Model-Driven Testing in Umple

Sultan Eid A. Almaghthawi

A thesis submitted in

partial fulfillment of the requirements for the degree

Ph.D in Computer Science

Ottawa-Carleton Institute for Computer Science

School of Electrical Engineering and Computer Science

University of Ottawa

Ottawa, Ontario, Canada

April 2020

© Sultan Eid A. Almaghthawi, Ottawa, Canada, 2020

Acknowledgement

Firstly, I would like to express my sincere gratitude to my supervisor Prof. Timothy Lethbridge for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Thomas R. Dean, Prof. Daniel Amyot, Prof. jean-Pierre Corriveau, and Prof. Stéphane Somé, for their insightful comments and encouragement, but also for the hard question which incented me to widen my research from various perspectives.

Thanks to my country who had provided all the support needed and has always looked after its students. Also, thank you Taibah University for giving me the opportunity to pursue my study and providing the support needed especially professor Mosaed Alsobhe for his great insight and support during my early years as an academic.

I also would like to thank everyone in the The Complexity Reduction in Software Engineering (CRUISE). All the discussion, brainstorming and code sprints we had over the years made this work possible.

Last but not least, I would like to thank my family: Thank you and much love my wife Lamia; your endless support during the endeavor is forever appreciated. Thanks to my parents Fann and Eid for everything, may my father soul rest in peace, words cannot describe how gratefully I am for everything they have provided. Thanks to my brothers and sisters for supporting me spiritually throughout writing this thesis and my life in general.

Abstract

In this thesis we present a language and technique to facilitate model-based testing. The core of our approach is an xUnit-like language that allows tests to refer to model entities such as associations. This language can be used by developers to describe tests based on an existing UML model. The tests might even be written before creating a UML model, and be based on requirements. The testing language, including its parser and generators, is written entirely in Umple, an open-source textual modeling tool with semantics closely based on UML, and which generates Java, PHP and several other target languages. Tests in our language can be embedded in Umple or in standalone files. The test language compiler converts our abstract testing language into JUnit, PHPUnit and other domain-language testing environments. In addition to allowing developers to write tests manually, we have created generators that create abstract tests for any Umple model. These generators can be used to verify the Umple compiler and to give Umple users extra confidence in their models. User-defined tests can be standalone or embedded in methods; they can be generic, referring to metamodel elements. Tests can also be located in traits or mixsets to allow testing of separate concerns or product lines. To test our language and the tests written in it, we have created an extensive test suite. We have also implemented mutation testing, that enables varying of features of the models to ensure that runs of the pre-mutation tests then fail.

Table of Contents

Chapter 1 Introduction 1
1.1 Thesis Topic and Contributions 1
1.2 Problem Statement 3
1.3 Motivation and Benefits 3
1.4 Methodology: Design Science 5
1.5 Research Questions 7
1.5.1 RQ1: What would be an effective syntax and semantics for an abstract model-level test language for testing UML class models? 7
1.5.2 RQ2: How can we evaluate the quality of the test specifications and the automatically generated tests? 8
1.6 Thesis Outline 8
Chapter 2 Background 9
2.1 Software Testing 9
2.2 Model-Based Testing 10
2.2.1 What is Model-Based Testing? 10
2.2.2 Why Model-Based Testing? 11
2.2.3 Where are the Weaknesses of Model-Based Testing? 11
2.2.4 The Difference Between Model-Based and Model-Driven 13
2.2.5 Is Our Approach Offline-Testing or Online-Testing? 14
2.3 Overview of Umple 14
2.3.1 Umple Features Relevant to This Thesis: Mixins, Traits and Mixsets 17
2.4 Model-Based Testing Tools and Technologies 17
2.4.1 Conformiq Designer/Creator, Previously Known as Leirios 18
2.4.2 MBTestSuite 18
2.4.3 MoMuT::UML 19

2.4.4 RT-Tester and RTT-MBT 20
2.4.5 TTCN-3 and UTP 20
2.4.6 ETSI Test Description Language 22
2.4.7 Spec Explorer / PyModel 22
2.4.8 ACL Contracts 22
2.4.9 MaTeLo 23
2.4.10 ModelJUnit / GraphWalker 23
2.5 Other Techniques That Are Related 24
2.5.1 Aspect-Oriented Programming 24
2.5.2 Generative Programming 25
2.6 Summary 25
Chapter 3 Basics of Model-Based Testing in Umple 27
3.1 Umple's Model-Based Testing Architecture 29
3.2 Model Transformations in Our Approach 30
3.3 The Umple Test Model Parser 32
3.4 Testing of Attributes 37
3.4.1 Testing Lazy Attributes 39
3.4.2 Testing Immutable Attributes 40
3.5 Testing Associations 41
3.5.1 Optional-One to Many Associations 44
3.5.2 One to Many Associations 46
3.5.3 N to Many Associations Where N Is a Fixed Number > 1 48
3.5.4 Directional Associations 50
3.5.5 Lower-Bounded and Upper-Bounded Associations: N..N -- * 52
3.5.6 Immutable Unidirectional Associations 54

3.5.7 Sorted Association in Umple 56
3.5.8 Chained Associations 57
3.5.9 Reflexive Associations 59
3.6 Interfaces 59
3.7 Abstract Classes 60
3.8 Inheritance 61
3.9 State Machines 61
3.10 Test Data 62
3.11 Coverage Criteria: 62
3.12 Summary 65
Chapter 4 Umple Testing Language 66
4.1 An Overview of the Umple Testing Entities 66
4.1.1 Test Model 68
4.2 The Test Model/File Generation Process 72
4.3 Grammar 76
4.3.1 Some Challenges in the Writing of the Grammar: 78
4.3.2 Integrating the Test Language Syntax into Umple 79
4.4 Translating of Abstract Test Code to Different Unit Test Systems 80
4.5 Summary 81
Chapter 5 User-Defined Model-Based Tests 82
5.1 Basics of User-Defined Tests 82
5.1.1 Why Support Both Mixin Test Code and Model-Integrated Test Code? 83
5.1.2 Related Work 84
5.2 Handling User-Defined Methods in Umple Tests 85
5.2.1 Handling User-Defined Testcases 86
5.2.2 Handling Testcase Order of Statements 87

5.3 Embedding Tests for Models With State Machines ... 89
5.4 Support for Flow Control in the Test Language ... 96
5.5 Test Sequence Control in Umple Test Models ... 97
5.6 Summary ... 97
Chapter 6 Test-Driven Modelling Using Umple ... 99
6.1 Test Driven Modeling Using Umple as a Modelling Language ... 99
6.2 Challenges in Test-driven Modeling ... 106
6.3 Summary ... 107
Chapter 7 Advanced Issues in Model-driven Testing In Umple ... 108
7.1 Testing with Traits ... 108
7.2 Test Generation for the Mixset Feature ... 110
7.3 Test Inheritance and Polymorphism ... 111
7.3.1 Inheriting Tests Defined in a Parent Class ... 111
7.3.2 Implementing Tests in Interfaces ... 115
7.4 Platform-Specific Tests ... 118
7.5 Before/After Keywords in Testing Elements ... 119
7.6 Test Backward Traceability ... 121
7.7 Summary ... 121
Chapter 8 Generic Model Tests ... 123
8.1 Generic Tests for Attributes ... 124
8.2 Generic Tests for Methods ... 124
8.3 Generic Tests for Associations ... 126
8.4 Discussion ... 128
8.5 Summary ... 129
Chapter 9 Dependency Handling for Instantiation of Test Objects ... 130
9.1 Object Model Instantiation ... 130

9.1.1 Problem 1: Dealing with Dependency Level ... 132
9.1.2 Problem 2: Starting Point and the Possibility of Cycles ... 132
9.1.3 Observation: Multiplicity Lower Bound of One Is a Key Constraint ... 132
9.1.4 Solution: Traversal Algorithm ... 133
9.2 Dependency Injection and Object Instantiation for the Airline Example ... 135
9.3 Summary ... 136
Chapter 10 Implementation of the Approach ... 137
10.1 MBTParser: Umple Abstract Test Parser ... 138
10.2 Generators ... 139
10.3 Summary ... 144
Chapter 11 Evaluation and Testing of the Approach Itself ... 145
11.1 Evaluation Objectives ... 145
11.1.1 Evaluation Objective 1: Output of TestGenerator ... 145
11.1.2 Evaluation Objective 2: The Output of TestModelParser ... 146
11.1.3 Evaluation Objective 3: The Output Coming From the xUnitGenerator ... 146
11.2 The Infrastructure for Testing in Umple ... 147
11.3 Test-Driven Development of Umple MBT ... 148
11.4 Umple Mutation Testing ... 152
11.4.1 Statement of the Evaluation Problem ... 154
11.4.2 Umple Mutation Test Tool ... 161
11.4.3 Case Study: Umple Java Testbed ... 167
11.4.4 Case Study: Advisor Example ... 169
11.4.5 Categories of Killed Mutants ... 171
11.5 Summary ... 172
Chapter 12 Conclusion ... 174
12.1 Summary of the Contributions ... 174

12.2 Summary of Evaluation 176
12.3 Summary of Answers to Research Questions 176
12.4 Summary Comparison of Our Work To Other Research Tools 180
12.5 Further Work 182
12.5.1 Expanding The Current Implementation 182
12.5.2 Conducting a Empirical Study of the Test Language and Technology 185
12.5.3 Mutation Enhancements 185
12.5.4 More support for Test-Driven Modelling 186
12.5.5 Graphical Representation 186
References 187

List of Figures

Figure 1: Design cycle 7
Figure 2: Thesis outline 8
Figure 3: Software testing V-Model [12] 9
Figure 4: Level of abstraction of programming languages 16
Figure 5: MbtSuite user interface [51] 19
Figure 6: MoMuT::UML test generation [53] 20
Figure 7: Umple test generation process 28
Figure 8: The Umple test generator architecture 29
Figure 9: Model transformation in Umple test generation 31
Figure **10:** Inadequate approach: directly generating executable tests without an abstract model. 31
Figure 11: Umple Test generator packages 33
Figure 12: Umple airline example diagram 35
Figure 13: Optional on- to-many association 45
Figure 14: One-to-many association 47
Figure 15: A unidirectional association 51
Figure 16: Bounded multiplicity 52
Figure 17: Class diagram for immutable directional optional one-to-many association 55
Figure 18: Chained associations 58
Figure 19: Metamodel for Umple test model and language 67
Figure 20: Student-Mentor example diagram 71
Figure 21: Test Generator output workflow 73
Figure 22: Visualizing the grammar for a single test case 78
Figure 23: Visualizing generic test in Umple 79
Figure 24: State machine for bulb 90
Figure 25: State machine for garage door 91
Figure 26: Garage door test 93
Figure 27: Test sequence for state testing 93
Figure 28: State machine with a specific sequence 95

Figure 29: Workflow for test-driven modelling in Umple 99
Figure 30: First initial error from running tests 103
Figure 31: compiling the model after adding class Student 103
Figure 32: Requirements tests after passing initialization errors 104
Figure 33: Incrementally passing test cases 104
Figure 34: Passing *isRegister* test case 104
Figure 35: the JUnit code for the test-driven modelling 105
Figure 36: Improved test-driven modeling workflow 106
Figure 37: Class hierarchy showing tests 112
Figure 38: Example inheritance model illustrating testing 112
Figure 39: A model with an interface 116
Figure 40: Result from model analysis of unimplemented tests 118
Figure 41: Umple Method parameters matching 125
Figure 42: A model with a number of associations 131
Figure 43: Umple optional association 132
Figure 44: Model Traversing and Dependency Check 135
Figure 45: Test generation in Umple 137
Figure 46: Workflow for creating a unit generator using Umple 138
Figure 47: Unit test generator with Php as an example 140
Figure 48: Evaluation of Umple outputs regarding tests 148
Figure 49: Test-driven development of a single feature 149
Figure 50: TDD in Umple Compiler 150
Figure 51: Tests for unit test generators 150
Figure 52: TDD Result 150
Figure 53: Generating mutants based on the metamodel instance 155
Figure 54: Second approach to process the Umple model for mutation 156
Figure 55: Telephone example in Umple 157
Figure 56: Model scattered mutations 161
Figure 57: Model mutation using double-random 163
Figure 58: Generated mutants for the generated tests for a regular 164
Figure 59: Generated mutants for generated tests for defaulted attribute in Umple. 164

Figure 60: Model mutation testing 165
Figure 61: Advisor example model 169
Figure 62: Result of unit tests 169
Figure 63: Failure caught by trying a fault input 170

List of Tables

Table 1: Comparison between different MBT tools – part 1 23
Table 2: Comparison between different MBT tools – part 2 24
Table 3: Generated test files for the airline example 37
Table 4: generated API for class RegularFlight in the airline model 42
Table 5: API methods added for sorting 56
Table 6: Test model elements 69
Table 7: Test runner templates and utility 74
Table 8: Model with separate test code 91
Table 9: GarageDoor State Tables Generated Using UmpleOnline 92
Table 10:Test cases 100
Table 11: Interface tests 117
Table 12: Umple Generic Association Testing 127
Table 13: Umple metamodel metrics 129
Table 14: Command-line parameters for unit-test generator 139
Table 15: Unit templates 146
Table 16: Umple Mutation Test Tool Parameters 162
Table 17: List of mutation operators in Umple 165
Table 18: Fault injection in the model file: TestHarness.ump 168
Table 19: Result of mutation tests for Advisor example 171

List of Code Snippets

Code Snippet 1: Umple code example 16
Code Snippet 2: Conformiq QML example [32] 18
Code Snippet 3: TTCN-3 example [55] 21
Code Snippet 4: Example of ACL code [59] 23
Code Snippet 5: Airline example textual Umple code 36
Code Snippet 6: Test model code example 39
Code Snippet 7: Executable test code in JUnit 39
Code Snippet 8: Lazy attribute in Umple 39
Code Snippet 9: Generated abstract model for lazy attribute 39
Code Snippet 10: Immutable attribute 40
Code Snippet 11: Generated test for immutable attribute 40
Code Snippet 12: Concrete code in JUnit used to find setter method for immutable attributes 40
Code Snippet 13: Umple template code for detecting attribute patterns 41
Code Snippet 14: An example of abstract test code for an association 43
Code Snippet 15: example of optional one to many 45
Code Snippet 16: Generated abstract test model for an optional one-to-many association........ 46
Code Snippet 17: Umple model for one-to-many association 47
Code Snippet 18: Generated abstract test model for one-to-many association 48
Code Snippet 19: Generated abstract test model for an N -- * association........ 50
Code Snippet 20: Directional association reference 51
Code Snippet 21: Umple Model for lower and upper bounded multiplicity 52
Code Snippet 22: Generated abstract test model for bounded multiplicity association 54
Code Snippet 23: Assert method using reflection........ 55
Code Snippet 24: Umple code showing an immutable class and an immutable association.. 55
Code Snippet 25: Sorted association in Umple........ 56
Code Snippet 26: Method assertions for sorted association 56
Code Snippet 27: Checking sorted association order 57
Code Snippet 28: Umple Model for testing chained association 58

Code Snippet 29: Reflexive association for class Course 59
Code Snippet 30: Abstract Umple class 60
Code Snippet 31: Generated abstract test class 60
Code Snippet 32: generating tests based on a given coverage criteria. 63
Code Snippet 33: Handling template inclusion based on coverage criteria 65
Code Snippet 34: Umple Student-Mentor example 71
Code Snippet 35: Test model vode, Student-Mentor example 71
Code Snippet 36: Test runner template 75
Code Snippet 37: Current language grammar 77
Code Snippet 38: Using a mixin to target different platforms 84
Code Snippet 39: Added assertion to user-defined method 85
Code Snippet 40: Generated abstract testcase for user-define method 86
Code Snippet 41: Umple model with test case 87
Code Snippet 42: Generated Abstract test model with user-defined test case 87
Code Snippet 43: Test case as defined in Umple model 88
Code Snippet 44: JUnit test case of testNO20 without preserved statement location 88
Code Snippet 45: JUnit test case of NO20 when location is preserved 89
Code Snippet 46: Merging test with model 90
Code Snippet 47: Garage door state machine 92
Code Snippet 48: Test model skeleton for a test embedded directly in the model 94
Code Snippet 49: Test embedded in the model 94
Code Snippet 50: Test file with three attempts to press buttonOrObstacle 96
Code Snippet 51: Flow control within a test case 96
Code Snippet 52: Test sequence 97
Code Snippet 53: Test model based on requirements 101
Code Snippet 54: Test-driven Umple model 101
Code Snippet 55: Test-driven modelling 102
Code Snippet 56: Adding methods not in Umple API to support testing 103
Code Snippet 57: JUnit code for isUnderAge 105
Code Snippet 58: Umple model with trait 108
Code Snippet 59: Generated tests for class using trait 109

Code Snippet 60: Trait with testcase 110
Code Snippet 61: Basic Mixsets 110
Code Snippet 62: Umple 2D Shape Model 113
Code Snippet 63: Check orientation test 113
Code Snippet 64: Test code injected in subclasses 114
Code Snippet 65: Overriding testcases 114
Code Snippet 66: Merging new test code in a subclass 115
Code Snippet 67: Umple code with an interface 117
Code Snippet 68: Umple Interface with tests 117
Code Snippet 69: Before test method 119
Code Snippet 70: After test method 119
Code Snippet 71: Before and after assertions in model 120
Code Snippet 72: Generated before/after assertions 120
Code Snippet 73: Source model tagging of generated code blocks 121
Code Snippet 74: Attribute generic test in Umple 124
Code Snippet 75: Umple Method Generic Testing 126
Code Snippet 76: Constructors generated by Umple 131
Code Snippet 77: Constructors for directed associations 133
Code Snippet 78: Airline dependency handling 136
Code Snippet 79: Class initiation in Umple test template and dependency injection 136
Code Snippet 80: An Umple abstract test model example 141
Code Snippet 81: Generated JUnit code 142
Code Snippet 82: Generated PHPUnit code 143
Code Snippet 83: Generated RubyUnit code for model 144
Code Snippet 84: Telephone system model 158
Code Snippet 85: Diff result on the model mutation 159
Code Snippet 86: Test harness model 167
Code Snippet 87: Command for the Umple mutant generator 167
Code Snippet 88: Mutant generation result for TestHarness model file 168
Code Snippet 89: AOP in Umple 183
Code Snippet 90: Result of AOP on setA 184

Code Snippet 91: AOP injection in Umple model for test 184
Code Snippet 92: Aspect-Oriented Programming for Testcase 184

Chapter 1 Introduction

In this thesis we present a new approach to model-based testing that enables developers to specify target-language-independent tests. These can be specified before embarking on modeling, enabling test-driven modeling. We give the design of a new high-level testing language whose elements refer to modeling constructs such as UML classes and associations. We discuss our implementation of this language, including generators that allow transforming it into target languages such as Junit. We also discuss our implementation of a generator for the language in Umple, a model-oriented programming technology. Finally, we show how our approach can enable model-based mutation testing.

