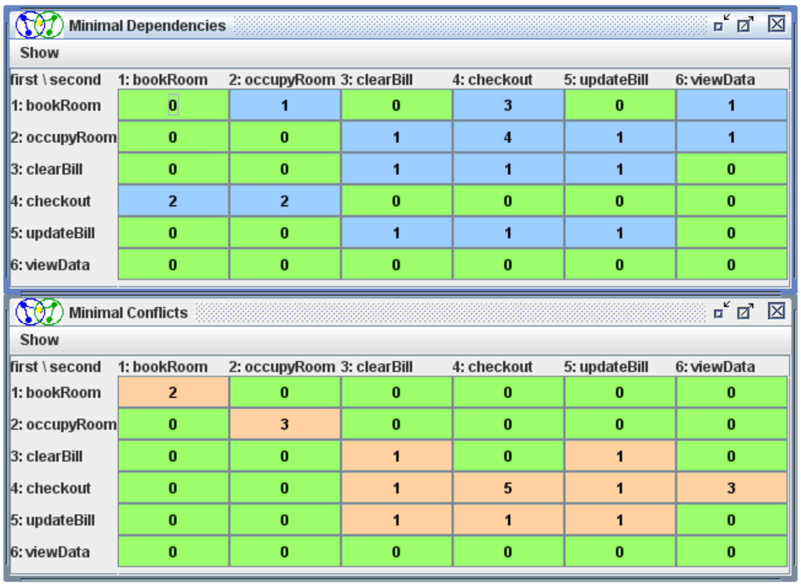
Chapter 5

Model-Based Coverage Criteria

This chapter is devoted to introducing our model-based coverage criteria. A user's view of a web service is through provided interfaces, which abstract from implementation details and prevent us from using traditional testing methods based on source code [15]. We propose to replace code-based by model-based coverage criteria using semantic service descriptions at the interface level. Specifying the provided operations by visual contracts, formally typed attributed graph transformation rules, we analyze their potential conflicts and dependencies [22]. We generate a dependency graph whose nodes represent rules while its edges indicate potential conflicts or dependencies between them. They also carry labels showing the nature of the relation, allowing us to record where data was defined, used, updated, or deleted. Our coverage criteria will make use of this information.

5.1 Dependency Graphs

In this section, we show how to extract a dependency graph for a system under test (SUT) from the available interface specification based on visual contracts. A dependency graph (DG) provides us with a visual representation of conflicts and dependencies allowing us to study coverage criteria at the interface level. The nodes represent rules while its edges indicate the nature of the relation between the nodes i.e., a conflict or dependency. The edges also bear annotations at the start and the end of the edge to represent if the data was created, read, updated or deleted by these rule applications. We use conflicts and dependency analysis explained in Section 2.5 and take $\prec$ and $\nearrow$ to represent asymmetric versions of these relations. We make use of dependencies and conflicts analysis to find out if two rule applications depend on each other or disable each other. Consider rules p and q, if q depends upon p, we write $p \prec q$ and if q disables p, we write $p \nearrow q$. The theoretical basis behind this analysis are provided below.


Minimal Dependencies

Show

first \ second	1: bookRoom	2: occupyRoom	3: clearBill	4: checkout	5: updateBill	6: viewData
1: bookRoom	0	1	0	3	0	1
2: occupyRoom	0	0	1	4	1	1
3: clearBill	0	0	1	1	1	0
4: checkout	2	2	0	0	0	0
5: updateBill	0	0	1	1	1	0
6: viewData	0	0	0	0	0	0

Minimal Conflicts

Show

first \ second	1: bookRoom	2: occupyRoom	3: clearBill	4: checkout	5: updateBill	6: viewData
1: bookRoom	2	0	0	0	0	0
2: occupyRoom	0	3	0	0	0	0
3: clearBill	0	0	1	0	1	0
4: checkout	0	0	1	5	1	3
5: updateBill	0	0	1	1	1	0
6: viewData	0	0	0	0	0	0

Figure 5.1: Critical pairs and dependencies

Definition *5.1.1 (asymmetric dependencies and conflicts) Given two rules* p_1, p_2 *we say that* p_2 may disable p_1 *, written* $p_1 \nearrow p_2$*, if there are steps* $G \overset{p_1,m_1}{\Longrightarrow} H_1$ *and* $G \overset{p_2,m_2}{\Longrightarrow} H_2$

without $k : L_1 \to D_2$ such that $m_1 = l_2^* \circ k$.

$$\begin{tikzcd}
R_1 \arrow[d, "m_1^*"'] & K_1 \arrow[l, "r_1"'] \arrow[r, "l_1"] \arrow[d, "k_1"'] & L_1 \arrow[dr, "m_1"] \arrow[drrr, dotted, "k"] & & L_2 \arrow[dl, "m_2"'] & K_2 \arrow[l, "l_2"'] \arrow[r, "r_2"] \arrow[d, "k_2"] & R_2 \arrow[d, "m_2^*"] \\
H_1 & D_1 \arrow[l, "r_1^*"] \arrow[rr, "l_1^*"'] & & G & & D_2 \arrow[ll, "l_2^*"] \arrow[r, "r_2^*"'] & H_2
\end{tikzcd}$$

We say that p_1 may enable *p_2, written $p_1 \prec p_2$, if there are steps $G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \overset{p_2,m_2}{\Longrightarrow} G_2$ without $j : L_2 \to D_1$ such that $m_2 = r_1^* \circ j$.*

$$\begin{tikzcd}
L_1 \arrow[d, "m_1"'] & K_1 \arrow[l, "l_1"'] \arrow[r, "r_1"] \arrow[d, "k_1"'] & R_1 \arrow[dr, "m_1^*"] & & L_2 \arrow[dl, "m_2^*"'] \arrow[dlll, dotted, "j"'] & K_2 \arrow[l, "l_2"'] \arrow[r, "r_2"] \arrow[d, "k_2"] & R_2 \arrow[d, "m_2^*"] \\
G & D_1 \arrow[l, "l_1^*"] \arrow[rr, "r_1^*"'] & & G_1 & & D_2 \arrow[ll, "l_2^*"] \arrow[r, "r_2^*"'] & G_2
\end{tikzcd}$$

We have represented conflicts and dependencies between rules in our running example in Table 5.1. For example we find a dependency

$$\textit{bookRoom} \prec \textit{occupyRoom}$$

Similarly, there exists a conflict

$$\textit{updateBill} \nearrow \textit{checkout}$$

i.e., *updateBill* reads the *BillData* object, changing the *unpaid* amount, while *checkout* deletes the object.

Using our definition of asymmetric conflicts and dependencies, we define the dependency graph as follows:

Definition *5.1.2 (dependency graph) A dependency graph $DG = \langle G, OP, op, lab \rangle$ is a structure where*

Table 5.1: Conflicts $\nearrow$ and dependencies $\prec$ between rules

First/Second (↓) / (→)	*bookRoom*	*occupyRoom*	*clearBill*	*checkout*	*updateBill*	*viewData*
bookRoom	$\nearrow$	$\prec$		$\prec$		$\prec$
occupyRoom		$\nearrow \prec$		$\prec$	$\prec$	$\mid$
clearBill			$\nearrow \prec$	$\prec \nearrow$	$\nearrow \prec$	$\mid$
checkout	$\prec$	$\prec$	$\nearrow$	$\nearrow$	$\nearrow$	
updateBill			$\nearrow \prec$	$\nearrow \prec$	$\nearrow \prec$	$\mid$
viewData		$\mid$	$\mid$	$\nearrow$	$\mid$	$\mid$

- $G = \langle V, E, src, tar\rangle$ *is a graph,*
- *OP is a set of (names of) operations,*
- $op : V \to OP$ *maps vertices to operation names,*
- $lab : E \to \{c, r, d\} \times \{\prec, \nearrow\} \times \{c, r, d\}$ *is a labeling function distinguishing source and target types* <u>c</u>reate, <u>r</u>ead, <u>d</u>elete *and dependency types* $\prec, \nearrow$.

We use the visual contracts specifying the interface to extract a dependency graph, where rules are represented by nodes labeled by operation names while edges represent dependencies and conflicts between them. Edge labels tell us whether an edge represents a dependency ($\prec$) or a conflict ($\nearrow$) and what roles are played by the source and target nodes.

Definition *5.1.3 (dependency graph of TAGTS with rule signatures) Given a TAGTS with rule signatures* $\mathcal{G} = (ATG, P, X, \pi, \sigma)$*, its dependency graph* $DG(\mathcal{G}) = \langle G, OP, op, lab\rangle$ *with* $G = (V, E, src, tar)$ *is defined by*

- $V = \bigcup\limits_{p \in P} (\{p\} \times \pi(p))$ *as the set of all rule spans tagged by their names. If* $s_1 \in \pi(p)$ *we write* $p_1 : s_1 \in V$.
- $E \subseteq V \times V$ *such that:*
  - $e = (p_1 : s_1, p_2 : s_2) \in E$ *if there are steps* $G \overset{p_1:s_1,m_1}{\Longrightarrow} H_1 \overset{p_2:s_2,m_2}{\Longrightarrow} H_2$ *such that the second step requires the first. The role labels are defined as follows.*
    1. *If an element created by the first step is read by the second,* $lab(e) = \langle c, \prec, r\rangle$.

2. *If an element created by the first step is deleted by the second, $lab(e) = \langle c, \prec, d\rangle$ such that label d takes precedence over r.*

– $e = (p_1 : s_1, p_2 : s_2) \in E$ *if there are steps* $H_1 \overset{p_1:r_1,m_1}{\Longleftarrow} G \overset{p_2:r_2,m_2}{\Longrightarrow} H_2$ *such that the second disables the first. The role labels are defined as follows.*

1. *If an element deleted by the first step is also deleted by the second, $lab(e) = \langle d, \nearrow, d\rangle$.*

2. *If an element read by the first step and deleted by the second, $lab(e) = \langle r, \nearrow, d\rangle$ such that label d takes precedence over r.*

- *$OP = P$ is the set of rule names.*
- *$op : V \to OP$ is defined by $op(p : s) = p$*

Example *5.1.1 [dependency graph] Using the example in Fig. 2.2 we can draw a dependency graph as shown in Fig. 5.2. Consider edge between nodes* bookRoom *and*

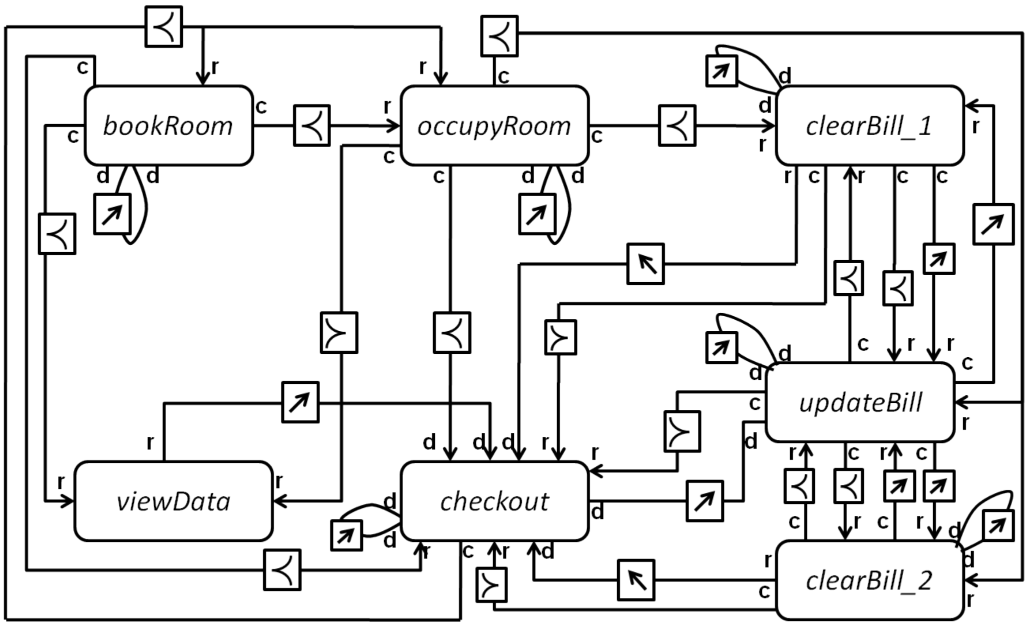


Figure 5.2: Dependency Graph of $TAGTS$ representing hotel web service

occupyRoom *where the labeling is $\langle c, \prec, r\rangle$. That means, an object created during the* *first operation* bookRoom *is read by the second operation* occupyRoom *with* $\prec$ *representing the dependency relation. Similarly, consider an edge between* clearBill *and* checkout *where the labeling is* $\langle r, \nearrow, d \rangle$*. It means,* clearBill *reads an object which is deleted by* checkout*. An examination of the rules reveals that* clearBill *operates on a* BillData *object which is deleted by* checkout*.*

Consider the cr edge between updateBill *and* checkout *which is due to attribute* unpaid*. The first rule sets it to a value* 0 *whereas the second reads it, as part of the match, to operate.*

5.2 Coverage Criteria

Dataflow graphs contain annotations on the nodes to mark places where data is defined and used in a program [3], [20], [67]. The locations where a variable is defined are annotated by *def*, use is indicated by *use*, and deallocated by *kill*. Paths are identified through the system such that they exercise particular coverage criteria. For example, $def-use$ coverage requires to find test cases such that all edges in the dataflow graph from nodes annotated with *def* to nodes annotated with *use* are exercised at least once.

Our dependency graphs carry information about dependencies between operations. We have annotated sources and targets of edges with *c* (create), *r* (read), and *d* (delete). These annotations are analogous to *def*, *use* and *kill* annotations for traditional versions of dataflow graphs which also represent the places where the data is defined, used, and deallocated. Using these labels on edges rather than nodes, we can focus on the type of access that gives rise to the particular dependency or conflict represented by the edge. Hence, the information contained in our *DG* allows us to propose model-based criteria which are suitable to the platform-independent nature of web services. If we consider a given set of test cases, we can analyze its coverage if we construct the sub-graph of *DG* covered by the set. The possibilities are summarized in Table. 5.2.

Table 5.2: Label combinations indicating conflicts and dependencies

Label Combination	**Conflict**	**Dependency**
cr	×	√
cd	×	√
rd	√	×
dd	√	×

Using the *DG*, we can devise different coverage criteria such as *cr*, *cd*, etc. The question is, which of these pairs to include into our criteria. If we demand all *cr* (create-read), and *cd* (create-delete) edges in $DG(\mathcal{G})$, we exercise all dependencies based on data being defined and used subsequently. If we add *cd* (create-delete) and *rd* (read-delete), we also cover situations of asymmetric conflict. In order to see if a set of test cases *T* provides the required coverage, we record all the nodes and edges that *T* is exercising.

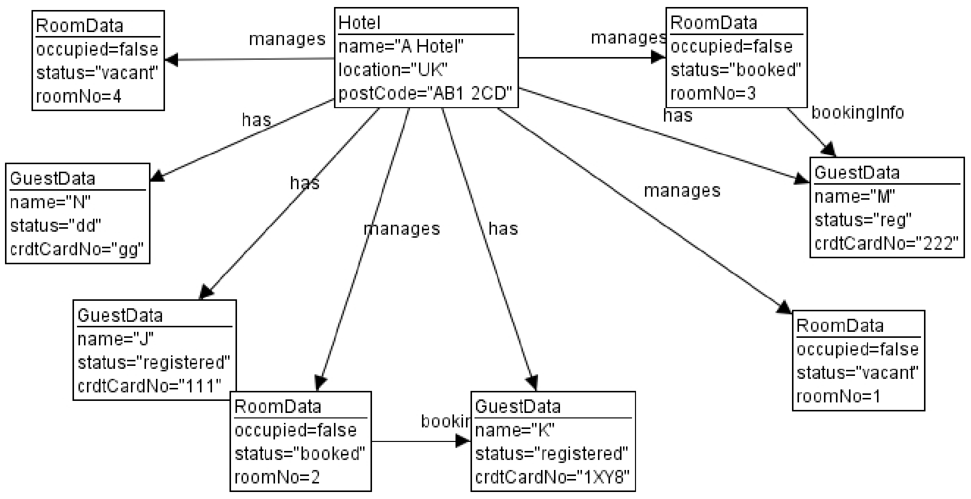


Figure 5.3: Start graph

Example *5.2.1 [coverage of test cases] Consider criterion cr + cd and the start state as shown in Figure 5.3, let T be the set of test runs shown in Table. 5.3.*

In the graph in Fig. 5.4, dotted red and the black lines of the show the edges not covered by the test cases in Table. 5.3, while the dotted blue indicate the edges in the dependency graph covered by T. This includes all the nodes, but we need to add test

Table 5.3: Test cases providing node coverage

test runs set-I
bookRoom(1,“J”);occupyRoom(1,“J”,3);updateBill(3,100)
occupyRoom(2,“K”,4);viewData(2);clearBill(4)
clearBill(3);updateBill(3,100)
updateBill(3,100);clearBill(3)
occupyRoom(3,“M”,5);checkout(3,“M”,5)
updateBill(3,100);updateBill(3,100);clearBill(3)

cases in order to get the required coverage cr + cd. In order to achieve this we add the

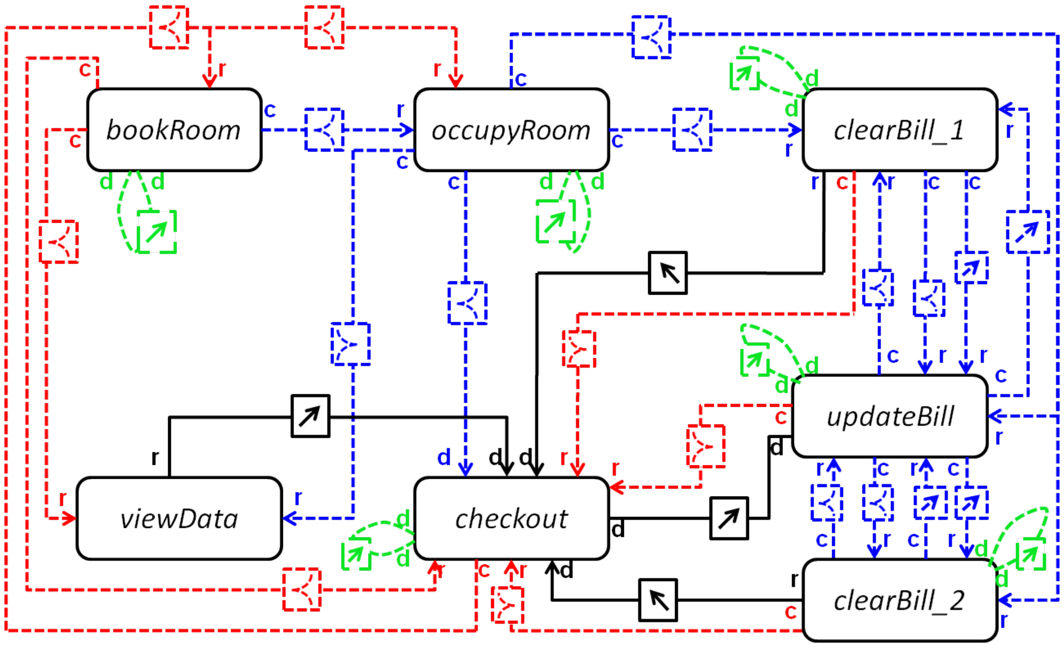


Figure 5.4: $Cov(DG,\ T)$

test cases in Table 5.4 and analyze the resulting coverage.

Table 5.4: Additional test cases to cover $cr + cd$

test runs set-II
bookRoom(3,“M”);occupyRoom(3,“M”,5);viewData(5)
bookRoom(4,“N”);occupyRoom(4,“N”,6);checkout(4,“N”,6)
checkout(1,“J”,3);bookRoom(1,“J”)
viewData(1);checkout(1,“J”,3)
updateBill(6,100);clearBill(6);checkout(4,“N”,6)

The first two test cases in Table 5.4 are required to cover the cr edges between bookRoom *and* viewData *and between* bookRoom *and* checkout *and the inclusion of* occupyRoom *is required. The third and forth test cases are required to cover the edges between* checkout *and* bookRoom *and between* viewData *and* checkout. *The last test case is required to exercise edges between* updateBill, clearBill *and* checkout. *Notice that the dotted green edges are not covered by any of the test cases as well as the edge between* checkout *and* occupyRoom *is not covered. We need stronger criteria to include test cases covering these edges, too.*

5.3 Summary

In this chapter, we have explored the use of graph transformation systems specifying service interfaces for the derivation of model-based coverage criteria. We have already discussed our approach to provide test oracles using the visual contract analysis. In the next chapter, we record if test runs exercised a statically existing dependency. We also record which of the steps within the test runs disables matches for the rules within the graph grammar to assess coverage of conflict relations.

Chapter 6

Dynamic Analysis of Dependencies and Conflicts

In this chapter, we discuss how we use dependency graphs providing static information about conflicts and dependencies and dynamic analysis of dependencies and conflicts to assess the coverage of a given set of test runs. We make use of AGG for simulating the model while tests are executed. In the course of the simulation, which also serves as a test oracle, conflicts and dependencies are observed and recorded. This allows us to see if the statically detected potential dependencies and conflicts are exercised at runtime. For evaluation purposes, we compare coverage with respect to model-based criteria and traditional structural ones.

The static information about dependencies and conflicts between rules does not guarantee that they are exercised during execution. We analyze the model state after each rule application and record if dependencies were exercised. We also examine if a step in the considered sequence has disabled a match for any other rule in the TAGTS. Our approach to coverage hence combines static and dynamic analysis of models.

$$\begin{tikzcd}
\dots & L_i \arrow[rr, bend left, "p_i"] \arrow[d, "m_i"] & K_i \arrow[l] \arrow[r] \arrow[d] & R_i \arrow[rr, bend left, "p_{j-1} \circ \dots \circ p_{i+1}"] \arrow[d, "m_i^*"] \arrow[drr, dotted, "m_i^{j-1}"] \arrow[drrrr, dotted, "m_i^j"] & \dots & L_j \arrow[rr, bend left, "p_j"] \arrow[d, "m_j"] & K_j \arrow[l] \arrow[r] \arrow[d] & R_j \arrow[d, "m_j^*"] & \dots \\
\dots & G_{i-1} \arrow[rr, bend right, "p_i^*"'] & D_i \arrow[l] \arrow[r] & G_i \arrow[rr, bend right, "p_{j-1}^* \circ \dots \circ p_{i+1}^*"'] & \dots & G_{j-1} \arrow[rr, bend right, "p_j^*"'] & D_j \arrow[l] \arrow[r] & G_j & \dots
\end{tikzcd}$$

Figure 6.1: Dependencies in a transformation sequence

6.1 Dynamic Analysis of Dependencies and Conflicts

We use AGG [2] as an oracle simulating the model while tests are executing. Monitoring the transformation sequences created, we can detect conflicts and dependencies at runtime and therefore measure the coverage of the dependency graph with respect to a given set of criteria. We consider dependencies and conflicts separately.

***Definition** 6.1.1 (coverage of dependencies) A dependency edge $p < q$ is covered by a transformation sequence $G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \overset{p_2,m_2}{\Longrightarrow} \cdots \overset{p_n,m_n}{\Longrightarrow} G_n$ if there are $i < j \leq n$ such that $p = p_i, q = p_j$ and the residual comatch m_i^{j-1} of p_i into G_{j-1} overlaps with the match m_j of p_j in accordance with the source and target types of the relation. That means, there exist a node or edge x or an attribute a in $m_i^{j-1}(L_i) \cap m_j(L_j) \subseteq G_{j-1}$ such that*

cr: *x is created by p_i and read by p_j*

cd: *x is created by p_i and deleted by p_j*

The residual comatch m_i^{j-1} of p_i into graph G_{j-1} is obtained by composing comatch m_i^* with the tracking morphism $p_{j-1}^* \circ p_{i+1}^*$ between G_i and G_{j-1} as illustrated in the diagram of Fig. 6.1.

This definition is implemented by Algorithm 1, whose input is a sequence of invocations s as generated by the test as well as the start graph of the grammar. The dynamic detection of conflicts is based on finding, for each graph in the sequence, all matches for all rules and comparing them via the tracking morphisms. If, for a given step $G \overset{p,m}{\Longrightarrow} H$, rule q has a match into graph G which is not present in H, this match is disabled by p. In this case, we have observed a conflict $q \nearrow p$.

Definition *6.1.2 (coverage of conflicts) A conflict edge $q \nearrow p$ is covered by a transformation sequence $G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \overset{p_2,m_2}{\Longrightarrow} \cdots \overset{p_n,m_n}{\Longrightarrow} G_n$ if there exists a step $G_{i-1} \overset{p_i,m_i}{\Longrightarrow} G_i$ such that $p = p_i$ and for any match m of q into G_{i-1} there is no match m' of q in G_i such that $p_i^* \circ m = m'$.*

The source and target labels of the edge are determined according to one of the following cases, for a node or edge x or an attribute a in $m_i(L_i) \cap m(L) \subseteq G_{j-1}$.

rd: *x is read by q and deleted by p_i*

dd: *x is deleted by both q and p_i*

The implementation is presented in Algorithm 2 with the same input as before and executing the same sequence of steps on invocation.

6.2 Implementation

We first discuss two algorithms that we use to mark the dependencies and conflicts at runtime. Algorithm 1 considers the input which is a sequence of invocations s as part of the test case as well as the start graph of the grammar. For each step in s we apply the corresponding rule schema or basic rule. AGG stores (co-)matches as mappings into a pool of graph elements. If an element is deleted it is removed with its details, leaving a hash value assigned upon creation of the element. We use these hash values to represent matches and comatches and to calculate their difference and intersection after each step to find out what was created, preserved, and deleted. We detect dependencies of type *cr* and *cd* by maintaining a list of partial comatches into all subsequent steps of the sequence, computing the overlaps between matches and partial comatches. The output of the algorithm is stored in the dynamic dependency matrix and compared to the static dependency matrix created by AGG when the model was first loaded. Note that the host graph can change without requiring to recalculate the stored static information, saving considerable execution time. The comparison of the two matrices provides the coverage data.

We make use of rule schemes if the signatures involve multi-objects as shown in Fig. 2.2. The concept of amalgamated transformations is already presented in [10, 27] where the implementation of this concept in AGG is explained in [63]. An amalgamated transformation returns the set of the nodes corresponding to the multi-object on the right-hand side of the rule. Rule schemes implement all-quantified operations on recurring graph patterns. The kernel rule is a common subrule of a set of multi-rules. It is matched only once, while multi-rules are matched as often as suitable matches are found. In AGG an amalgamated rule is constructed from all matches found for multi-rules that share the match of the kernel rule.

Consider the dependency graph in Fig. 5.2 and rules *bookRoom(r: int, n: String), occupyRoom(r: int, n: String, out int bNo), checout(r: int, n: String, out int bNo)* with the following test runs.

1. $bookRoom(1, \text{“}Tim\text{”}); occupyRoom(1, \text{“}Tim\text{”}, 100); checkout(1, \text{“}Tim\text{”}, 100)$
2. $bookRoom(1, \text{“}Tim\text{”}); occupyRoom(2, \text{“}Fim\text{”}, 100); checkout(3, \text{“}Sim\text{”}, 200)$

The first sequence exercises a direct *cr* dependency between steps *bookRoom(. . .)* and *occupyRoom(. . .)* since the first produces a mapping between the nodes *RoomData* and *Guest* while the second uses it and creates a *bill* object. Steps *bookRoom(. . .)* and

Algorithm 1 algorithm for marking dependencies

```
Input: size(s)>= 2
  set host graph to start graph of GraphGrammar
  for <i=0; i<size(s); i++> do
    if Rule $r_i$ instanceof RuleScheme then
      apply Rule Scheme
    else
      apply Rule $r_i$
    end if
    store hash values of involved graph elements in arrays matches and comatches
  end for
  for <i=0; i<size(s); i++> do
    createdElements[i] = difference(matches[i], comatches[i])
    deletedElements[i] = difference(comatches[i], matches[i])
    preservedElements[i] = intersection(matches[i], comatches[i])
  end for
  for <i=size(s); i>0; i- -> do
    for <j=0; j<size(s); j++> do
      if (intersection(createdElements[j], matches[i-1] <> $\phi$)) then
        mark dependency between Rule r[i] and r[j]
        repeat the intersection calculation for attributes lists
      end if
      if (intersection(preservedElements[j], matches[i-1] <> $\phi$)) then
        mark dependency between Rule r[i] and r[j]
        repeat the intersection calculation for attributes lists
      end if
    end for
  end for
```

checkout(...) have an indirect *cd* dependency since the mapping between *RoomData* and *Guest* is created by the first invocation and is deleted by the third. Observe that these dependencies rely on the matches of the steps as determined by the parameters of the operations. The second sequence does not exercise the dependency between *bookRoom* and *occupyRoom* because here the guest is assigned is allowed to check in and a *bill* object is created for a previously existing booking instead of one done by *bookRoom(1,"Tim")*. Similarly, steps *bookRoom(1, "Tim")* and *checkout(3, "Sim", 200)* are unrelated since the latter considers a different guest, room and a bill number not produced by any of the first two.