In the remainder of this chapter we outline the problem, and our research questions.

1.1 Thesis Topic and Contributions

Testing is crucial to software development. Historically, the cost of testing dramatically increased as systems grew, with the testing cost commonly exceeding half the cost of the project [1], [2]. Much work has focused on reducing testing costs through techniques such as test automation [3], [4].

Model-based testing [5] is considered a particularly good approach to reducing the cost, time and effort of testing. It encourages that a test specification should be described at a high level of abstraction along with the system model, and created early in software development.

Umple [6] is a textual language for representing UML models with extensive features for code generation, separation of concerns, diagram drawing and embedding of native code. Umple is designed to facilitate modeling by developers who are used to code, but want to use the abstractions found in diagrammatic models. Umple models, described in Section 2.3, follow the syntactic conventions of languages such as Java and C++, for ease of use. Umple also allows embedding of traditional code directly in models.

In this thesis, we focus on model-based testing for UML [6] class diagrams. We develop our ideas in the context of Umple because Umple is textual, like our language, and because it has

extensive and robust code generation with which we can readily experiment. Our work also supports other UML diagrams, such as state diagrams, but we elected not to focus on those.

Our work enables automatic test generation for models. We also consider what to test from a UML [7] model and how such a test model should be represented as a domain-specific language (DSL) [8] for testing at the abstract level. In addition to being the target of automatically-generated tests, the DSL enables user-defined model-based tests, that can be standalone or embedded directly in methods. We explore validation of a model-based test suite by mutating the model and running tests written in the DSL against code generated from the modified model.

The key contributions of this thesis include:

1. **Automatic test generation for class diagrams**. Such tests can deepen confidence in generated code. Although the Umple compiler is already thoroughly tested, and users of Umple could simply *trust* the compiler, there is a benefit of automatic test generation for auditing generated systems and encouraging Umple adoption. Our work has been released as open source with its own (meta-)tests to validate it. Our contribution 4, below validates it further.
2. **An abstract (model-level) and platform-independent test modelling language, including an implementation of the language**. This is similar in nature to 'xUnit' testing languages such as 'jUnit', except it is for models. It is a separate language from Umple so could theoretically be used by any UML-based modeling tool, although our implementation is designed to be used with Umple. It is generated by the test generator described in contribution 1, and hence its first version is in production and is available as open source. The language can also be used by the software developers using Umple to write their own tests that refer to model entities. Such user-written tests can be written in separate files, as is common with other xUnit languages, but can also be embedded directly in methods to improve cohesion, maintainability and discovery of the relevant tests.
3. **Generators that translate the abstract testing language from contribution 2 into concrete testing languages for Java (i.e. jUnit), Ruby and PHP**.

4. **A capability for mutation testing in model-based testing**. This modifies models and then verifies the failure of both the abstract tests generated from the original model using the language from contribution 2 as well as the manually embedded tests. Various coverage aspects can be specified: For example, the mutation generator can be directed to simply change multiplicities, or to change other aspects such as association directionality.

1.2 Problem Statement

Software testing is a process that requires extensive effort. There is a high need for automation of the process, and this is where model-based testing can come into play. Model-based testing aims to automate the testing process by allowing the developers to focus on the abstract specification of the tests and leave the low-level details to a well-tailored set of generators. It should save developers considerable time and cost if they had the option to using Umple to generate test cases in target languages from both the Umple model itself and also from developers' own model-level tests that are ideally created *before* the model. It should also raise the level of quality assurance as the set of test cases generated can be based on real test patterns developed by expert developers.

There is a strong argument regarding the above made by Bertolino [9] in one of the most important papers in the software testing and the model-based testing field. It casts doubts on the way tests are typically driven in model-based testing. It argues that whereas most model-based testers derive their test specification from the design model, they should instead derive tests from requirements so that the design model, and any generated code, conforms to these requirements-driven tests. With that in mind, in model-driven development, our work should draw first steps towards what Bertolino called "the tester dream," where the design-model is driven by the test model rather than being derived from requirement directly; which is called *test-driven-modelling* [9].

1.3 Motivation and Benefits

Back in the early days of computers, people used to code in machine language. Then came the high-level languages with compilers, which enabled working at higher levels of abstraction. Programmers relatively quickly learned to trust compilers to generate the correct outputs and it became superfluous to test whether the output of the compiler was correct with respect to the input

or to write tests in machine language: Unit tests came to be written referring to compile-time entities such as functions and data structures, even though the test cases themselves would still be compiled to machine language.

The work in this thesis is no different from the perspective of abstraction: When it comes to model-based testing, the developer should be able to tackle tests at the model-level and inspect the result of the test without the need to look at test cases generated from models as target-language code (e.g. JUnit/Java).

Trust for early compilers was not instantly achieved. Indeed, even today for high-security work only certified compilers may be used. Therefore, the creation of tests that essentially double-check the compiler can be beneficial as trust is built for that compiler.

The need to generate tests for an Umple system has several benefits to the end user, even though the Umple compiler is heavily tested. Generating tests for an Umple system has the following benefits:

- The user does not necessarily trust the Umple compiler. An average user does not know Umple well enough to build that trust. So providing the tests for the generated system can develop this trust in code and build confidence for developers using Umple.
- The user might not be certain whether to continue adopting the Umple technology or not. Although not what the Umple proponents would prefer, in some cases developers using Umple might only use Umple as a *kick-starter* to generate the architecture and skeleton of the system and then abandon the technology for a variety of reasons (management dictate, personal preference). Or they might plan to continue using Umple, but need a fallback as insurance against the situation where development of the Umple compiler stops. In this case, tests of the generated code would be needed to be present along with the generated code. It would be difficult to determine which code has been generated by Umple and which has been added by the user. In any case, whether the technology is to be abandoned or to be adopted, generating tests is beneficial for the sake of raising the quality assurance of the generated code.
- Writing abstract tests allows the user to test methods that are user-defined yet language independent (e.g. there are versions for C++ and Java). Providing abstract tests for these methods helps prevent code duplication.

The advantages are not in fact limited to the end-users of Umple, the work also brings benefits to the Umple developers themselves in terms of testing the Umple compiler. First, Umple relies on its compiler developers to employ test-driven development. Compiler developers are supposed to write test cases that are expected to cover as much Umple functionality as possible. This work will help Umple compiler developer to approach Umple testing in a more systematic and meticulous way by automatically generating tests based on certain Umple test-coverage patterns. The work should also help in detecting missing test cases and enhance the code coverage of Umple itself.

1.4 Methodology: Design Science

The research method we are using in this thesis is Design Science. We tackled the design problem of improving test specification at the abstract (modeling) level with the objective of helping developers improve the quality of their software. The specific "treatment" we designed was a test-specific language that allows specifying tests at the model level. We performed the work in an iterative manner, as indicated in Figure 1.

March and Smith [10] point out that, "design science attempts to create things that serve human purposes." They also say, "Its products are assessed against criteria of value or utility – does it work? is it an improvement?". In our research we clearly show that our approach works, since there is an open source implementation of our language and related tools with many test cases. We also claim that our work is an improvement by describing a set of capabilities for which no other technology exists that is capable of doing them all in an integrated manner. These improved capabilities for the practitioner, to be considered together, are:

a) To be able to describe a set of test cases that refer to model elements (UML associations);

b) to be able to generate unit tests in multiple languages from these test cases;

c) to be able to choose to either integrate such test cases directly in a textual modeling language, or to record them in separate test files;

d) to be able to generate test cases in the language automatically from a UML model, while at the same time allowing developer-written test cases;

e) to be able to write generic test cases that can refer to metamodel elements;

f) to be able to do mutation testing with the language;

g) to be able to use the testing language and technology in an integrated manner that leverages features of Umple technology such as traits and mixins.

March and Smith further indicate that, "design science products are of four types, constructs, models, methods, and implementations." The constructs are the "concepts" that are combined to create the "models". In our research the constructs include test patterns for associations, and the models include our modeling language that has a metamodel and syntax. We also provide a method (model-based testing) for using our language, and an implementation. Our language and architecture are described in a general way, enabling others to create a different implementation of either.

March and Smith also point out, "Notably absent from this list are theories." Indeed, in our research we are not attempting to produce or demonstrate new theories.

Finally, March and Smith say, "design science consists of two basic activities, build and evaluate. These parallel the discovery-justification pair from natural science. Building is the process of constructing an artifact for a specific purpose; evaluation is the process of determining how well the artifact performs." The building aspect of this thesis is self-evident: We build a language, describe it in detail, and provide an implementation. The evaluation consists of: a) Numerous test cases for our language (produced using test-driven development) that verify it does the intended job correctly (but which could be subject to bias as we created the tests); b) Mutation testing to verify with greater independence that our test suites work, and c) demonstration through example that the technology works well enough to be used in practice (which for a new approach is sufficient in design science).

Gregor and Hevner [11] point out that in Design Science there are two "camps", the "design theory camp" and the "pragmatic-design camp", with the latter putting focus on designed artifacts, as is the case in this thesis. In their Figure 3 they describe four types of work, Routine Design, Improvement, Exaptation (extending and adapting known solutions) and Invention. All but Routine Design constitute, "Research Opportunity and Knowledge Contribution." Our work is a

knowledge contribution in their framework because, as in their framework it is in an area where the application maturity in industry is low and the maturity of our solution is medium (meaning our work can be considered partly invention and partly exaptation).

Our method is considered solution-oriented design science since we focus on the provided solution. The solution is validated in part through a fault injection mechanism (mutation) that shows the quality of the provided output.

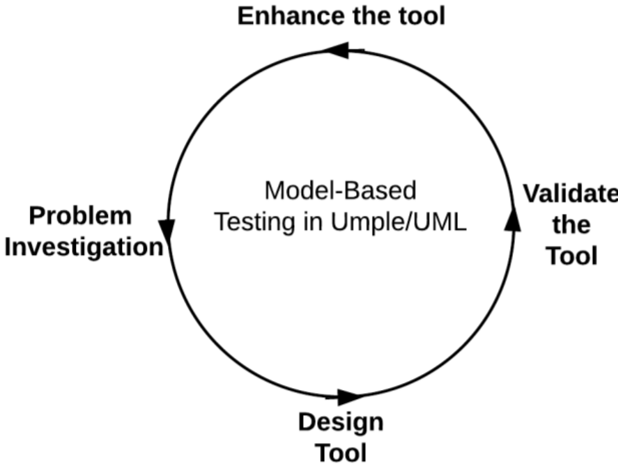

Figure 1: Design cycle

1.5 Research Questions

As we performed this work, our objectives were to answer the following research questions. In the concluding chapter we will return to these questions, and provide summaries of the answers we found.

1.5.1 RQ1: What would be an effective syntax and semantics for an abstract model-level test language for testing UML class models?

- RQ1.1: What kind of defects should be detected by this language?
- RQ1.2: How can test generation using this language best be integrated within Umple to allow testing of code automatically generated by the Umple compiler?

- RQ1.3: How can this language be designed to facilitate test-driven modelling – i.e. creation of model-level tests by developers using Umple before they actually write Umple models?
- RQ1.4: What should the testing architecture be for the use of this language?

1.5.2 RQ2: How can we evaluate the quality of the test specifications and the automatically generated tests?

- RQ3.1: What types of evaluation techniques should be applied in order to ensure the quality of the tests using the language from RQ1?

1.6 Thesis Outline

The rest of the thesis is organized according the outline shown in Figure 2.

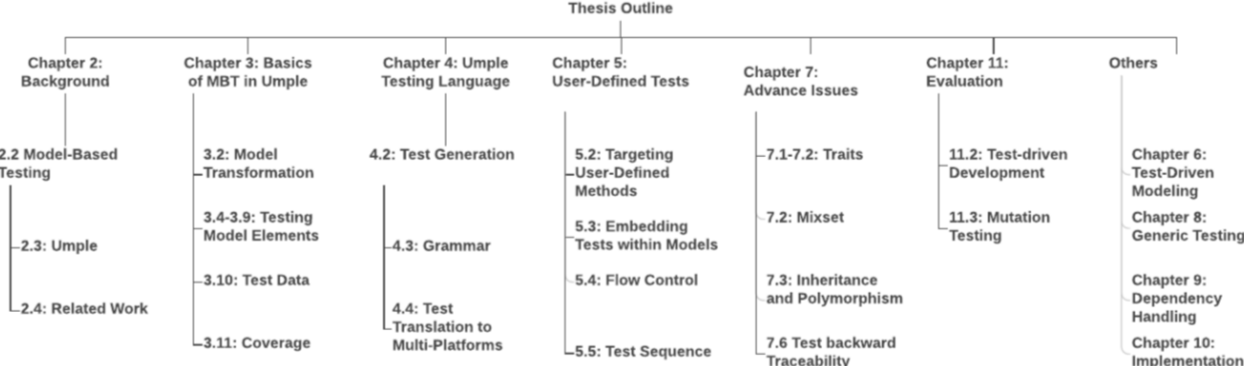

Figure 2: Thesis outline

Chapter 2 Background

In this chapter we describe certain background required for understanding the thesis, including a brief overview of testing in general, a look at model-based testing and its tools, and an overview of Umple.

2.1 Software Testing

Software testing is one the most important tasks in software engineering.

Deriving tests is a challenging process. Tests can and should be derived mainly from the software requirements, however, in some cases, tests can be derived from design. Either way, the tester must write tests targeting different levels of the system, including system tests, integration tests and unit tests [1]. Large-scale purchasers of software might also write acceptance tests.

These levels are central to Beizer's v-model [1] for software testing. Figure 1 demonstrates the relation between each system level and the corresponding test activity required to assess its correctness.

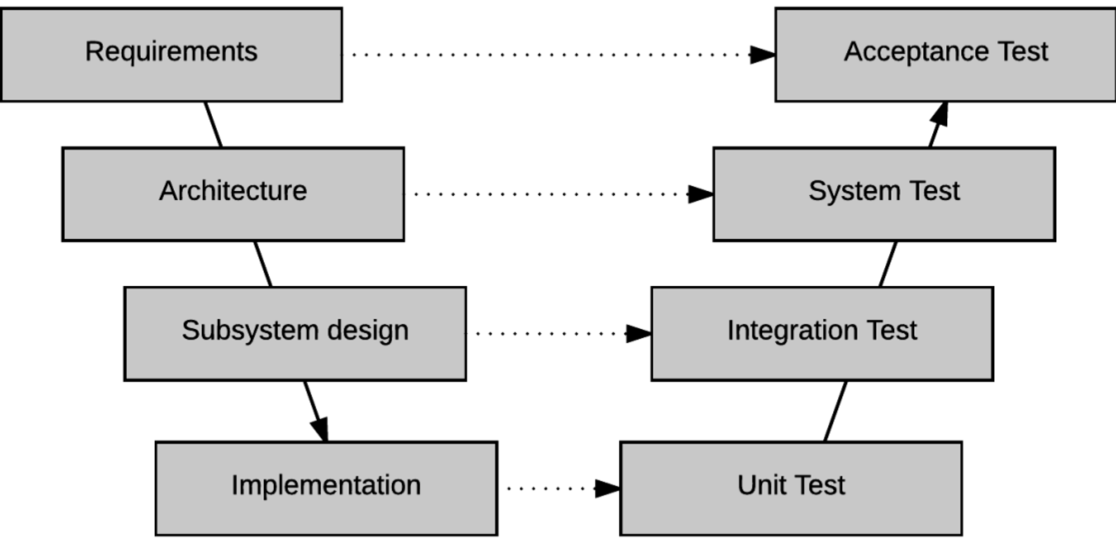

Figure 3: Software testing V-Model [12]

The software testing V-model illustrated in Figure 1 has been criticized by agile methods advocates for several reasons [2] : it is believed by such advocates to be too simplistic to reflect realistic software development behavior and process. Secondly, it is believed to have a problem

with flexibility in that it does not allow rapid and effective response to changes in requirements or in technological obstacles that might arise. On the other hand, V-model advocates believe that it can indeed deal with these problems. In this thesis, we consider the V-model as a milestone in software testing that had been an influence on software testing and a guideline to consider when starting to test a system when the requirements are clear and method used for development is well understood. The work in this thesis can be employed if the V-model is being employed, or if a more agile approach is being followed.

2.2 Model-Based Testing

Testing grows in significance as a system gets larger [13]. In the traditional software development lifecycle, the testing phase comes after the implementation is done. In such a process, cost of testing is very high since re-writing of tests and re-testing is required when flaws in requirements, design or implementation are detected. Statistically, the testing cost has been shown to often reach half the cost of the project [14]. Many studies have focused on reducing testing costs by proposing techniques such as test automation, without sacrificing the quality of the code [15] [16].

Agile methods [17], specifically test-driven development [18], have been proven to lessen the cost of testing by requiring the user to write automated tests *before* the actual implementation of code. Test-driven development ensures users end up with a good set of test cases when the implementation phase is complete and prevent regression because the tests are run every time any code is changed. This method was introduced by the American software developer Kent Beck in 2003 [19]. Through the technique, he aimed to promote simplicity and confidence [20] [21], [22].

2.2.1 What is Model-Based Testing?

This term has had an ambiguous definition since the early 2000's. Developers have been using the term differently to embed several test-generation techniques under the name. For instance, it had been utilized in the following areas [22] :

- Test input generation from domain models.
- Test case generation from environmental models.

- Generation of a test-case oracle from behavioral models.
- Generation of concrete tests from abstract test scripts.

Mark Utting listed the above four uses in his book *Practical Model-Based Testing* (MBT) in 2007 [23]. He discussed that the difference between these four lies behind how developers look at the term and what is to be generated from a model. He also reviewed the ambiguity in the term "model" which is somewhat of a buzzword and has varied in usage itself. Therefore, we can conclude that we must first agree on a definition of the term "model-based testing" and the definition of the term 'software model' before we proceed in this thesis.

Our definition is very similar to Mark Utting's fourth meaning of model-based testing. "Model-based testing is the generation of executable test cases from a software model in which the generated code must contain information that dictates what output values are considered correct. And software models are any machine-processable description of aspects of software that contain abstractions not commonly available in programming languages. Models can be graphical, textual or both".