Algorithm 2 is used for marking conflicts at the runtime which takes the input of a sequence of invocations s as generated by the test as well as the start graph of the grammar as in the case of Algorithm 1. At each step we find and store all matches for all rules in the grammar, computing the difference between the sets of matches into graph i and graph $i + 1$ to find out those that were disabled by that step. Each disabled match represents an asymmetric conflict, which is recorded in the dynamic conflict matrix. Like for dependencies, this is compared to the result of the static analysis to calculate the coverage.

Algorithm 2 algorithm for marking conflicts

```
Input: size(s)>= 2
  set host graph to start graph of GraphGrammar
  for <i=0; i<size(s); i++> do
    if Rule $r_i$ instanceof RuleScheme then
      apply Rule scheme
    else
      apply Rule $r_i$
    end if
    store the hash value of graph elements in an array
    for all Rule r in GraphGrammar do
      find all possible matches and store in an array
    end for
  end for
  for (i=0; i<size(s)-1; i++) do
    select all the matches found for $i^{th}$ row
    select all the matches found for $(i + 1)^{th}$ row
    for (j=0; j<size(row);j++) do
      analyze matches details to mark conflict between rule[i] and rule[j]
    end for
  end for
```

Consider the following two test runs.

1. $occupyRoom(1, \text{“}Tim\text{”}, 100); clearBill(100); checkout(1, \text{“}Tim\text{”}, 100)$

2. $occupyRoom(1, \text{“}Tim\text{”}, 100); checkout(1, \text{“}Tim\text{”}, 100)$

An *rd* conflict between *clearBill(. . .)* and *checkout(. . .)* in the first sequence means that these operations are not executable in the reverse order. Step *occupyRoom(. . .)* has a *dd* conflict with itself, meaning that it can only occur once in that sequence. The second sequence results from choosing *checkout(. . .)* before, and instead of, *clearBill(. . .)* in the *rd* conflict.

We consider a test run containing three invocations for explaining our implementation, as given below. We have numbered the invocations within the test run for ease of referring to them in the forthcoming discussion and we assume that the matches for all of the invocations were found with a bill number 100 assigned to the output variable in $occupyRoom(\ldots)$.

1. bookRoom(1, “Tim”);

2. occupyRoom(1, “Tim”, 100);

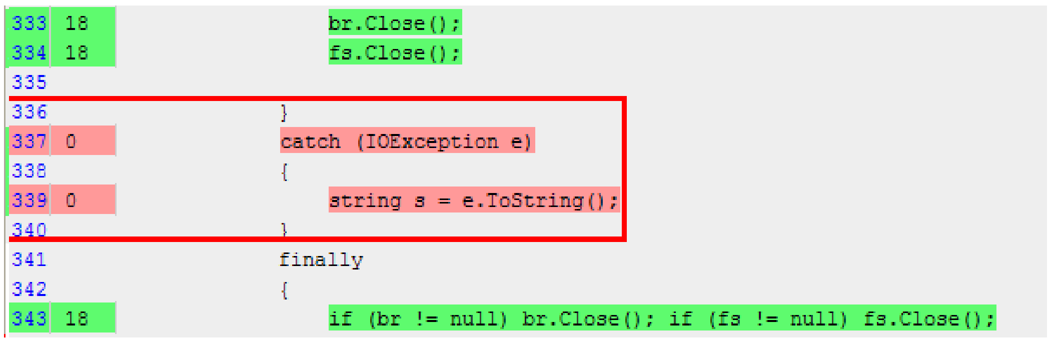
3. checkout(1, “Tim”, 100);

Considering the first and the second invocation in the above test run, we record the graph elements in the match and the co-match of the rule application. We record them in terms of the hash values they were assigned on creation. We keep the hash values of the nodes and of the attribute lists in two separate lists. The code for storing hash values for attribute lists involved in a (co-)match is shown in Listing 6.1.

Listing 6.1: Hash Values for Attributes Lists

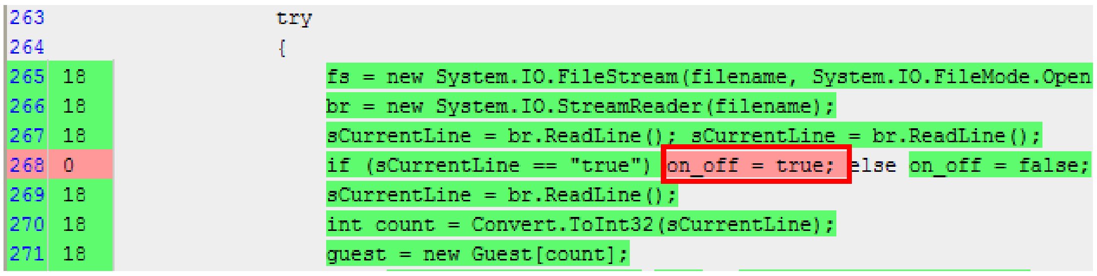
```
 ...
ArrayList<GraphObject> l = new ArrayList<GraphObject>(m.getCodomainObjects());
for (int i=0; i<l.size(); i++)
{
    if (l.get(i).isNode() == true)
    {
        List<String> result = new ArrayList<String>();
        int hCode = l.get(i).hashCode();
        result.add(Integer.toString(hCode));
        Node n = (Node) l.get(i);
        for (int j=0; j<n.getNumberOfAttributes(); j++)
            result.add(Integer.toString(n.getAttribute().hashCode()))
...
```

Once, we have stored hash values for matches and co-matches for all of the invocations in our test run, we check if a dependency was exercised following Algorithm 1. Our analysis of the test run reveals that there exists a create-read dependency between the first two and a create-delete dependency between the first and the third invocations. While processing the same test run for conflicts, we consider the start graph and the list of rules in the graph grammar and find out all possible matches. We require this information since each rule application can destroy matches. Consider the first invocation in our test run, *bookRoom(1, "Tim")* and visual contract for *bookRoom* in Figure 2.2, this creates an edge between nodes *RoomData* and *Guest* and therefore the application of *bookRoom* with same data, i.e., *(1, "Tim")* is not possible. We capture this information together with the reason and record which of the invocations have disabled any matches. we record the number of matches after each rule application, together with the details for each of the matches, to verify if a match is disabled and a new match is enabled reporting same number of matches before and after application. Our approach for dynamic conflict analysis is shown in Algorithm 2. Examining the considered test run, we find out that the first invocation, i.e, *bookRoom(1,"Tim")* renders match of its successive use, i.e., use of *bookRoom(1,"Tim")* again with the same parameters as disabled and same is the case for the second and the third invocation. Similarly, the third invocation, i.e., *checkout(1,“Tim",100)* renders *updateBill(100)* and *clearBill(100)* disabled since it is deleted by the third invocation in the considered sequence. Considering the considered sequence, the SUT should not allow the booking of the same room by the same guest again or creating bill 100 again and inclusion of a negative test run is indeed required to test the SUT.



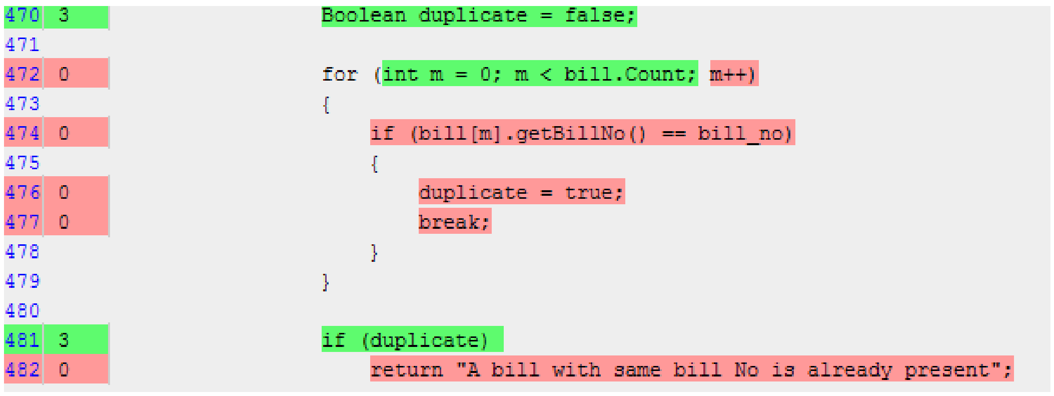
```
18              br.Close();
18              fs.Close();

            }
0           catch (IOException e)
            {
0               string s = e.ToString();
            }
            finally
            {
18              if (br != null) br.Close(); if (fs != null) fs.Close();
```

(a) Covered and uncovered code example



```
            try
            {
18              fs = new System.IO.FileStream(filename, System.IO.FileMode.Open
18              br = new System.IO.StreamReader(filename);
18              sCurrentLine = br.ReadLine(); sCurrentLine = br.ReadLine();
0               if (sCurrentLine == "true") on_off = true; else on_off = false;
18              sCurrentLine = br.ReadLine();
18              int count = Convert.ToInt32(sCurrentLine);
18              guest = new Guest[count];
```

(b) Dead code example



```
3       Boolean duplicate = false;

0       for (int m = 0; m < bill.Count; m++)
        {
0           if (bill[m].getBillNo() == bill_no)
            {
0               duplicate = true;
0               break;
            }
        }

3       if (duplicate)
0           return "A bill with same bill No is already present";
```

(c) Uncovered alternate behavior code example

Figure 6.2: (Un)covered code (a), Dead and alternate code(b) and (c)

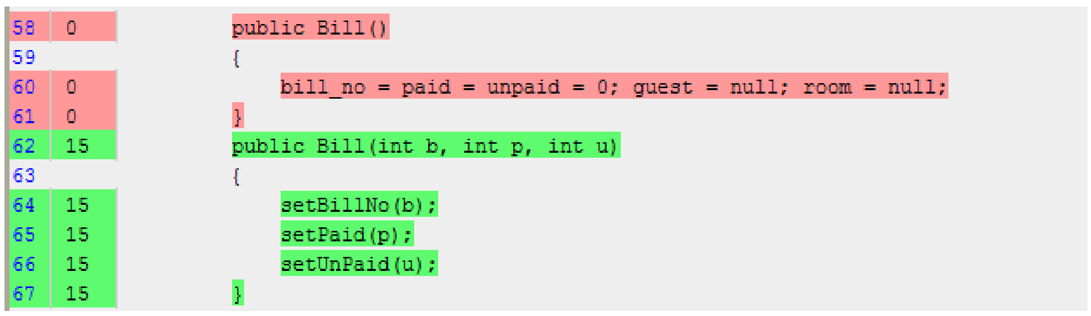
6.3 Application to the Running Example

In order to evaluate the effectiveness of our coverage criteria, we selected the test runs such that our coverage criteria was to cover all *cr* + *cd* edges through the dependency graph. In order to compare model level coverage with traditional code based coverage criteria, we implemented our running example as a web service. We calculated the latter using NCover 2.0 [1]. The resulting symbol coverage, branch coverage and method coverage is shown in Figure 6.4. We inspected the instrumented code of the application using NCover to see which of the statements were not covered. Our analysis revealed the following reasons for a lower coverage metric.

1. Exceptions handling code was not executed. This is shown in Figure 6.2(a) and in Figure 6.2(c) respectively.

2. Some dead code shown in Figure 6.2(b) was no more accessible in the final version of the service.

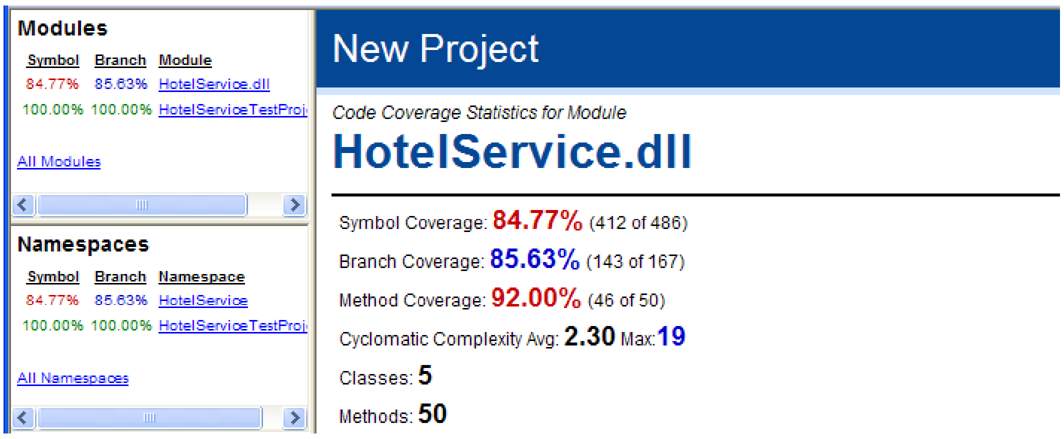
3. Additional glue code added by the IDE and default constructors in the classes. An example of constructors that are present in the code, but not executed during testing is shown in Figure 6.3.



```
0           public Bill()
            {
0               bill_no = paid = unpaid = 0; guest = null; room = null;
0           }
15          public Bill(int b, int p, int u)
            {
15              setBillNo(b);
15              setPaid(p);
15              setUnPaid(u);
15          }
```

Figure 6.3: Method Coverage

[1]`http://www.ncover.com/download/current`



Modules

Symbol	Branch	Module
84.77%	85.63%	HotelService.dll
100.00%	100.00%	HotelServiceTestProj

All Modules

Namespaces

Symbol	Branch	Namespace
84.77%	85.63%	HotelService
100.00%	100.00%	HotelServiceTestProj

All Namespaces

New Project

Code Coverage Statistics for Module

HotelService.dll

Symbol Coverage: 84.77% (412 of 486)

Branch Coverage: 85.63% (143 of 167)

Method Coverage: 92.00% (46 of 50)

Cyclomatic Complexity Avg: 2.30 Max: 19

Classes: 5

Methods: 50

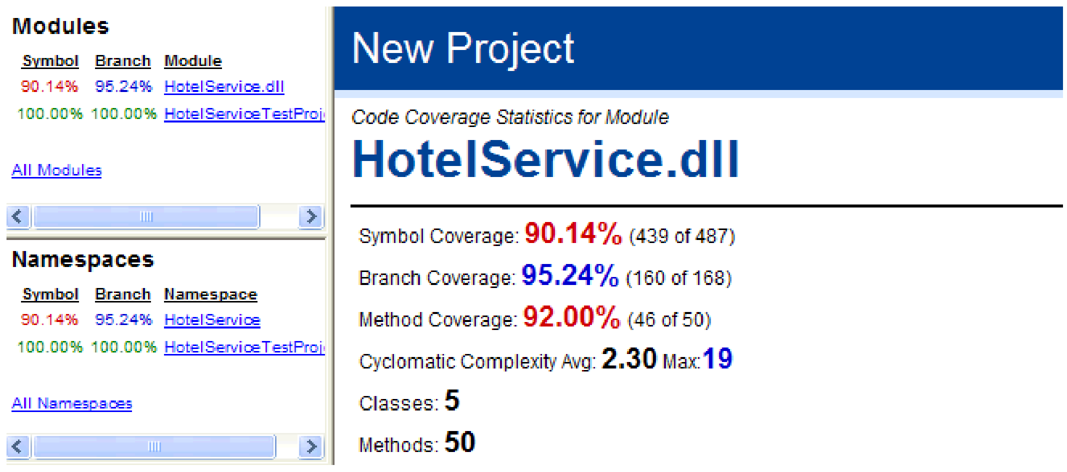
Figure 6.4: Code Coverage with $cr + cd$

We repeated the same experiment with criteria $rd + dd$ paths. We consider the same hotel example and the test runs we have already used in Chapter 5. We additionally consider the test runs resulting from our conflicts analysis. However, it is important to note that we are still not able to attain a complete statement or branch coverage. Our analysis of the instrumented code reveals that there are the following reasons for a lower than complete statement or branch coverage.

1. The exception related code, as shown in Figure 6.2(a) was also not exercised by any of the test run. Exception handling code was triggered by technical errors outside the specification, e.g., a failure to connect to a flat file on the disk drive.

2. The unreachable or dead code, as shown in Figure 6.2(b) (Chapter 5) was still not exercised.

There is however an improvement in the symbol and the branch coverage but the method coverage remains the same which reflects that only the test runs were added with negative expectation.


Modules

Symbol	Branch	Module
90.14%	95.24%	HotelService.dll
100.00%	100.00%	HotelServiceTestProj

All Modules

Namespaces

Symbol	Branch	Namespace
90.14%	95.24%	HotelService
100.00%	100.00%	HotelServiceTestProj

All Namespaces

New Project

Code Coverage Statistics for Module

HotelService.dll

Symbol Coverage: **90.14%** (439 of 487)

Branch Coverage: **95.24%** (160 of 168)

Method Coverage: **92.00%** (46 of 50)

Cyclomatic Complexity Avg: **2.30** Max: **19**

Classes: **5**

Methods: **50**

Figure 6.5: Code Coverage with all paths

6.4 Limitations of the Approach

We have proposed an approach to model-based coverage based on a two-step process combing static and dynamic analysis. Statically, we use AGG's critical pair and minimal dependency analysis to create a dependency graph over rules representing visual contracts. These graphs, which distinguish different types of dependencies and conflicts, are the basis for coverage criteria. The evaluation of a set of tests based on the criteria is performed dynamically while executing the model as an oracle.

The approach requires further evaluation in particular in terms of scalability to longer sequences. It is clear that improvements are possible by reducing the number of matches kept and compared, using information from the static analysis which provides a conservative approximation of the real dependencies and conflicts.

The major cost factor is the creation and maintenance of the models. If and when these costs are outweighed by the benefits can only be evaluated in a more realistic industrial setting. However, scenarios where specifications are created once and used repeatedly, e.g., as part of standards, are likely to provide good tradeoffs. In particular, model-driven development provides a framework where models are core artefact. In this case the use of models for testing becomes an issue of reusing rather than creating models.

We need to consider the negative application conditions which would strengthen the visual contracts specification. This will create additional types of dependency, such as create-forbid, where one rule creates part of the structure preventing the application of another, resulting in new coverage criteria.

6.5 Summary

We have proposed, in this chapter, an approach to model-based coverage based on a two-step process combing static and dynamic analysis. Statically, we use AGG's critical pair and minimal dependency analysis to create a dependency graph over rules representing visual contracts. These graphs, which distinguish different types of dependencies and conflicts, are the basis for coverage criteria. The evaluation of a set of tests based on the criteria is performed dynamically while executing the model as an oracle.

In the next chapter, we work on the proposal for a regression testing approach considering the model-level information and focussing on web services.

Chapter 7

Model Based Regression Testing

This chapter introduces our approach to model-based regression testing of services. We discuss two evolution scenarios to our running example to explain our analysis of impact of evolution. Since regression testing is the process of verifying the quality of the evolved system, there is a preexisting test suite.

Following the classification in [48], a test case in a regression test suite can be *obsolete* (*OB*) if it is no longer applicable to the new version, *reusable* (*RU*) if it is still applicable and *required* (*RQ*) if it tests functionality affected by the changes. Assuming that the version before evolution is SUT and the version after evolution is SUT', our assumption is that we have the model, the analysis information and the set of test cases for SUT. The analysis of visual contracts for SUT' allows us to differentiate the operations that have been added or deleted.

Our approach considers an operation with modified signature as obsolete and the same operation with revised signature as newly added. We select those operations where we do not see any evolution in the pre- or the post-condition as not affected by the evolution at the model level. We collect information about the implementation related changes to operations which do not impact the model-level information as well. Based on our model-level information about the evolution, we categorize test cases as obsolete, reusable and required from the regression test suite, RTS, which contains all the test cases for the new version. The method is applicable to all software systems that have interfaces specified in this way, but is particularly relevant for services because of the lack of access to implementation code and the potential cost involved in running a large number of tests through a remote and potentially payable provider.

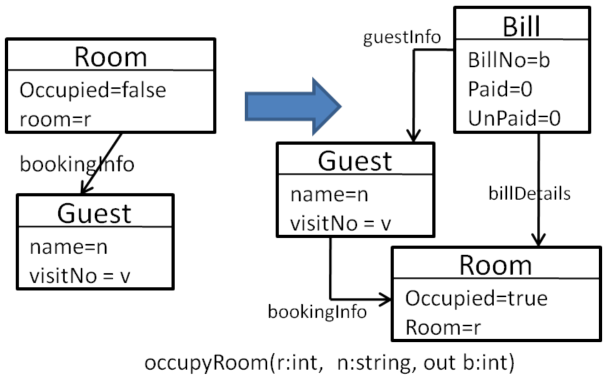
7.1 Evolution Scenarios

In order to explain our treatment of regression testing, we present two evolution scenarios.

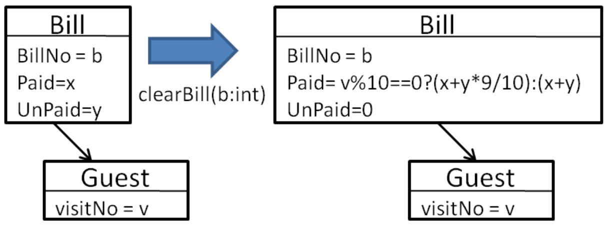
We consider our running example as the base version $V1$ shown in Figure 2.2. This allows us to do room bookings, check-in and checkout. The model also provides rules for updating and clearing bills as well as a rule to query the stored data. The type graph is shown in Figure 2.4 and a selection of rules are shown in Figure 2.2 and Figure 2.4.

7.1.1 Scenario I

As an evaluation scenario, we start to record the number of visits of each customer to the hotel, to introduce a promotional 10% discount on all payments of every 10^{th} visit. This is achieved by an update to rule *occupyRoom* to count visits, and a change to *clearBill* to distinguish the cases where the bill is paid with or without discount. Note the use of a conditional expression for calculating the value of the *paid* attribute. The evolved part of the type graph and the concerned rule are highlighted and shown in Figure 7.1. We refer to this evolution scenario as version $V2$ throughout the rest of this chapter.



(a) Rule occupyRoom After Evolution



(b) Rule clearBill

Figure 7.1: Rules occupyRoom and clearBill after evolution

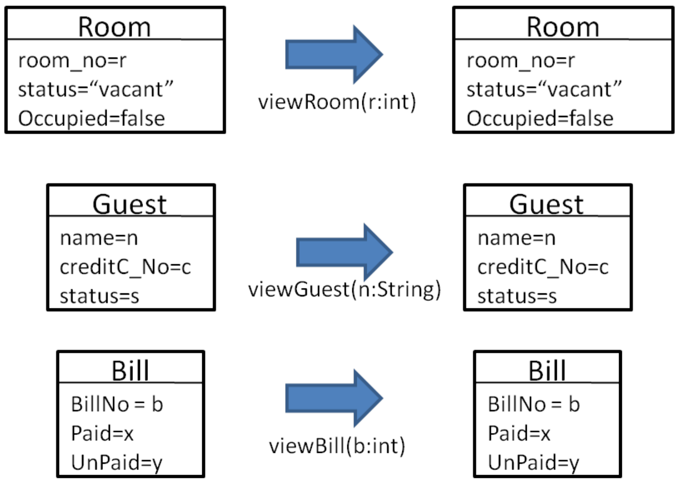
7.1.2 Evolution Scenarios II

For the second evolution scenario, there is a requirement for querying information about the bills and guests, just as operation *viewData* already provides information about rooms. The additional rules are shown in Fig. 7.2. We also allow the manager to give an additional discretionary discount of up to 5% to the guests if he thinks it would have a positive impact on the overall sales.

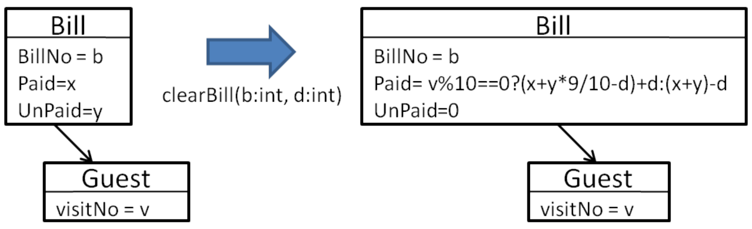
We refer to this evolution scenario as version $V3$ throughout the rest of this chapter.

7.2 Classification of Test Cases

Given a regression test suite RTS for one version SUT of the system under test, we are going to provide a classification of test cases with respect to an evolution of SUT into SUT' that will distinguish



(a) Additional Rules



(b) Revised Rule clearBill

Figure 7.2: Additional Rules and clearBill after evolution

- *obsolete* test cases OB, that are no longer executable in SUT', either because signatures have changed or part of the functionality has seized to exist;
- *reusable* test cases $RU = RTS \setminus OB$, that are still executable in SUT';
- *required* test cases $RQ \subset RU$, that are still executable and test new or modified functionality in SUT'.

Test runs may become *obsolete* because of changes in the operation signatures. Assuming P is the set of signatures for SUT, $P \setminus P'$ are signatures that are valid for SUT, but invalid in SUT', e.g., due to missing or incorrectly typed parameters where P' represents the set of signatures of the new version. All test runs containing obsolete signatures are obsolete.

In the evolution step $V1 \rightarrow V2$, all operations are preserved with their signatures. Hence all tests are preserved and therefore $OB = \emptyset$. For $V2 \rightarrow V3$, signatures have evolved and some new rules have been added. In particular, operation *clearBill(bill_no)* is obsolete, so all test cases containing its invocations are obsolete as well. Instead there are new invocations based on the extended signature *clearBill(bill_no, discount)*.

Test runs in RQ, which exercise operations that have changed are classified as *required*. Denote by $M \subseteq P'$ the set of operations such that either their specification or implementation has changed. The set of required test cases is therefore given by the set of all reusable ones RU which contain at least one affected rule invocation.

In evolution $V1 \rightarrow V2$, RQ is set of all test runs involving *occupyRoom* and *clearBill* since there are modifications in the visual contracts of these. Considering $V2 \rightarrow V3$, we find that *clearBill* has been modified and therefore any test runs involving its invocations are required.

7.3 Coverage Analysis

We have presented model-based coverage criteria in Chapter 5 and a mechanism to assess the coverage achieved by a test suite in Chapter 6. Considering two versions $V1$ and $V2$ of a system as presented in Section 7.1, we already know the coverage provided by the existing test suite for $V1$. For version $V2$, coverage is affected due to test cases becoming obsolete. However, there are test cases that are executable for both the versions, i.e., $V1$ and $V2$. Therefore, coverage provided by these test cases for version $V1$ could be considered for $V2$ as well. This information is helpful in deciding where to add test cases in order to retain the required coverage.

Example *7.3.1 Consider evolution scenario $V2 \rightarrow V3$ together with a set of test cases as given in Table 7.1. From $V2$ to $V3$, we notice that the operation $clearBill(bNo)$ has evolved to $clearBill(bNo, desc)$ and some new operations are added.*

Table 7.1: Example test runs for $V2$

S No	test runs
1	bookRoom(1,“J”);occupyRoom(1,“J”,3);viewData(1)
2	checkout(1,“J”,3);bookRoom(1,“J”)
3	occupyRoom(1,“J”,3); updateBill(3,100)
4	clearBill(3);updateBill(3,100)
5	occupyRoom(2,“K”,4);clearBill(4);checkout(2,“K”,4)
6	bookRoom(2,“K”);occupyRoom(2,“K”,4);viewData(2)
7	bookRoom(3,“M”);occupyRoom(3,“M”,5);checkout(3,“M”,5)
8	updateBill(3,100);clearBill(3);checkout(1,“J",3)

We notice that test runs $\{1, 2, 3, 6, 7\}$ *are reusable for version* $V3$ *whereas test runs* 4,5 *and* 8 *have gone obsolete. However, we cannot retain the coverage by required set of test runs since we need to rerun them again and we do not know if their execution would be successful. Therefore, we can consider the coverage provided by* $RU \setminus RQ$ *for* $V3$*. Inspecting Table 7.1, we find out that the test runs going obsolete due to* clearBill *have also rendered* updateBill(...) *not covered. We, therefore, need to add test cases, so that operation updateBill(...) is also covered.*

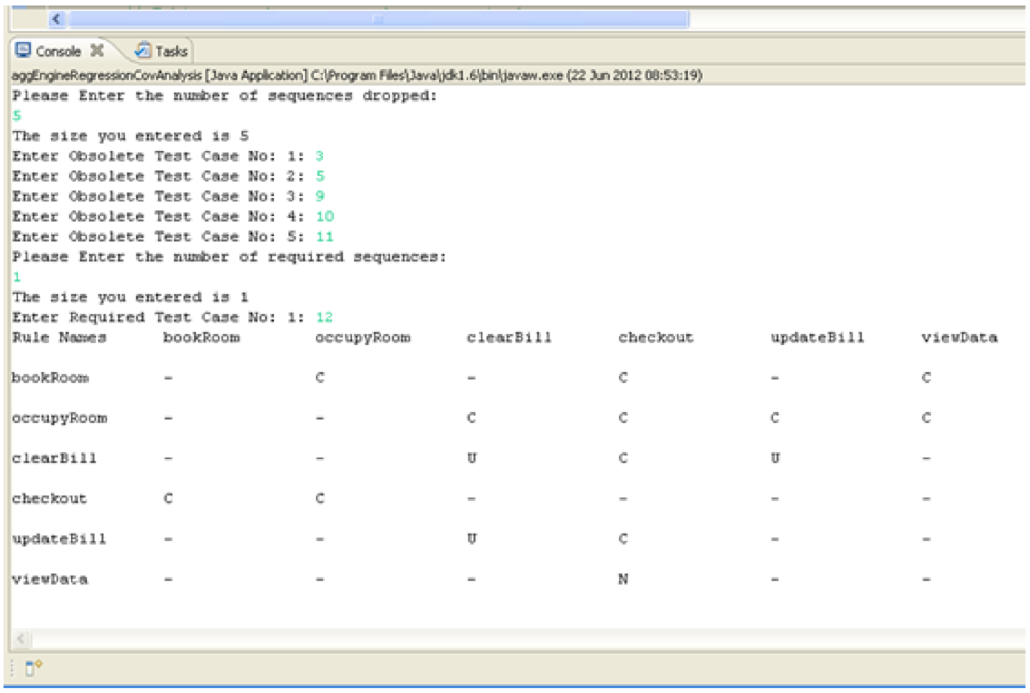
Algorithm 3 algorithm for marking conflicts

```
Input: Coverage report for each test case
Input: sequence numbers of $OB$ and $RQ$
  for (i=0; i<size($OB + RQ$); i++) do
    for (j=0; j<size(total test cases); j++) do
      if coverage due to $OB$ OR coverage due to $RQ$ then
        mark potentially uncovered
      end if
    end for
  end for
  for (i=0; i<size(total test cases); i++) do
    for (j=0; j<size($OB + RQ$); j++) do
      if $RU$ cover edges marked as potentially uncovered then
        mark as covered
      end if
    end for
  end for
```

If we record and analyze coverage with respect to $RU \setminus RQ$, we can execute the required set of test cases and update the coverage accordingly. We can reuse coverage information of reusable test cases if we maintain information about the coverage by each test case separately. We record which of the test cases have gone obsolete and which ones are required for retesting.

Algorithm 3 presents how we implement our coverage analysis for regression test suites. The algorithm considers coverage information for each test case and the set of obsolete as well as required test cases as the input. We first mark which of the edges in the dependency graph have become potentially uncovered by dropping obsolete and required test cases. We take one more iteration where we consider all the reusable test cases removing markings if they exercise the marked edges.

Example *7.3.2 Considering versions V*2 *and V*3 *and test cases given in Table 7.1; our coverage analysis is shown in Figure 7.3.*



```
aggEngineRegressionCovAnalysis [Java Application] C:\Program Files\Java\jdk1.6\bin\javaw.exe (22 Jun 2012 08:53:19)
Please Enter the number of sequences dropped:
5
The size you entered is 5
Enter Obsolete Test Case No: 1: 3
Enter Obsolete Test Case No: 2: 5
Enter Obsolete Test Case No: 3: 9
Enter Obsolete Test Case No: 4: 10
Enter Obsolete Test Case No: 5: 11
Please Enter the number of required sequences:
1
The size you entered is 1
Enter Required Test Case No: 1: 12
Rule Names     bookRoom       occupyRoom     clearBill      checkout       updateBill     viewData

bookRoom       -              C              -              C              -              C

occupyRoom     -              -              C              C              C              C

clearBill      -              -              U              C              U              -

checkout       C              C              -              -              -              -

updateBill     -              -              U              C              -              -

viewData       -              -              -              N              -              -
```

Figure 7.3: Fault seeding with L-Care

Here, the symbol C *reflects that the edge was previously covered and is still covered* *by the reusable set of test cases,* N *means that it was previously not covered and* U *means that the edge was previously covered but has become uncovered due to test cases going obsolete.*

7.4 Application to the Running Example

For each evolution, the evaluation is performed in five steps that are outlined below and explained in more detail throughout the section.

1. Generation of test cases.
2. Validation of the quality of the entire test suite.
3. Classification of test cases into OB, RU, and RQ.
4. Validation of the quality and required size of RQ by comparing the results of executing RQ and RU.
5. Analyzing resulting coverage.

7.4.1 Generation of Test Suite

We generate test cases manually, based on the information in the model, but without applying a formal notion of coverage. The completeness of the test set is evaluated instead trough *fault seeding*, i.e., deliberate introduction of faults to be detected by the execution of test cases. We generate 11 test cases for version $V1$, 11 test cases for version $V2$ and 14 test cases for version $V3$. The test runs for $V1$ are shown in Table 7.2 from which we select RQ for rerunning on $V2$ and the start state of the model is shown in Figure 7.4.

Table 7.2: Example test runs for $V1$

S No	test runs for V1
1	bookRoom(1,"J");occupyRoom(1,"J",3);updateBill(3,100)
2	occupyRoom(2,"K",4);viewData(2);clearBill(4)
3	clearBill(3);updateBill(3,100)
4	updateBill(3,100);clearBill(3)
5	occupyRoom(3,"M",5);checkout(3,"M",5)
6	updateBill(3,100);updateBill(3,100);clearBill(3)
7	bookRoom(3,"M");occupyRoom(3,"M",5);viewData(5)
8	bookRoom(4,"N");occupyRoom(4,"N",6);checkout(4,"N",6)
9	checkout(1,"J",3);bookRoom(1,"J");occupyRoom(1,"J",3)
10	viewData(1);checkout(1,"J",3)
11	updateBill(4,100);clearBill(4);checkout(2"K",4)

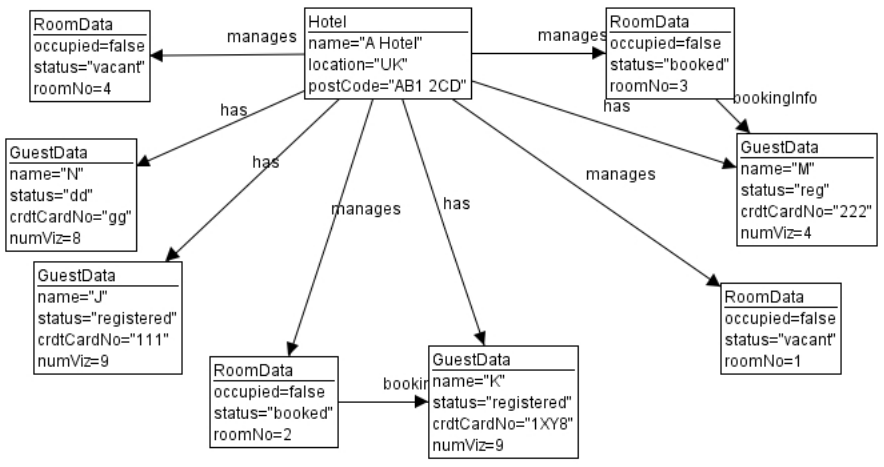


Figure 7.4: Start graph for version V2

7.4.2 Validation of Quality of Entire Test Suite

The percentage of the seeded faults detected provides a statistical measure of the capability of the test set to find similar errors in the system, i.e., a measure of confidence in our test suite [59]. In order to decide which faults to introduce we identified suitable fault types, and then developed rules for seeding them automatically. After applying the rules to the code of the system, we execute the entire test suite to assess its quality. In an iterative process we add test cases until all of the seeded errors were detected.

Faults are classified by [37], into *domain* and *computation* faults. A domain fault results from control flow errors, where programs follow the wrong path, while a computation fault occurs when the programme delivers incorrect results while following a correct path (usually due to errors in assignments or invocations). More specifically, we have followed the fault types discussed in [56], which also supports calculating a measure of confidence in a test suite as shown in Table 7.3.

Table 7.3: Distribution of seeded faults

Code Examples	
Correct Statement	Mutant Statement
(int i = 0; i < x; i++)	**(int i = 0; i < x; i+=2)**
return true	//**return** false
filename = path + "tak12";	filename = “C:" + “tak12";
if (this.bill[i].getBillNo() == billNo)	if (this.bill[i].getBillNo() != billNo)
file_read()	//file_read()

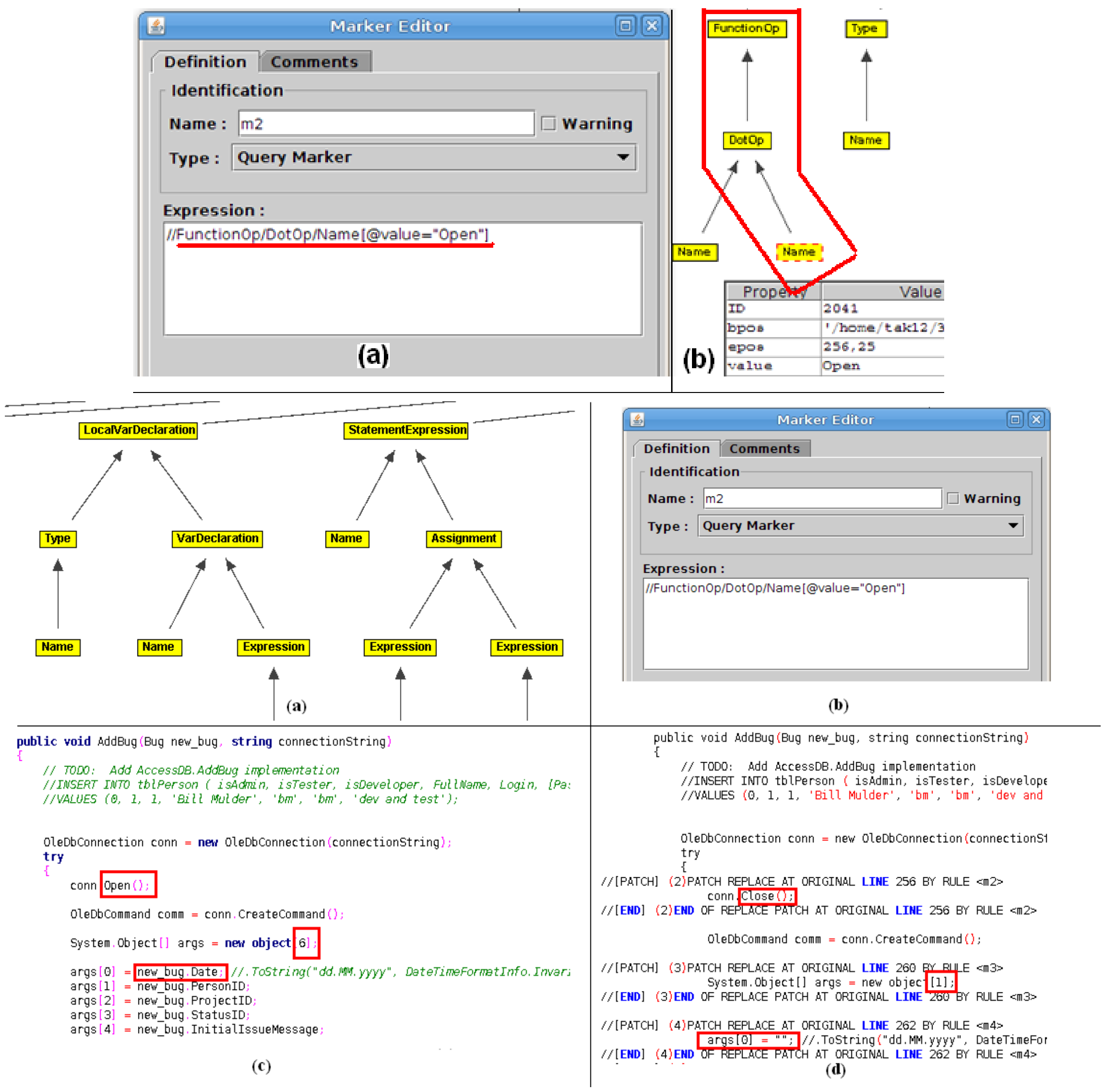
Our choice of fault based techniques for evaluation was motivated by our requirement to select a subset of test cases as *RQ* and see if they can uncover all of the seeded faults to ensure that they execute all parts of the code identified as impacted by the change.

We develop rules for seeding them automatically. After applying the rules to the code of the system, we execute the entire test suite to assess its quality. In an iterative process we add test cases until all of the seeded errors were detected. Rules for seeding faults according to these types are implemented in the source code transformation tool L-Care[1], which allows to define markers based on *XPath* queries as shown in Fig. 8.14(a) on an XML representation of the code.

A sketch of this XML in tree form is shown in Fig. 7.5(b). Examples of the original and the fault-seeded code are shown in Fig. 7.5 (c) and (d) respectively. Table 7.4 shows the total number of faults seeded for each version where a breakdown into the different types along with typical representatives is shown in Table 7.3.

We added test cases such that they were enough to uncover all the seeded faults. used fault seeding technique introduced by Mills (1972) as explained in [59]. Fault

[1]A product of `http://www.atxtechnologies.co.uk/`



Name : m2
Type : Query Marker
Expression : //FunctionOp/DotOp/Name[@value="Open"]

Property	Value
ID	2041
bpos	'/home/tak12/3
epos	256,25
value	Open

```
public void AddBug(Bug new_bug, string connectionString)
{
    // TODO:  Add AccessDB.AddBug implementation
    //INSERT INTO tblPerson ( isAdmin, isTester, isDeveloper, FullName, Login, [Pa:
    //VALUES (0, 1, 1, 'Bill Mulder', 'bm', 'bm', 'dev and test');


    OleDbConnection conn = new OleDbConnection(connectionString);
    try
    {
        conn.Open();

        OleDbCommand comm = conn.CreateCommand();

        System.Object[] args = new object[6];

        args[0] = new_bug.Date; //.ToString("dd.MM.yyyy", DateTimeFormatInfo.Invari
        args[1] = new_bug.PersonID;
        args[2] = new_bug.ProjectID;
        args[3] = new_bug.StatusID;
        args[4] = new_bug.InitialIssueMessage;
```

```
        public void AddBug(Bug new_bug, string connectionString)
        {
            // TODO:  Add AccessDB.AddBug implementation
            //INSERT INTO tblPerson ( isAdmin, isTester, isDevelope
            //VALUES (0, 1, 1, 'Bill Mulder', 'bm', 'bm', 'dev and


            OleDbConnection conn = new OleDbConnection(connectionSt
            try
            {
//[PATCH] (2)PATCH REPLACE AT ORIGINAL LINE 256 BY RULE <m2>
                conn.Close();
//[END] (2)END OF REPLACE PATCH AT ORIGINAL LINE 256 BY RULE <m2>

                OleDbCommand comm = conn.CreateCommand();

//[PATCH] (3)PATCH REPLACE AT ORIGINAL LINE 260 BY RULE <m3>
                System.Object[] args = new object[1];
//[END] (3)END OF REPLACE PATCH AT ORIGINAL LINE 260 BY RULE <m3>

//[PATCH] (4)PATCH REPLACE AT ORIGINAL LINE 262 BY RULE <m4>
                args[0] = ""; //.ToString("dd.MM.yyyy", DateTimeFor
//[END] (4)END OF REPLACE PATCH AT ORIGINAL LINE 262 BY RULE <m4>
```

Figure 7.5: Fault seeding with L-Care

seeding gives us the confidence measure $C = 1$ if $n > N$ and $S/(S - N + 1)$ if $n \leq N$ where S is the number of seeded faults, N is the total number of non-seeded (indigenous) faults. The estimated total number of faults can be found by $N = Sn/s$ where n is actual number of non-seeded faults and s is the number of seeded faults detected during testing.

We tested all the three versions, extending our test suites until all the seeded faults were detected. We seed faults in the modified classes of $V2$ and $V3$ only and execute the

Table 7.4: Distribution of seeded faults

Fault Type	# of	Seeded	Faults
	V1	V1	V2
Wrong declaration	26	26	29
Wrong assignment	2	2	2
Wrong proc. handling	8	8	11
Control faults	0	0	2
I/O faults	6	6	9
Total	42	42	53

two sets of required test cases *RQ* to determine if all of the seeded faults are discovered and how many test cases are actually required to discover them. We have seeded 13 and 24 faults in $V2$ and $V3$, respectively, the smaller numbers owing to the size of the changed classes in comparison to the entire code base. The results are reported in Table. 7.5.

7.4.3 Classification of test cases into OB, RU, and RQ

We classified test runs for $V1$ into OB, RU, and RQ and seeded and repeated the same for $V2$. We record the number of test cases in each category *produced* by our classification as well as the number of test cases actually *successful* in finding faults. Of step $V1 \rightarrow V2$ we recall that $OB = \varnothing$ because none of the existing operations were modified but number of tests in RQ are 10 since there was an internal modification in the rule *occupyRoom* and *clearBill*.

Table 7.5: Test case classification and success rate

	$V1 \rightarrow V2$		$V2 \rightarrow V3$	
Test cases	produced	successful	produced	successful
Obsolete (OB)	0	0	5	0
Reusable (RU)	11	0	6	0
Required (RQ)	10	4	0	0
New (NT)	–	0	–	8

With the second evolution step, 5 out of 11 existing test cases were classified as obsolete OB due to evolution in *clearBill*. The new test cases required to replace OB

were able to find all the seeded faults and none of the remaining test cases in RU found any fault, but 3 new test cases in NT had to be produced to detect faults seeded into newly added operations. That means, our reduction in the size of test suites has not resulted in missing any faults.

7.4.4 Validation of the quality and required size of RQ

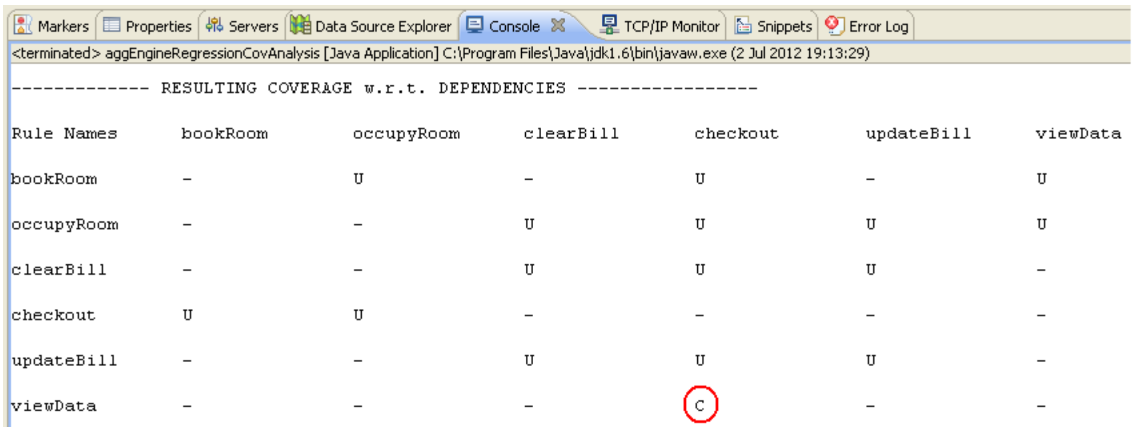
After applying to the resulting test set the classification described in Section 7.2, we validate the completeness of RQ against RU by seeding errors into the classes of our service implementation that were modified in the recent evolution step. We then run the tests in both RQ and RU, comparing their results. The evaluation is based on implementations in C# of the three versions of the running example. The programming environment Pex[2] has been used for automated unit testing of individual classes. Pex is able to generate test cases based on analyzing the source code, with the aim of detecting faults that could lead to runtime errors such as inappropriate exception handling. In our report below we do not include these tests because unit testing is part of the coding at the provider's site while we are concerned with service-level acceptance testing by the client. Therefore, test cases we have generated are concerned with deviations from the public specification of the service interface. The actual testing revealed that a smaller number of tests from RQ i.e., 4 would have been sufficient. For version $V2$ there was no test case selected as RQ and this was confirmed by our actual executions as well.

7.4.5 Coverage Analysis

We analyze the coverage by $RU \setminus RQ$ tests and reapply RQ to find out if the coverage is retained. For the coverage analysis, we first consider versions $V1$ and $V2$. The test

[2] `http://research.microsoft.com/en-us/projects/pex/`

runs for $V1$ are given in Table 7.2. Our test case classification reports 10 out of 11 tests as required. The only reusable test run is given in the 10th row of Table 7.2 and, therefore, the coverage analysis reveals that only the edge between operations *viewData* and *checkout* is still covered as shown in Figure 7.6.



```
<terminated> aggEngineRegressionCovAnalysis [Java Application] C:\Program Files\Java\jdk1.6\bin\javaw.exe (2 Jul 2012 19:13:29)
------------- RESULTING COVERAGE w.r.t. DEPENDENCIES -----------------

Rule Names      bookRoom        occupyRoom      clearBill       checkout        updateBill      viewData

bookRoom        -               U               -               U               -               U

occupyRoom      -               -               U               U               U               U

clearBill       -               -               U               U               U               -

checkout        U               U               -               -               -               -

updateBill      -               -               U               U               U               -

viewData        -               -               -               C               -               -
```

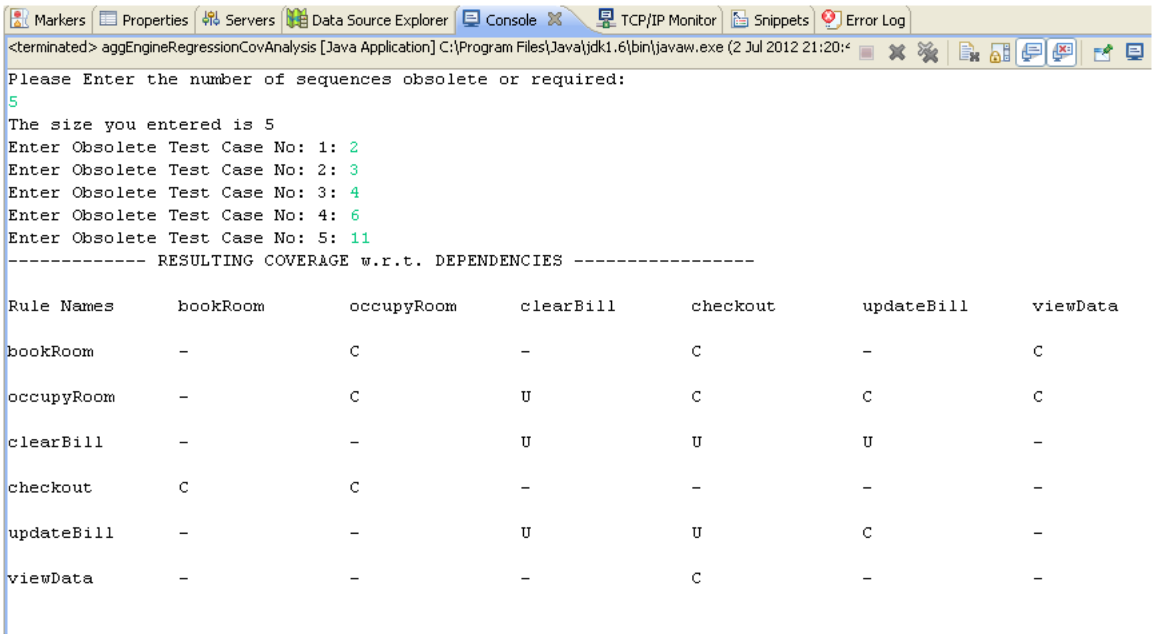
Figure 7.6: Coverage analysis output for version V1

Considering evolution from $V2$ to $V3$, we find that the operation signature for *clearBill* has gone obsolete and three new operations are added. We found that we required 8 new test cases where 5 went obsolete with $RQ = \emptyset$. Our coverage analysis shows that the following edges are covered after this evolution. The coverage analysis revealed that the edges marked as U require test cases to retain the coverage as shown in Figure 7.7

7.5 Limitations of the Approach

There are a few limitations of the approach. The approach classifies more than the optimally required number of test cases as required RQ. The reason for this is that we consider all potential situations where the impact of change in one part could have affected the quality.

If we can further pinpoint the impact of change, we can provide a more fine tuned



```
<terminated> aggEngineRegressionCovAnalysis [Java Application] C:\Program Files\Java\jdk1.6\bin\javaw.exe (2 Jul 2012 21:20:
Please Enter the number of sequences obsolete or required:
5
The size you entered is 5
Enter Obsolete Test Case No: 1: 2
Enter Obsolete Test Case No: 2: 3
Enter Obsolete Test Case No: 3: 4
Enter Obsolete Test Case No: 4: 6
Enter Obsolete Test Case No: 5: 11
------------- RESULTING COVERAGE w.r.t. DEPENDENCIES -----------------

Rule Names      bookRoom        occupyRoom      clearBill       checkout        updateBill      viewData

bookRoom        -               C               -               C               -               C

occupyRoom      -               C               U               C               C               C

clearBill       -               -               U               U               U               -

checkout        C               C               -               -               -               -

updateBill      -               -               U               U               C               -

viewData        -               -               -               C               -               -
```

Figure 7.7: Coverage analysis output for version V2

version of the selection criteria such that a lower number of test cases are selected as required for rerunning. Another possibility is to classify different changes and select the test cases accordingly.

New test cases will be required to validate the newly added operations, but the generation of test cases is out of the scope of this thesis.

7.6 Summary

In this chapter, we have presented a method to reduce the size of a regression test suite based on an analysis of the dependencies and conflicts between visual contracts specifying the preconditions and effects of operations. We have also evaluated the approach through the development of a case study, showing that (1) the reduced test sets could find all the faults detected by the larger sets while (2) being significantly smaller, which we present in the following chapter.

Chapter 8

Case Study: Bug Tracking System

In this chapter, a case study is presented to evaluate the developed approaches and to perform a critical analysis of the results. This chapter also presents the limitations as well as threats to the validity of the evaluation. We first discuss the experimental setup before we turn to the results and evaluation.

8.1 Bug Tracking System

This section is devoted to introducing the implementation and model of the case study before we use it for evaluation of our approaches.

8.1.1 Service Implementation

We have derived a web service from an open source desktop application[1], originally called BTsys, and replaced its GUI by a service interface. Such a service could be useful, for example, in order to allow automatic bug reports through applications detecting faults or in order to integrate bug tracking data into higher level functions. The service is implemented in C# and it provides operations to manage projects and users, report

[1]available at `http://btsys.sourceforge.net/`

faults and issues. Development teams can access fault reports and update their status.

The signatures for the operations are shown in Listing 8.1. The complete interface contains more than 30 operations. Throughout the rest of the chapter, we speak of the web service as the *implementation* and refer to the visual contracts as the *model*.