2.2.2 Why Model-Based Testing?

Model-based testing is intended to reduce the cost of testing by reducing the time and effort involved. It encourages developers, in the early phases of software development, to create that test specifications referring to model elements [24]. Model-based testing will allow developers to achieve higher level automation, more comprehensive testing and allow for testing new changes in the system. In addition, it will allow for more flexible and comprehensive coverage of the system. However, the most important advantage of MBT is reducing testing cost and time [4], [25], [26].

2.2.3 Where are the Weaknesses of Model-Based Testing?

Although model-based testing brings strength to the system, there are several areas where current methods need improvement [27]. Some of these are:

- **Coverage specification**: Since model-based testing cannot guarantee 100% test coverage of all aspects of a given model and its generated code, good coverage criteria must be

specified to dictate what is to be tested in a model. We can only claim a percentage of test coverage when we are given a set of coverage criteria. For instance, we cannot claim to be testing a system 100%, but we can claim testing 100% of its method calls or its state machine transitions.

- **Verifying the generated tests**: Model-based testing ultimately aims to test the generated code. The process involves test generation. But these generated tests need verification themselves. One way to do this would be by performing mutation testing, which is a mechanism we discuss later in the thesis.

- **Model verification**: A thoroughly tested system generated from a bad model will be a bad system. One of the biggest problems in model-based testing is the potential infestation of defects in the model, as discussed in [28]. When a model has logical errors, these errors will be most likely undetected by the test model since the test model is derived based on that model. This issue can be solved in several ways. One way is by tools that analyze the model. One of the reasons why we adopted Umple is that it includes several model analysis capabilities, including numerous built-in analyses, as well as extraction of formal methods languages for external analysis [29], [30]. Another way to deal with the problem is to allow a mix of automatically generated model-based tests, and manually written model-based tests: Our work enables both of these.

- **Model growth leading to more-than-linear test resource requirements**. As the main model grows, the test model becomes larger. This inflation will result in scalability issues where tools can fall behind in terms of keeping up with the size of the model; specExplorer [31] is an example of this, the tool generates a large number of graphical state machines for each possible test path. It ends up in a combinatorial explosion problem where the number of generated outputs exceeds what can be realistically analyzed each time the system changes. In this thesis, the tools we have developed do not exhibit this problem [3].

- **Difficulty of use**: Many software projects recruit professionals to do the modeling [32]. This results in the number of industrial MBT tools in the market, but their usability and

interoperability can be weak. A few tools like Umple [6] are designed with the intention of making modeling easy [33] and are used in education.

- **Adaptability**: Another issue with model-based testing is the gap between generic test solutions and the system implementation [34]. Many of these tools like TTCN-3 are designed to work with different systems, therefore, these systems need to be adapted. Therefore, writing adapters is key in order to connect to TTCN-3 when testing systems [35]. In Umple, we try to fill such a gap when the technology is adapted.

In this thesis, we will be focusing on a model-based testing approach for UML class diagrams, represented using Umple. The work can readily be extended to other features of Umple such as state machines, but we have limited our work to class diagrams (associations in particular) to reduce thesis scope.

The discussion will include automatic test generation for UML models, what to test from a UML model and the representation of the test model in a domain-specific language (DSL) for testing at the abstract level.

2.2.4 The Difference Between Model-Based and Model-Driven

The concept of being model-based can in some sense be applied to any well-developed testing setup, since the developer would have had to cognitively develop a mental model of the system under test and the tests, even if not documented. Hence, it all comes down to whether the developer decides to document such models or not. When test models are documented using either a design language or test-specific language, then we can call the test system model-based testing.

On the other hand, model-*driven* testing requires more. We can say the system is model-driven if those documented test models were derived from another model in the system such as a class diagram or state machine model.

Our approach in this thesis is considered both model-based and model-driven due to the fact that we document the tests using a test-specific language and also provide a mechanism where some or all of this abstract test model can be automatically derived (generated) from a design model of the system; which is Umple/UML as a design language.

2.2.5 *Is Our Approach Offline-Testing or Online-Testing?*

In model-based testing, there are two main types of testing. Offline-testing and online-testing. They differ in the way test are being executed, concretized and reported back to the system.

On-line testing: In this approach, each test case is executed separately with results reported live to the developer. Therefore, whenever a value has been sent to the test system, it is evaluated immediately, and the developer sees the verdict. Tools that use this approach often require adapters to be implemented in order to execute the tests [36], [37].

Offline testing: In this case, the test specification is derived from the system using the model-based testing tool. When the tests are generated, they are then executed using a test runner. This process can be automated. However, the key factor here is that the tests are not executed separately and must be derived from the system model and executed later, where results are presented to the developer [38], [39].

In our thesis, we consider our approach to be closer to the latter. We derive the tests from the system model, generate the test specification using a test-specific language. Then we concretize the tests into an executable language where we finally get the result of the tests. We don't require adapters and do not evaluate the tests individually. In fact, we provide a test runner with the test suite that runs the test classes and report the result back either using HTML report file or in the command line.

2.3 *Overview of Umple*

Umple is an open-source modeling and programming language [6][40]. Umple enables modeling of abstract UML concepts such as associations, constraints and state machines, as well as to add these as a layer of abstraction on top of high-level languages such as: Java, C++, and Php [41].

Umple is developed by CRuiSE team at the University of Ottawa, supervised by professor Timothy Lethbridge [6], [40], [42], [43]. Umple enables generation of the code of complete systems, on various platforms. Umple can to be used to develop systems starting with rapid prototyped initial versions, evolved in an agile manner into fully fledged mature systems.

The developers envision Umple as representing the continued evolution of textual programming, as illustrated in Figure 4.

Umple is targeted for a wide variety of developers including the open-source community, who historically have favoured textual languages. Umple is also designed as an educational tool for teaching UML modeling in classrooms [33].

In Umple, a developer or modeler has the option of writing code in three different ways:

1- Write a pure Umple model, or draw it, with real-time generation of the textual form). This allows generation of a 'pure' model-driven system in the desired target-language code, perhaps as part of a larger system.
2- Mix target-language code with the Umple model. This allows the developer to write platform-specific code *within* the Umple model. This is done when there are features (e.g. methods with their algorithms, that are not directly modellable in UML and are best represented in traditional structured code. It is also done to provide main methods that can initiate the system.
3- Write the system *mostly* in target-language code such as Java, C++, etc. and link in a limited amount of Umple-generated code to provided support for certain aspects of the design such as state machines.

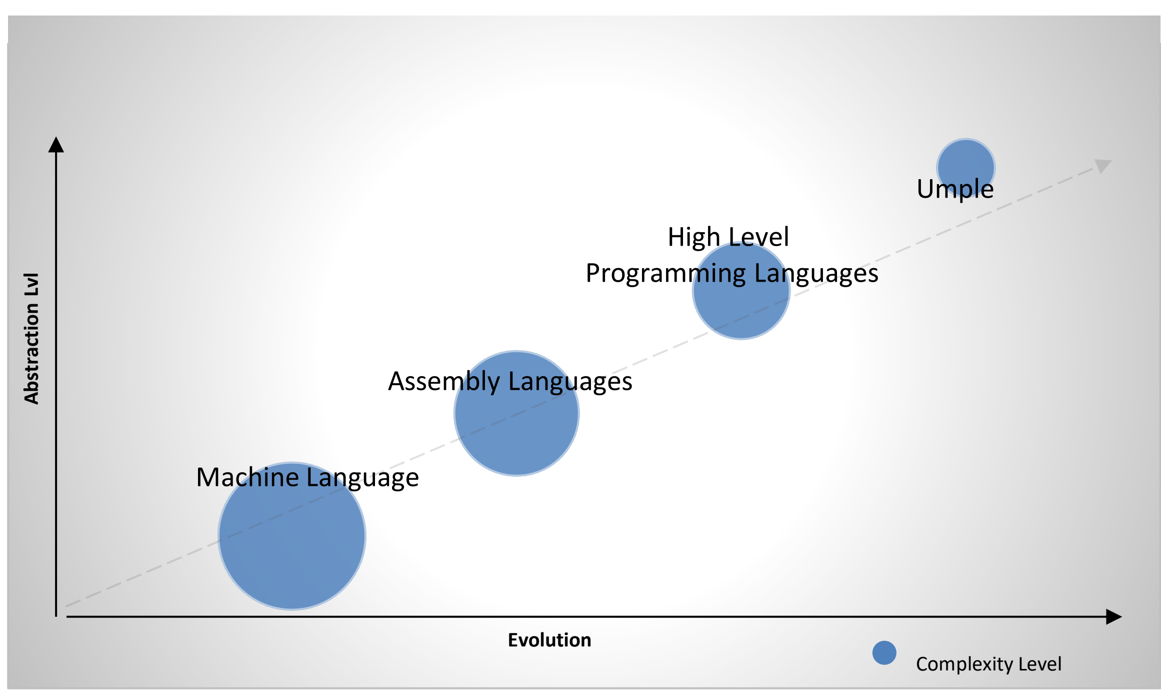

Figure 4: Level of abstraction of programming languages

These multiple ways of using Umple provide flexibility to developers, and provide a good environment for model-driven development [44].

In addition to modeling UML elements, Umple also supports several features that increase the power and synergy of both modelling and programming: These include aspect-orientation, traits [45][46], mixins, mixsets [47] (see next subsection), tracing [48] and built-in design patterns. A simple example in Umple is presented in Code Snippet 1, which is also found in Umple's online user manual. Note the syntax for the association between class Student and class Address.

```
class Student {
  firstName;   // attribute - defaults to String
  lastName;
  Integer number;  // attribute with type Integer
  * -- * Address;  // Many-to-many association

  public String fullName()  // Method, whose content is not processed by Umple {
    return getFirstName() + " " + getLastName();
  }
}

class Address{
  String[] line;  // Multi-valued attribute
}
```

Code Snippet 1: Umple code example

2.3.1 Umple Features Relevant to This Thesis: Mixins, Traits and Mixsets

Umple has been evolving for the last ten years. Many features have been integrated into it. In this thesis we will be referring to a subset of these features that need additional explanation. These are:

- **Mixins**: This is a feature that allows an entity in Umple, such as a class, to be split into multiple parts, where each part can be in one or several files. These parts will be weaved together during compilation. Two definitions for the same class, for example, will be weaved together, so there is just one class in the metamodel instance after compilation.
- **Traits**: This feature allows for the reusability of certain elements [45]. A trait can be defined as an entity containing a selection of elements that can be found in a class (attributes, associations, state machines, methods) that are *copied* into concrete classes that are referred to as *clients*; the clients will then have all the elements defined by the trait. Use of traits differs from inheritance, in that separate copies of the elements are made in the clients. Traints are discussed in detail in section 7.1.
- **Mixsets**: These are entities that add arbitrary Umple code to specified elements that can be switched on and off [47], allowing for the definition of variants or product lines. They are essentially named sets of mixins. We discuss this feature in detail further in section 7.2.

2.4 Model-Based Testing Tools and Technologies

Several studies have been conducted in model-based testing since the late 1990's. Many factors motivate these studies; some commercial tools aim to reduce the cost of testing to sell their product while others aim to push quality assurance to a higher level. The following is an overview of tools that have been developed to support model-based testing and to verify the behavior of the model by (in most cases) ultimately generating executable test cases. Later we will present a comparison among the tools, in the concluding chapter we will refer back to this comparison to show how our work advances the state of the art.

2.4.1 Conformiq Designer/Creator, Previously Known as Leirios

Conformiq [49] is a commercial tool focusing on software test automation, functional testing and software quality assurance. It was first developed in 1998 and has evolved over the years. Conformiq automatically derives black-box unit test from a system behavioral model and does not look into the source code of the system. The main artifact of Conformiq is the system model, the tool has its own DSL (domain-specific language) to describe the model in a textual language Qtronic Modelling Language (QML)[32][50]. An example is shown in Code Snippet 2.

Conformiq Qtronic uses its own modelling language QML to write test models. The language has a list of specification that can be found in their website. The following is a simple example written in QML:

```
1  record R { public int value; }
2  void main()
3  {
4    R r1, r2;
5    r1.value = r2.value = 1;
6    assert r1 == r2; }
```

Code Snippet 2: Conformiq QML example [32]

The keyword 'record' stands for a user-defined container that behaves just like a class according to the tool user manual. The previous code creates a record 'R' with an Integer value. Then two objects of type 'R' are created in the 'main' method. In line number 5 the values of attribute 'value' for both object had been set to '1'. Finally, in line '6' the assertion is written to check whether the values are equal. The QML language also provides a key aspect that we do not support, the ability to trace test objectives. Although our system has a monitoring feature using the tracing language in the model, the tracing in Umple does not support test elements in the model. This is, however, more beneficial for online-testing systems since test objectives are traced as the system is being executed and of a less value when the system is being tested offline since the tool provides test reports.

2.4.2 MBTestSuite

This is a commercial testing framework that supports automatic test generation from behavior models [51] [52]. The tool, illustrated in Figure 5, has a test generator that provides the following:

- Concrete executable test cases (manual and automatic);
- Visualization of test case tree;
- Showing the coverage within the model;
- Information about test management and change handling.

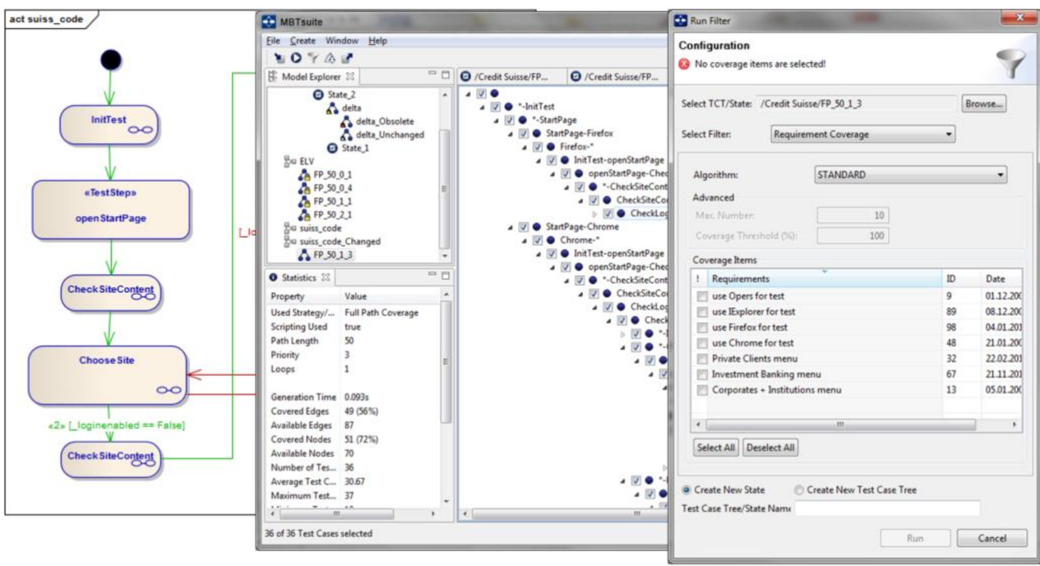

Figure 5: MbtSuite user interface [51]

2.4.3 MoMuT::UML

MoMuT::UML is a software model-based testing tool that uses model-based mutation of tests to check the correctness of behavior [53]. It is illustrated in Figure 6. The tool maps system state machine to formal semantics in order to run conformance checks between the original and mutated models in order to automatically generates tests.

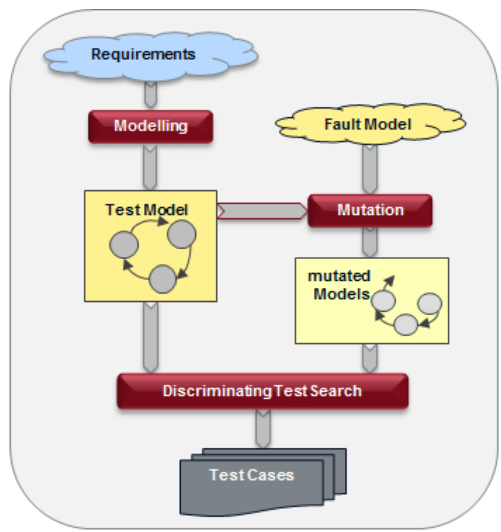

Figure 6: MoMuT::UML test generation [53]

In order to check for errors, the tool goes through the following steps: First it enables modeling of the system using UML, then It generates three types of tests: random sets of tests cases, a mutation-based test suite and a combination of the two. The tests are finally executed to reveal errors.

2.4.4 RT-Tester and RTT-MBT

RT-Tester and RTT-MBT are commercial tools that automatically generate test cases, test executions and real-time test evaluation. They support UML 2.0, domain-specific languages. They also support a multithreading environment in test execution [54] as well as requirements tracing by creating a map between generated tests and a given set of requirements. This mapping is done by annotating the model with the given requirement tag. Ultimately, a report with which requirements have been successfully test is generated.

2.4.5 TTCN-3 and UTP

TTCN-3 is a testing language notation that has the look of a regular programming language. The language is widely used in testing behavior of telecommunication systems. TTCN-3 is an international standard in software testing. It has been developed by experts from industry as well as engineers in the European Telecommunications Standards Institute (ETSI).[55], [56]. An example is shown in Code Snippet 3.

```
module tRequestAnswer {
 // Types for messages
 type record mRequest { integer param1 };
 type record mAnswer { integer param1 };

 // Port type for the interface with the SUT
 type port sut_port_type message {
  out mRequest;
  in  mAnswer;
  };

 // Component type for the MTC and system interface
 type component mtc_type {
  port  sut_port_type sut_port;
  };

 // Templates for message exchange: the parameter in mAnswer should be the one sent in mRequest + 1
 template mRequest request := { param1 := 42 };
 template mAnswer right_answer := { param1 := 43 };

 // Testcase
 testcase tc_request_answer() runs on mtc_type {
  timer answer_timeout := 2;

  // Send request to SUT
  sut_port.send(request);
  // Wait for answer
  answer_timeout.start;
  alt {
   // Correct answer
   [] sut_port.receive(right_answer) {
    setverdict(pass);
    }
   // Any other answer is wrong
   [] sut_port.receive {
    setverdict(fail);
    }
   // If answer is never received, test is failed too
   [] answer_timeout.timeout {
    setverdict(fail);
    }
   }
  }

 // Control part: just execute testcase
 control {
  execute(tc_request_answer());
  }
 }
```

Code Snippet 3: TTCN-3 example [55]

UTP (UML Testing Profile), on the other hand is a profile for testing UML. It had been developed as a standardized language based on the OMG Unified Modelling Language (UML). The main target for UTP is provide a set of domain-specific and process-independent concepts for test modeling. It also targets reusability by benefiting from model transformation vertically and horizontally.