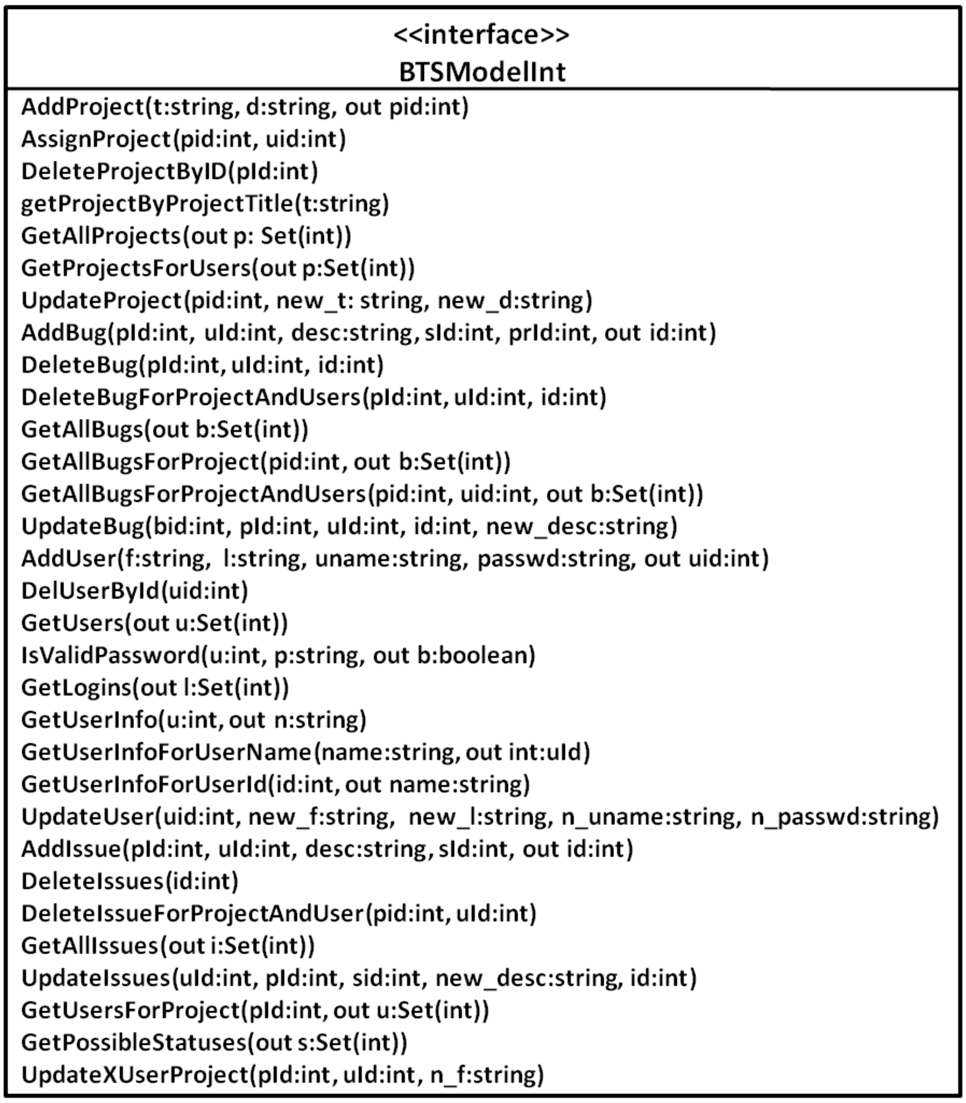
Listing 8.1: BTSImplementationInt

```
    ...
public String AddProject(String title, String description) { ... }
public string AssignProject(Int32 userID, Int32 projectID) { ... }
public String DeleteProjectByID(int projectID) { ... }
public ProjectInfo getProjectByProjectTitle(String projectTitle) { ... }
public List<ProjectInfo> GetProjects() { ... }
public List<ProjectInfo> GetProjectsForUser(Int32 UserId) { ... }
public String UpdateProject(Int32 projectId, String title, String description)
public String AddBug(Int32 creatorId, Int32 projectId, String initialIssueMessage,
                        short issueStatusId,Int32 priorityID) { ... }
public String DeleteBug(Int32 bugId) { ... }
public void DeleteBugForProjectAndUser(Int32 projectId, Int32 userId) { ... }
public List<Bug> GetAllBugs()
public List<Bug> GetAllBugsForProject(Int32 projectId) { ... }
public List<Bug> GetAllBugsForProjectAndUser(Int32 projectId, Int32 userId) { ... }
public string UpdateBug(Int32 ID, Int32 creatorId,Int32 projectId,
                            short statusId) { ... }
public String AddUser(String fullName, String userName, String password,
            Boolean isDeveloper, Boolean isTester, Boolean isAdmin) { ... }
public String DeleteUserByID(int personID) { ... }
public List<String> GetUsers() { ...
public Boolean IsValidPassword(String userName, String password) { ... }
public List<String> GetLogins() { ... }
public UserInfo GetUserInfo(Int32 userId) { ... }
public String GetUserInfoForUserName(String userName) { ... }
public String GetUserInfoForUserId(Int32 userId) { ... }
public String UpdateUser(Int32 userID, String fullName, String userName,
   String password,Boolean isDeveloper, Boolean isTester, Boolean isAdmin) { ... }
public String AddIssue(int projectID, String issueMessage,
                        Int32 creatorId, int statusId) { ... }
public String DeleteIssues(Int32 issueId) { ... }
public String DeleteIssuesForProjectAndUser(Int32 projectId,
                        Int32 userId) { ... }
public List<Issue> GetAllIssues() { ... }
public string UpdateIssue(Int32 ID, Int32 projectID, String issueMessage,
                                Int32 creatorId, int statusId) { ... }
public List<String> GetUsersForProject(Int32 projectId) { ... }
public List<String> GetPossibleStatuses(Int32 userId, Int32 projectId,
                                                Int32 statusId) { ... }
public void UpdateXUserProject(ProjectInfo project, ArrayList users) { ... }
    ...
```

8.1.2 Model Artifacts

We specify the operations as visual contracts. Visual contracts for a selection of operations are shown in Figure 8.2 whereas Appendix A is used to represent visual contracts for all the operations shown in Figure 8.1. We have appended "out" with the output parameters in the signatures to distinguish them from inout parameters, as shown in Figure 8.1. Parameters in the operations are used as variables in attribute expressions in the visual contracts. Consider Figure 8.2(a), where the signature *addProject(t:String, d:String, out Id:int)* has parameters *t* and *d* which are also used in the contract *addProject* to represent the possible sets of values for *title* and *description*. The parameter *Id* is appended with "out" to represent the fact that it is an output parameter and it represents the project id which is returned by the system. The signature associated with the visual contract in Figure 8.2(b) has a set of integers as output. The interface listing these signatures is shown in Figure 8.1. The signature *getAllProjects(out p:Set(int))* associated with the visual contract in the Figure 8.2(b) has a multi-object as output.

The pre-condition of the visual contract shown in Figure 8.2(b) allows us to find a match for any number of *Project* nodes. The post-condition states that the rule application has no effect. This allows us to model a query to select data from the system state. The models specifies functional requirements only, without addressing error handling or reporting. This may result in differences in signatures. For example, at implementation level *addProject* has return type *String* to carry success or error messages of the operation, while the model only records applicability or otherwise of the corresponding rule.



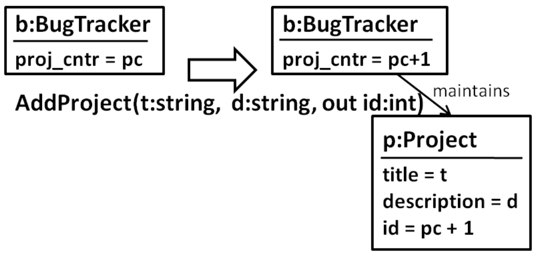
```
<<interface>>
BTSModelInt
AddProject(t:string, d:string, out pid:int)
AssignProject(pid:int, uid:int)
DeleteProjectByID(pId:int)
getProjectByProjectTitle(t:string)
GetAllProjects(out p: Set(int))
GetProjectsForUsers(out p:Set(int))
UpdateProject(pid:int, new_t: string, new_d:string)
AddBug(pId:int, uId:int, desc:string, sId:int, prId:int, out id:int)
DeleteBug(pId:int, uId:int, id:int)
DeleteBugForProjectAndUsers(pId:int, uId:int, id:int)
GetAllBugs(out b:Set(int))
GetAllBugsForProject(pid:int, out b:Set(int))
GetAllBugsForProjectAndUsers(pid:int, uid:int, out b:Set(int))
UpdateBug(bid:int, pId:int, uId:int, id:int, new_desc:string)
AddUser(f:string, l:string, uname:string, passwd:string, out uid:int)
DelUserById(uid:int)
GetUsers(out u:Set(int))
IsValidPassword(u:int, p:string, out b:boolean)
GetLogins(out l:Set(int))
GetUserInfo(u:int, out n:string)
GetUserInfoForUserName(name:string, out int:uId)
GetUserInfoForUserId(id:int, out name:string)
UpdateUser(uid:int, new_f:string, new_l:string, n_uname:string, n_passwd:string)
AddIssue(pId:int, uId:int, desc:string, sId:int, out id:int)
DeleteIssues(id:int)
DeleteIssueForProjectAndUser(pid:int, uId:int)
GetAllIssues(out i:Set(int))
UpdateIssues(uId:int, pId:int, sid:int, new_desc:string, id:int)
GetUsersForProject(pId:int, out u:Set(int))
GetPossibleStatuses(out s:Set(int))
UpdateXUserProject(pId:int, uId:int, n_f:string)
```

Figure 8.1: Model level signatures

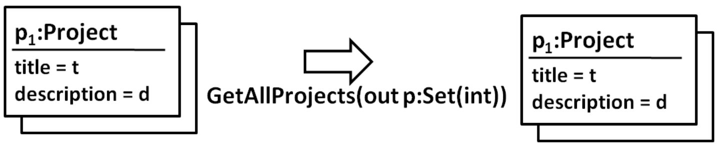
8.2 Test Oracles

We have presented a model-based approach to test oracles in Chapter 4. This section discusses the evaluation of our proposal using the bug tracker case study. In particular, we intend to find out the overheads of using our approach to test oracles. For evaluating our approach, we ask the following questions:

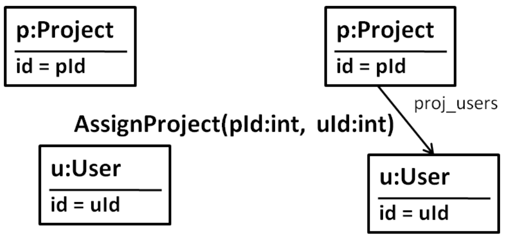
- Are there cases where the oracle reports false positives?



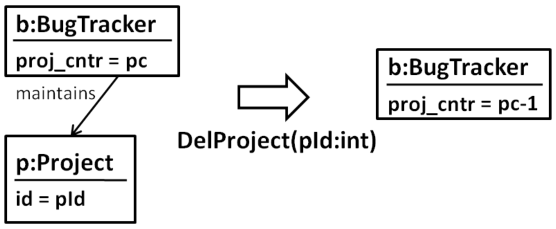
(a) visual contract: addProject



(b) visual contract: getAllProjects



(c) visual contract: assignProject



(d) visual contract: deleteProjById

Figure 8.2: visual contracts (a), (b), (c) and (d)

- What is the execution overhead of the oracle?

8.2.1 Setup and Execution

We tested operations in our bug tracker service by writing a client in Java using the integrated development environment (IDE) of *Eclipse*[2]. A test case is shown in Listing 8.2.

[2]available at `http://www.eclipse.org/`

Listing 8.2: JUnit Test:AddProject

```
...
    org.tempuri.BTSysServiceStub.AddProject addProject160 = ...;
    addProject160.setTitle(title);
    addProject160.setDescription(description);
    myAssert.successMessage = result;
    AddProjectResponse resp = stub.AddProject(addProject160);
    assertNotNull(resp);
    String res1 = resp.getAddProjectResult();
    aggEngine agg = new aggEngine();
    ArrayList<String> list = new ArrayList<String>();
    list.add("\"" + title + "\"");
    list.add("\"" + description + "\"");
    boolean res2 = agg.aggResult("C:\\localapp\\bts.ggx", "addProject", list);
    assertNotNull(res2);
    assertTrue(res2);
    myAssert.assertBothSucceededOrBothFailed((Object) res2, (Object) res1);
    agg.save();
    ...
   org.tempuri.BTSysServiceStub.AddProject addProject161 = ...;
   addProject161.setTitle(title);
   addProject161.setDescription(description);
   AddProjectResponse resp2 = stub2.AddProject(addProject161);
   assertNotNull(resp2);
   String res3 = resp2.getAddProjectResult();
   boolean res4 = agg.applyRule("addProject", list);
   assertNotNull(res4);
   assertFalse(res4);
   myAssert.assertBothSucceededOrBothFailed((Object) res4, (Object) res3);
    ...
```

In order to provide the test oracle, we need to include, in our project, the AGG engine[3] which provides the API, Our adapter, and our set of custom assertions which help in comparison. In order to answer the questions stated in the preceding subsection, we implement unit test cases using the *JUnit* framework provided by *Eclipse*. We provide custom assertions for comparing the model and and execution responses and

[3]available at `http://user.cs.tu-berlin.de/~gragra/agg/down_V203/index.html`

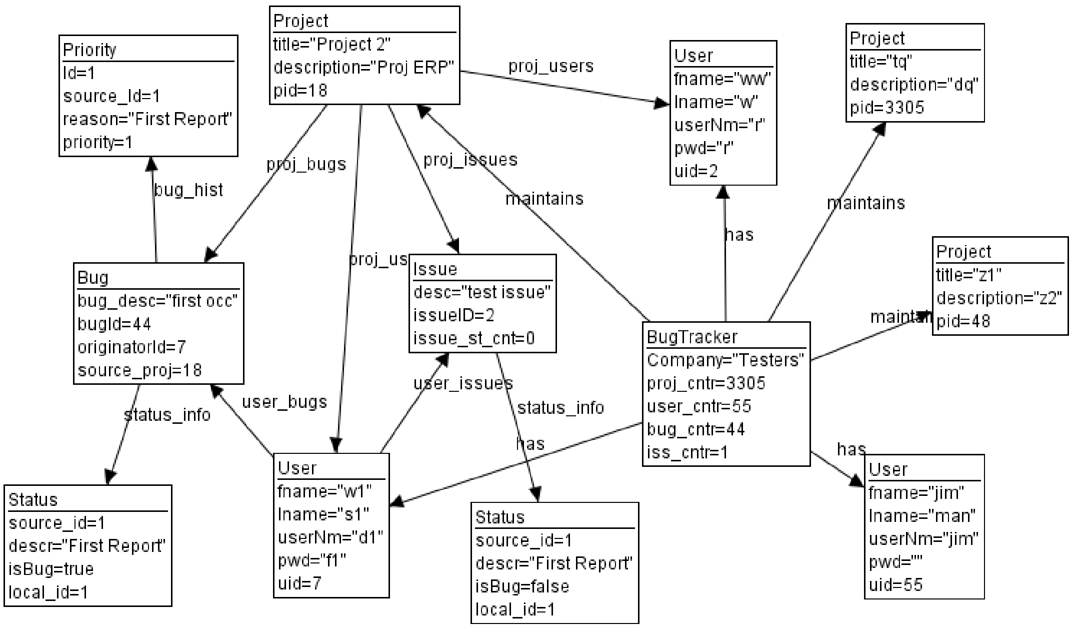


Figure 8.3: Initial state of the model

we also provide an execution log in case of a failure for the developer to inspect and decide if the failure is due to under-specification of visual contract. We consider the initial state of the system as shown in Figure 8.3 for test runs in Table 8.1 and the initial state as shown in Figure 8.4 for test runs in Table 8.2.

For the second question regarding the execution overheads, we write parameterized test case trying to add two projects with the same credentials in the model and in the implementation. We compare the execution responses considering the model as an oracle and providing the results manually for comparison.

8.2.2 Results and Evaluation

The results are reported based on Table 4.1. The model and the implementation for the cases shown in Table 8.1 report comparable results. Therefore, the assert statements report either “true” representing that the model and the implementation report comparable responses or both report “false” to represent that both the model as well as

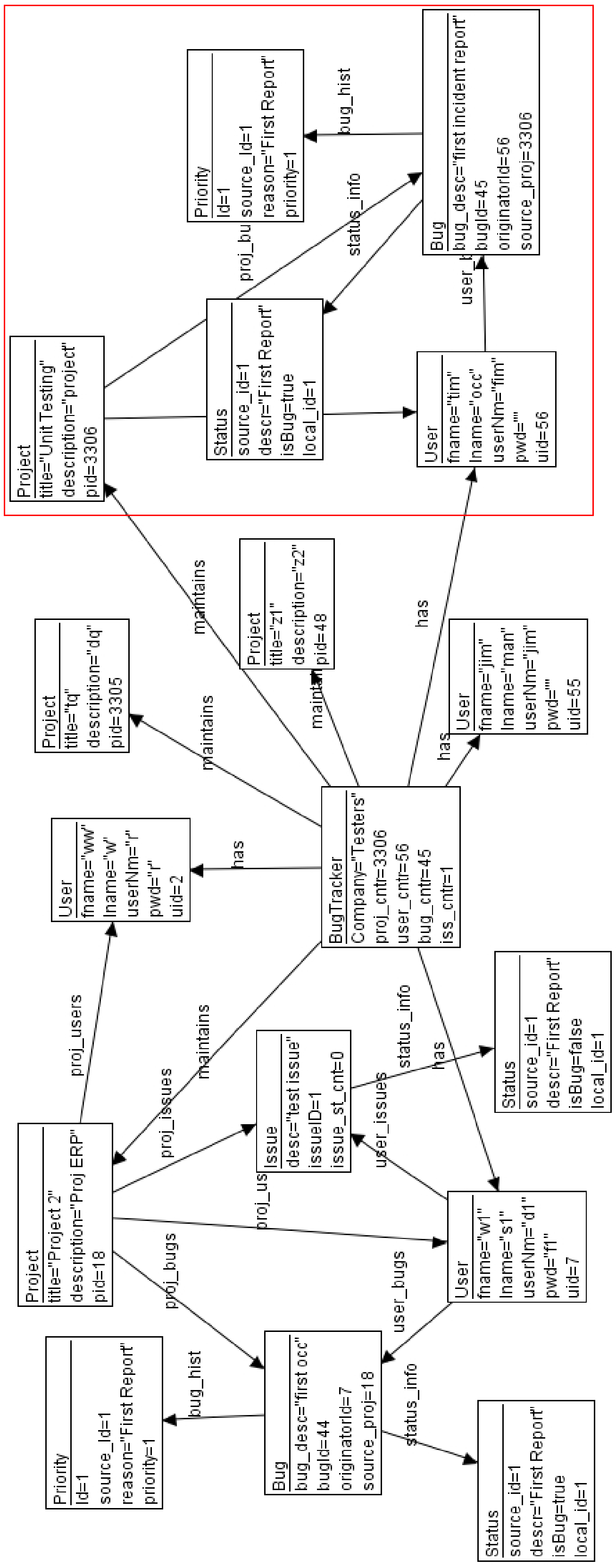


Figure 8.4: Initial model state

implementation register a failure.

Table 8.1: Successful test runs

Operation	Test Run
AssignProject	testAssignProject(48, 55)
DeleteProject	testDeleteProject(3065)
GetProject	testGetProject()
GetProjectByProjectTitle	testGetProjectByProjectTitle("Project 2")
GetProjectsForUsers	testGetProjectsForUsers(7)
UpdateProject	testUpdateProject(3305, "new Unit Testing", "new project")
DeleteBug	testDeleteBug(45)
DeleteBugForProjectAndUser	testDeleteBugForProjectAndUser(48,55)
GetAllBugs	testGetAllBugs()
...	...

However, the assert statements evaluated to false for the operations shown in Table 8.2 considering the start state as shown in Figure 8.4. The model was showing that the operation execution was successful whereas the implementation was registering a failure. These operations are shown in Table 8.2.

Table 8.2: Test runs related to under-specified visual contracts

Operation	Test Run
AddProject	testAddProject("Unit Testing", "project")
AddBug	testAddBug(48, 55, "first incident report", 1, 1)
AddUser	testAddUser("tim", "occ", "fim")
...	...

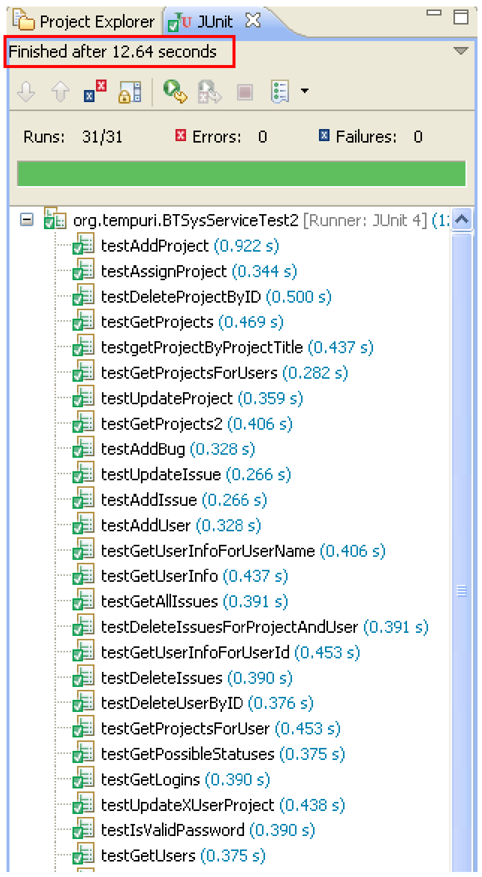
Our detailed investigation considering the log files reveals that the failure is due to a mismatch between the model and oracle response related to 5th row of Table 4.1. Considering the example of operation *addPoject*, we find that the visual contract is under-specified, allowing to add a project with same title twice, whereas the implementation restricted such entries. The fault trace is shown in Listing 8.3. We do not test for technical or communication failure since this is out of scope, as the model covers the functional aspects of the system.

Listing 8.3: Stack Trace Example

```
        ...

INFO:  --- Test Case run: class org.tempuri.addProjectTests: IMPLEMENTATION RESPONSE↩
     ----

12-Jun-2012 17:42:06
STDOUT: The result was: Error occurred while saving data...
Error Message:The changes you requested to the table were not successful because
they would create duplicate values in the index, primary key, or relationship.

Stack Trace  ...

12-Jun-2012 17:42:06
STDOUT: File name: C:\localapp\aggEngine_V202\Examples_V164\BasisUsing\bts.ggx

12-Jun-2012 17:42:06
STDOUT: Transformation  non-deterministically ...
12-Jun-2012 17:42:06 STDOUT: The gragra was set or not: true

STDOUT: Rule  addProject : step is done

        ...

12-Jun-2012 17:42:06 STDOUT: ---------------  TEST FAILURE -----------------
        ...
12-Jun-2012 17:42:06 STDOUT: ---------------  STACK TRACE -----------------
        ...
12-Jun-2012 17:42:06 STDERR:
junit.framework.ComparisonFailure: expected:<[true]> but was:<[Error occurred while ↩
    saving data...
        ...
```

We minimized the execution overhead by creating the object to communicate with the model and the implementation only once. The total execution time for our unit test cases was 12.64 seconds as shown in Figure 8.5.

The result of conducting the experiment with and without the use of model as oracle to find out the execution overheads is shown in Figure 8.6(a) and Figure 8.6(b)

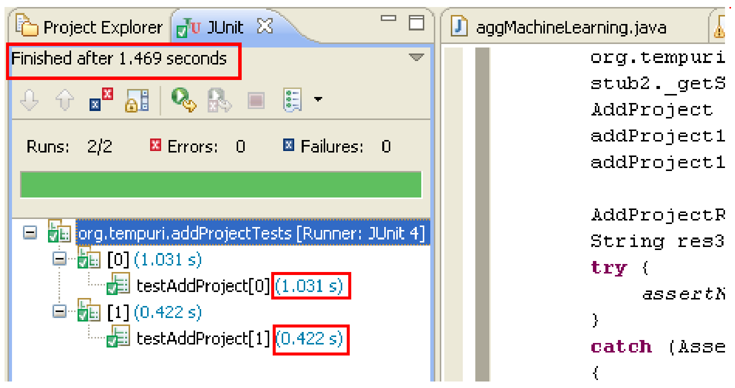


```
Finished after 12.64 seconds

Runs:   31/31        Errors:   0        Failures:   0

org.tempuri.BTSysServiceTest2 [Runner: JUnit 4] (1:
    testAddProject (0.922 s)
    testAssignProject (0.344 s)
    testDeleteProjectByID (0.500 s)
    testGetProjects (0.469 s)
    testgetProjectByProjectTitle (0.437 s)
    testGetProjectsForUsers (0.282 s)
    testUpdateProject (0.359 s)
    testGetProjects2 (0.406 s)
    testAddBug (0.328 s)
    testUpdateIssue (0.266 s)
    testAddIssue (0.266 s)
    testAddUser (0.328 s)
    testGetUserInfoForUserName (0.406 s)
    testGetUserInfo (0.437 s)
    testGetAllIssues (0.391 s)
    testDeleteIssuesForProjectAndUser (0.391 s)
    testGetUserInfoForUserId (0.453 s)
    testDeleteIssues (0.390 s)
    testDeleteUserByID (0.376 s)
    testGetProjectsForUser (0.453 s)
    testGetPossibleStatuses (0.375 s)
    testGetLogins (0.390 s)
    testUpdateXUserProject (0.438 s)
    testIsValidPassword (0.390 s)
    testGetUsers (0.375 s)
```

Figure 8.5: Test Case Execution Time

respectively. The execution time for a test case using the oracle was around double for both the successful and the failure case as shown in Figure 4.4.

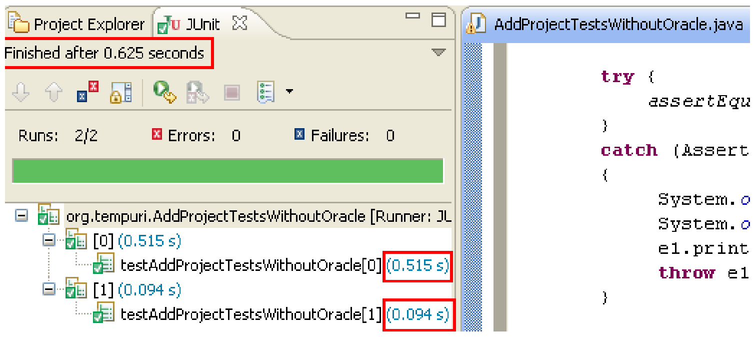
Referring to the questions raised in Section 8.2 and answering the question regarding false positives, we get a difference in the two responses if the visual contracts are under-specified. We provide execution logs for all such cases to document false positives so that the developers can analyze them more carefully. Our results regarding execution overheads show that the time taken for model execution is double the time it would take without considering model as an oracle, as shown in Figure 4.4. However, the maintainability of test cases seems to be an advantage in use of oracles as opposed to manual oracles since the rules are specified only once and can be executed as many times as required. Therefore, the execution time does not seem to be a problem if we



```
Finished after 1.469 seconds

Runs:  2/2     Errors:  0     Failures:  0

org.tempuri.addProjectTests [Runner: JUnit 4]
    [0] (1.031 s)
        testAddProject[0] (1.031 s)
    [1] (0.422 s)
        testAddProject[1] (0.422 s)
```

```
org.tempuri
stub2._getS
AddProject
addProject1
addProject1

AddProjectR
String res3
try {
    assertN
}
catch (Asse
{
```

(a) Execution time running model as oracle



```
Finished after 0.625 seconds

Runs:  2/2     Errors:  0     Failures:  0

org.tempuri.AddProjectTestsWithoutOracle [Runner: JL
    [0] (0.515 s)
        testAddProjectTestsWithoutOracle[0] (0.515 s)
    [1] (0.094 s)
        testAddProjectTestsWithoutOracle[1] (0.094 s)
```

```
try {
    assertEqu
}
catch (Assert
{
    System.o
    System.o
    e1.print
    throw e1
}
```

(b) Execution time without running model as oracle

Figure 8.6: Test case execution time

consider the benefits of using automated oracles.

8.2.3 Threats to Validity

Currently the execution time required to execute the tests does not contain the time required by the developers to understand how visual contracts are developed, how AGG works, and how to use our oracle proposal. There is a requirement to conduct experiments to see the effectiveness and usefulness of the approach such that one group tests a web service with the help of our adapter and the second group considers the traditional measures. For our experiments, we considered an average sized SUT as a case study and we need to consider bigger examples to see the scalability of the approach.

8.3 Coverage Analysis

In this section, we evaluate the relation of our model-based approach with traditional code-based coverage criteria where the system specifications in terms of $TAGTS$ are analyzed for conflicts and dependencies between the visual contracts or operation signatures specified as production rules. The *DG* is developed and the test cases considering suitable coverage criteria are selected and the code based coverage is analyzed for results. We also use multiple rules to represent different outcomes of one operation depending upon different data values passed to the operation signatures which give us a chance to cover more paths through our SUT. For evaluating the quality of our coverage criteria as well as the effectiveness of the test cases, we have conducted experiments considering the following questions.

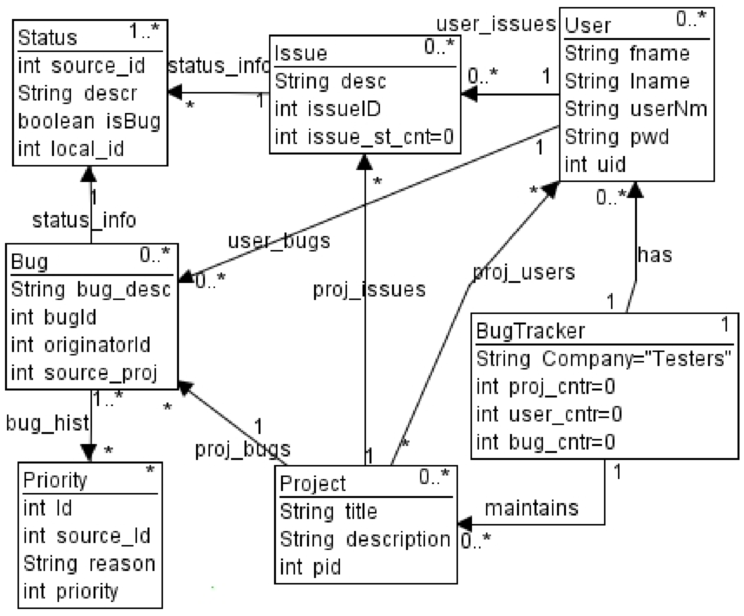
Since the code is not available for services or components, the tester would not have access to code-based coverage data. Therefore, using model-based criteria instead, we are interested in the following:

- We want to compare the model-based coverage reported by dynamic dependency analysis for a given test suite with code based coverage.
- We also evaluate the scalability of our approach in terms of the size of the specifications, the length of a test case, and number of test cases that can be executed in a given period of time.

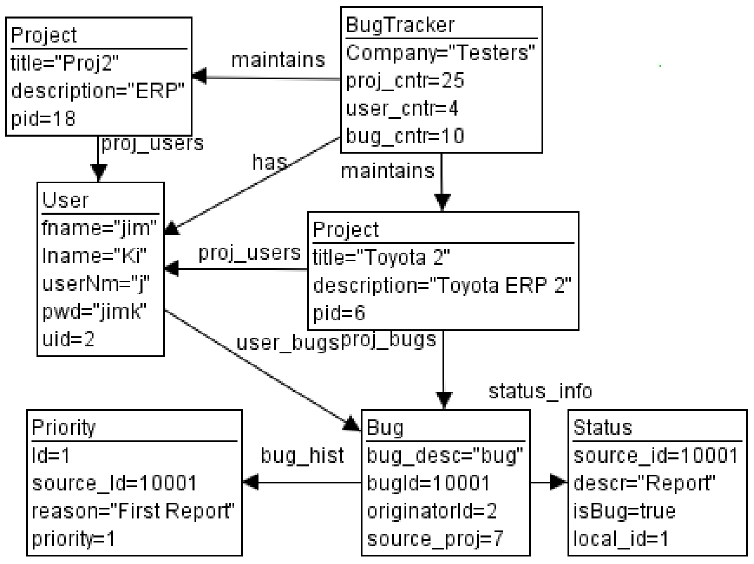
8.3.1 Setup and Execution

The model is formally represented as a typed attributed graph transformation system (TAGTS) where the visual contracts are specified as production rules over the type graph representing the class model of the service as shown in Figure 8.7.

We evaluate coverage with respect to model-based coverage criteria in relation to



(a) type graph



(b) start graph

Figure 8.7: type graph (a) and start graph (b)

code-based coverage. Using our own AGG-based tool to measure coverage with respect to the selected criteria on the model, we determine code-based coverage with respect to the most common criteria using the *NCover* tool.[4] In particular, we consider symbol and branch coverage. The first is essentially a more fine-grained version of

[4]See `http://www.ncover.com/`

statement coverage, including elements in expressions. The second requires that, for each condition of a branch (such as in an *if, while, do while*, etc.) both positive and negative outcome should be tested.

Model-based coverage is based on the dependency graphs in Fig. 8.8 and 8.9.

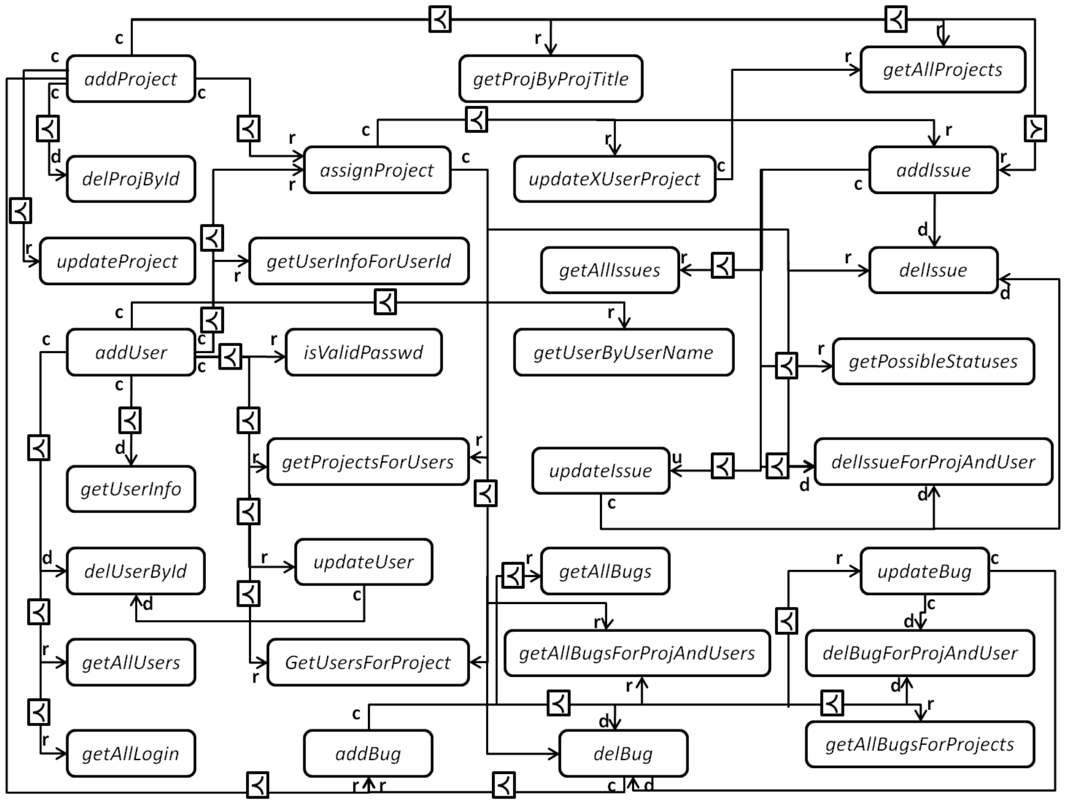


Figure 8.8: Dependency graph for ≺ relation.

For measuring the scalability of approach, we conduct experiments with large sequences of invocations.

8.3.2 Results and Evaluation

Results are reported in Table 8.3, where each row represents a selection of basic or combined model-based coverage criteria. We report the number of test cases necessary to achieve this coverage, the average length of these test cases, and the corresponding code-based coverage achieved with respect to the two criteria.

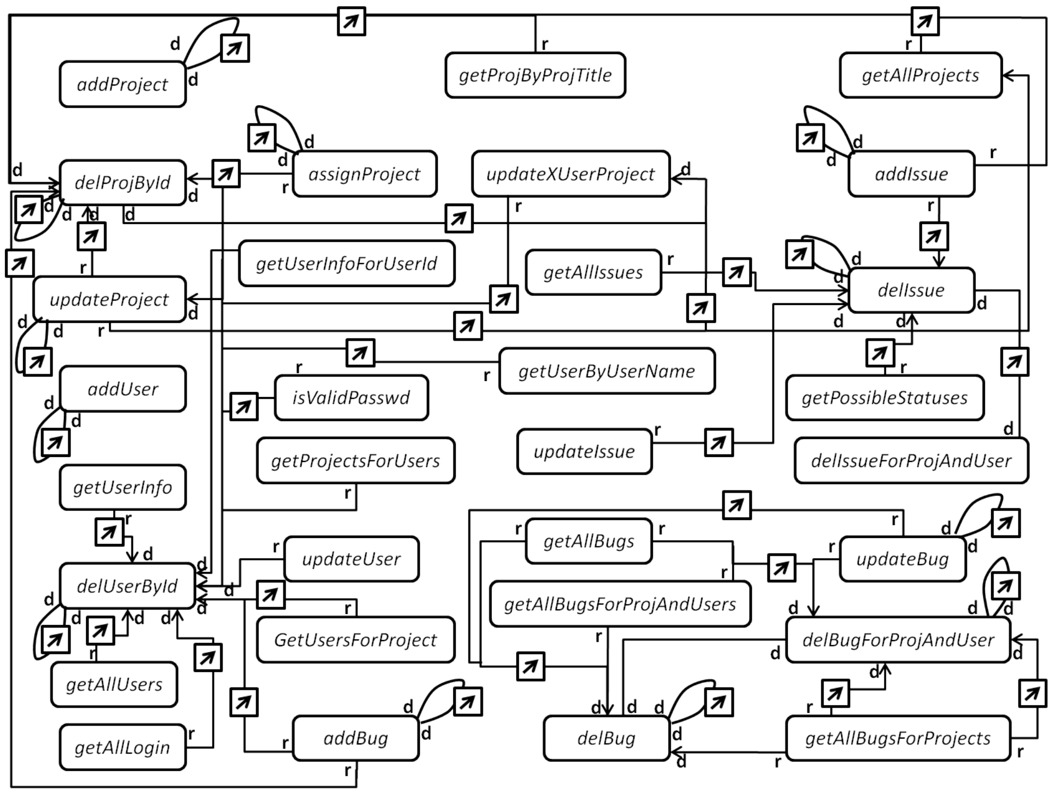


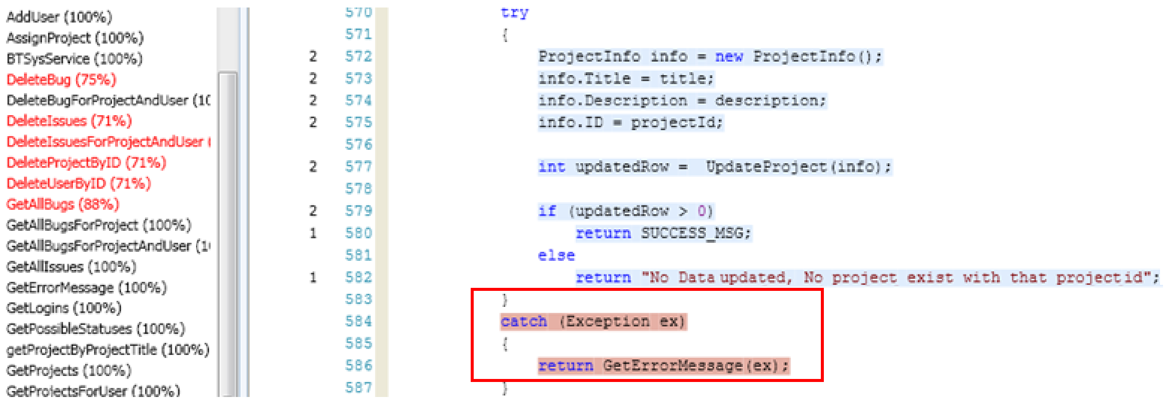
Figure 8.9: Dependency graph for ↗ relation.

Table 8.3: Label combinations indicating conflicts and dependencies

			SUT		
S/N	Criteria	# of test cases	average length of test case	Symbol Coverage	Branch Coverage
1.	*cd*	10	3	49.19%	45.07%
2.	*cr*	8	5	52.10%	56.34%
3.	*cr* + *cd*	10	7	83.50%	87.32%
4.	*cr* + *cd* +*rd* + *dd*	14	9	91.91%	92.96%

We consider the results discussed Subsection 8.3.2 and present our evaluation of these results. Considering the first row of Table 8.3 where the coverage criterion *cd* is used, we require eight test cases of average length 3 achieving 49.19% symbol and 45.07% branch coverage. For criterion *cr* the values are slightly higher, while combing the two coverage jumps above 80%. Obviously, some of the nodes are required by both criteria, such as *addProject* which is involved in both *cr* and *cd* edges.

The forth row represents the results for complete coverage of dependencies and conflicts in the dependency graph, but failed to provide complete coverage with respect to code-based criteria. Further analysis revealed that the remaining 8 – 9% correspond to code that is not executable as part of the normal behavior. This includes exception handling code triggered by technical errors outside the specification, e.g., a failure to connect to the data base, or glue code added by the IDE. An example is shown in Figure 8.10 where the code fragment in the rectangle is triggered by a technical failure. Since the approach is concerned with testing against functional specifications, technical exception handling is out of scope and the failure to cover it based on functional testing is not surprising.



```
AddUser (100%)
AssignProject (100%)
BTSysService (100%)
DeleteBug (75%)
DeleteBugForProjectAndUser (1(
DeleteIssues (71%)
DeleteIssuesForProjectAndUser (
DeleteProjectByID (71%)
DeleteUserByID (71%)
GetAllBugs (88%)
GetAllBugsForProject (100%)
GetAllBugsForProjectAndUser (1
GetAllIssues (100%)
GetErrorMessage (100%)
GetLogins (100%)
GetPossibleStatuses (100%)
getProjectByProjectTitle (100%)
GetProjects (100%)
GetProjectsForUser (100%)
```

```
     try
     {
2        ProjectInfo info = new ProjectInfo();
2        info.Title = title;
2        info.Description = description;
2        info.ID = projectId;

2        int updatedRow =  UpdateProject(info);

2        if (updatedRow > 0)
1            return SUCCESS_MSG;
         else
1            return "No Data updated, No project exist with that projectid";
     }
     catch (Exception ex)
     {
         return GetErrorMessage(ex);
     }
```

Figure 8.10: Unreachable code example

We have evaluated the scalability of our approach by considering the size of the specifications in terms of the number of rules and the size of the start graph as well as the length of the test case. Our case study has 31 rules with the start graph having 12 projects, 9 users, and 1 bug, plus priority and issue objects. We compute the static information, i.e., critical-pairs and minimal dependencies, only once using AGG's API and store them locally for repeated use. The time taken for this calculation is 783.53 seconds, while loading the stored data for subsequent use takes 1.72 seconds. This is acceptable given that the effort is only incurred once, but we have to be aware that the runtime is quadratic in the number of rules and exponential in the size of the rules themselves. That means, large numbers of complex operations will continue to pose challenges. On the other hand, service interfaces should be high-level, and split up if they become too large.

We also conducted experiments with different lengths of test cases. We produced a routine to automatically provide inputs for test case repeating only two rules, i.e., for adding and deleting a project repeatedly. That means, the size of the graph remains stable. The time taken for executing test cases of lengths $25, 50$ and 100 was 4.579, 12.844 and 65.189 seconds, respectively, while the system crashed with an *out of memory* exception for a sequence of 106 steps. The problem here is the maintenance of partial matches for all earlier rules into later graphs of the sequence, which incurs a cost quadratic in the length of the sequence in terms of the memory used. It should be noted that in practice, most test cases will be short (e.g., [9] states that the majority of faults are revealed by test cases of size 2), but the size of the longest possible sequence is still a limiting factor.

Executing 14 test cases of average length 9, as required but the 4th combination of criteria in Table 8.3, takes about 15 seconds. The effort is in fact linear in the number of test cases, so there is no significant barrier to executing large test suites.

8.3.3 Threats to Validity

The validity of the evaluation, in case of model-based coverage criteria and use of dynamic dependencies is limited by several factors. First, the implementation of the case study, if non-trivial, is relatively small, although the interface (and model) are of reasonable size. The sequences for evaluating scalability in terms of the length of test cases are clearly artificial, but based on our knowledge of the implementation we can say that the actions of the rules in the sequence are marginal for the effort, which is caused by maintaining and comparing matches into a large number of graphs. The start graph of 25 nodes used for the evaluation is probably realistic for tests with specifically created data, but tests using real data will require much larger graphs. These may represent not only a challenge to scalability, but also call for automation of the generation of start graphs from initial test data.

8.4 Regression Testing

In this section we evaluate both the correctness of our method and the reduction in the set of test cases obtained. We have used the same case study to verify our regression test suite selection methodology. For evaluation purpose, we have measured the sufficiency of our test suite using *fault seeding*, i.e., deliberate introduction of faults to be detected by the execution of test cases. The percentage of the seeded faults detected provides a statistical measure of the capability of the test set to find similar errors in the system, i.e., a measure of confidence in our test suite [59].

In order to decide which faults to introduce we identified suitable fault types, and then developed rules for seeding them automatically. After applying the rules to the code of the system, we execute the entire test suite to assess its quality. In an iterative process we add test cases until all of the seeded errors were detected. We evolve the system and classify the test cases into *RQ*, *RU* and *OB* and rerun the tests in both *RQ* and *RU* for comparison of results. We answer the following questions:

- Do the smaller sets of required test cases *RQ* find the same faults as the larger sets of reusable test cases *RU*?
- What is the difference in size between *RQ* and *RU* and what would be the smallest test set able to find the faults seeded?

For each evolution step the evaluation is performed in four steps that are outlined below and explained in more detail throughout the section.

1. Generation of test cases.
2. Validation of the quality of the entire test suite.
3. Classification of test cases into OB, RU, and RQ.
4. Validation of the quality and required size of RQ by comparing the results of executing RQ and RU.
5. Analyzing resulting coverage.

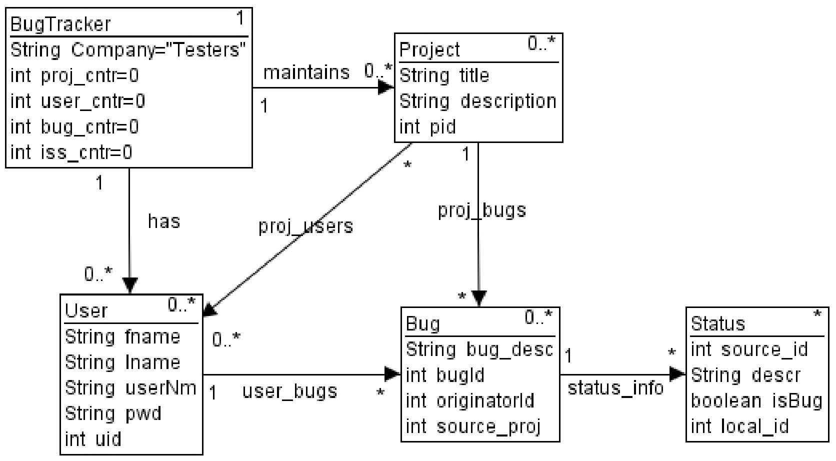
We, first of all, introduce three versions of the case study in which the evolution in the system is discussed below.

8.4.1 Setup and Execution

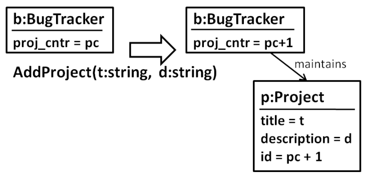
In this section, we first present an evolution scenario in two steps. Then we use the scenario to illustrate our approach to regression test suite reduction. The base case for the Bug Tracker allows us to record projects, users and reported faults. A selection of rules and type graph are shown in Figure 8.11.

In the first evolution step, the Bug Tracker service is extended in order to record Issues with the projects, i.e., concerns raised by users that may not be faults yet indicate deviations from their actual needs. The additional rules and extended type graph are shown in Fig. 8.12.

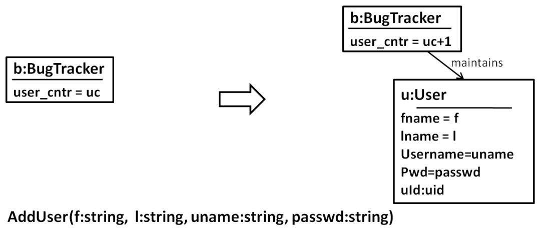
In the second evolution step we include a feature to record, among other details, a priority level while adding a bug. That means, the signature of $addBug(\ldots)$ is changed as well as its specification by the rule. Not surprisingly therefore, the modified operation will have additional dependencies, such as $addUser(\ldots) < addBug(\ldots)$. A minor update to $addIssue(\ldots)$ means that the description of the *status* is preset to "First Report" when the issue is initially reported. There is no change to the signature in this case, and the dependencies and conflicts are not affected. The new version of the changed rule along with the type graph are shown in Fig. 8.13.



(a) start graph



(b) addProject



(c) addUser

Figure 8.11: type graph (a) and rules addProject (b) and addUser (c)

Generation of test cases: We generated test cases manually, based on the information in the model, but without applying a formal notion of coverage. The completeness of the test set is evaluated instead trough *fault seeding*, i.e., deliberate introduction of faults to be detected by the execution of test cases.

Table 8.4: Distribution of seeded faults

Fault Type	# of	Seeded	Faults
	V1	V1	V2
Wrong declaration	6	8	9
Wrong assignment	23	34	35
Wrong proc. handling	27	32	35
Control faults	22	27	29
I/O faults	27	32	35
Total	105	133	143

Following the same approach as reported in subsection 7.4.1, we have generated 41 test cases for version $V1$, 51 test cases for version $V2$ and 54 test cases for version $V3$.

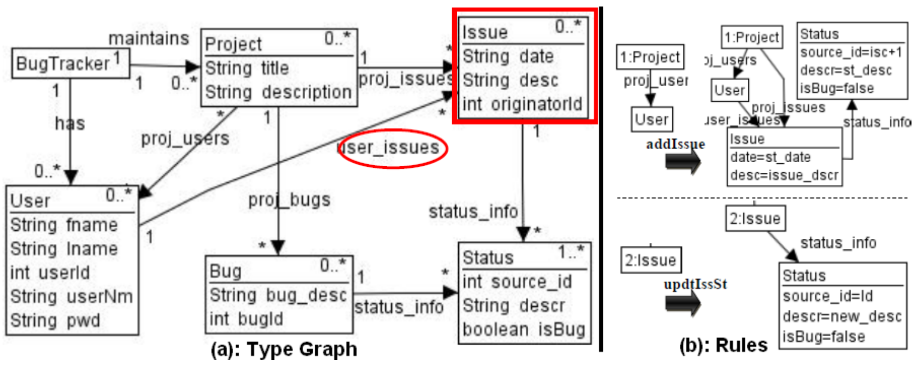


Figure 8.12: BugTracker, evolution to Version 2

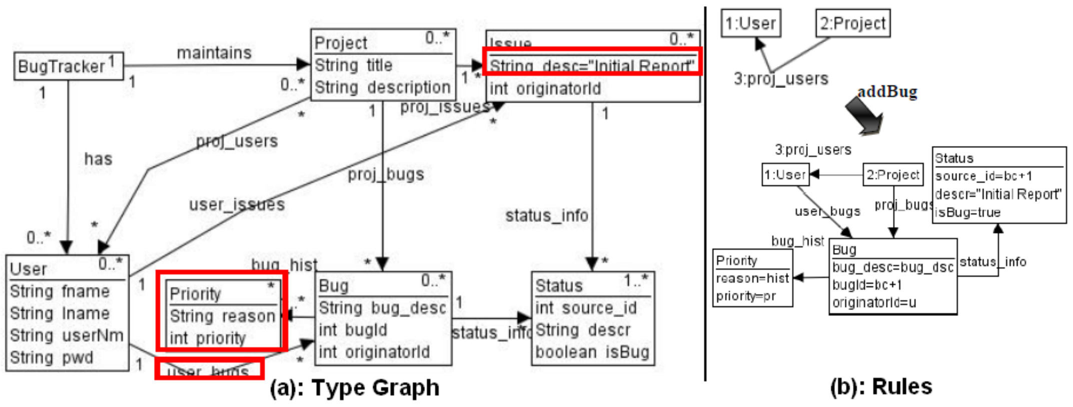
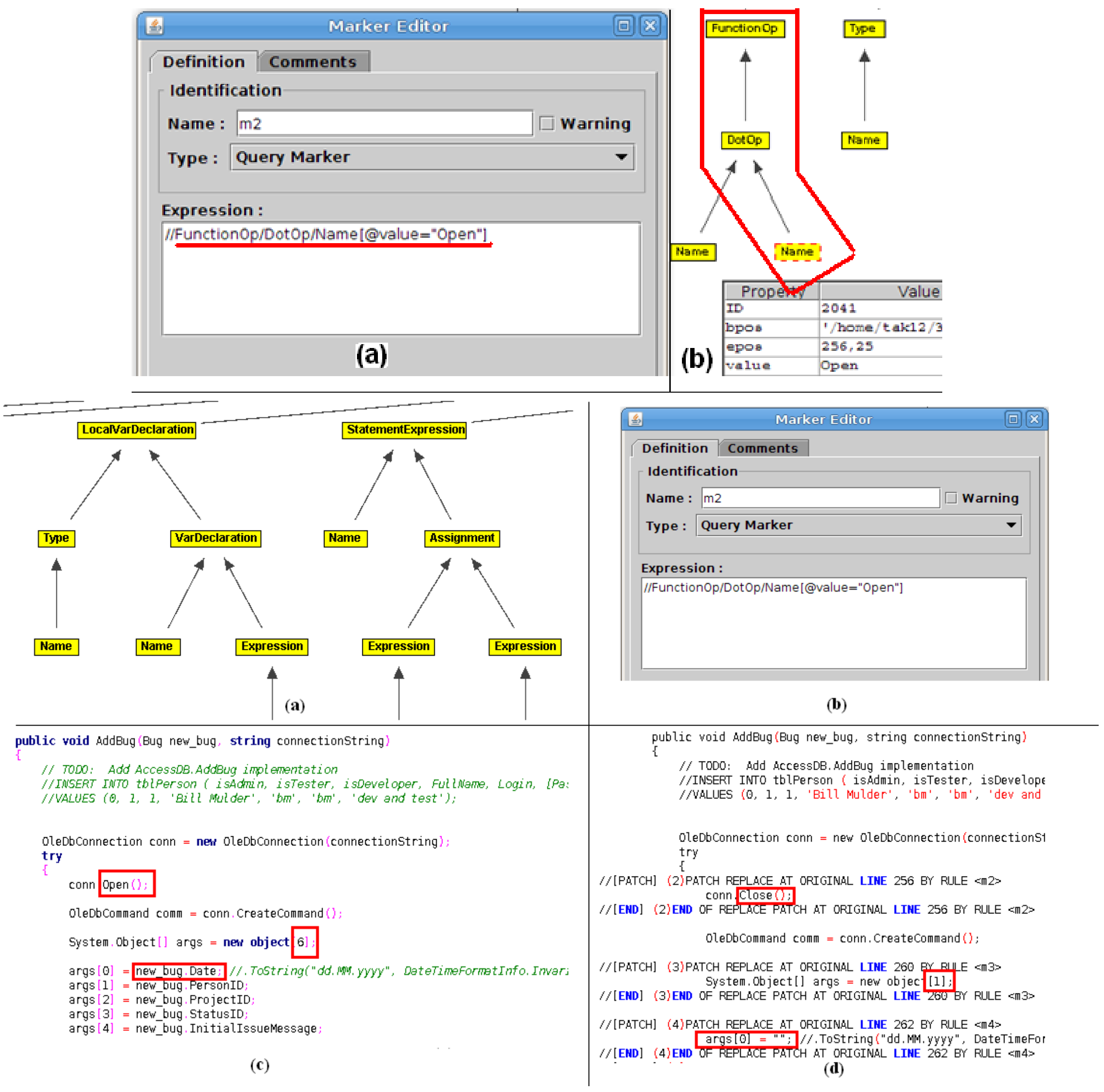


Figure 8.13: BugTracker, evolution to Version 3

Validation of quality of entire test suite: Rules for seeding faults according to these types are implemented in the source code transformation tool L-Care[5], which allows to define markers based on *XPath* queries as shown in Fig. 8.14(a) on an XML representation of the code. A sketch of this XML in tree form is shown in Fig. 8.14(b). Examples of the original and the fault-seeded code are shown in Fig. 8.14 (c) and (d) respectively. Table. 8.5 shows the total number of faults seeded for each version as well as a breakdown into the different types along with typical representatives.

Classification of test cases into *OB*, *RU*, and *RQ*: We classified the test cases of one version into obsolete, reusable and required subsets and reran the subset *RQ* on the next version. Considering step $V1 \rightarrow V2$, we classified 41 test cases that were existing

[5]A product of `http://www.atxtechnologies.co.uk/`



Property	Value
ID	2041
bpos	'/home/tak12/3
epos	256,25
value	Open

```
public void AddBug(Bug new_bug, string connectionString)
{
    // TODO:  Add AccessDB.AddBug implementation
    //INSERT INTO tblPerson ( isAdmin, isTester, isDeveloper, FullName, Login, [Pa:
    //VALUES (0, 1, 1, 'Bill Mulder', 'bm', 'bm', 'dev and test');


    OleDbConnection conn = new OleDbConnection(connectionString);
    try
    {
        conn.Open();

        OleDbCommand comm = conn.CreateCommand();

        System.Object[] args = new object[6];

        args[0] = new_bug.Date; //.ToString("dd.MM.yyyy", DateTimeFormatInfo.Invari
        args[1] = new_bug.PersonID;
        args[2] = new_bug.ProjectID;
        args[3] = new_bug.StatusID;
        args[4] = new_bug.InitialIssueMessage;
```

```
        public void AddBug(Bug new_bug, string connectionString)
        {
            // TODO:  Add AccessDB.AddBug implementation
            //INSERT INTO tblPerson ( isAdmin, isTester, isDevelope
            //VALUES (0, 1, 1, 'Bill Mulder', 'bm', 'bm', 'dev and


            OleDbConnection conn = new OleDbConnection(connectionSt
            try
            {
//[PATCH] (2)PATCH REPLACE AT ORIGINAL LINE 256 BY RULE <m2>
                conn.Close();
//[END] (2)END OF REPLACE PATCH AT ORIGINAL LINE 256 BY RULE <m2>

                OleDbCommand comm = conn.CreateCommand();

//[PATCH] (3)PATCH REPLACE AT ORIGINAL LINE 260 BY RULE <m3>
                System.Object[] args = new object[1];
//[END] (3)END OF REPLACE PATCH AT ORIGINAL LINE 260 BY RULE <m3>

//[PATCH] (4)PATCH REPLACE AT ORIGINAL LINE 262 BY RULE <m4>
                args[0] = ""; //.ToString("dd.MM.yyyy", DateTimeFor
//[END] (4)END OF REPLACE PATCH AT ORIGINAL LINE 262 BY RULE <m4>
```

Figure 8.14: Fault seeding with L-Care

for $V1$ and we found out that all of them are reusable. The result of classification is shown in Table 8.6.

Validation of the quality and required size of *RQ*: We seed faults in the modified classes of $V2$ and $V3$ only and execute the two sets of required test cases RQ to determine if all of the seeded faults are discovered and how many test cases are actually required to discover them. We have seeded 28 and 18 faults in $V2$ and $V3$, respectively, the smaller numbers owing to the size of the changed classes in comparison to the

Table 8.5: Distribution of seeded faults

Code Examples	
Correct Statement	Mutant Statement
new object[6]	**new object**[0]
args[0] = DateTime.Now;	args[0] = “ ”;
throw ex	//**throw** ex
if (conn.Open == ...)	if (conn.Open != ...)
conn.Open()	conn.Close()

entire code base.