The combination of using TTCN-3 with UTP in UML has been shown in several studies [57] to be very strong and provides great results. SDL is used in the context of describing the behavior of the system with TTCN-3.

2.4.6 ETSI Test Description Language

TDL is language for specifying tests focusing on the black-box design of the system. The language targets model-based testing for writing test descriptions that can be derived from test objectives. Also, it can be beneficial for representation of tests from resources such as test execution traces. The language is standardized, yet has little tool support [58].

2.4.7 Spec Explorer / PyModel

SpecExplorer [3] is a model-based testing tool that allows for modelling system behavior. It also allows for the analysis of the model through graphical visualization and ultimately generating stand-alone test code from the model. There are two ways to model behavior in Spec Explorer: 1) Writing rules in C#; 2) Writing scenario scripts in a regular-expression style in a separate configuration file. The tool requires building models using C# and running a model state checker pattern to verify the system behavior.

2.4.8 ACL Contracts

ACL is a model-based scenario-testing approach using contracts and comes with a verification framework. It has its own domain-specific language to specify tests using contracts. The tool was developed at Carleton University and had been used in validation of outsourced implementations provided by an external developer. This works by providing an ACL model that contains contracts and test specification and then through a verification framework [59] to ensure that the provided implementation satisfies the given abstract ACL model [60]. We will be looking into this tool/language for comparison in terms of technology and in terms of testing language design. In the ACL Code of Code Snippet 4, a contract is created with parameters and variables then a list of scenarios to be executed to validate the correctness.

```
Contract Librarian {
Parameters
{
  [0-10] Scalar Integer MaxLoanCount;
  [0-2] Scalar Integer MaxLoanRenewalCount; [0-3] Scalar Integer MaxLoanInterval;
}
```

```
// contract variables
String potentialNewUserInfo = null; Integer userIdToBeDeleted=0;
…..
Scenario addUser {
Integer uId;
String newUInfo; Trigger(observe(userAdditionRequested(newUInfo))); choice(DoesUserExist(newUInfo)) true
{
  Belief reportUserExists("UI should inform the librarian that user
  already exists with the given information");
}
….
```

Code Snippet 4: Example of ACL code [59]

2.4.9 MaTeLo

Matelo is a framework that focuses on generating test cases and test suites from behavior models. The tool supports tracing between each requirement and the corresponding tests. The tool allows for test path generation as well as providing several test strategies such as risk-based testing, regression testing and agile sprint testing. It also has two options for test coverage (custom or full) [61].

2.4.10 ModelJUnit / GraphWalker

ModelJUnit is a Java library that enables model-based testing by extending the Junit framework. The tool allows one to write FSM and EFSM as Java classes then allows generating tests and measuring test coverage. It was developed by Mark Utting [62].

Table 1: Comparison between different MBT tools – part 1

Tool / Feature	OS supported	Languages supported	DSL	Customizable criteria
Conformiq	Windows, Linux	Java	Yes (external QML)	Yes
MBTSuite	Windows	Java, Python, C , C++	none	Yes
MoMuT::UML	Windows	Java	none	Yes
RT-Tester and BBT	Windows	Python	none	No
TTCN-3	Independent	SDL, UML	Yes, External	NA
UTP	Independent	UML	none	NA
specExplorer	Windows	C#, Python	Internal (annotation)	No
ACL Contract	Windows (verification framework)	C++	Yes, external	NA
MaTeLo	Windows	Java	none	Yes
ModelJUnit	Independent	Java	none	No

Table 2: Comparison between different MBT tools – part 2

Tool / Feature	UML	Type	Test Data	Constraints	Model-Driven ?
Conformiq	state machine	commercial	Only with QML	Yes	Yes
MBTSuite	State machine	commercial	NA	No	No
MoMuT::UML	State machine	commercial	Yes	No	Yes
RT-Tester and BBT	UML/SysML	commercial	Yes	Yes	No
TTCN-3	No	Open-source	Yes	No	No
UTP	UML Profile	Open-source	Yes	Yes	Yes
specExplorer	State machine	Open-source	No	Yes	Yes
ACL Contract	Use-Case	Open-source	Yes	Yes	No
MaTeLo	State machine	commercial	NA	No	No
ModelJUnit	State machine	Open-source	No	Yes	No

2.5 Other Techniques That Are Related

We have mentioned before that there are a several tools that have similarities to our work on different levels. However, in a majority of cases, these tools encapsulate techniques applied in the underlying architecture in order to offer the final product. Some of these techniques we consider related to our work since they can achieve similar result.

2.5.1 Aspect-Oriented Programming

Aspect-oriented programming is a technique that allows for the separation of cross-cutting concerns [63]. AOP achieves this goal by introducing behavior to the existing code without modifying that code. The process is done by separately modifying *pointcut* specifications. These specifications may include functions like logging certain methods based on patterns, or executing a block of code after a method has been executed.

There are several studies in the literature that manipulate such techniques in order to either generate tests or bridge the gap between the system and the test logic. The work by Kumar and Baar [64] discusses a technique that uses AOP in order to discover executable tests in the system to feed the test module. This is done by adding a number of pointcuts that allows for this. In addition, the work by Benz [65] introduces an aspect-oriented language for the instantiation of abstract tests while aiming at reducing effort by modularizing the test concerns in the form of aspects. The main difference between our work and the work of Benz is that they allow for one abstract test case to target different properties of the system using the same abstract test case. We promote reusability by allowing the tests to be inherited or encapsulated within traits or mixset in

Umple. Also, their work is based on the Eclipse Modeling Framework so is limited in its availability in other platforms. We provide a solution that is platform-independent and can be used to test model-level aspects.

2.5.2 Generative Programming

Generative programming (GP), a form of program transformation whereby programs are created by other programs. It can be used for on-demand automatic test generation of a manually retrieved and assembled component of the system. It has been used to achieve quality and productivity, separation of concerns and to separate problem space from solution space. The work by Boussaa [66] provides a generative programming approach to automatically generate non-functional tests for code generators to support domain-specific languages and help achieve quality. Our work is different since we provide a test-specific language to specify the tests at the abstract level.

2.6 Summary

There are multiple existing model-based testing tools as seen in Tables 1 and 2 and they vary in several aspects. We can see for instance that some tools, such as Conformiq, ACL and TTCN-3, come with a DSL for modelling while other tools don't have a language to specify their tests. We can also see different platforms supported. For example, a tool like ACL/FR does not support Java. What is important is the fact that most of these tools are not model-driven, which means that one has to write the abstract test code by hand. Take ACL for example; it is a tool/language that will allow the developer to write test scenarios to verify a particular implementation regardless of the code quality. As long as the given abstract tests are satisfied, then the result should be viable. However, this does not answer the critical questions, what is the coverage criteria followed? And is the testing code driven by a given model? If it is not indeed derived from a model, then we wouldn't be saving a reasonable amount of time.

Our model-based testing approach aims to answer the above questions. Our goal is that one should be able to select among several coverage criteria. Also, the test model should not have to be written by hand but derived from the model that represents the system. This idea changes how the developer thinks, since automatically generating tests will allow him or her to focus on the logic of the system and testing rather than spending extra time writing the test model. Other critical

aspects where our work advances in the state of the art are 1) that it is embedded in Umple, so can make use of many unique features of Umple; 2) that it is open-source, and 3) that it supports multiple languages and platforms, hence it can bring together developers with background in several programming languages.

Being based in Umple means that the language we will present in this thesis enables reusability of test cases by allowing the developer to write their tests in traits that can be inherited by child traits of classes. We will show that our approach also allows developers to write test code inside Umple methods in a way similar to contracts, where these tests will be generated into executable unit tests that should be satisfied. In addition, it takes the first steps towards test-driven modelling which is not yet facilitated by any of the tools above. This is an approach to incrementally add model elements based on tests written by the user, and which conform to the requirements; this is explained in Chapter 6 .

Chapter 3 Basics of Model-Based Testing in Umple

Umple was developed using a test-driven approach [43]. This means that for each change to the compiler, a set of test cases is written, prior to the submission of that change, to attempt to cover all the effects the change may have. This results in a large number of test cases that Umple developers have to carefully maintain. Maintenance can include adjusting many previous tests, since changes can affect behavior in such a way that previous tests would give different results.

Umple compiler tests include tests that verify the Umple code is parsed correctly, that it is analyzed correctly (finding errors and building an abstract syntax graph that is an instance of the Umple metamodel), that outputs such as Java are generated correctly, and that the outputs execute correctly.

However, prior to this research Umple could not *generate* any tests for the generated code; any such testing had to be done manually, or else the Umple compiler had to be trusted. Given an arbitrary Umple model, many developers will want some way to automatically test the generated code (in Java for instance) so they can have greater confidence in the compiler. Developers of Umple itself would also benefit from generated tests as a way to detect potential errors.

Umple allows the construction of complex models, and embedding of arbitrary code. Just like Java developers want to be able to write tests in terms of Java constructs, developers using Umple would benefit from being able to specify tests at the 'Umple' level that refer to Umple constructs such as associations and state machine as well as to embedded arbitrary code. It would be best if developers can follow a 'test first' approach – in other words developing the tests before the Umple model is even developed.

Hence, incorporating model-based testing in Umple would be highly beneficial. In this chapter, we will discuss how model-based testing is approached in Umple. We will cover the main concept and the Umple Test Model/Language we are presenting in this thesis syntactically and semantically. We also describe the process to generate executable tests from an Umple model.

We aim to provide the following features to enable model-based testing in Umple:

- **A language to specify tests at the modelling level:** We want to allow the developer to specify in an abstract model, separate from the Umple model, what output values should be correct, given an input model and a set of queries. This test model/language should refer to elements in the original Umple model. This language will have two uses. The first use, illustrated in the left three boxes of Figure 7) is automatic test case generation of any Umple model: In this case an Umple Test Model (set of abstract test cases) will be automatically generated. If the Umple model changes, such tests must be regenerated. The second use is to allow the developer to write their own abstract tests at the modeling level; the developer would be responsible to edit these if they change the model (as would be the case for testing languages such as jUnit).
- **Automatic generation of executable tests in xUnit languages:** This is done by transforming the abstract test model into one of several target languages, and is illustrated by the 'Test Model Parser' and 'Executable Test Case' boxes in Figure 7.

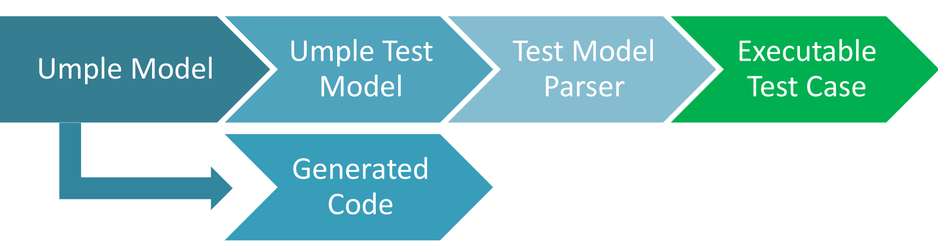

Figure 7: Umple test generation process

When we first considered the topic of automatically generated test cases, there were several questions to be answered, these were:

1- What to test from an Umple model?
2- How can we integrate test artifacts within the Umple architecture?

There were couple of approaches to consider in order to determine what elements to be tested from the model: First, we studied the hundreds of existing test files in the Umple project to see what previous developers covered with these tests, then decide whether there was a certain coverage criterion selected at the time of writing these test cases.

Secondly, we looked into the literature to see what are test coverage criteria that had been used to test UML class diagrams and UML state machines. We found from the literature that the focus is on behavioural UML models (primarily state machines) but not as much on static models like class diagrams. Beizer's work, however, discusses what to test from an UML model and how [1]. Also, Offutt discusses [67] what to test from a class diagram providing several algorithms for testing.

3.1 Umple's Model-Based Testing Architecture

Figure 8 shows the model-based testing architecture we have developed.

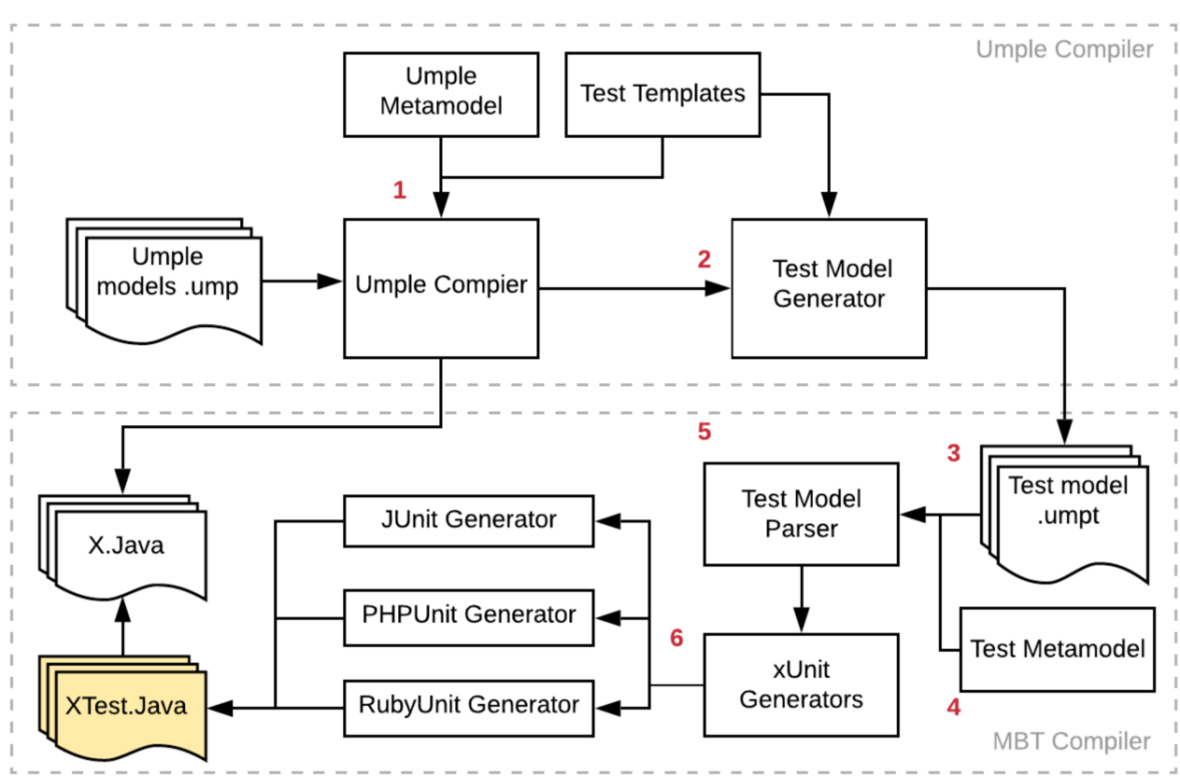

Figure 8: The Umple test generator architecture

This thesis has contributed all the rectangles in the figure except the Umple Metamodel and Umple Compiler, which have only been modified. The numbers in the figure represent the order in which the tests are processed.

The Umple compiler parses and analyses a given Umple model (#1). This would happen in the same way regardless of what the compiler is being later asked to generate. In the current process

we will consider generation of tests, but the same analysis would be performed if the compiler was generating diagrams or code. This step can result in syntactic or semantic errors that have to be corrected before the process proceeds.

The class *TestGenerator* (#2), one of many different generators plugged into the Umple compiler, then runs. The model instance is processed in order to find elements to be tested. For each attribute, association, or other model element, the test generator will add a set of tests to a 'test buffer' (where all the test code is accumulated). This is eventually written into a file with the extension '.umbt'; which stands for Umple Model-Based Test. This file is the abstract test model represented as #3 in the figure.

3.2 Model Transformations in Our Approach

A model transformation is a process that accepts a model as an input; its output can vary based on the purpose of the transformation – it could be code, a different model or any other product. Model transformation is often used in software development, in particular when doing migration toward a certain technology or when working with a diverse selection of languages. Model-driven development builds on the core concept of model transformation since it focuses the editing and processing of models with a well-rounded model transformation mechanism. There are several types of transformations in model-driven development depending on the type of target platform, these are:

- **Model-To-Model transformation**: This type of transformation takes a model as input and generates an output of another model. This type of transformation mostly targets platform independent models and is more abstract than final output results. This type of transformation is also called *Horizontal Transformation* [68][23].
- **Model-To-Text transformation**: This type of transformation takes a model as an input and generates code, despite what this code stands for (it could represent a model). The key here is that the code it generates is concrete and at a lower level of abstraction than the original input model. This type of transformation is also called *Vertical Transformation* referring to the fact that we are going from a higher level of abstraction to a lower one.

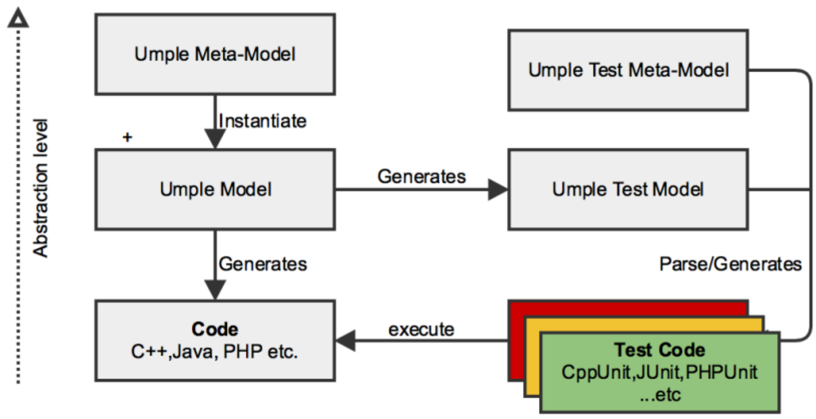

Figure 9: Model transformation in Umple test generation

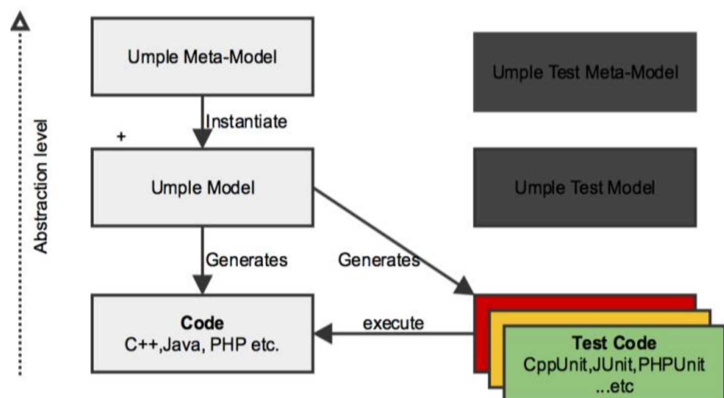

Figure **10:** Inadequate approach: directly generating executable tests without an abstract model.

Figure 9 shows the Umple approach, including two horizontal transformations and two vertical transformations. More details of this are explained in the next section.

In Figure 10, one horizontal transformation is ignored, and the transformation is done directly from the *UmpleModel-TestCode*. This approach is insufficient because it does not conform to the best practices of model-based testing. According to several studies [23][24][25], an abstract representation of the model must be present and the best way to create it is by deriving the abstract test model from the Umple model since they are at the same level of abstraction which makes this transformation a horizontal.

3.3 The Umple Test Model Parser

A key goal of this work is to generate executable test cases. This is achieved by first generating the abstract test model/language. However, this test model/language must be parsed and interpreted in order to be concretized into executable test cases. We have developed a parser, marked as #5 in Figure 8, for this purpose. This parser is written in Umple using the Umple Parser technology.

The following demonstrates the input and output of the *TestGenerator*. Figure 11 shows the packages view of the Umple test generator. As seen in the figure, the class *TestModelgenerator* is part of the Umple compiler, it uses the instance of the Umple metamodel in order to generate the abstract test. On the right side, the main package is *UmpleMBTestGenerator*, this has all the internal packages for generating executable tests. *xUnit Meta Model* is used by the 'MBT Parser' to parse and analyze the abstract model. *TestGenerator* is a concrete test generator with several classes, *xUnitTestCaseGenerator* an abstract class used as a template for each targeted implementation. Then we have *TestCaseJUnitGenerator* which uses the assertion templates and utility class for generating the test code. The Utility class is specifically used for converting values, calculating the random values for tests at runtime and other useful functions for testing.

Component: TestGenerator, Umple Compiler

Input: Umple Model . (e.g.: model.ump). This is an instance of the Umple metamodel that had been parsed and analyzed by the Umple compiler.

Process: Examine Umple model elements: attributes, associations, etc. and create the tests based on the values in the Umple model.

Output: An Abstract Test model (modelTest.umbt)

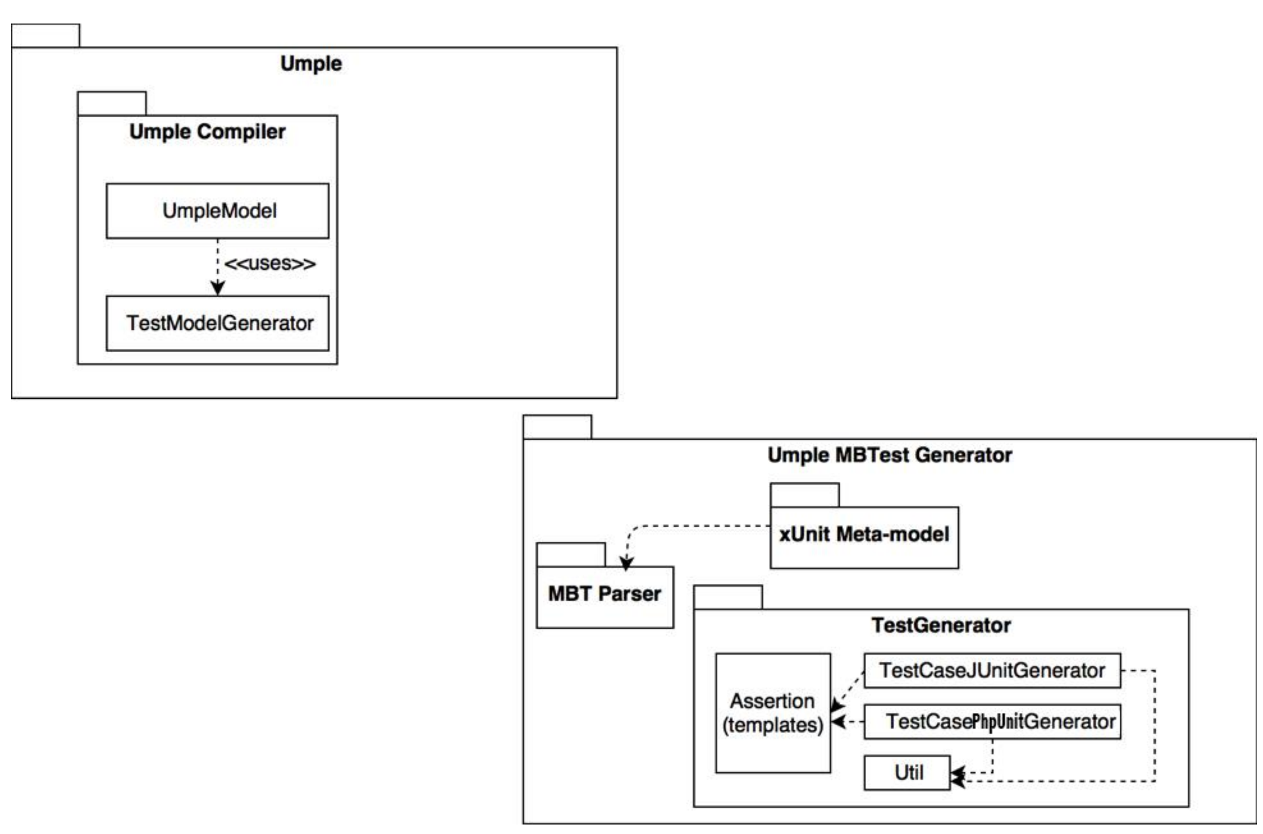

Figure 11: Umple Test generator packages

The above is the main modification made to the Umple compiler in order to incorporate the test generator as part of the system. However, this process will only generate abstract tests that are non-executable. In order to execute the test model, it must be concretized into a specific platform such as Junit, RubyUnit or PhpUnit. In order to do this, we have to process the test model. Processing the test model involves parsing its content and generating the corresponding executable platform. This is also shown in Figure 8 as items #5 (parsing the abstract test model) and #6 (generating an appropriate xUnit language).

As outlined in Section 3.2, one important design decision we had to make was between the following two options:

- Having the Test Generator (#2 in Figure 8) directly generate the xUnit tests (#6), from its internal memory representation of the abstract tests.
- Writing out tests in the abstract language and then parsing this language.

Although the first option would have some benefits in terms of efficiency and amount of code we would have to write, we chose the second option for several reasons:

1. It allows for a cleaner separation of concerns: Others could write independent tools for our language, or we could adapt our generators to generate a different testing language.
2. A parser is needed for the abstract test language anyway to give the developer the option to write their own tests abstractly. The approach we chose would therefore not save much code, and in fact helps ensure that parser is more thoroughly tested.

In order to parse the abstract test model, we need a grammar (discussed later) for it. As with a compiler, our parser uses this grammar to generate an abstract syntax tree containing parsed tokens, These are then analysed for correctness as an instance of our xUnit metamodel is parsed. Looking again at Figure 8, component #4 shows the xUnit metamodel we used for this purpose. Component #5 is the test model parser itself. This parser takes a test model file as an input 'testModel.umbt' and analyzes it in order to finally generate the executable tests, as seen in #6.

Component: TestModelParser, XUnit_Metamodel

Input: Umple Test Model . (exp: modelTest.umbt). This is a textual representation of the abstract tests.

Process: Parse the content, get the tokens, analyze the tokens and finally generate test files.

Output: Executable test files (class1Test.java , class1Test.php)

The generated executable test cases (Junit files for instance) should be able to run against the Java, Php, C++ or other code generated from the Umple model, and we should be able to see the result of the tests. More details about each component will be discussed later in this thesis.

After the initial work in the thesis, we decided to add the link in Figure 8 between the Test Templates and the Umple Compiler. This is to allow the test syntax to be embedded directly into the Umple model. The grammar and processing logic of the test model parser therefore also appears in the Umple compiler.

Later, we will look deeper into the metamodel content, the content of the test model and also test patterns used for coverage. Next we will look into a system written in Umple with a sample of the generated tests.

Figure 12 and the following Umple textual form (Code Snippet 5) are used as a basis for further explaining our approach.

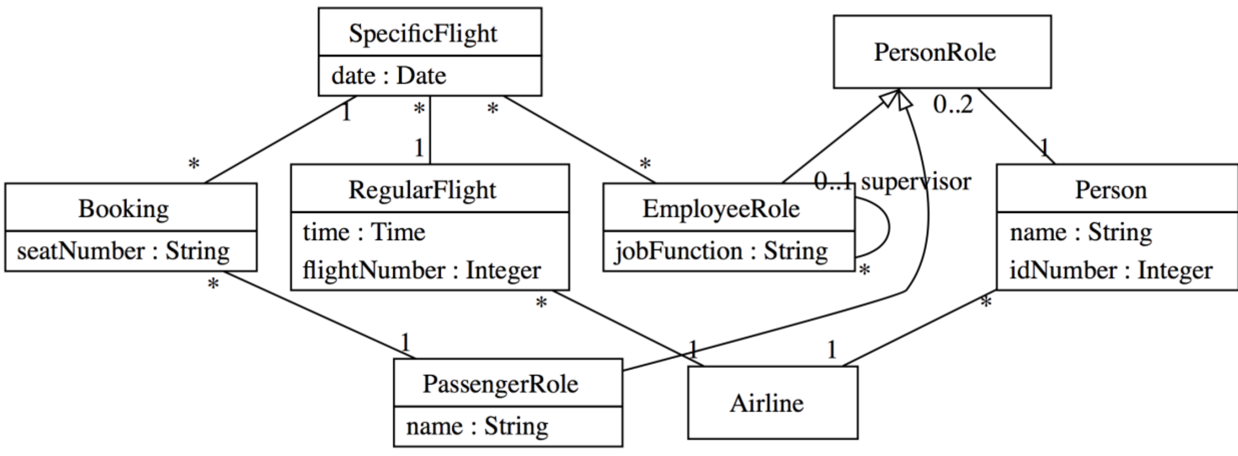

Figure 12: Umple airline example diagram

```
// Airline system - sample UML class diagram in Umple
// From Book by Lethbridge and Laganiere, McGraw Hill 2004
// Object-Oriented Software Engineering: Practical Software Engineering using UML and
Java

namespace Airline;

class Airline{
  1 -- * RegularFlight;
  1 -- * Person;
}

class RegularFlight{
  Time time;
  unique Integer flightNumber;
  1 -- * SpecificFlight;
}

class SpecificFlight{
  unique Date date;
}

class PassengerRole {
  isA PersonRole;
  immutable String name ;
  1 -- * Booking;
}

class EmployeeRole {
```

```
  String jobFunction ;
  isA PersonRole;
  * -- 0..1 EmployeeRole supervisor;
  * -- * SpecificFlight;
}

class Person {
  settable String name;
  Integer idNumber;
  1 -- 0..2 PersonRole;
}

class PersonRole{}

class Booking{
  String seatNumber;
  * -- 1 SpecificFlight;
}
```

Code Snippet 5: Airline example textual Umple code[1]

For this airline Umple model, the test generator will generate the following:

- **A test model file for each individual class:** This contains all the tests related to the class individually for model aspects that do not involve interaction among classes. This includes testing the API of attributes and state machines. The name of this part of the test model is by convention the class name followed by the word 'test', *classTest* for instance. The tests involve initializing attributes and calling their API; important kinds of tests are those that ensure, for example, that immutability and other stereotypes are respected. Tests of state machines in a class verify that the state machine ends up in the correct states after sets of carefully-generated events are applied.
- **A test model file for the Umple model as a whole:** In this test model file, we include all the tests for associations, since they refer to multiple classes. The process of generating this model requires dependency analysis and initialization. Tests require initializing instances of multiple classes, calling the generated API to link them and then verifying that the objects are linked as expected.

Table 3 shows the generated files for the airline example:

[1] http://try.umple.org/

Table 3: Generated test files for the airline example

Filename
AirlineExample_ModelTest.umpt
AirlineTest.umpt
BookingTest.umpt
EmployeeRoleTest.umpt
PassengerRoleTest.umpt
PersonRoleTest.umpt
PersonTest.umpt
RegularFlightTest.umpt
SpecificFlightTest.umpt

For the rest of this chapter we will be discussing how different elements are tested and what type of test template we used to test the behavior of the element in regard to the generated code. These templates are written using *Umple Template Language*. To read more about these templates please refer to the Umple repository on GitHub here:

https://github.com/umple/umple/tree/master/UmpleToTest

The templates are under the project UmpleToTest. This will generate two implementation classes that are used by the Umple compiler. The first implementation class is ClassTestGenerator, which will generate a test file for each Umple class. The second implementation class is called ModelTestGenerator, which will generate a test file for the Umple model. This is the first file in Table 3. Templates are inspired by three main oracles: The Umple testbed, Meszaros' Book [69] and Binder's book on testing object-oriented systems [70].

3.4 Testing of Attributes

An attribute in Umple (as in UML) represents a simple element that holds a value of a specific type [71]. It can hold a primitive value , for example an Integer for 'id' or an instance of a class such as 'Address'. Attributes are more than just 'fields' or 'instance variables', since they have a much richer semantics. For example, they can be subject to constraints of various kinds, can have their changes used to trigger events in state machines.

From the testing perspective, we can take into account the different types of attribute that can be used in the model , and find a methods to automatically generate test for each attribute. Hence, we are interested in investigating the following:

- What types of value can the attribute may hold?
- Is the attribute a list or single value attribute?
- Is there any stereotype or pattern applied to this attribute that may affect its behavior in this system?
- Are there any generated methods that are related to this attribute: such as getter/setter or query methods?

In Umple there are several stereotypes of attributes. However, for most attributes, Umple generates set and get methods. Therefore, we need to test the behavior of these methods by providing specific test data that we will discuss later. Regular attributes should be able to be set by a setter method and retrieved by a getter method. For some types of attributes like immutable ones (i.e. preceded by the keyword *immutable*), we need to generate test cases to verify the *absence* of a setter method for the attribute; this is done using reflection. Also, attributes that are defined as lists, which is indicated in Umple (following C-family syntax) with square brackets [], are also taken in consideration in testing by providing test cases to check the list-specific methods to make sure they are working correctly.

There other stereotypes of attributes in Umple such as: *defaulted*, *const* and *lazy*. Attributes also can have a built-in type such as Date or Time, or be of an arbitrary type. Attributes can also be subject to constraints.

For instance, the attribute 'Integer idNumber' in the class Person should have the abstract tests generated shown in Code Snippet **6** and the executable tests shown in Code Snippet 7, assuming Java is the target language, and hence jUnit the testing language.

```
…
WHEN:
Person person("John", "123");
THEN:
test aTestCase {
 AssertTrue(Person.getIdNumber() == "123");
```

```
 AssertTrue(Person.getIdNumber() == RANDOMINTEGER);
}
```

Code Snippet 6: Test model code example

```
@Before
    public void setup(){
      Integer randomInt = util.randomGenerator(1000);
    }

@Test
    public void idNumberTest(){

  //------------------------
  // Assertions
  //------------------------

  Assert.assertTrue(person.getIdNumber("123"));
  Assert.assertTrue(person.setIdNumber(randomInt);
  Assert.assertTrue(person.getIdNumber(randomInt));
    }
```

Code Snippet 7: Executable test code in JUnit

3.4.1 Testing Lazy Attributes

In the case of lazy attributes, Umple generates initializing values for its built-in data types in the constructor (e.g. the empty string, zero, or false), whereas for regular (not lazy) attributes, the value must be specified as an argument to the constructor. This means the value of the lazy attribute cannot be null following initialization of the object. Therefore, we test such attributes by adding extra tests to make sure the values is not null. We also have to ensure that when we generate tests, that there is no argument for the attribute on the constructor. Code Snippet 9 shows the abstract test model for a sample attribute given in Code Snippet 8.

```
generate Test;

class LazyA {
  lazy id;
}
```

Code Snippet 8: Lazy attribute in Umple

```
Test LazyA {
  //tests for :id
  test attribute_id {

    AssertTrue( lazya.setId("RandomString1"));
    AssertTrue( lazya.getId() =="RandomString1");
    AssertTrue( lazya.setId("RandomString1"));
    AssertTrue( lazya.getId() =="RandomString1");
    AssertNotNull (id);
    AssertNotNull (LazyA.getId != null)
  }
}
```

Code Snippet 9: Generated abstract model for lazy attribute

3.4.2 Testing Immutable Attributes

In the case of immutable attributes, they are created without a setter method. As a result, the value cannot be changed after construction. In this case, we use Java reflection in order to look into the generated code to find the method immutableA.setId(). In Code Snippet 12, we can see the generated method hasMethod will return true if the method exists in the generated code. Code Snippet 11 shows the abstract test that will generate concrete calls to hasMethod when jUnit is generated. This way, we can use assertFalse (hasMethod(setId)) on the class ImmutableA to make sure method is not generated and the Umple generated code as expected.

```
generate Test;

class immutableA {
  immutable id = "someId";
}
```

Code Snippet 10: Immutable attribute

```
Test immutableA {
  //tests for :id

    test attribute_id {

    AssertMethodFalse( immutablea.setId());
    AssertTrue( immutablea.getId() == "someId");

               }
}
```

Code Snippet 11: Generated test for immutable attribute

```
public Boolean hasMethod (Class<A> cls, String methodName)
    {
       Boolean hasMethod = false;
       Method[] methods = cls.getMethods();

       System.out.print("\n");
       System.out.print(methods.toString());
       System.out.print("\n");

       for (Method m : methods )
       {
              System.out.print("\n");
              System.out.print(m.getName());
              System.out.print("\n");
              if ( m.getName() == methodName)
              { hasMethod = true; }
       }

       return hasMethod;
    }
```

Code Snippet 12: Concrete code in JUnit used to find setter method for immutable attributes

All of the software we have developed in this thesis is written in Umple, including Umple's native template language (UmpleTL) for generating text, and Umple's parsing library.