8.4.2 Results and Evaluation

The results are reported in Table. 8.6. We record the number of test cases in each category *produced* by our classification as well as the number of test cases actually *successful* in finding faults. Of step $V1 \rightarrow V2$ we recall that $OB = RQ = \emptyset$ because none of the existing operations were modified. Unsurprising, therefore, none of the remaining test cases in RU found any fault, but 10 new test cases NT had to be produced to detect faults seeded into newly added operations. With the second evolution step, 10 out of 39 existing test cases were classified as required RQ, of which 5 where successful in finding faults. Again, 15 new test cases where added such that 12 were added to replace the test cases going obsolete due to evolution in signature of *addBug* and 3 new ones to cover features not addressed by existing test cases. That means, our reduction in the size of test suites has not resulted in missing any faults, i.e., the numbers of faults discovered using RU and RQ are the same.

Table 8.6: Test case classification and success rate

	$V1 \rightarrow V2$		$V2 \rightarrow V3$	
Test cases	produced	successful	produced	successful
Obsolete (OB)	0	0	12	0
Reusable (RU)	41	0	29	5
Required (RQ)	0	0	10	5
New (NT)	–	10	–	15

The reduction in size is significant, but probably still not optimal, because a smaller set of 5 rather than 10 test cases would have been sufficient. This is despite an exhaustive error seeding strategy, which produced faults of the designated types wherever this was possible in the code. The reason could be in over approximation of dependencies and conflicts which, like in many static analysis approaches, leads us to err on the cautious side.

Analyzing resulting coverage: We find that the coverage is retained in case of $V1 \rightarrow V2$. However, the edges shown in Table 8.7 have become uncovered due to test cases going obsolete in case of evolution $V2 \rightarrow V3$.

Table 8.7: Uncovered edges

S No	Edges
1	Edge between assignProject() and delBug()
2	Edge between addBug() and deleteBug()
3	Edge between addBug and delBugForProjectAndUser()
4	Edge between updateBug() and delBug()
5	Edge between addBug() and getAllBugsForProjAndUser()
6	Edge between addBug() and getPossibleStatuses()
7	Edge between getPossibleStatuses() and delBug()

8.4.3 Threats to Validity

Considering our results of regression test suite reduction, the assessment depends on the quality of the original test suite, which was evaluated by fault seeding while using the set of reusable test cases RU as a benchmark for the required ones RQ. But fault seeding will only deliver relevant results for the types of faults actually sown, while unexpected or unusual faults are not considered. Our approach here was to use approaches to fault classification from the literature, but in order to gather relevant statistics about the costs savings possible we would require data on error distributions from real projects. Scalability of the approach as well as the coverage analysis needs to be verified considering large scale case studies

8.5 Summary

We considered a publicly available desktop application and converted into a web service for evaluation of our presented approaches. We have shown how we can benefit from the use of a model as an oracle and how effective are our model-based coverage criteria. We have presented the limitation of the proposal regarding models as oracles and explained treatment to the false positives reported by our approach. We have demonstrated our coverage analysis based on dynamic dependencies and conflicts and evaluated the scalability of our approach. We have also applied our regression testing approach and evaluated its effectiveness and efficiency.

Part III

Comparison and Conclusion

Chapter 9

Comparison to the State of the Art

In this chapter, we give a critical account of some of the competing approaches for model-based test oracles, coverage criteria, and regression testing. We also compare and contrast our results to highlight the salient points as well as the limitations of our approach. We compare our use of visual contracts for test oracles and model-based testing. We summarize our research objectives as follows:

- Since web services hide implementation details, we require a model-based approach to test oracles.
- Web service testing bears additional overheads and testing is, anyway, a time consuming activity. Therefore, we require model-based coverage criteria to assess the completeness of test suites.
- We also need a model-based approach to regression testing allowing selective retesting.

Our requirements are motivated by non-availability of implementation details in case of web services, and the fact that web services testing involves additional costs such as service invocation costs, usage of network resources, etc. We evaluate our contributions with respect to our objectives, provide a comparison with competing approaches and comment on the suitability and effectiveness of our proposal.

9.1 Test Oracles

Our approach for test oracles, model-based coverage criteria as well as regression testing is based on the analysis of visual contracts formalized as a TAGTS. We first describe how visual contracts have been used previously before commenting on our approach.

Visual contracts have been used in [31] for interface specification of services as well as for model-based testing [33, 50, 29]. The use of visual contracts in [33, 50, 29] is supported by a formal interpretation as graph transformation rules. The approach introduced in [33] specified visual contracts as rules in a TAGTS for deriving functional tests and used *JTest* and *Parasoft* for test execution.

Visual contracts are also used in [29] for formalizing pre- and post-conditions of use cases to be used as test models for the generation of logical test cases. This work provides the basis for establishing a relation between UML specifications and visual contracts and proposes a test suites generation mechanism for required and provided interfaces. The work reported in [29] proposes to make them executable for the generation of test cases.

Visual contracts have been used in [50] not only for testing individual operations but also for operation sequences. The work proposes a mapping between visual contracts and Java modeling language (JML) assertions that can be considered as providing an oracle. A mapping between visual contracts and the Java classes using JML is also reported in [51] where the authors have proposed how correctness of the implementation can be verified against the specification. The proposal considers a runtime assertion checker where the use of visual contracts is to specify the data state transformation between a pair of UML object diagrams representing pre- and post-conditions. Visual contracts are also used in model-driven development (MDD) e.g., in [24] where the authors have made use of graph transformation principles for solving horizontal and vertical model transformation issues. The models are then transformed into JML assertions and monitoring is done through the help of an Eclipse plug-in and JML assertions.

Our use of visual contracts provides a means of executing specifications through an adapter linking the model signatures with rules in AGG. We derive a dependency graph (DG) using critical-pair analysis. We base our model-based coverage criteria on annotations on the start and end points of the edges between nodes in our DG. We check, dynamically, if a static dependency was exercised. This is novel to our approach.

We exploit the information present in visual contracts for providing a test oracle by filling gaps in abstraction between service implementation and visual models. Oracles can be realized by a manual process, a separate program implementing the same process, a simulator producing parallel results, an earlier version of the same software, checking values against known responses, etc [35]. An oracle should be independent of the SUT in providing its responses and could be based on specification or the implementation artifacts [72]. Several approaches for test oracles are presented based on system specifications [61], documentation [58], and parallel implementation [71, 6].

Test oracles for web service testing are proposed in [68] where each service has several implementations and there is a training phase to make available the oracle in the first place. Another approach [17] is based on the idea of metamorphic testing which emphasizes the usage of relations between the inputs and the outcomes of several executions of a method under scrutiny. This allows the reuse of existing test cases to generate more test cases. [17] use these metamorphic services as access wrappers providing encapsulation to the actual SUT. The test oracle is in the metamorphic services which compute the follow-up messages and predicts results of the SUT.

Table 9.1: Comparison test oracles approaches

Approach and Author	**Comparison with Our Proposal**
Visual Contracts used for unit testing with JTest [33]	We use visual contracts and provide executable specifications for testing
Visual contracts for test oracles using JML where Graph transformation rules are translated into JML assertions [50] and a runtime assertion checker [51]	We use model-level information and provide model as an oracle where we specify visual contracts as rules in AGG and execute them using API
Group testing by applying one test case to all web services in a group with outputs stochastically ranked and oracle established if majority of web services report the same output [68]	Our proposal does not need a training phase and the result is not based on a majority vote
Metamorphic services are used as access wrappers providing encapsulation to the actual SUT and providing test oracle by comparing results of metamorphic service that computes follow-up message and predict results of actual service [17]	We directly execute the visual contracts specifying the operation formalized as rules in TAGTS where the model is made executable as an oracle
Visual Contracts are used for test case generation [29]	We execute visual contracts as oracles

Test sheets for oracles where UML and OCL are used to represent the operational features of web services under test [4]	Outputs are computed by executing visual contract considering system start state given as start graph

Test oracles for web services are also proposed in [4] where tables describing sets of test cases, called test sheets, are used. These tables contain inputs and the sets of possible outputs. The approach builds on concepts defined in the Framework for Integrated Test [1]. The approach suggests a manual process for the provision of test oracles for web services [12].

Our approach [43] is different from those mentioned above. We propose a mechanism to execute service specifications and do neither rely on a training phase nor do we require the additional overhead required for metamorphic testing. Our work is also different from [29] where visual contracts are made executable for generation of test cases.

We provide a comparison of our approach with the competing approaches in Table 9.1. Our objective was to use model-level information to provide test oracles which neither require additional wrappers, as required in [17], nor depends upon the stochastic analysis of outputs for establishing an oracle. Our proposal of executable specifications is an automatic process, in contrast to the proposal of [4] where test oracles are manual.

9.2 Model-based Coverage Criteria

Coverage criteria for dataflow-based methods include all def paths, all $def-use$ paths, etc., where def and use are the labels on nodes to annotatate where a data element

[1]available at `http://fit.c2.com`

was created and used respectively [67, 26, 60]. Coverage criteria for control-flow graphs are statement coverage, which ensures that all statements are executed at least once; decision coverage that every decision point is invoked at least once[11, 39, 8], etc. We have analyzed visual contracts and presented our analysis in the shape of a dependency graph based on which we determine model-based coverage criteria. For this, we considered model-level information about dependencies and conflicts between operations. Dataflow graphs considering specifications are presented in [8], where the specifications are first analyzed for correctness. A DG is then constructed and the test paths are identified.

Specifications have been considered for control-flow graph (CFG) development for web services in [49] using resource description framework (RDF) schema. RDF graphs are extracted from the functional specifications of web services comprising axioms defining pre-conditions and effects of invocations. A CFG is derived from these graphs and the extracted CFG is then interpreted as extended finite state machine (EFSM) to define coverage criteria. Semantic Web service descriptions in (WSDL-S) are used to construct a finite state machine (FSM) by first expressing pre-conditions and effects using the web ontology language (OWL) having semantic web rule language (SWRL) extensions [65].

There are several approaches to testing web services based on dataflow graphs extracted from semantic information [7, 65, 36]. The approach discussed in [7] is aimed at testing service composition using $BPEL$ specifications. $BPEL$ is also considered in [36], where dependency analysis is carried out over variables acquired from $WSDL$ interface description to extract paths through the graph of the $BPEL$ specification. Criteria for data flow testing, originally established in [26], are applied by [52] to the functional testing of services using $BPEL$ and $WSDL$. The authors of [54] have made use of call-based dependence graphs for coverage in object-oriented systems. They incorporate both control and data dependence. In our approach, the combination of data and control flow analysis could be interesting when considering service specifications complementing visual contracts with orchestrations. However, our handling of data dependencies is more advanced than what can be extracted from operation signatures in *WSDL*.

Rather than using finite state machine or control-flow information, we have constructed dependency graphs through visual contracts analysis formalized by rules in a TAGTS. Approaches to model-based testing using data dependencies for object-oriented systems are also considered in [18, 13].

Table 9.2: Comparison of model-based coverage approaches

Approach and Author	**Comparison with Our Proposal**
Model-based conformance testing of web services where extended WSDL specifications are used to derive finite state machine for generating test cases and providing predicate coverage [65]	We use information about conflicts and dependencies between visual contracts specifying operations in a web service and propose coverage criteria to find out completeness of a given set of test cases
BPEL processes are analyzed for collecting dataflow information to test compositions [36]	We consider model-level information and analyze visual contracts formally specified as rules in a TAGTS
Control-flow analysis for coverage extracted from service specifications considering RDF schema [49]	We propose a dataflow based coverage criteria by representing the dataflow related information as a dependency graph

Using Frankl-Weyuker dataflow criteria to test BPEL services where BPEL and WSDL are considered for test case generation and all-uses is considered as coverage criteria [52]	We identify data creation, read and deletion for coverage and, instead of all-uses, we propose create-read, create-delete, read-delete and delete-delete criteria covering more options than all-uses
Dataflow based analysis for web service compositions [7]	We propose coverage considering operations within a web service
Dependence representation for coverage [54] based on control and dataflow information using notation for object-oriented systems	We consider dependency graph and use labels on the edges for the proposal of coverage criteria for web services
Testing of web service compositions is proposed by considering BPEL and timed extended finite state machine for criteria, e.g., transition coverage [46]	Our proposal of coverage criteria is dataflow based and does not consider state machine related coverage
Integration testing using dataflow information where the authors introduce a formal language to develop dataflow diagram for path coverage for object-oriented systems [18]	We formalize dataflow information as dependency graph to test web services where the coverage criteria is proposed on the basis of edge labels resulting from the use of formal concepts of conflicts and dependency analysis in theory of graph transformation [22]

Converting UML state machines into flow graphs based on events and actions where OCL expressions associated with nodes in the flow graph are analyzed for refining state-based criteria [13]	Our focus is web services where we represent the information as a dependency graph to propose dataflow based coverage criteria for a given set of test cases

We have made use of web service specifications by means of visual contracts for deriving a dependency graph to define coverage [32]. Dependencies and conflicts extracted by critical pair analysis provide a simple representation of the system at the interface level, abstracting from detailed control flow. Our model-based analysis makes us independent of programming languages and platform related details, which is suitable for the platform-independent nature of web services. We provide a comparison of our approach [42] with the competing approaches in Table 9.2.

Considering the comparison given in Table 9.2, our approach covers combinations of create, read and delete effects on data than [52] which only proposes to cover all-uses as coverage crieteria. We consider operations within one service in contrast with [7, 36]. Our coverage criteria are based on conflicts and dependency analysis where approach in [65] uses guard conditions and extended finite state machines to provide predicate coverage. The approaches in [54, 18] consider dependency graphs derived through a different process and provide coverage for object-oriented systems whereas our focus is web services.

9.3 Regression Testing

Several techniques [45, 19, 14] have been using model level information for regression testing. Extended finite state machine (EFSM) are considered in [45], where interaction patterns between functional elements represented by transitions are used for test set reduction. Two tests are considered equivalent if they represent the same interaction pattern. Therefore, whenever a transition is added or deleted, the effect of the model on the transition, the effect of the transition on the model and any side effects are tested for. That means test cases are selected with respect to elementary modifications of the state machine model.

EFSM are also considered in [19] where a set of elementary modifications (EM) is identified. Two types of dependencies, data dependencies (DD) and control dependencies (CD) are discussed. A state dependence graph (SDG) represents DD and CD visually and a change in the SDG leads to a regression testing requirement to verify the effect of the modification.

The technique presented in [14] uses UML use case and class diagrams with operations described by pre and post conditions in object constraint language (OCL). A unique sequence diagram is associated with a use case to specify all possible object interactions realizing the use case. Changes in the model are identified by comparing their XML metadata interchange (XMI) representations. An approach to regression testing of web services suggested by [57] makes use of unit tests based on *JUnit*. Test cases are produced by the developer, who generates *QoS* assertions and XML-encoded test suites and monitors I/O data of previous test logs to see if the behavior is changed.

The approach discussed in [64] constructs a global flow graph by requiring a flow-graph from each party in the collaboration. It then defines call nodes as a special nodes for remote service calls to record the operation name and the service uniform resource identifier (URI). A CFG containing a call node is converted by inserting the call graph corresponding to that call node. Whenever an operation is modified, the previous and the resulting call graphs are compared to find the differences and all downstream edges are marked as “dangerous” once a modified node is marked.

Table 9.3: Comparison of regression testing approaches

Approach and Author	Comparison with Our Proposal
Maintaining JUnit test cases for regression testing where service integrator maintains a log of test cases and executions and QoS assertions for preexisting test cases are generated by a Java toolkit where tests and assertions constitute executable contracts [57]	We analyze visual contracts to identify impact of change and select test cases into subsets and perform coverage analysis to see if the resulting test suite retains coverage
Develop control-flow graphs for composite web services where control-flow information for web services is considered. The approach is applicable to web services compositions where the change information is analyzed by maintaining and updating call graph and by identifying dangerous edges [64]	We analyze the information in terms of visual contracts where our focus is how the change affects the model. We also need to know which of the code artifacts were accessed during update. Our focus is different as we consider one web service and not compositions
Extended finite state machine considering control and data dependence to construct a state dependence graph which is used to identify the effect of modification [19]	We evaluate the change in model through visual contracts and do not consider control dependence

Model-based regression testing using data dependence analysis where Extended finite state machine is used for dependence analysis in object-oriented systems for reducing regression test suites [45]	Our analysis is based on visual contracts analysis and not on extended finite state machines and the application domain is web services
UML designs are considered where pre- and post-conditions are given as OCL constraints and the changes in model are analyzed by analyzing their XML representation [14]	We consider visual contracts to represent pre- and post-conditions formalizing them as rules in a TAGTS for analysis and propose a regression testing approach for web services

Our approach [41] is different in the sense that not only we consider the impact of evolution for selective retesting but we also propose a coverage analysis mechanism to see if there is any requirement of new test cases to retain coverage as well. Dependency information used, e.g., in [45, 19] is instead derived from state machines. Pre and post conditions on application data are also used with [14].

While conceptually close, our visual contracts are more easily usable than a textual encoding in OCL and provide a formal operational semantics with a well-established theory of concurrency as a basis for verifying formally the correctness of our approach. We provide a comparison with our approach in Table 9.3.

9.4 Summary

We have presented in this chapter a comparison of our approach to some of the state-of-the-art approaches. Considering our approach to test oracles, we were able to use model-level information to provide test oracles which not only handle a variety of challenges. We have also provided a comparison where specifications have been considered for coverage criteria. We have also covered approaches where visual contracts were used for testing purposes. Lastly, we have presented a comparison of our approach with approaches considering model level information as well as approaches for web services.

Our use of visual contracts for proposing test oracles, coverage criteria and regression testing is unique for providing an executable model as an oracle. Our analysis of visual contracts specifying operation in a web service formalized as rules in a TAGTS to arrive at dependency graph and annotating the edges with labels to propose coverage criteria is also novel. Our proposal of analyzing visual contracts for regression testing and coverage analysis for evolved version of the system is also original.

Chapter 10

Conclusion and Outlook

This chapter provides a conclusion and discusses possibilities of future extensions to the work presented in this thesis.

10.1 Conclusion

We have used high level visual specifications for oracle development, in line with the platform-independent nature of web services and mainstream software modeling languages. Our contribution is not in the generation of test cases, but in helping the tester to implement them by automating the decision, if the response from the operation being tested is correct. This information is present in visual contracts and should be reused rather than re-implemented. Other test-related activities, such as debugging, are not directly affected. In order to generate test oracles, an adapter needs to translate invocations of services under test into rule applications, passing and converting parameters and interpreting replies. We differentiate logical failures from technical or communication failures and provide support for the developers to handle issues pertaining to partial specification of visual contracts. We also provide a mechanism to handle differences in how web services report the success or failure.

We have proposed an approach to model-based coverage which is based on a two-step process that combines static and the dynamic analysis. Statically, we use AGG's critical pair and minimal dependency analysis to create a dependency graph over rules representing visual contracts. These graphs, which distinguish different types of dependencies and conflicts, are the basis for coverage criteria. The evaluation of a set of tests based on the criteria is performed dynamically while executing the model as an oracle.

We have presented a method to reduce the size of a regression test suite based on an analysis of the dependencies and conflicts between visual contracts specifying the preconditions and effects of operations. The method is applicable to all software systems that have interfaces specified in this way, but it is particularly relevant for services because of the lack of access to implementation code and the potential cost involved in running a large number of tests through a remote and potentially payable provider. Finally, we have evaluated all of the work on test oracles, model-based coverage criteria as well as the model-based regression testing on small scale examples as well as using average sized cases studies.

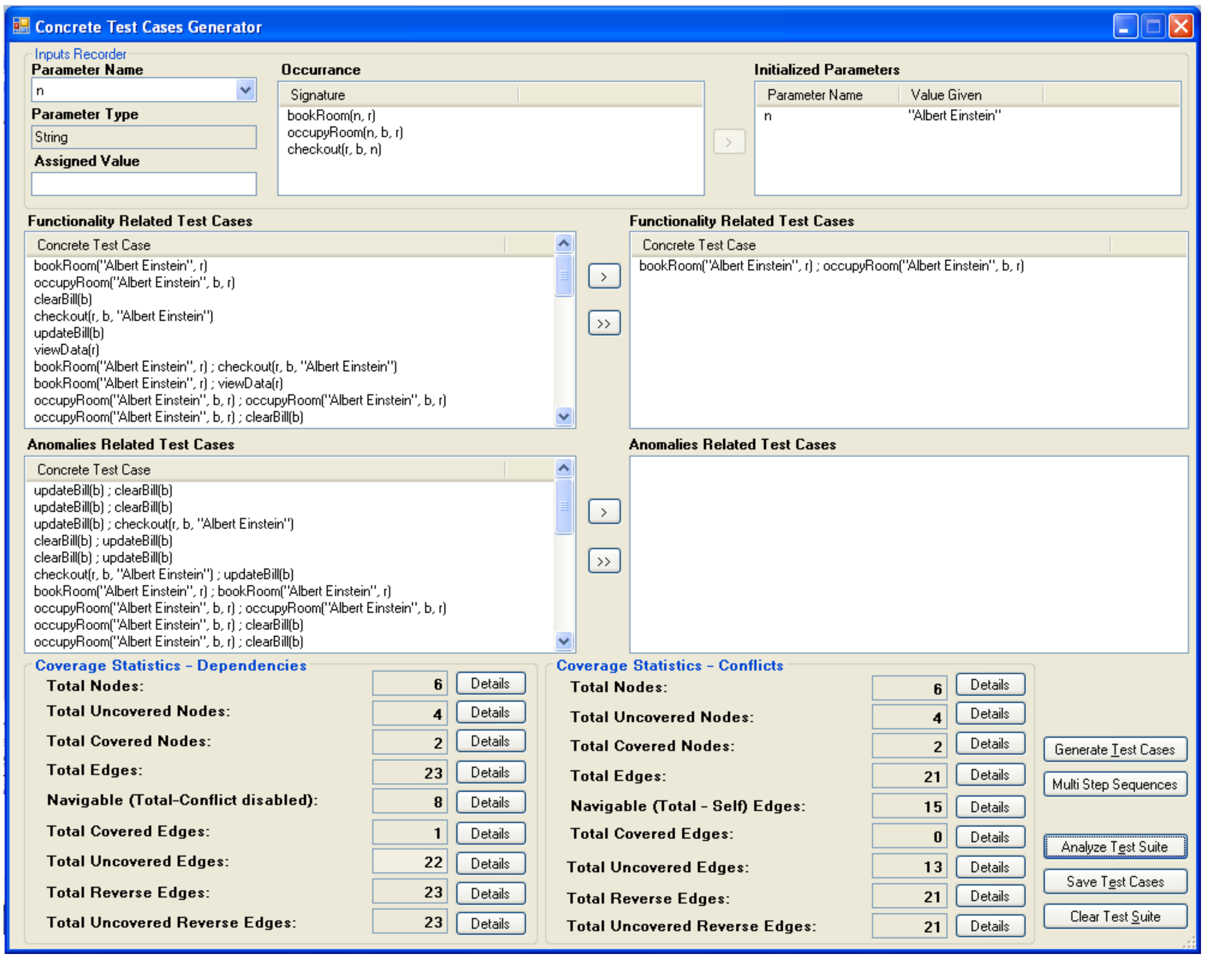
However, there are a number of future research directions that were identified during the course of this doctoral research and which are presented in the forthcoming section.

10.2 Outlook

The work reported in this thesis has a number of possible future directions. We discuss a selection, where the order is not significant.

10.2.1 Test Case Analysis and Generation

We have assumed a given set of test cases. A future research direction is to consider test case generation from visual contracts using path expressions over the dependency graph.



Concrete Test Cases Generator

Inputs Recorder

Parameter Name: n

Parameter Type: String

Assigned Value:

Occurrance

Signature
bookRoom(n, r)
occupyRoom(n, b, r)
checkout(r, b, n)

Initialized Parameters

Parameter Name	Value Given
n	"Albert Einstein"

Functionality Related Test Cases

Concrete Test Case
bookRoom("Albert Einstein", r)
occupyRoom("Albert Einstein", b, r)
clearBill(b)
checkout(r, b, "Albert Einstein")
updateBill(b)
viewData(r)
bookRoom("Albert Einstein", r) ; checkout(r, b, "Albert Einstein")
bookRoom("Albert Einstein", r) ; viewData(r)
occupyRoom("Albert Einstein", b, r) ; occupyRoom("Albert Einstein", b, r)
occupyRoom("Albert Einstein", b, r) ; clearBill(b)

Functionality Related Test Cases

Concrete Test Case
bookRoom("Albert Einstein", r) ; occupyRoom("Albert Einstein", b, r)

Anomalies Related Test Cases

Concrete Test Case
updateBill(b) ; clearBill(b)
updateBill(b) ; clearBill(b)
updateBill(b) ; checkout(r, b, "Albert Einstein")
clearBill(b) ; updateBill(b)
clearBill(b) ; updateBill(b)
checkout(r, b, "Albert Einstein") ; updateBill(b)
bookRoom("Albert Einstein", r) ; bookRoom("Albert Einstein", r)
occupyRoom("Albert Einstein", b, r) ; occupyRoom("Albert Einstein", b, r)
occupyRoom("Albert Einstein", b, r) ; clearBill(b)
occupyRoom("Albert Einstein", b, r) ; clearBill(b)

Anomalies Related Test Cases

Coverage Statistics - Dependencies

Total Nodes:	6	Details
Total Uncovered Nodes:	4	Details
Total Covered Nodes:	2	Details
Total Edges:	23	Details
Navigable (Total-Conflict disabled):	8	Details
Total Covered Edges:	1	Details
Total Uncovered Edges:	22	Details
Total Reverse Edges:	23	Details
Total Uncovered Reverse Edges:	23	Details

Coverage Statistics - Conflicts

Total Nodes:	6	Details
Total Uncovered Nodes:	4	Details
Total Covered Nodes:	2	Details
Total Edges:	21	Details
Navigable (Total - Self) Edges:	15	Details
Total Covered Edges:	0	Details
Total Uncovered Edges:	13	Details
Total Reverse Edges:	21	Details
Total Uncovered Reverse Edges:	21	Details

Figure 10.1: Test Case Generation Example

We would allow test cases to be added to the test suite incrementally where we analyze the resulting coverage for the newly added test case as well as the benefit of the addition. The process should go on until coverage with respect to selected model-based coverage criteria is achieved. The automation should also support developers in finding suitable and minimal sets of test cases that provide them with full dependency graph coverage. In case they intend to add test cases manually, there should also be a process for manual addition. Automation should also support visual contract analysis for input space partitioning [55] for test suite generation.

We have made initial progress on a tool shown in Figure 10.1, that proposes test cases for addition analyzing which additional conflicts and dependencies this new test case exercises.

10.2.2 Learning Visual Contracts

The research proposed in this thesis assumes that the visual contracts are available. We can imagine several methods to arrive at visual contracts. Apart from extracting the contracts from available semantic information, we could make use of learning techniques considering observation sequences from which the pre- and post-conditions forming visual contracts are learnt.

10.2.3 Rules Signature for Multi Objects

Currently, our theoretical contribution addresses TAGTS with rule signatures returning basic data types only. The research goal is to extend the theory so that rule signatures could be associated with rule schemes as well.

This would require to allow the output parameter to be of type $Set(s)$ for a sort $s \in S$ to return a response in case of a rule scheme with a multi object. The mapping function given in our definition of TAGTS with rule signatures would assign to each rule name $p \in P$ a list of formal input and output parameters $\sigma(p) = \bar{x} = (q_1 x_1 : c_1 s_1, \ldots, q_n x_n : c_n s_n)$ where $q_i \in \{\epsilon, out\}$, $c_i \in \{\epsilon, Set\}$ and $x_i \in X_{s_i}$ for $1 \leq i \leq n$.

This would also change the way we handle labels and we would need to extend our definition of observation from transformation sequences, accordingly introducing rules with multi objects and their applications.

10.2.4 Application Conditions

We plan to extend our theoretical formulation to include negative application conditions (NAC). This would allow us to specify visual contracts in a more expressive manner. Consider rule *occupyRoom* in our case study presented in Figure 2.2, addition of NAC as shown in Figure 10.2 prohibits more than one nodes of type *BillData* to be associated with one room booking.

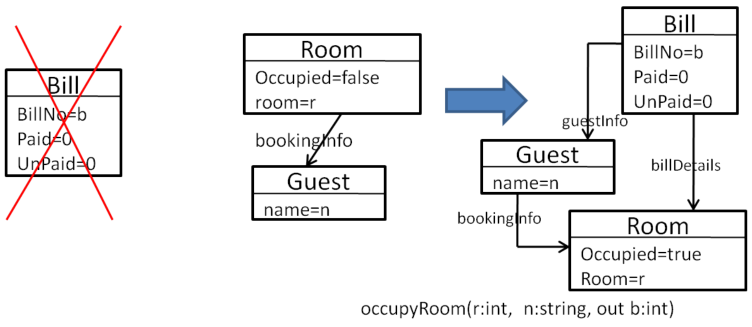


Figure 10.2: Test Case Generation Example

This would require us to introduce more edges in the dependency graph to cover additional relations between rule applications where one rule produces a graph element which is forbidden by an application application condition.