As an example of our implementation, Code Snippet 13 is a template file written in UmpleTL (Umple Template Language) [72] that detects the different patterns for attributes by looking into all attributes in a given class. The code will construct the test case structure for the attribute then decide which template to use for the body based on the type of pattern detected.

```
if (!uClass.hasAttributes())
  { appendln(realSb, ""); }

  else {
    for(Attribute at : uClass.getAttributes()) {
      String typeName = at.getType() == null ? "String" : at.getType();
      String upperBound = at.getIsList() ? " upperBound=\"-1\"" : "";
      String attrName = StringFormatter.toPascalCase(at.getName());
            #>> //tests for :<<=at.getName()>>

                    test attribute_<<=at.getName()>> {
            <<#
            if (!at.isImmutable())
                {#>>
                        <<@UmpleToTest.attribute_typed>><<#
                    }
             if (at.isIsLazy())
                {#>>
                        <<@UmpleToTest.attribute_Lazy>>
                <<#}
      if ( at.isImmutable() == true)
                {#>>
                        <<@UmpleToTest.attribute_Immutable>>
                <<#}
            appendln(realSb, "\t}\n");
    }
  }
```

Code Snippet 13: Umple template code for detecting attribute patterns

3.5 Testing Associations

Associations in Umple are one of the core elements of the technology [73]. Since Umple was originally designed to abstract associations from programming languages, they are used heavily in the design of systems. Also, the majority of the generated Umple API revolves around handling and processing association variables. Hence, focusing on testing associations will significantly improve the quality of the systems generated by Umple.

An association represents a relation between two classes in the system. In Umple, association statements can be injected in any class with a reference to the class at the other end. They can also

be written externally to either class. There are several types of association in Umple, we will discuss each case with examples in terms of automatic test generation for associations. We will also discuss the generic test generation for association variables in models. When it comes to testing associations, we focus on the following points:

Table 4: generated API for class RegularFlight in the airline model

Return Value	Method and Parameters
boolean	addOrMovePersonAt(Person aPerson, int index)
boolean	addOrMoveRegularFlightAt(RegularFlight aRegularFlight, int index)
boolean	addPerson(Person aPerson)
Person	addPerson(java.lang.String aName, int aIdNumber)
boolean	addPersonAt(Person aPerson, int index)
boolean	addRegularFlight(RegularFlight aRegularFlight)
RegularFlight	addRegularFlight(java.sql.Time aTime, int aFlightNumber)
boolean	addRegularFlightAt(RegularFlight aRegularFlight, int index)
void	delete()
Person	getPerson(int index)
java.util.List<Person>	getPersons()
RegularFlight	getRegularFlight(int index)
java.util.List<RegularFlight>	getRegularFlights()
boolean	hasPersons()
boolean	hasRegularFlights()
int	indexOfPerson(Person aPerson)
int	indexOfRegularFlight(RegularFlight aRegularFlight)
static int	minimumNumberOfPersons()
static int	minimumNumberOfRegularFlights()
int	numberOfPersons()
int	numberOfRegularFlights()
boolean	removePerson(Person aPerson)

- **Logic check:** Making sure the logic of the association in the model is reflected correctly in the generated code, including referential integrity: This can be verified through several layers of checks on the associations construct. This may include multiplicity checking and other aspects such as whether the association is directional or not.
- **API test:** We also want to ensure that the behavior of the generated API is working as expected. Umple supports associations with a list of API methods which we use to process association variables. Table 4, shows the generated Umple API for class RegularFlight from the Airline model presented earlier.

To test associations, we are particularly interested in testing the following elements: association ends, association multiplicity upper-bounds, lower-bounds, and directionality of associations. The latter need to be tested using reflection to verify the absence of methods in the API that would otherwise be present. Code Snippet 14 shows an example of the abstract code we generate for one of the associations in the airline example.

```
File: <Airline.ump>
01:   /*-------------------------*/
02:   /*  Class RegularFlight    */
03:   /*-------------------------*/
04:
05:               /////// Association: -- [*,1] Airline ///////
06:               //create .....
07:               Test CreateRegularFlightWithoutAirline {
08:               RegularFlight someRegularFlight = new RegularFlight();
09:               AssertTrue (someAirline.getAIRLINE()!= null);
10:               }
11:               //replace
12:
13:               Test ReplaceAirlineInRegularFlight {
14:
15:               Airline someAirline = new Airline();
16:               Airline someAirline2 = new Airline();
17:               someAirline.addRegularFlight(someRegularFlight);
18:               AssertEqual (1, someAirline.getNumberOfRegularFlights());
19:               AssertEqual (1, someAirline.getNumberOfRegularFlights());
20:               someRegularFlight.setAirline(someAirline2);
21:               AssertEqual (someRegularFlight, someAirline2.getRegularFlight(0));
22:               }
23:               //delete
24:               Test DeleteAirline {
25:               // delete Airline in RegularFlight
26:               someAirline2.delete();
27:               AssertEqual (null, someAirline2.getRegularFlight(0));
28:               }
29:               //add to new
30:
31:               Test AddAirlineToNewRegularFlight {
32:               Airline someAirline = new Airline();
33:               Airline someAirline2 = new Airline();
34:
35:               RegularFlight someRegularFlight = new RegularFlight(someAirline);
36:               someRegularFlight.addAirline(someAirline);
37:               someRegularFlight.setAirline(someAirline2);
38:
39:               AssertEqual(someAirline2, someRegularFlight.getAirline();
40:               AssertEqual (someRegularFlight, someAirline2.getRegularFlight(0));
41:               AssertEqual (1, someAirline.getNumberOfRegularFlights());
42:               }
```

Code Snippet 14: An example of abstract test code for an association

In the above code snippet, we can see that we first set up the testing data based on given values for one object of PassengerRole and two objects from Booking. In the 'Then' area where we list our assertions, we first test the attribute 'name' and then we test the API as follows:

- Test the function of adding these objects to Booking association variable List<Booking> bookings in Umple generated code.
- Confirm the size of the list is as expected.
- Check that the object exists in the list as expected.
- Test the function for removing these objects from the list.
- Confirm the size of the list again to ensure it reflects changes.

Similar tests will be generated for each association in the system, checking the functionality of the API and the attributes in each class.

Some cases are more complex and require chains of association between classes. Class A may require an instance of B, yet B requires an instance of C. Such dependency must be analyzed ahead of the initialization process. A concrete example of such dependency can be seen in the following requirements: *Each Booking is for one SpecificFlight; each SpecificFlight is for one ScheduledFlight; each ScheduledFlight is for one Airline*. In a model for such a chain, there would be a series of * -- 1 associations. The test generator would have to create an Airline, then add to this some ScheduledFlights, then add to each some SpecificFlights, then add to this some Bookings (which, by the way, would also require Passengers to be generated first). We will discuss this issue more in Chapter 9

Coverage of the Umple API associated with associations includes a functionality to manage objects in the sets of links. In addition, we need to examine the association's multiplicities and test various scenarios, such as assuring that upper and lower bounds are respected. Associations can be classified based on patterns corresponding to the multiplicities at both ends. The test generator detects the pattern of association and based on that choses the corresponding test template. The following sections outline the different patterns.

3.5.1 Optional-One to Many Associations

In this association pattern, exemplified by Figure 13 and Code Snippet 15, we test the following:

- Create an object of A without B (should be successful).
- Replace a B in A (this should be allowed)

- Delete a B from A (this should be allowed)
- Add another object of A in B (this should be allowed)
- Test the upper bound if the number of Bs at the maximum value (it should not be possible to add a second B).

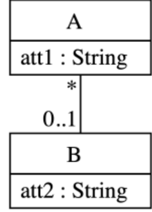

Figure 13: Optional on- to-many association

```
class A {
  att1;
}

class B{
  att2;
  0..1 -- * A;
}
```

Code Snippet 15: example of optional one to many

The resulting test code generated appears in Code Snippet **16**.

```
File: <model.ump>
01: …
02:
03: ////---- Tests for  OptionalOneToManyAssociation ----/////
04:
05: Test OptionalOneToManyAssociation.ump {
06: depend  A,B ;
07:
08:
09:
10:   /*-------------------------*/
11:   /*  Association Test       */
12:   /*-------------------------*/
13:   /*-------------------------*/
14:   /*  Class A       */
15:   /*-------------------------*/
16:
17:               /////// Association: -- [0,1] B ///////
18:               //create ... without ...
19:               Test CreateAWithoutB {
20:               A someA = new A();
21:               AssertTrue (someA.getB()!= null);
22:               }
23:               //replace
24:
25:               Test ReplaceBInA {
26:
```

```
27:                B someB = new B();
28:                B someB2 = new B();
29:                someB.addA(someA);
30:                AssertEqual (1, someB.getNumberOfAs());
31:                AssertEqual (1, someB.getNumberOfAs());
32:                someA.setB(someB2);
33:                AssertEqual (someA, someB2.getA(0));
34:                }
35:                //delete
36:                Test DeleteB {
37:                // delete B in A
38:                someB2.delete();
39:                AssertEqual (null, someB2.getA(0));
40:                }
41:                //add to new B
42:
43:                Test AddBToNewA {
44:                B someB = new B();
45:                B someB2 = new B();
46:
47:                A someA = new A();
48:                someA.addB(someB);
49:                someA.addB(someB2);
50:
51:                AssertEqual(someB2, someA.getB();
52:                AssertEqual (someA, someB2.getA(0));
53:                AssertEqual (someA, someB.getNumberOfAs());
54:                }
55:
56:                //boundary test
57:
58:                Test BoundaryTest {
59:                private int size = B.getNumberOfAs();
60:                AssertTrue ( size > 0 &&  size < -1)
61:
62:                }
63:                Test BoundaryAtMax {
64:                //Maximum A allowed : 1
65:                int size = B.getNumberOfAs();
66:                B someB = new B();
67:                A obj0 (someB);
68:                A obj1 (someB);
69:                AssertEqual(size, someB.getNumberOfAs());
70:                AssertEqual(someB, obj1.getSomeA(0));
71:                }
72:                /*-------------------------*/
73:                /*  Class B       */
74:                /*-------------------------*/
75: ..
76:  }
```

Code Snippet 16: Generated abstract test model for an optional one-to-many association

3.5.2 One to Many Associations

In this type of association, we run very similar tests from the set in Section 3.5.1, with additional checks on the multiplicity of 1, making sure any object of A always has an instance of

B and it cannot be null. Figure **14** shows the UML diagram generated by Umple, Code Snippet 17 shows the Umple textual form, and Code Snippet 18 shows the generated abstract tests.

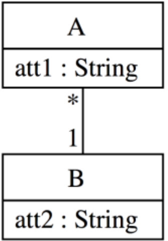

Figure 14**: One-to-many association**

```
class A {
att1;
}

class B{
  att2;
  1 -- * A;
}
```

Code Snippet 17: Umple model for one-to-many association

```
File:
01:
02: ////---- Tests for  OneToManyAssociation ----/////
03:
04: Test OneToManyAssociation.ump {
05: depend  A,B ;
06:
07:   /*-------------------------*/
08:   /*  Association Test       */
09:   /*-------------------------*/
10:   /*-------------------------*/
11:   /*  Class A       */
12:   /*-------------------------*/
13:
14:               /////// Association: -- [*,1] B ///////
15:               //create .....
16:               Test CreateAWithoutB {
17:               A someA = new A();
18:               AssertTrue (someA.getB()!= null);
19:               }
20:               //replace
21:
22:               Test ReplaceBInA {
23:
24:               B someB = new B();
25:               B someB2 = new B();
26:               someB.addA(someA);
27:               AssertEqual (1, someB.getNumberOfAs());
28:               someA.setB(someB2);
29:               AssertEqual (someA, someB2.getA(0));
30:               }
```

```
31:                 //delete
32:                 Test DeleteB {
33:                 // delete B in A
34:                 someB2.delete();
35:                 AssertEqual (null, someB2.getA(0));
36:                 }
37:                 //add to new B
38:
39:                 Test AddBToNewA {
40:                 B someB = new B();
41:                 B someB2 = new B();
42:
43:                 A someA = new A();
44:                 someA.addB(someB);
45:                 someA.addB(someB2);
46:
47:                 AssertEqual(someB2, someA.getB();
48:                 AssertEqual (someA, someB2.getA(0));
49:                 AssertEqual (someA, someB.getNumberOfAs());
50:                 }
51:
52:                 //boundary test
53:
54:                 Test BoundaryTest {
55:                 int size = B.getNumberOfAs();
56:                 AssertTrue ( size > 0 &&  size < -1)
57:
58:                 }
59:
60:                 Test BoundaryAtMax {
61:
62:                 //Maximum A allowed : 1
63:                 int size = B.getNumberOfAs();
64:
65:
66:                 A obj1 ();
67:
68:                 B someB = new B(obj1,);
69:                 A obj1 (someB);
70:                 AssertEqual(size, someB.getNumberOfAs());
71:                 AssertEqual(someB, obj1.getSomeA(0));
72: }
73:
74:
75:  }
```

Code Snippet 18: Generated abstract test model for one-to-many association

3.5.3 N to Many Associations Where N Is a Fixed Number > 1

In this case, we have an association with a numerical multiplicity, for instance, *4* --* . The way we handle this is the same; we handle the lower-bounded and upper-bounded association, we treat it the same way we handle the association of type *4..4*--* . Code Snippet 19 shows the generated abstract test model for this type of association.

```
File:
001: ////---- Tests for  NNToManyAssociation ----/////
002:
```

```
003: Test NNToManyAssociation.ump {
004: depend  A,B ;
005:
006:   /*-------------------------*/
007:   /*  Association Test       */
008:   /*-------------------------*/
009:   /*-------------------------*/
010:   /*  Class A       */
011:   /*-------------------------*/
012:
013:              /////// Association: -- [4,4] B ///////
014:              //create ... without ...
015:              Test CreateAWithoutB {
016:              A someA = new A();
017:              AssertF (someB.getBS()!= null);
018:              }
019:              //replace
020:
021:              Test ReplaceBInA {
022:
023:              B someB = new B();
024:              B someB2 = new B();
025:              someB.addB(someA);
026:              AssertEqual (1, someB.getNumberOfAs());
027:              AssertEqual (1, someB.getNumberOfAs());
028:              someA.setB(someB2);
029:              AssertEqual (someA, someB2.getA(0));
030:              }
031:              //delete
032:              Test DeleteB {
033:              // delete B in A
034:              someB2.delete();
035:              AssertEqual (null, someB2.getA(0));
036:              }
037:              //boundary test
038:
039:              Test BoundaryTest {
040:              private int size = B.getNumberOfAs();
041:              AssertTrue ( size > 0 &&  size < -1)
042:
043:              }
044:
045:              Test BoundaryAtMax {
046:              //Maximum A allowed : 4
047:              int size = B.getNumberOfAs();
048:
049:              A obj1 ();
050:              A obj2 ();
051:              A obj3 ();
052:              A obj4 ();
053:              B someB = new B(obj1,obj2,obj3,obj4,);
054:              A obj4 (someB);
055:              AssertEqual(size, someB.getNumberOfAs());
056:              AssertEqual(someB, obj1.getSomeA(0));
057:              AssertEqual(someB, obj2.getSomeA(0));
058:              AssertEqual(someB, obj3.getSomeA(0));
059:              AssertEqual(someB, obj4.getSomeA(0));
060:
061:
062:              }
063:   /*-------------------------*/
064:   /*  Class B       */
065:   /*-------------------------*/
```

```
066:
067:                ///////  Association: -- [0,*] A ///////
068:                //create ... without ...
069:                Test CreateBWithoutA {
070:                B someB = new B();
071:                AssertF (someA.getAS()!= null);
072:                }
073:                //replace
074:
075:                Test ReplaceAInB {
076:
077:                A someA = new A();
078:                A someA2 = new A();
079:                someA.addA(someB);
080:                AssertEqual (1, someA.getNumberOfBs());
081:                AssertEqual (1, someA.getNumberOfBs());
082:                someB.setA(someA2);
083:                AssertEqual (someB, someA2.getB(0));
084:                }
085:                //delete
086:                Test DeleteA {
087:                // delete A in B
088:                someA2.delete();
089:                AssertEqual (null, someA2.getB(0));
090:                }
091:                //boundary test
092:
093:                Test BoundaryTest {
094:                private int size = A.getNumberOfBs();
095:                AssertTrue ( size > 4 &&  size < 4)
096:
097:                }
098:  }
```

Code Snippet 19: Generated abstract test model for an N -- * association

3.5.4 Directional Associations

Associations in Umple can be declared with a specific direction. Directionality, also known as navigability, determines whether instances on one associated class maintain links to the other, or vice-versa, or both. Directionality can be declared as either ‘bidirectional’ or ‘unidirectional’. In ‘bidirectional’ associations, instances of each of the two classes has one or more references to instances of the other class, and referential integrity is maintained by all the API methods (i.e. if and only if instance A has a link to instance B in the association, B will have a link to A). On the other hand, in ‘unidirectional’ association instances of only one of the associated classes maintains references to the other.

Bidirectional associations are the default in Umple, using the textual notation ‘--', and with no arrows when drawn graphically. This is the original UML 1.0 notation, which we decided to maintain in order to keep diagrams easier to read, rather than putting arrows on both ends of most

associations. The associations illustrated so far are therefore all bidirectional, and we have discussed the testing of them.

Unidirectional associations use the textual notation '->' to correspond to the graphical arrows that appear in UML. When it comes to testing unidirectional associations, we want to check the following aspects, with A and B referring to the diagram given in Figure 15:

- **There is no reference in the class B to A**. This is be done using 'assertMethod' and 'assertAttribute'. We generate tests that check whether class B has method 'getA', 'setA', etc. We want to check the generated code for having any reference to A generated in B.
- **Class A does have references to class B**. We do this by generating the same test template we have discussed in Section 3.5.1:

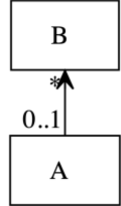

Figure 15: A unidirectional association

Umple Model

```
class A {
  0..1 -> * B;
}
class B{
}
```

Generated Class A

```
  //------------------------
  // MEMBER VARIABLES
  //------------------------

  //A Associations
  private List<B> bs;
```

Generated Class B

```
  //------------------------
  // MEMBER VARIABLES
  //------------------------
```

Code Snippet 20: Directional association reference

We can see from Code Snippet 20 that class B has no reference to class A. Hence, we generated different set of tests for each class. Since our approach is model-driven, the number and type of tests generated are not determined by the content of the generated implementation but rather by

the specification given in the model. If we would generate tests for class B without looking at the model specification, we would have no idea that this class has 'unidirectional' association unless we investigate the content of all other classes. Hence, specific tests for checking association references are generated based on the declarations given at the model level.

*3.5.5 Lower-Bounded and Upper-Bounded Associations: N..N -- **

This type of association is lower and upper bounded with values greater than 1. In the example shown in Figure 16 and Code Snippet 21, we cannot construct A without at least 2 objects of B assigned at construction and cannot add more than 4 objects of B to A. The way we handle this is by providing the basic pattern for associations in Umple in addition more tests to handle boundaries and also handling the construction of objects around the boundary. The test generator analyzes the Umple model to look for this pattern, when this pattern is detected, we add several test cases, Boundary testing and testBoundaryAtMax.

- **Boundary Test**: makes sure that the number of B's is between the lower bound and the upper bound, in this case between 2-4.
- **Test Boundary at Max**: creates objects of the maximum number allowed and adds them to the list, then verifies that no further objects are allowed.

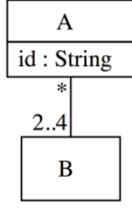

Figure 16: Bounded multiplicity

The resulting abstract test code is shown in Code Snippet 22.

```
class A {
   id;
}

class B {
  2..4 -- * A;
}
```

Code Snippet 21: Umple Model for lower and upper bounded multiplicity

```
File: <modelTest.umpt>
001: …
002:
003: ////---- Tests for  NNToManyAssociation ----/////
004:
005: Test NNToManyAssociation.ump {
006: depend  A,B ;
007:
008:
009:
010:   /*-------------------------*/
011:   /*  Association Test       */
012:   /*-------------------------*/
013:   /*-------------------------*/
014:   /*  Class A       */
015:   /*-------------------------*/
016:
017:               /////// Association: -- [2,4] B ///////
018:               //create ... without ...
019:               Test CreateAWithoutB {
020:               A someA = new A();
021:               AssertFalse (someB.getBS()!= null);
022:               }
023:               //replace
024:
025:               Test ReplaceBInA {
026:
027:               B someB = new B();
028:               B someB2 = new B();
029:               someB.addB(someA);
030:               AssertEqual (1, someB.getNumberOfAs());
031:               AssertEqual (1, someB.getNumberOfAs());
032:               someA.setB(someB2);
033:               AssertEqual (someA, someB2.getA(0));
034:               }
035:               //delete
036:               Test DeleteB {
037:               // delete B in A
038:               someB2.delete();
039:               AssertEqual (null, someB2.getA(0));
040:               }
041:               //boundary test
042:
043:               Test BoundaryTest {
044:               private int size = B.getNumberOfAs();
045:               AssertTrue ( size > 0 &&  size < -1)
046:
047:               }
048:
049:               Test BoundaryAtMax {
050:
051:
052:               //Maximum A allowed : 4
053:               int size = B.getNumberOfAs();
054:
055:
056:               A obj1 ();
057:               A obj2 ();
058:
059:               B someB = new B(obj1,obj2);
060:
061:
062:
```

```
063:              A obj3 (someB);
064:              A obj4 (someB);
065:
066:              AssertEqual(size, someB.getNumberOfAs());
067:              AssertEqual(someB, obj1.getSomeA(0));
068:              AssertEqual(someB, obj2.getSomeA(0));
069:              AssertEqual(someB, obj3.getSomeA(0));
070:              AssertEqual(someB, obj4.getSomeA(0));
071:              }
072:   /*-------------------------*/
073:   /*  Class B       */
074:   /*-------------------------*/
075:
076:              /////// Association: -- [N,*] A ///////
077:              //create ... without ...
078:              Test CreateBWithoutA {
079:              B someB = new B();
080:              AssertFalse (someA.getAS()!= null);
081:              }
082:              //replace
083:
084:              Test ReplaceAInB {
085:
086:              A someA = new A();
087:              A someA2 = new A();
088:              someA.addB(someB);
089:              AssertEqual (1, someA.getNumberOfBs());
090:              AssertEqual (1, someA.getNumberOfBs());
091:              someB.setA(someA2);
092:              AssertEqual (someB, someA2.getB(0));
093:              }
094:              //delete
095:              Test DeleteA {
096:              // delete A in B
097:              someA2.delete();
098:              AssertEqual (null, someA2.getB(0));
099:              }
100:              //boundary test
101:
102:              Test BoundaryTest {
103:              private int size = A.getNumberOfBs();
104:              AssertTrue ( size > 2 &&  size < 4)
105:
106:              }
107:          }
```

Code Snippet 22: Generated abstract test model for bounded multiplicity association

3.5.6 Immutable Unidirectional Associations

For immutable associations, one class must have an association that is directional, and the other class must be immutable. When these two conditions are met, we cover all different cases of multiplicities. Figure 17 shows an immutable association between class B and immutable class A. Also, the multiplicity is Optional-One to Many. This means that we can create an instance of B without any A's. However, because the association is unidirectional, A does not have an association variable for B. Now when we want to test this, we want to make ensure the following:

- We can create instance of B without A.
- We can create B with any number of instances of A's.
- B does not contain any adder or remover method for the list A.

This requires us to look into the code via reflection using Java.lang.reflect in order to match a particular regex to capture adder/remover methods that were not supposed to be generated. The method shown in Code Snippet 23 is abstract and should work with any object when invoked.

```
private boolean objectClassHasSettersAddersOrRemovers(Object obj)
  {
    boolean hasMethods = false;

    Class<? extends Object> clazz = obj.getClass();
    Method[] methods = clazz.getMethods();
    String setterNameRegex = "(set|add|remove)[A-Z]+[a-zA-Z]*";

    for (Method m : methods)
    {
      if (m.getName().matches(setterNameRegex))
      {
        hasMethods = true;
        break;
      }
    }
    return hasMethods;
  }
```

Code Snippet 23: Assert method using reflection

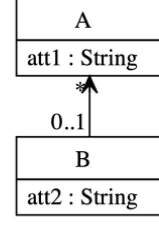

Figure 17: Class diagram for immutable directional optional one-to-many association

```
class A {
  immutable;
  att1;
}

class B{
  att2;
  immutable 0..1 -> * A;
}
```

Code Snippet 24: Umple code showing an immutable class and an immutable association

3.5.7 Sorted Association in Umple

Umple associations can be declared to be automatically sorted based on a given attribute value from the other association end [2]. This will inject several methods related to sorting of the association variable set. This can be declared as seen in Code Snippet 25.

```
class A {
  0..1 -- * B sorted {name};
}
class B{
name;
}
```

Code Snippet 25: Sorted association in Umple

Declaring a class *sorted* will add the following methods to the Umple API:

Table 5: API methods added for sorting

Method name	Description
setBsPriority	This will set the value for bsPriority in case we want to sort according to another value
getBsPriority	This will return the value that is currently used for sorting

When it comes to automatically generated test cases for sorted associations, we add two main test cases: First, a test case that checks the presence of the sorting-related methods in the generated code. 'checkHasPriorityMethods', which checks run the following:

```
test checkHasPriorityMethods {
   A aA ();
   assertMethod(setBsPriority);
   assertMethod(getBsPriority);
}
```

Code Snippet 26: Method assertions for sorted association

This will eventually run the two above checks at the concrete level using code reflection and the supporting test method addressed previously in Code Snippet 12. Secondly, we want to check that objects have been sorted according to the derived value provided in the association declaration.

2 https://cruise.eecs.uottawa.ca/umple/SortedAssociations.html

The declaration of A-B association is declared as sorted using ‘name’ as the value to be prioritized. For such tests we generate the following:

```
test checkBsOrder {
   A aA ();
   B b1 ("Adam");
   B b3 ("Zak");
   B b2 ("Jane");
   aA.addB(b1);
   aA.addB(b2);
   aA.addB(b3);
   assertTrue(aA.getB(0).equals(b1));
   assertTrue(aA.getB(0).equals(b3));
   assertTrue(aA.getB(0).equals(b2));
}
```

Code Snippet 27: Checking sorted association order

The values used for test data are obtained from the *TestGenerator* translation map that will determine the value based on the type of the derived value used for sorting. If the value is ‘String’ then we invoke the test translator with the following call; the arrow represents the returned value by the translator:

gen.translate(“sortedAssociationString1”); ➔ Adam

gen.translate(“sortedAssociationString2”); ➔ Zak

gen.translate(“sortedAssociationString3”); ➔ Jane

In the case the derived attribute used for sorting was ‘Integer’ we invoke the translator with the following calls.

gen.translate(“sortedAssociationInteger1”); ➔ 2

gen.translate(“sortedAssociationInteger2”); ➔ 9

gen.translate(“sortedAssociationInteger3”); ➔ 5

Therefore, we cover the two main aspects we discussed previously using these two main test cases whenever a class has sorted association detected.

3.5.8 Chained Associations

Test generation becomes tougher when there are chains and trees of associations at a deeper level that have ‘mandatory’ multiplicities (1 or higher). In particular, the association 1-1 creates a

paradox in which we run into a situation where in order to create B we need at least an instance of C, however, in order to create an instance of C we also need an instance of B. This type of dependency is handled in a specific way in Umple (creating both objects simultaneously).

The dependency chain in Figure 18 and Code Snippet 28 is hence another case that the test generator must analyze.

In Chapter 9 we will discuss the analysis of the dependencies among multiple classes in a model and how our test generator prepares and handles the pre-analysis of the generation process.

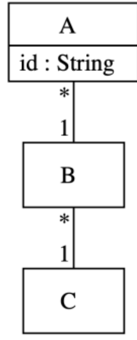

Figure 18: Chained associations

```
class A {
  id;
}

class B {
  1 -- * A;
}

class C{
  1 -- * B;
}
```

Code Snippet 28: Umple Model for testing chained association

There are several issues that need to be addressed when it comes to generating tests for chained associations. Such issues include dependency injection handling and cycles detection. These issues are addressed in Chapter 8 in detail.

3.5.9 Reflexive Associations

Reflexive association is a type of association where a class refers to itself. Declaring a reflexive association in Umple requires using role names to differentiate between the two end objects' names. An example is given in Code Snippet 29.

```
class Course {
  * successor -- * Course prerequisite;
}
```

Code Snippet 29: Reflexive association for class Course

Reflexive association cannot be declared using one-to-many type of associations. This is due to self-dependency which is impossible to satisfy. In order to create any object of class Course, one needs to feed the constructor an object of class Course; this is illogical and creates a paradox at the top of the hierarchy. Hence reflexive associations must be declared either optional-one-to-many or many-to-many.

In addition, the other association end must have a role; otherwise the compiler will raise an error so as to avoid generating two API methods with the same name. In order to test reflexive associations, we use the same templates we used earlier. We always prioritize role name to be used for object instantiation when testing Umple classes. Therefore, test templates will be applied automatically on reflexive association and the compiler will help by adding a layer of validation in order to block the processing of any model that does not meet the requirements.

3.6 Interfaces

Interfaces in UML are class-like elements (classifiers) that cannot be instantiated directly [74], [75], but describe the signatures of a set of methods that must be implemented. Conforming with this, in Umple, an interface represents a list of methods that are to be implemented by any class. When it comes to automatic test generation for interfaces, we do not generate any test case for the interface per se, but we allow generic tests to be added that appear in every implementing class, in the manner to be discussed in Section 7.3.2.

3.7 Abstract Classes

Abstract classes are meant to define properties of a particular entity without allowing the user to directly instantiate the entity [76]. The rationale behind using abstract classes is to introduce reusability of elements and reduce redundancy of attributes and other features. Umple supports 'abstract' classes by allowing the use to declare any Umple class as 'abstract'.

```
class Person {
  name;
  id;
  abstract;
}
```

Code Snippet 30: Abstract Umple class

Umple generates abstract 'classes' in the implementation language based on which platform is targeted for code generation. Hence, our goal when it comes to testing abstract classes is to generate an abstract test class for the abstract Umple class. For the model demonstrated in Code Snippet 30, we generate the following abstract model:

```
abstract Test Person {

generate JUnit ;

 GIVEN:
 model.ump;

 THEN:
  //tests for :name
      … omitted
 //tests for :id
      … omitted
}
```

Code Snippet 31: Generated abstract test class

When a test class is tagged 'abstract' in the test language, it will be generated as an abstract test class. In this case, this will be generated in jUnit as 'abstract public class PersonTest'. In the case where we have another class 'Student' that extends the Person class then the declaration of the Student test class will include the following: 'public class StudentTest extends PersonTest'. Hence, when we run the StudentTest class, the two test cases in PersonTest:attribute_name PersonTest::attribute_id will be invoked.

The other issue that needs to be addressed is whether the abstract class *Person* contains any user-defined tests at the model level. This issue is explored further in Section 7.3 later in the thesis.

3.8 Inheritance

Each class in Umple is dealt with separately when it comes to test generation. This means we include any test case for each class separately and then finally analyze whether the test class needs to be declared in a specific manner to address issues such as inheritance or implementation. In the case where we have a regular class that has a parent class, we include the following keyword in its test class declaration: ‘extends ParentTest’. Like we mentioned above in Section 3.7, invoking a child test class will eventually call the parents tests too regardless whether the parent class is abstract or not. However, In the case where a model has inheritance while the parent classes do have embedded tests, then we handle this differently. Handling tests embedded in the Umple model for model class inheritance is discussed in detail in Section 7.3.

3.9 State Machines

To avoid over-extending the thesis, the work in this thesis does not cover the automatic test generation for state machines. Therefore, when the model is compiled using the test generator it will not include any tests in the abstract test model for state machines. The literature has a decent number of studies and tools those target the automatic test generation for state machine. The work by [77] [78] [79] discusses the automatic generation of test data and test cases from UML state machines. Also, tools like [47] [62], [78] discusses model-based approach to generate test specification from UML state machines. Including automatic test generation for state machines in Umple would over-broaden the focus of the thesis. Hence, with the recommendation of the committee, we decided not to include state machines for the automatic test generation.

However, state machines can still be tested using the user-defined tests embedded in the Umple model or using an external test-generation tool for state machines. This still requires the user to compose more specific tests rather than a systemic solution; more explanation on how to test state-machines using the Umple testing language is given in Chapter 4.

3.10 Test Data

The data we use for testing are saved within the test generator using the translator pattern. When we write the templates, we refer to the value with a key, and always retrieve the expected result for testing. This way we can draft the test template and oracle knowing the value that had been used for the initiation of these objects beforehand. The test data map can be overridden by the values of certain keys. This is useful if we want to provide an interface to allow for the test data to be retrieved from an external source. Also, in some cases, we can trigger the randomly generated values using the test utility class that will provide a random value based on a given type.

3.11 Coverage Criteria:

Coverage criteria are predefined heuristics that aim to determine what specific aspects of the system are focused on by testing [80]. A typical Umple model has many elements including classes, associations, attributes, state machines and possibly advanced features such as traits, tracing or composite structure.

Testing such models can be a time-consuming task as test developers may come up with large number of tests from a simple model with a weak plan [81]. Unless testing is driven by predefined coverage criteria, there are likely to be unnecessary tests that don't contribute to the enhancement of the quality assurance of the system.

For our automatic test generator in Umple, we allow the user to choose between several coverage criteria, where each represents a specific type of Umple element; in addition to the default coverage which is comprehensive (i.e. generate tests for all the defined coverage criteria).

There are many types of coverage criteria in use today in standard programming contexts, ranging from structural coverage criteria to data-flow coverage criteria [12], [70].

In Umple, we select from criteria that serve the developer from the modelling perspective. The work by Erriksson and Lindstrom [82] discusses the test coverage of UML associations and lists several elements that identify the characteristics of associations in UML. Our work in Umple deals with all of them. They also discuss important coverage types for associations such as: navigational coverage, logic-based and iteration coverage. The following is the list of coverage criteria that are currently developed and will be expanded as the system develops:

Umple default: With this coverage criteria set, Umple call upon all available test patterns to be generated for a given model.

Associations: When this is specified, Umple only generate tests related to associations, including multiplicity tests. Currently, the compiler interprets the coverage 'association' to request coverage of all aspects of associations. However, we later intend to include the following additional, more specialized coverage types:

- One-To-Many
- Optional-One-To-Many
- Many-To-Many
- Unidirectional

Attributes: With this, Umple only generates tests for attributes including design patterns for attributes such as: Lazy, Immutable and Autounique.

Methods: With this, Umple only generates tests for user-defined methods. This may include user-defined assertions within methods.

Custom: This will only generate tests that had been defined by the developer within the Umple model. This includes manually written test cases and user-defined assertions within Umple methods.

Coverage criteria can be dictated in 'generate' statements using the Umple '-suboption' keyword on the command-line or in the model. The '-suboption' keyword is a feature that had been implemented in Umple by former developers which allow an Umple developer to pass the compiler optional parameters to control code generation for certain features (the feature here being test generation). To handle test coverage criteria, we pass the name of the desired coverage type as a suboption as the following:

```
generate test –s "attribute" ;
```

Code Snippet 32: generating tests based on a given coverage criteria.

When this is passed to the Umple compiler, the compiler will simply add these sub-options to the Umple model as part of the element ‘generate target’. When we process the template to generate test code, we look for the ‘generate target’ objects within the Umple model instance to check if there are any coverage criteria indicated by the user. The templates are defined in separate files that can be included in the main directive using the template inclusion statement; which helps in the pluggablity of features in the test code. Hence, we do a check if the model has coverage criteria and based on that we include the desired template and exclude the ones not in the coverage criteria.

The following code shows how this is handled within the test templates:

```
for (GenerateTarget tr : model.getGenerates() )
        {
                if (tr.hasSuboptions())
                {
                        coverageCriteria = tr.getSubOption(0);
                }
        }
```

When we assign the value to the coverageCriteria attribute, we then handle the inclusion of the template as shown in Code Snippet 33.

```
..
 if (coverageCriteria.equals("")){#>>
  <<@ UmpleToTest.members_AllAttributes >>
  <<@ UmpleToTest.members_AllTestCases >>
  <<@ UmpleToTest.members_AllMethods >>
  <<#}
 if (coverageCriteria.equals("attributes"))
 {#>>
  <<@ UmpleToTest.members_AllAttributes >>
  <<#}
 if (coverageCriteria.equals("methods"))
```

```
{#>>
 <<@ UmpleToTest.members_AllMethods >>
 <<#}
…
```

Code Snippet 33: Handling template inclusion based on coverage criteria

When state machines are covered in Umple test generation, this will result in extra options for coverage. We would look into the work discussed by Friske and Schlingloff [83] to add state-machine test directive in order to expand the coverage criteria. The test templates for the state machines had been created initially but left out when the decision to omit it from the work in the thesis was taken during the proposal of the thesis. We can still in future toggle the coverage criteria in case this work is to be expanded to cover these aspects, which may include the option to choose specific structural coverage such as: all-transition, all-state, and so on. The other options would be to consider more behavior coverage as discussed by Offut [12].

3.12 Summary

In this chapter we have presented how we incorporate model-based testing in Umple in general. We have discussed what elements from an Umple model are used to automatically generate tests. The automatic test approach covers associations, attributes and other features of Umple. We have also discussed automatic test generation for interfaces, abstract classes and parent classes. In addition, we have explained the different types of coverage that can be specified to narrow down the type of elements being targeted for tests in a given Umple model.