10.2.5 Evaluation of Effectiveness of Proposal

We intend to evaluate the approach with developers in order to assess the benefit of a model-based approach, where oracle and test coverage are provided, against a more informal documentation of the service interface where no such help is given. Seeding faults in the service implementation, this would allow us to assess the added value of the model-based approach.

Appendix A

Visual Contracts: Bug Tracker Service

In this appendix, we provide a list of visual contracts for our case study of the Bug Tracker service in this appendix. In case of rule schemes, we also show how the visual contract is given by a kernel rule and multi rules. We show this in a square box where the upper part represents the kernel rule and the lower represents the multi rule, as shown, e.g., in A.5, below the visual contract.

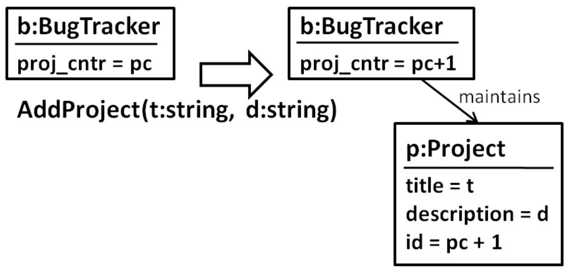


Figure A.1: Visual Contract for Rule AddProject

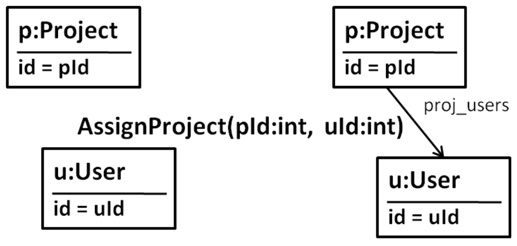


Figure A.2: Visual Contract for Rule AssignProject

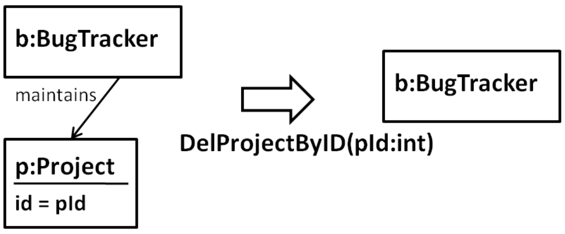


Figure A.3: Visual Contract for Rule DeleteProjectById

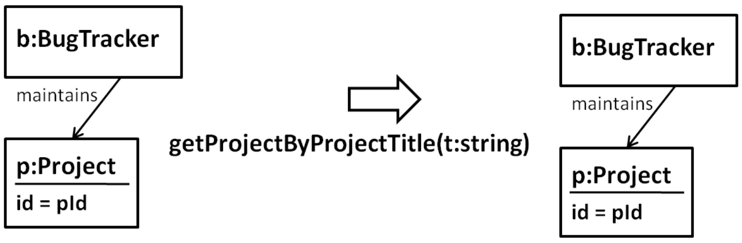


Figure A.4: Visual Contract for Rule GetProjectByProjectTitle

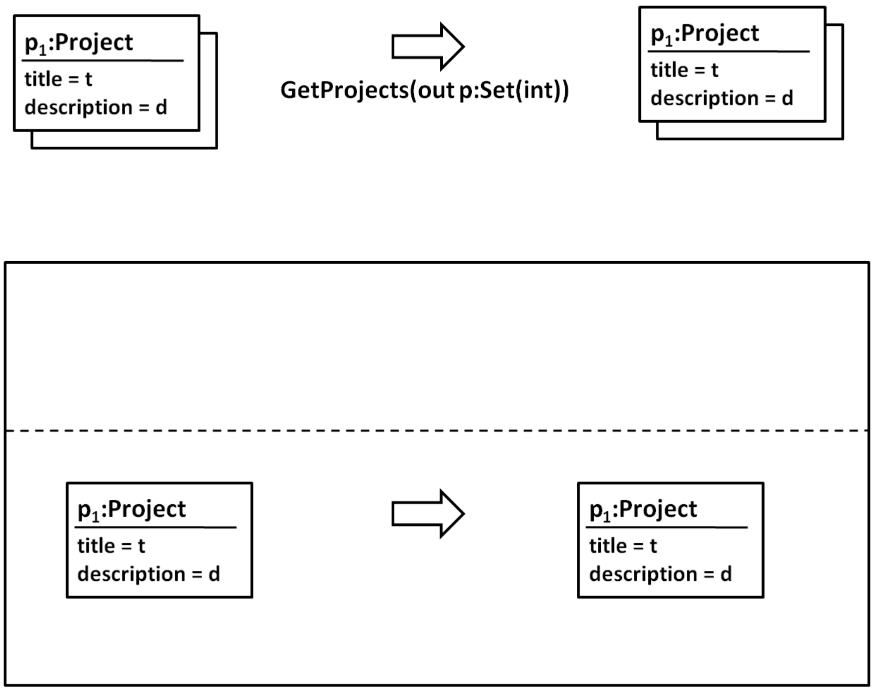


Figure A.5: Visual Contract for RuleScheme GetProjects

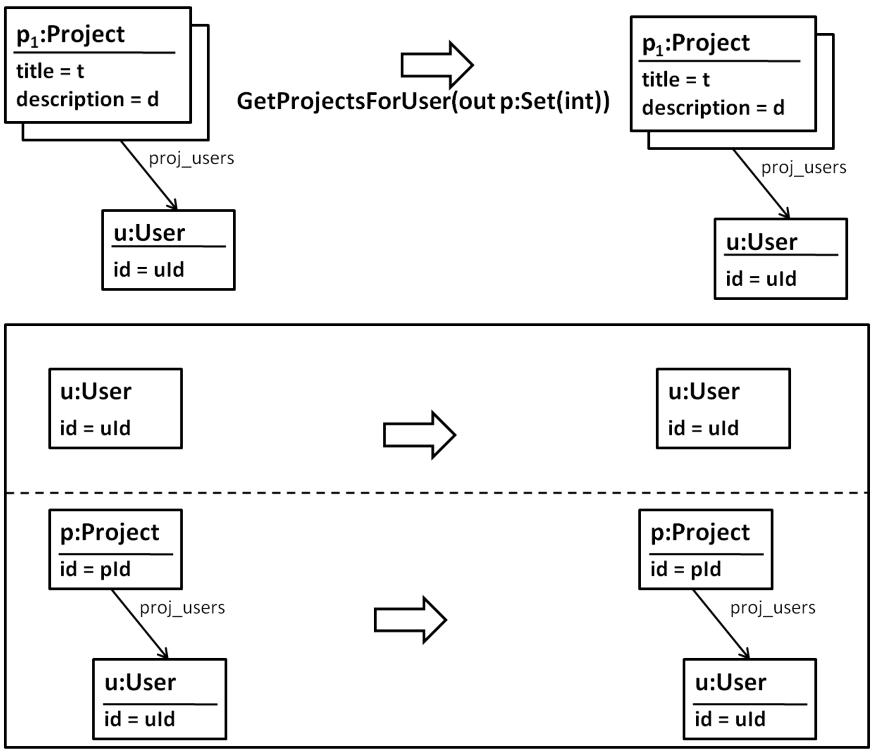


Figure A.6: Visual Contract for GetProjectsForUsers

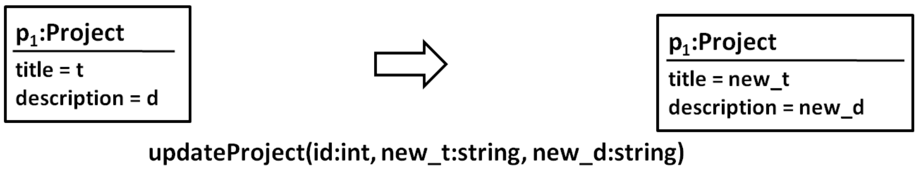


Figure A.7: Visual Contract for Rule UpdateProject

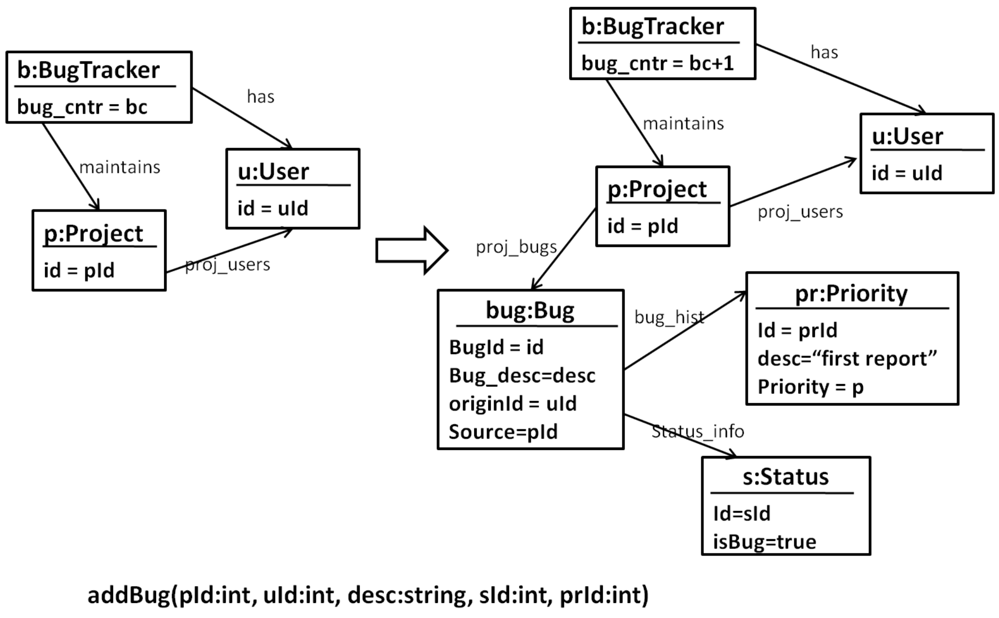


Figure A.8: Visual Contract for Rule AddBug

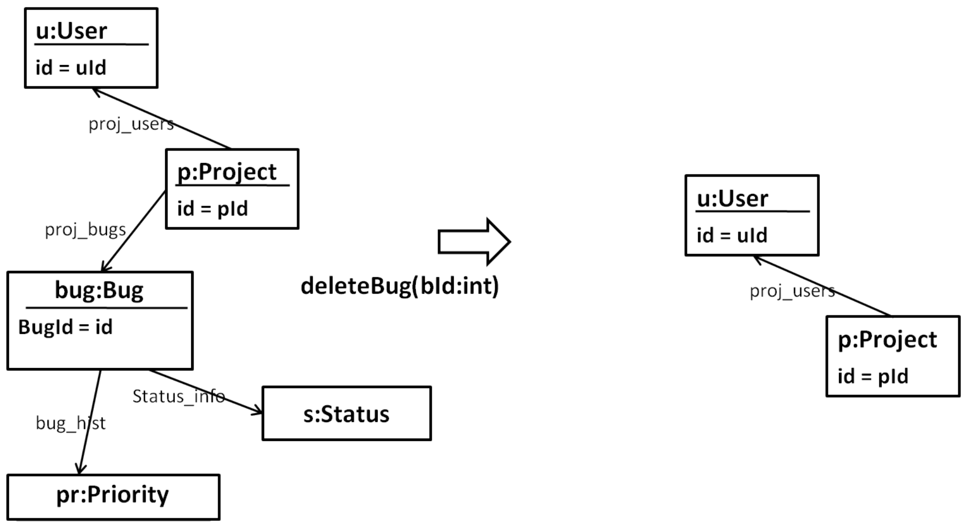


Figure A.9: Visual Contract for Rule DeleteBug

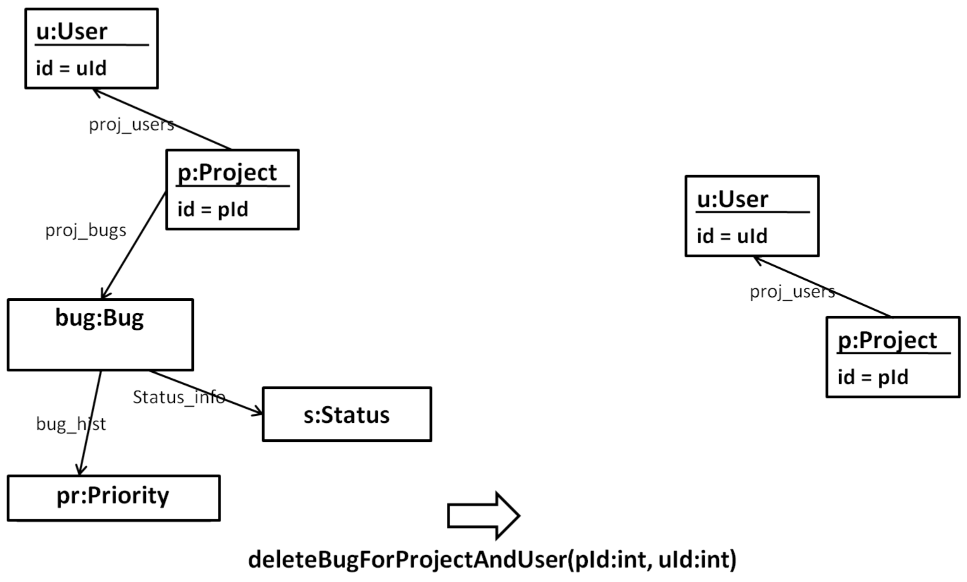


Figure A.10: Visual Contract for Rule DeleteBugsForProjectAndUsers

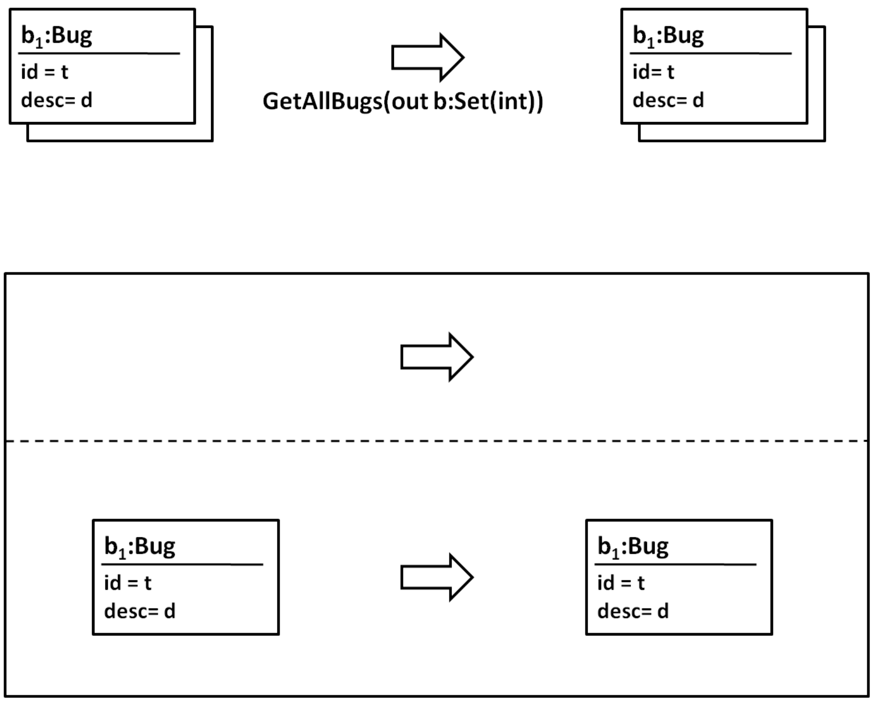


Figure A.11: Visual Contract for RuleScheme GetAllBugs

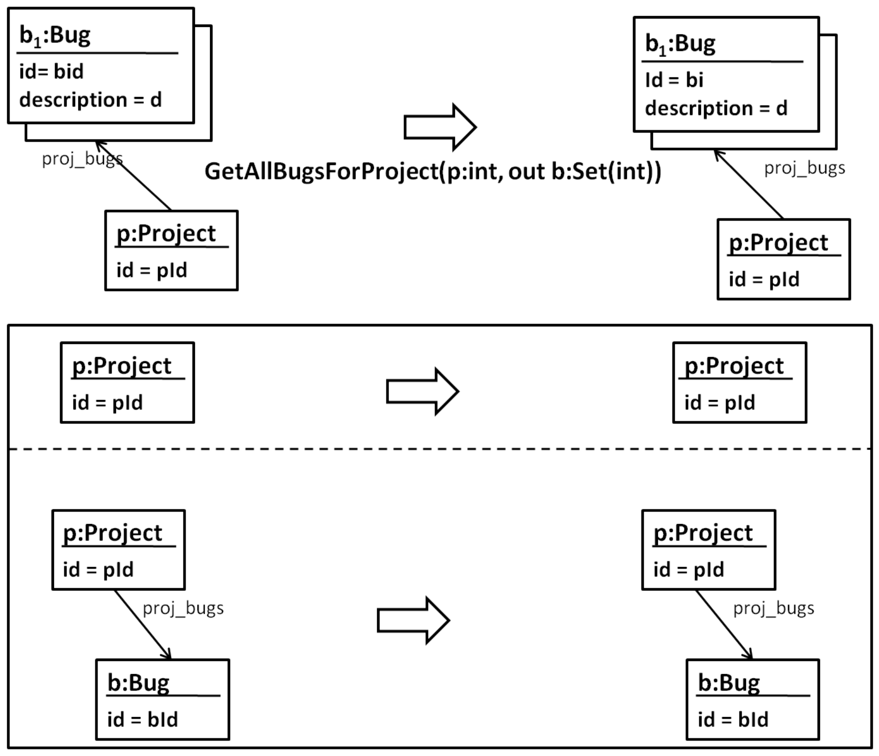


Figure A.12: Visual Contract for RuleScheme GetAllBugsForProject

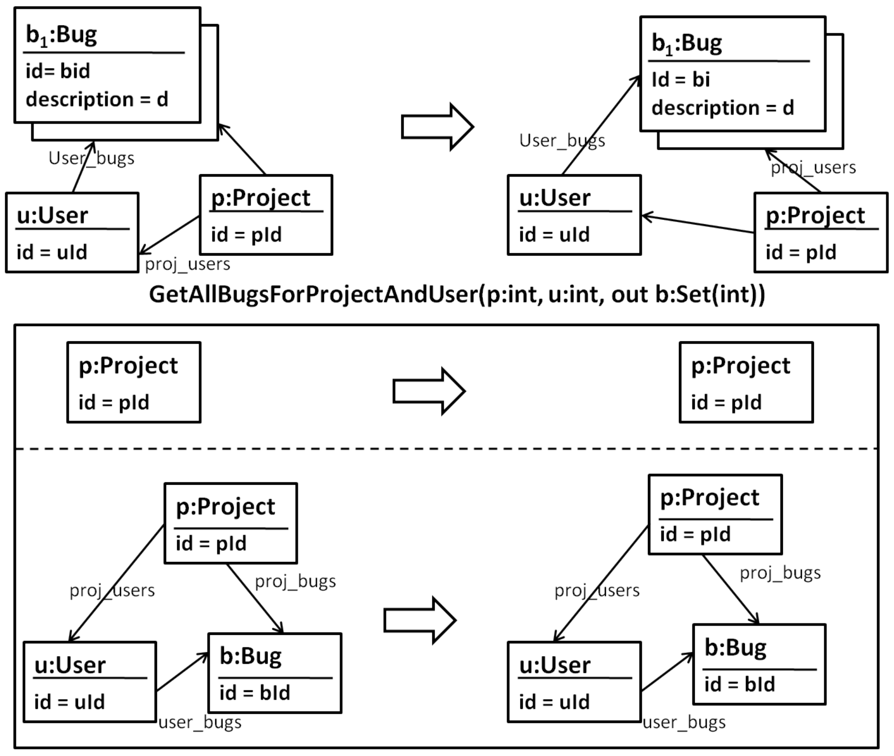


Figure A.13: Visual Contract for RuleScheme GetAllBugsForProjectAndUser

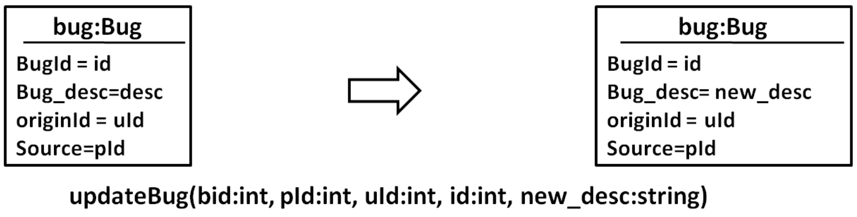


Figure A.14: Visual Contract for Rule UpdateBug

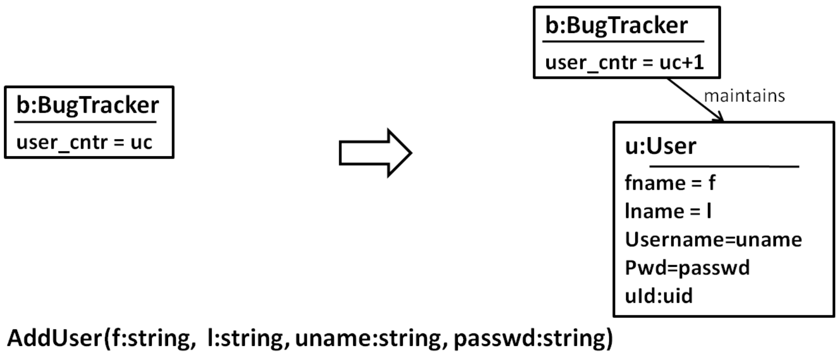


Figure A.15: Visual Contract for AddUser

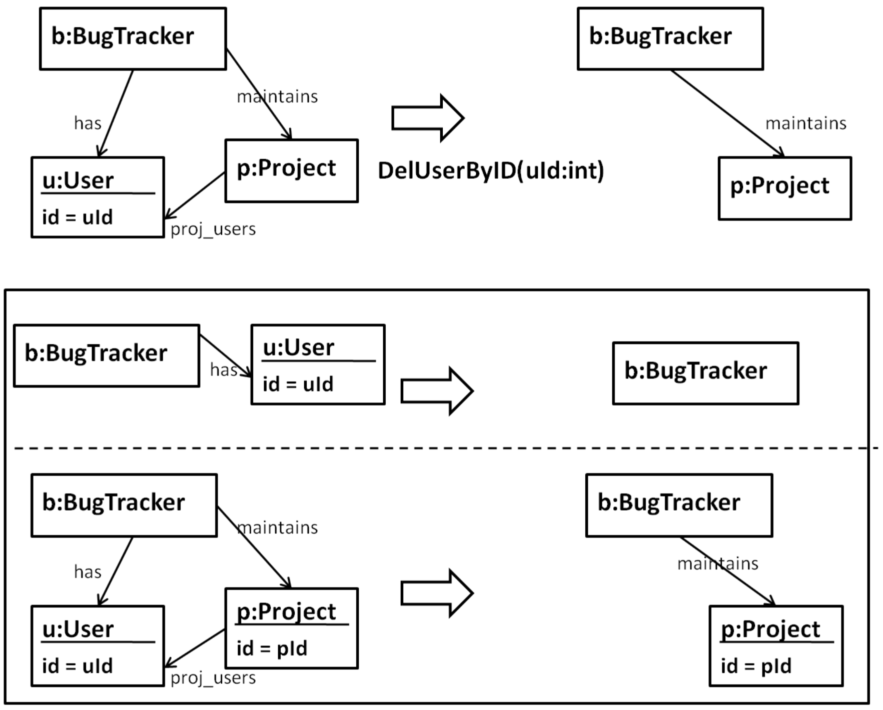


Figure A.16: Visual Contract for RuleScheme DeleteUserById

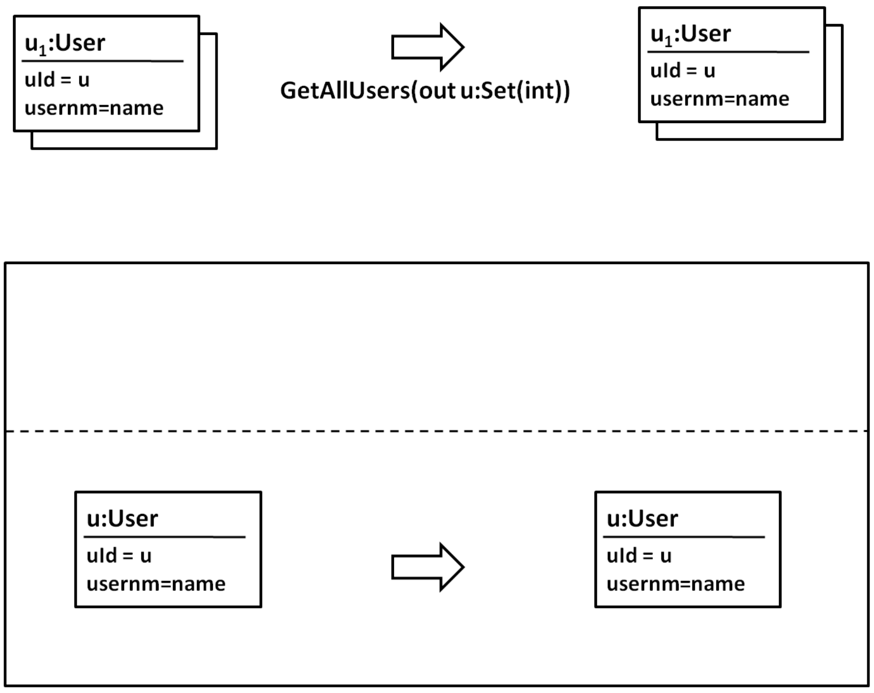


Figure A.17: Visual Contract for RuleScheme GetUsers

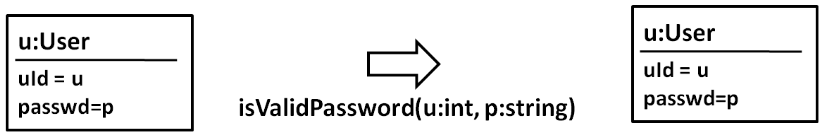


Figure A.18: Visual Contract for Rule IsValidPassword

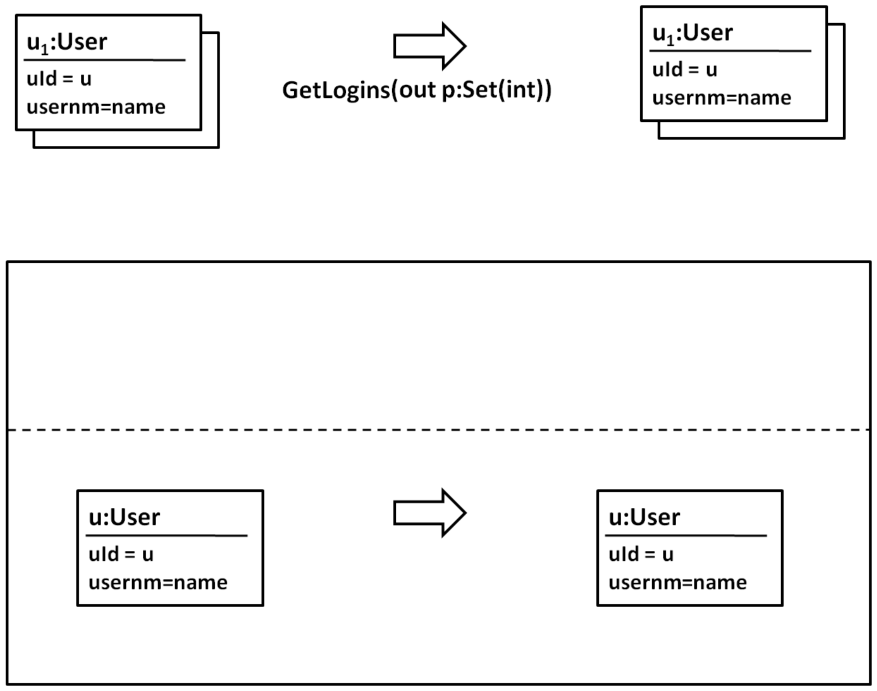


Figure A.19: Visual Contract for Rule GetLogins

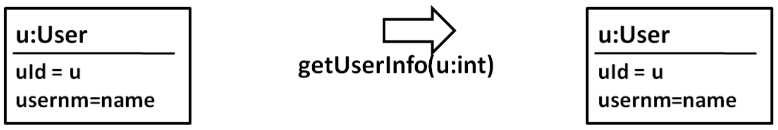


Figure A.20: Visual Contract for Rule GetUserInfo

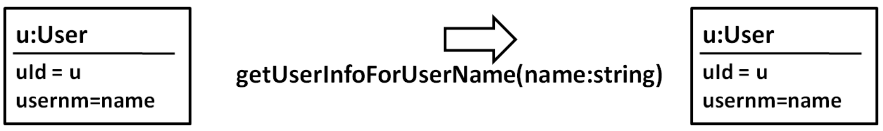


Figure A.21: Visual Contract for Rule GetUserInfoForUserName

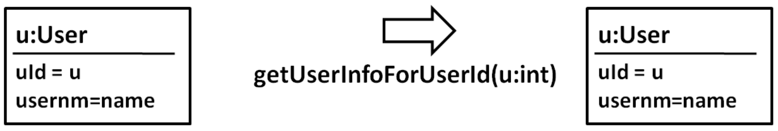


Figure A.22: Visual Contract for Rule GetUserInfoForUserId

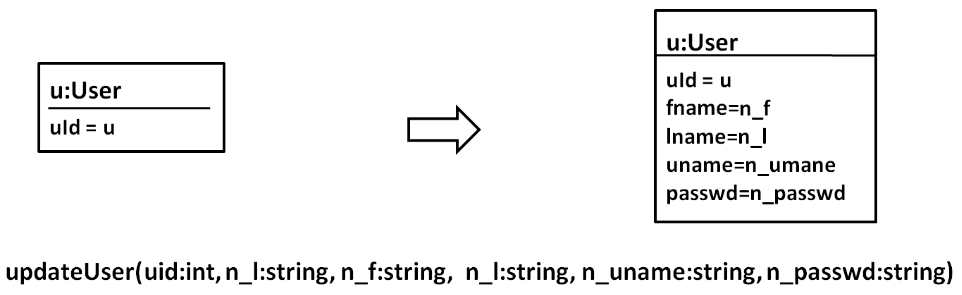

u:User
uId = u
u:User
uId = u
fname=n_f
lname=n_l
uname=n_umane
passwd=n_passwd
updateUser(uid:int, n_l:string, n_f:string, n_l:string, n_uname:string, n_passwd:string)

Figure A.23: Visual Contract for Rule UpdateUser

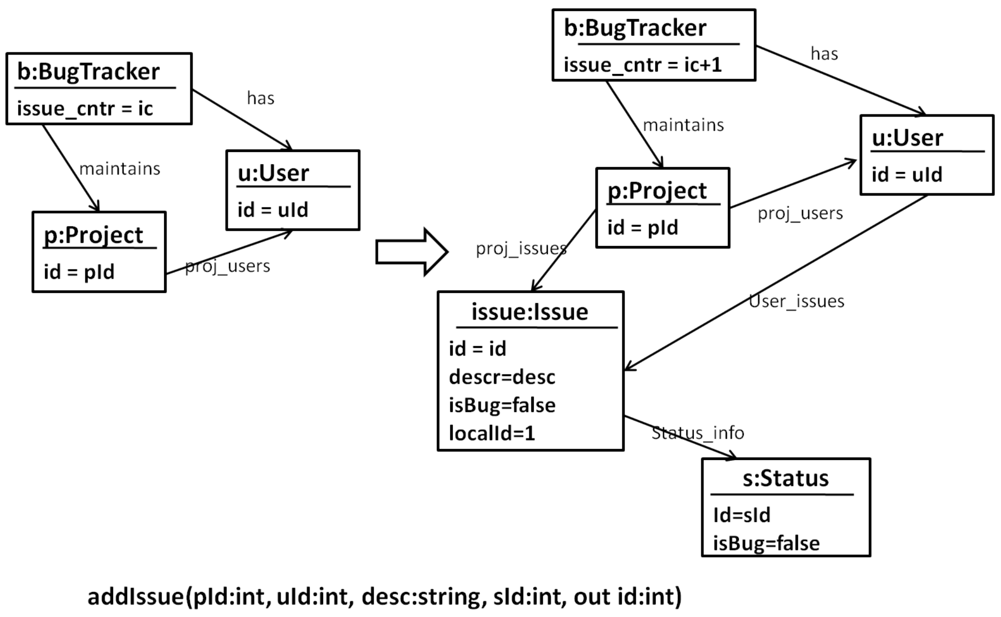


Figure A.24: Visual Contract for Rule AddIssue

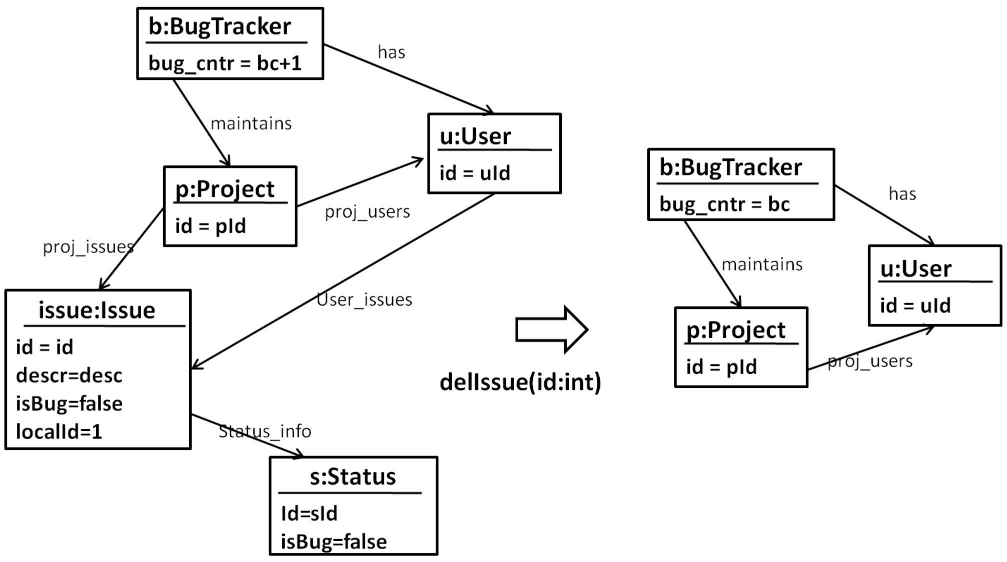


Figure A.25: Visual Contract for Rule DeleteIssues

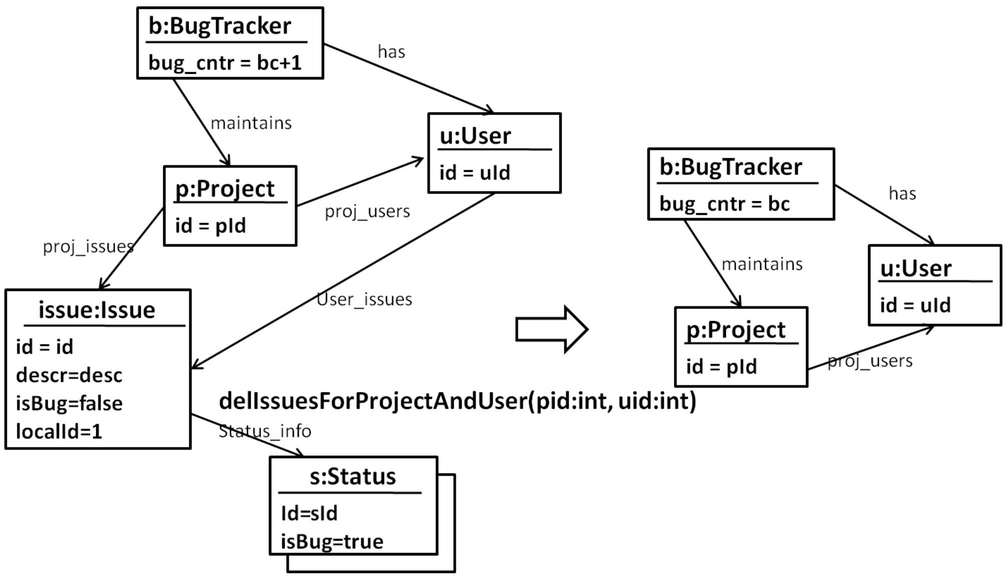


Figure A.26: Visual Contract for Rule DeleteIssuesForProjectAndUser

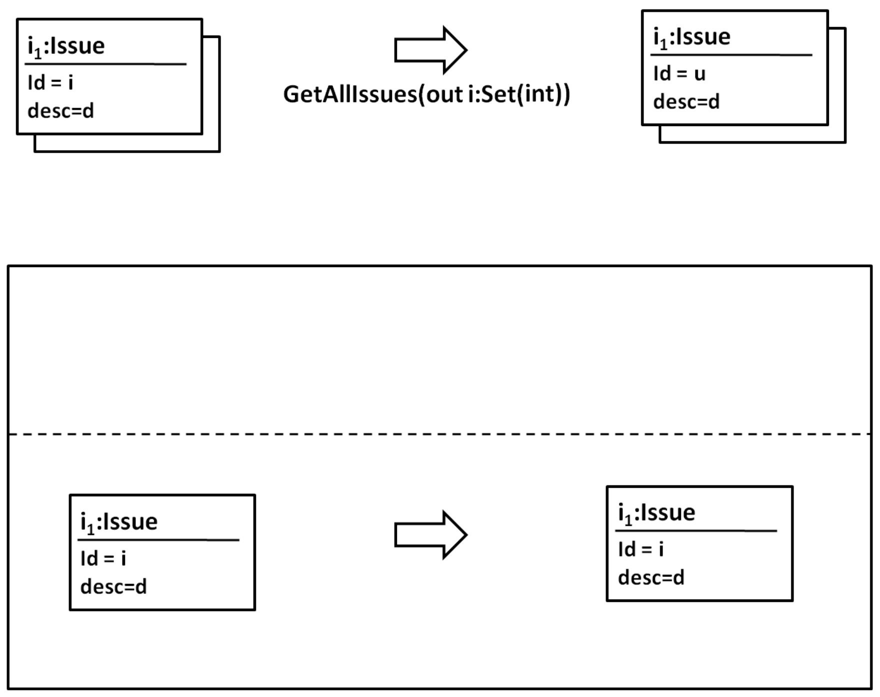


Figure A.27: Visual Contract for RuleScheme GetAllIssues

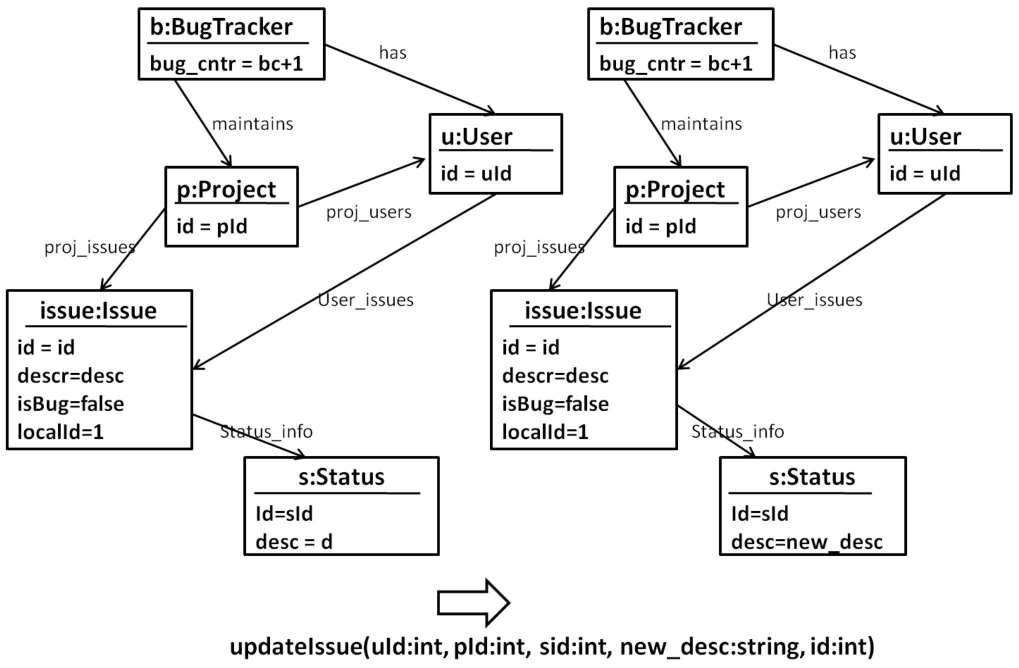


Figure A.28: Visual Contract for Rule UpdateIssue

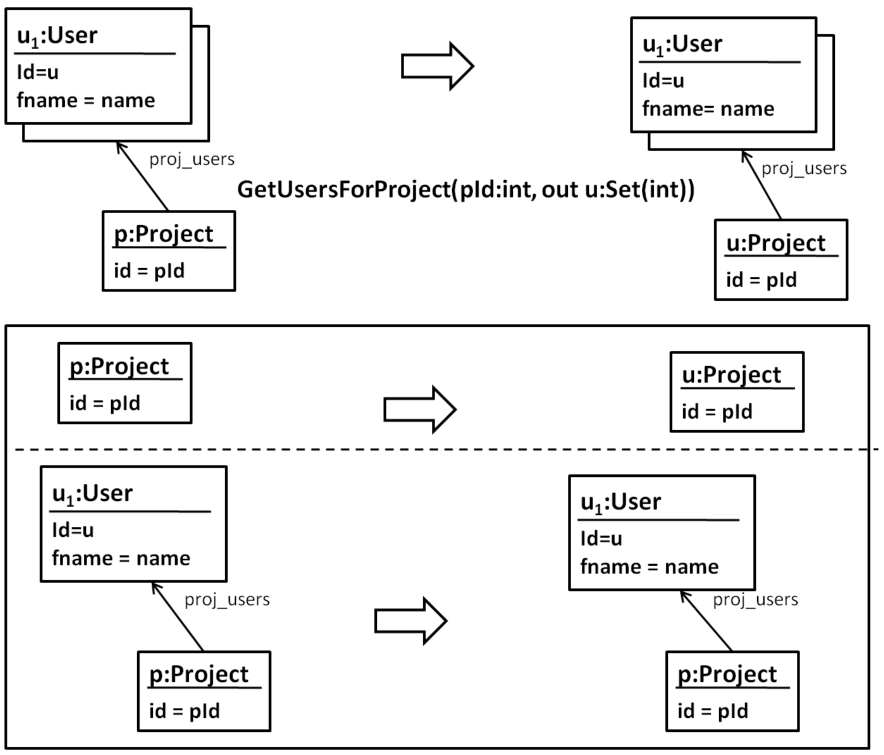


Figure A.29: Visual Contract for RuleScheme GetUsersForProject

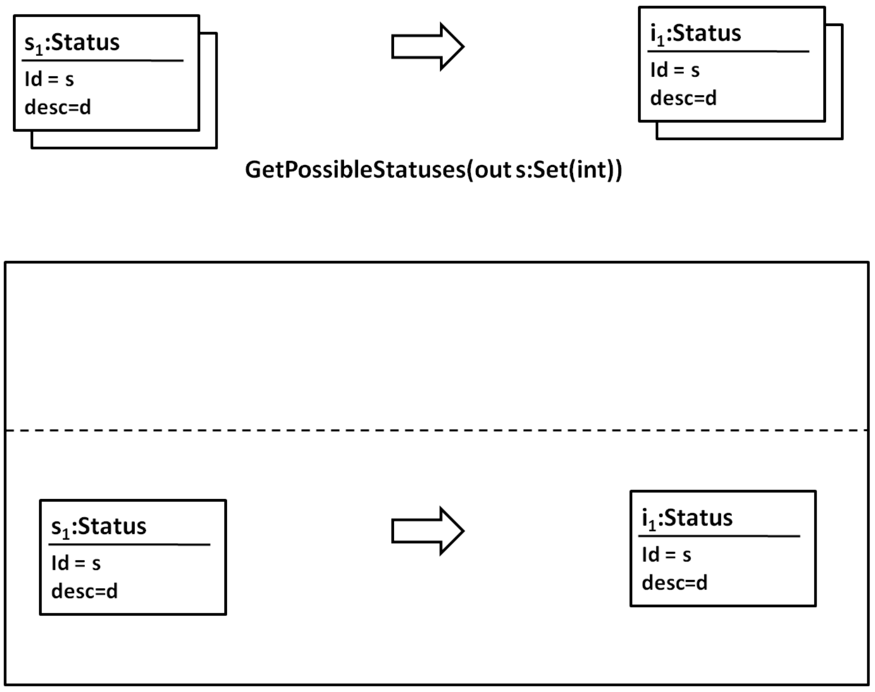


Figure A.30: Visual Contract for RuleScheme GetPossibleStatusses

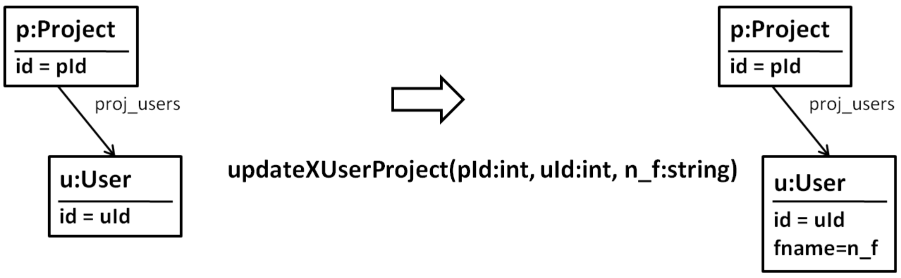


Figure A.31: Visual Contract for Rule UpdateXUserProject

References

[1] IEEE Standard Glossary of Software Engineering Terminology. Technical report, 1990.

[2] AGG - Attributed Graph Grammar System Environment. `http://tfs.cs.tu-berlin.de/agg`, 2007.

[3] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[4] C. Atkinson, D. Brenner, G. Falcone, and M. Juhasz. Specifying high-assurance services. *Computer*, 41(8):64–71, Aug. 2008.

[5] L. Baresi and R. Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*, volume 3256 of *Lecture Notes in Computer Science*, pages 402–429. Springer, 2004.

[6] L. Baresi and M. Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001. `http://www.cs.uoregon.edu/~michal/pubs/oracles.html`.

[7] C. Bartolini, A. Bertolino, E. Marchetti, and I. Parissis. Data flow-based validation of web services compositions: Perspectives and examples. In R. Lemos, F. Giandomenico, C. Gacek, H. Muccini, and M. Vieira, editors, *Architecting Dependable Systems V*, LNCS, pages 298–325. Springer-Verlag, Berlin, Heidelberg, 2008.

[8] B. Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[9] F. Belli, N. Güler, and M. Linschulte. Are longer test sequences always better? - a reliability theoretical analysis. In *Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on*, pages 78 –85. IEEE, June 2010.

[10] E. Biermann, H. Ehrig, C. Ermel, U. Golas, and G. Taentzer. Parallel independence of amalgamated graph transformations applied to model transformation. In G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel, editors, *Graph transformations and model-driven engineering*, LNCS, pages 121–140. Springer-Verlag, Berlin, Heidelberg, 2010.

[11] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools (The Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, October 1999.

[12] M. Bozkurt, M. Harman, and Y. Hassoun. Testing & verification in service-oriented architecture: A survey. *Software Testing, Verification and Reliability (STVR)*, To Appear.

[13] L. Briand, Y. Labiche, and Q. Lin. Improving the coverage criteria of UML state machines using data flow analysis. *Software Testing, Validation, and Reliability (Wiley)*, 20(3), 2010.

[14] L. C. Briand, Y. Labiche, and S. He. Automating regression test selection based on UML designs. *Information & Software Technology*, 51(1):16–30, 2009.

[15] G. Canfora and M. D. Penta. Testing services and service-centric systems: Challenges and opportunities. *IT Professional*, 8(2):10–17, 2006.

[16] J. F. Cem Kaner and H. Q. Nguyen. *Testing computer software*. Second edition, Van Nostrand Reinhold, October 1993.

[17] W. K. Chan, S. C. Cheung, and K. R. P. H. Leung. Towards a metamorphic testing methodology for service-oriented software applications. In *Proceedings of the Fifth International Conference on Quality Software*, QSIC '05, pages 470–476, Washington, DC, USA, 2005. IEEE Computer Society.

[18] Y. Chen, S. Liu, and F. Nagoya. An approach to integration testing based on data flow specifications. In Z. Liu and K. Araki, editors, *Theoretical Aspects of Computing - ICTAC 2004*, volume 3407 of *Lecture Notes in Computer Science*, pages 235–249. Springer Berlin / Heidelberg, 2005.

[19] Y. Chen, R. L. Probert, and H. Ural. Model-based regression test suite generation using dependence analysis. In *A-MOST '07: Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 54–62, New York, NY, USA, 2007. ACM.

[20] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A Formal Evaluation of Data Flow Path Selection Criteria. *IEEE Trans. Softw. Eng.*, 15(11):1318–1332, 1989.

[21] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-Based Testing in Practice. In *International Conference on Software Engineering*, pages 285–294, 1999.

[22] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, 2006.

[23] G. Engels and R. Heckel. Graph transformation as a conceptual and formal framework for system modeling and model evolution. In *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, volume 1853 of *LNCS*, pages 127–150, London, UK, 2000. Springer-Verlag.

[24] G. Engels, M. Lohmann, S. Sauer, and R. Heckel. Model-driven monitoring: An application of graph transformation for design by contract. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, volume 4178 of *LNCS*, pages 336–350. Springer, 2006.

[25] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[26] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.*, 14(10):1483–1498, Oct. 1988.

[27] U. Golas, E. Biermann, H. Ehrig, and C. Ermel. A Visual Interpreter Semantics for Statecharts Based on Amalgamated Graph Transformation. *In Proceedings of Int. Workshop on Graph Computation Models (GCM'10), Electronic Communications of the EASST*, 39, 2011.

[28] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, 2001.

[29] B. Güldali, M. Mlynarski, A. Wübbeke, and G. Engels. Model-based system testing using visual contracts. In *Proceedings of Euromicro SEAA Conference 2009, Special Session on "Model Driven Engineering"*, pages 121–124, Washington, DC, USA, 2009. IEEE Computer Society.

[30] R. Gupta, M. Jean, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In *In Proceedings of the Conference on Software Maintenance*, pages 299–308. IEEE Computer Society Press, 1992.

[31] J. H. Hausmann, R. Heckel, and M. Lohmann. Model-based development of web services descriptions enabling a precise matching concept. *Int. J. Web Service Res.*, 2(2):67–84, 2005.

[32] R. Heckel, T. A. Khan, and R. Machado. Towards test coverage criteria for visual contracts. *In Proceedings of Graph Transformation and Visual Modeling Techniques, GTVMT 11, Electronic Communications of the EASST*, 41, 2011.

[33] R. Heckel and M. Lohmann. Towards contract-based testing of web services. *In Proc. of the TACoS 2004, Electronics Notes Theory Computer Sciences in Electronic Notes in Theoretical Computer Science*, 116:145–156, 2005.

[34] K. Hölscher, P. Ziemann, and M. Gogolla. On translating uml models into graph transformation systems. *Journal of Visual Languages and Computing*, 17(1):78 – 105, 2006.

[35] D. Hoffman. A taxonomy for test oracles. *Quality Week 1998*, page 8, 1998.

[36] J. Hou, B. Xu, L. Xu, D. Wang, and J. Xu. A testing method for web services composition based on data-flow. *Wuhan University Journal of Natural Sciences*, 13:455–460, 2008.

[37] W. Howden. Reliability of the path analysis testing strategy. *Software Engineering, IEEE Transactions on*, SE-2(3):208 – 215, September 1976.

[38] S. Jurack, L. Lambers, K. Mehner, and G. Taentzer. Sufficient criteria for consistent behavior modeling with refined activity diagrams. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *Lecture Notes in Computer Science*, pages 341–355, Berlin, Heidelberg, 2008. Springer.

[39] G. M. Kapfhammer. *The Computer Science and Engineering Handbook*, chapter Chapter 105: Software Testing. CRC Press, Boca Raton, FL, second edition, 2004.

[40] T. A. Khan and R. Heckel. A methodology for model-based regression testing of web services. In *Testing: Academic and Industrial Conference - Practice and Research Techniques, 2009, TAIC PART '09*, pages 123 –124. IEEE, September 2009.

[41] T. A. Khan and R. Heckel. On model-based regression testing of web-services using dependency analysis of visual contracts. In *Fundamental Approaches to Software Engineering - 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6603 of *Lecture Notes in Computer Science*, pages 341–355. Springer, 2011.

[42] T. A. Khan, O. Runge, and R. Heckel. Testing against visual contracts: Model-based coverage. In *Proceedings of 6th International Conference on Graph Transformation, ICGT 12*, LNCS. Springer, 2012. To Appear.

[43] T. A. Khan, O. Runge, and R. Heckel. Visual contracts as test oracle in AGG 2.0. *In Proceedings of Graph Transformation and Visual Modeling Techniques, GTVMT 12, Electronic Communications of the EASST*, 47, 2012.

[44] E. Kit and S. Finzi. *Software testing in the real world: improving the process*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[45] B. Korel, L. H. Tahat, and B. Vaysburg. Model based regression test reduction using dependence analysis. In *18th International Conference on Software Maintenance (ICSM 2002), Maintaining Distributed Heterogeneous Systems, 3-6 October 2002, Montreal, Quebec, Canada*. IEEE Computer Society, 2002.

[46] M. Lallali, F. Zaidi, and A. Cavalli. Timed modeling of web services composition for automatic testing. In *Signal-Image Technologies and Internet-Based System, 2007. SITIS '07. Third International IEEE Conference on*, pages 417 –426, dec. 2007.

[47] L. Lambers, H. Ehrig, and F. Orejas. Conflict detection for graph transformation with negative application conditions. In *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, volume 4178 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2006.

[48] H. Leung and L. White. Insights into regression testing. In *Software Maintenance, 1989., Proceedings., Conference on*, pages 60 –69. IEEE, October 1989.

[49] L. Li, W. Chou, and W. Guo. Control flow analysis and coverage driven testing for web services. In *Web Services, 2008. ICWS '08. IEEE International Conference on*, pages 473 –480. IEEE, 2008.

[50] M. Lohmann, L. Mariani, and R. Heckel. A model-driven approach to discovery, testing and monitoring of web services. *Test and Analysis of Web Services*, pages 173–204, 2007.

[51] M. Lohmann, S. Sauer, and G. Engels. Executable visual contracts. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 63–70, Washington, DC, USA, 2005. IEEE Computer Society.

[52] L. Mei, W. Chan, T. Tse, and F.-C. Kuo. An empirical study of the use of Frankl-Weyuker data flow testing criteria to test bpel web services. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 1, pages 81 –88, July 2009.

[53] B. Meyer. Applying "design by contract". *IEEE COMPUTER*, 25:40–51, 1992.

[54] E. Najumudheen, R. Mall, and D. Samanata. A dependence representation for coverage testing of object-oriented programs. *Journal of Object Technology*, 9(4):1–23, July 2010.

[55] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, 1988.

[56] A. Pasquini and E. D. Agostino. Fault seeding for software reliability model validation. *Control Engineering Practice*, 3(7):993 – 999, 1995.

[57] M. Penta, M. Bruno, G. Esposito, V. Mazza, and G. Canfora. Web services regression testing. In L. Baresi and E. D. Nitto, editors, *Test and Analysis of Web Services*, pages 205–234. Springer, 2007.

[58] D. Peters, S. Member, I. David, L. Parnas, and S. Member. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24:161–173, 1998.

[59] S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[60] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, 1985.

[61] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th international conference on Software engineering*, ICSE '92, pages 105–118, New York, NY, USA, 1992. ACM.

[62] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22, 1996.

[63] O. Runge, C. Ermel, and G. Taentzer. AGG 2.0 – new features for specifying and analyzing algebraic graph transformations. In *Applications of Graph Transformation with Industrial Relevance, 4th International Symposium, (AGTIVE'11), Proceedings*, volume 7233 of *LNCS*. Springer, 2012.

[64] M. Ruth, S. Oh, A. Loup, B. Horton, O. Gallet, M. Mata, and S. Tu. Towards automatic regression test selection for web services. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 729–736, Washington, DC, USA, 2007. IEEE Computer Society.

[65] A. Sinha and A. Paradkar. Model-based functional conformance testing of web services operating on persistent data. In *Proceedings of the 2006 workshop on*

Testing, analysis, and verification of web services and applications, TAV-WEB '06, pages 17–22, New York, NY, USA, 2006. ACM.

[66] I. Sommerville. *Software Engineering: (Update) (8th Edition) (International Computer Science Series)*. Addison Wesley, 8 edition, June 2006.

[67] B. Tsai, S. Stobart, and N. Parrington. Employing data flow testing on object-oriented classes. *Software, IEE Proceedings*, 148(2):56 –64, Apr. 2001.

[68] W.-T. Tsai, Y. Chen, and R. Paul. Specification-based verification and validation of web services and service-oriented operating systems. In *WORDS05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 139–147, Washington, DC, USA, 2005. IEEE Computer Society.

[69] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 2011.

[70] Q. Xie and A. M. Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Trans. Softw. Eng. Methodol.*, 16(1), Feb. 2007.

[71] H. Zhu. A note on test oracles and semantics of algebraic specifications. In *Quality Software, 2003. Proceedings. Third International Conference on*, pages 91 – 98, November 2003.

[72] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29:366–427, December 1997.