Chapter 4 Umple Testing Language

In this chapter, we present in detail the testing language and model we have created to represent tests at the abstract level. This language is generated by our automatic test generator, and also can be written manually by developers, either as standalone tests, or as tests embedded in methods.

Being abstract, it should allow the developer to focus on the logical issues of the tests rather than struggling with low-level technical difficulties often faced when using testing platforms. This language aims for expressiveness, consistency and coherence. Figure 19 shows the metamodel of the language as it stands. The figure is generated by Umple, which is possible because all our software is written in Umple.

The inspiration behind the syntax of the language comes from several sources. The main syntax of the unit testing comes out as an abstraction of widely-used unit test systems such as JUnit, PhpUnit and others. Other modeling related syntax such as the division of the language blocks using *given*, *when*, *then*, etc. is inspired from behavior-driven development tools such as Cucumber, RSpec [84] and JBehave [85]. Some high-level model-oriented syntax has been drafted so as to effectively fit with other aspects of the language.

The main element of the metamodel is the test model that contains a number of test suites that may have a number of test cases. The class *TestCase* is a core element in the metamodel. It is used heavily through the propagation phase of the test model.

4.1 An Overview of the Umple Testing Entities

The test-specific language we have designed consists of a number of elements. Each element represents a specific concept in testing. By using a combination of those according to the correct syntax, the developer can implement their test model. Here, we present the list of elements that compose our testing language.

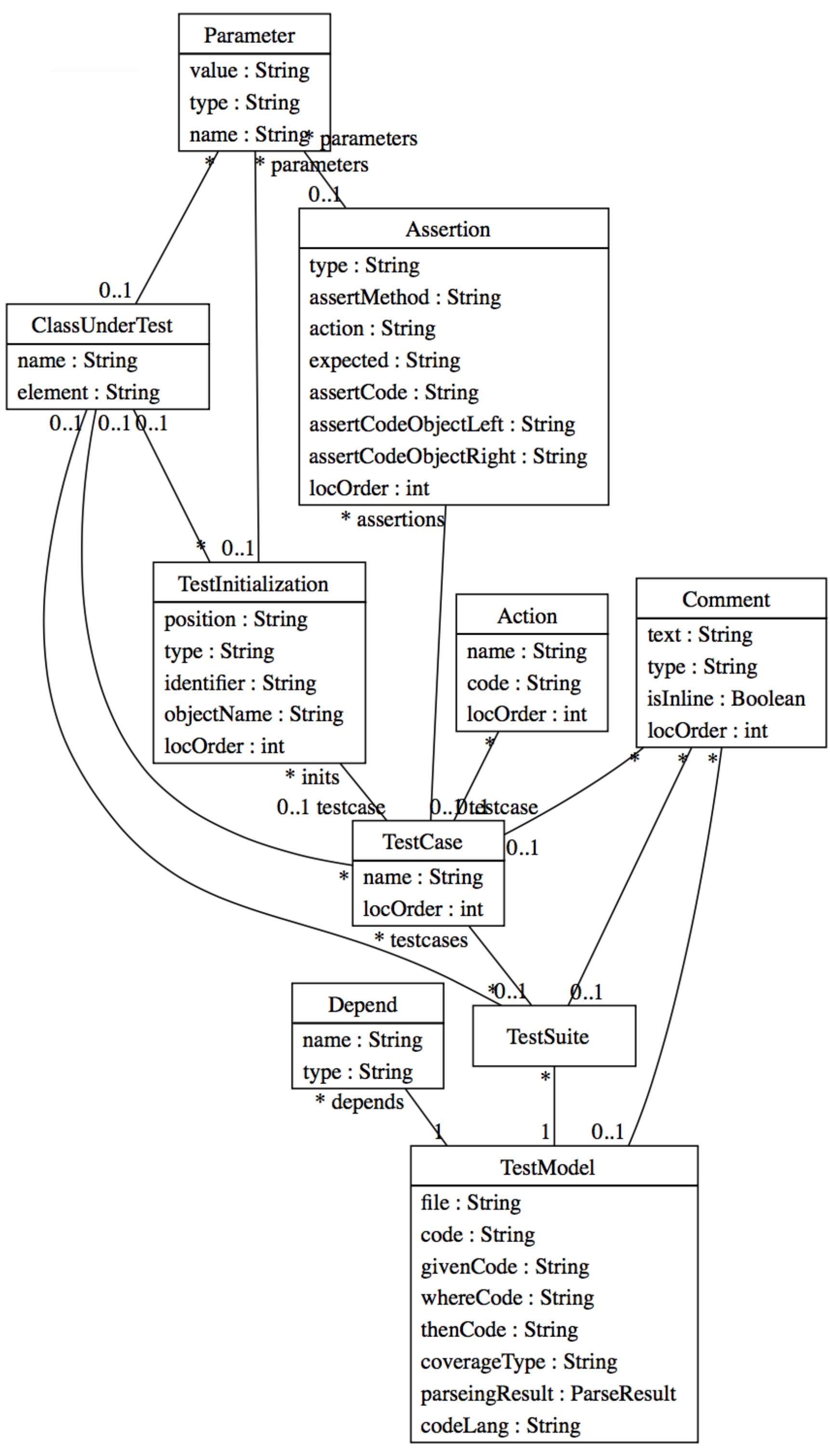

Figure 19: Metamodel for Umple test model and language

4.1.1 Test Model

A test model is a representation of abstract elements and specifications of tests written in order to encapsulate what is to be tested from a given system. When we defined the test model, we took into account the assumption that the developer (or tester) is interested in the following aspects:

- The developer should see encapsulation of entities and what classes are being tested
- Focusing on the test logic without getting distracted by other details
- Clear separation of different part of test code: initialization, dependency and testcases
- Clear separation between test cases and avoiding overlapping
- Sequence of events to identify a test sequence

The point of having an abstract test model is to separate test logic in order to focus on objectives of tests and avoid struggling with technical and concrete details. It also helps in understanding the test code and observe changes.

In the following explanation we give elements of the Umple grammar for tests. The language we use is the language the Umple Parser uses natively and is a form of EBNF. It should thus be broadly understandable by readers familiar with other EBNF dialects. For clarity, ? means the preceding element is optional (zero or one of them may be found), * means zero or more may be found. Double square brackets surround references to other non-terminals. Single square brackets refer to terminals. A tilde (~) after the open square brackets means the terminal is an arbitrary alphanumeric identifier. An equals sign (=) after the square bracket specifies specific strings. Full details of the grammar notation can be found at http://grammar.umple.org.

Table 6: Test model elements

Element	Description	Syntax

Test Class	A test class written for test purposes. Often contains the keyword ‘Test’ as postfix.	test <class> {
		…
		}

Depend	Dependent classes	depend <class> , … ;
Given	Contains model files used to generate this test model	Given: <modelFile>.ump ;
When	Lists initializations at the class level; in particular to set up objects for testing using sequences.	When: Person p1 (…) ;
Then	All test cases and assertion	Then: <testcase> …

Testcase	A test method in a unit test system, targeting a specific behavior of the system.	test <testMethod> {
		…
		}

Assertion	An assertion statement to evaluate whether the input is true or false.	assertTrue, assertFalse, assertMethod, assertequal …
Action	An event triggered, such as a method call or an assignment.	

Code Snippet 35 is an example a test case from the test model/language that is testing the Student-Mentor Umple file (Code Snippet 34, Figure 20).

The keyword *Test* indicates a test case declaration followed by the name of the test case *StudentMentor*. *Depend* is a keyword referring to a dependency on other classes; in this case, we are dependent on class Student and Mentor. This will map to ‘import’ for Java. The content of each test case is divided into three main blocks:

- **Given**: Any given input to the test model, in this case an Umple model is given as an input. There are two ways of declaring an Umple model within the GIVEN block, either by typing the actual model definition or by typing the Umple file name for example ‘Mentor.ump’. The grammar is:

```
givenCont: (GIVEN:)? [[givenUmpleModel]]*
givenUmpleModel : [~modelName].ump;
```

- **When**: Contains any initialization made to objects in order to setup the testing environment. The grammar is:

```
whenCont: (WHEN:)? [[initialization]]*
```

- **Then**: Contains the list of testcases. Each contains a number of the following wrapped in a testcase code block:
  - **Testcase:** which represents a set of initialization, actions and assertions.
  - **Initializations:** these are statements that aim to construct objects and initialize them at the test case level.
  - **Actions:** actions are any statements injected in the test case such as assignments, method calls.
  - **Assertions:** Assertions are part of the test case entity and they represent statements of expected results that should be satisfied.
    - AssertTrue
    - AssertFalse
    - AssertNull
    - AssertEqual
    - AssertMethod
    - AssertAttribute

  The grammar is:

```
thenCont: (THEN:)? [[testCase]]*
testCase: [=abstract:abstract]? [=isConcrete:concrete]?
          [=isTargeted:JUnit|PhpUnit|RubyUnit]? test [~testCaseName] {(
          [[initialization]] | [[assertion]] | [[testInitAtt]] | [[comment]]
          )* }
```

```
File: <Model.ump>
1: class Mentor {
2:  name;
3:  id;
4: }
5: class Student {
6:  name;
7:  id;
8:  0..2 -- 1 Mentor; //multiplicity
9: }
```

Code Snippet 34: Umple Student-Mentor example

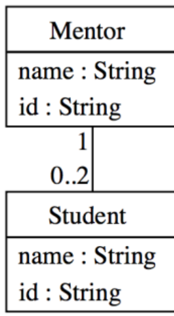

Figure 20: Student-Mentor example diagram

```
Test Model_ModelTest{
            generate JUnit;

       Depend Mentor, Student;

       GIVEN:
              Model.ump

       THEN:
            …..
            Test ReplaceMentorInStudent {
              Mentor someMentor ();
              Mentor someMentor2 ();
              Student someStudent ();
              someMentor.addStudent(someStudent);
              AssertEqual (1, someMentor.getNumberOfStudents());
              someStudent.setMentor(someMentor2);
              AssertEqual (someStudent, someMentor2.getStudent(0));
              }
           …….
}
```

Code Snippet 35: Test model vode, Student-Mentor example

When generating tests for the above model, Umple generates three test models from the *model.ump* file,

1) A test model that focuses on testing the elements from a broader view of system such as associations, multiplicities etc. This file is usual name as the following ‘xx_ModelTest.umpt’ where ‘xx’ represents the name of the model file; in our example it should be ‘Model_ModelTest.umpt’;

2) A test model for the class ‘Mentor’ as ‘MentorTest.umpt’, and

3) ‘StudentTest.umpt’ for the class ‘Student’. Class-specific test models focus on the internal elements of the class. Those don’t necessarily require interaction with other classes, such as attributes, state machines, methods and class user-defined test cases.

4.2 The Test Model/File Generation Process

The workflow of test file generation can be summarized in Figure 21. As seen in the figure, Step 1 shows the input to the Umple compiler – an Umple model with extension *.ump*. The compiler then processes the file and based on the number of classes it will determine the number of files to be generated.

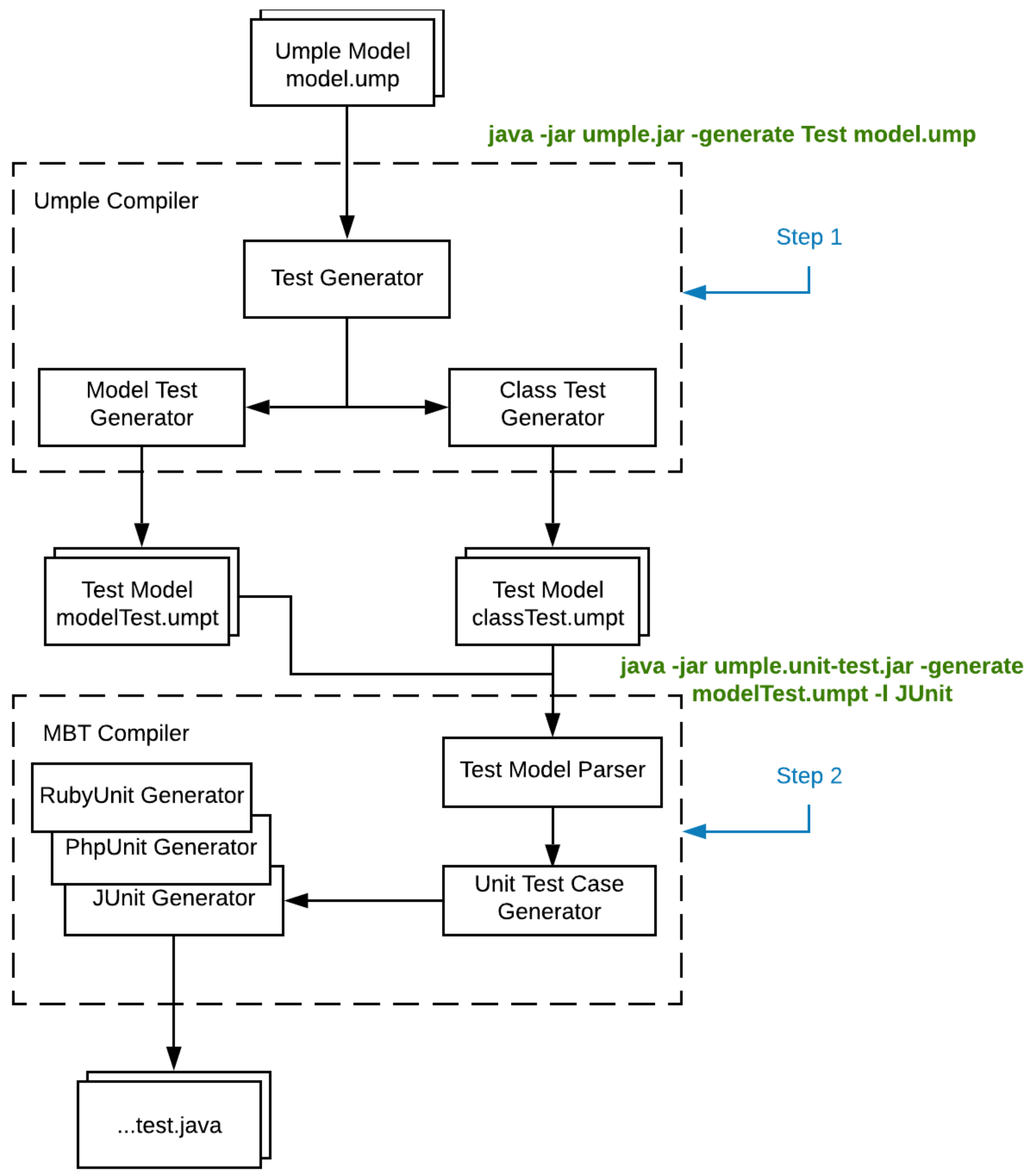

Figure 21: Test Generator output workflow

The compiler will always generate a test file for the Umple model file, this file includes the tests that are related to issues at the model level. In addition, a test file is created for each class that has tests that correspond to the issue at the class level. These tests do not have any dependencies on other classes. Step 2 shows the test model files that have been generated from the Umple compiler as an input to the MBT Compiler. These model tests will be processed by the MBT compiler and then directed to the corresponding unit generator based on the language specified

either on the command line or in test model file. Ultimately, we should have the concrete and executable unit test files. Figure 21 shows a Junit file as a demonstration of unit test generation.

As seen in the figure, Step 1 is invoking compiler using the ‘umple.jar’ tool while Step 2 is executing ‘umple.unit-test.jar’ as the tool for xUnit generation. Finally the execution of the concrete test files can be done either manually on the command line using corresponding unit language system tools or using the system build tool such as ‘Gradle’ ‘Ant’ or ‘Maven’ . In case a user is using JUnit, tests can be executed using the following command line *java org.junit.runner.JUnitCore* [86] [87]. The latter option requires to write the test running tasks in the build files which can be invoked later in the command line. We have upgraded our templates to support JUnit5-jupiter and JUnit5-jupiter-api. If the tests are compiled using JUnit5 console launcher then it will automatically detect test containers in the folder. Therefore, even if a manually written concrete test is written using JUnit4, backward-compatibility can be achieved when using the JUnit5-console-launcher. Further issues such as backward-traceability is discussed in Section 7.6.

In order to make test execution more efficient, we have written a template for a JUnit test runner that can be used in order to generate test class runner. Table 7 shows how we used the Umple *mixin* feature in order to write test runner templates within the ‘Utility’ class that is part of the test parser project.

Table 7: Test runner templates and utility

Filename
Utility_TestRunnerTemplate_JUnit.ump
Utility_TestRunnerTemplate_PhpUnit.ump
Utility_TestRunnerTemplate_RubyUnit.ump
Utility.ump

The ‘Utility’ class has a list of supporting methods that can support the analysis and processing of tests. For instance, looking at the file ‘Utility_TestRunnerTemplate_JUnit.ump’, it has the code as seen in Code Snippet 36.

```
File: <Utility_TestRunnerTemplate_JUnit.ump>
01: namespace cruise.umple.testgenerator;
02:
```

```
03: class Util{
04:              emit    getTestRunnerTemplateJUnitCode    (List<String>    classes)
(TestRunnerJUnitGenerator);
05:    // Template for JUnit Test Runner
06:    TestRunnerJUnitGenerator <<!<<#
07:    String classesCode = "";
08:    for (int x = 0; x < classes.size(); x++)
09:    {
10:       if (x == 0)
11:        {classesCode+= classes.get(x)+".class";}
12:        else
13:        {classesCode+=", " +classes.get(x)+".class";}
14:    }
15:
16:    #>>
17:    import java.io.File;
18:    import org.junit.After;
19:    import org.junit.Assert;
20:    import org.junit.Before;
21:    import org.junit.Test;
22:    import org.junit.internal.TextListener;
23:    import org.junit.runner.JUnitCore;
24:    import org.junit.runner.Result;
25:
26:     public class TestRunner {
27:      public static void main (String[] args)
28:      JUnitCore  junit = new JUnitCore();
29:      junit.addListener(new TextListener(System.out));
30:      Result result = junit.run(<<=classesCode>>);
31:      resultReport(result);
32:     }
33:
34:     public static void resultReport(Result result) {
35:              System.out.println("Finished. Result: Failures: " +
36:                result.getFailureCount() + ". Ignored: " +
37:                result.getIgnoreCount() + ". Tests run: " +
38:                result.getRunCount() + ". Time: " +
39:                result.getRunTime() + "ms.");
40:          }
41:     }
42:   !>>
43: }
```

Code Snippet 36: Test runner template

Code Snippet 36 shows how we feed the template with the class name as in <<classesCode>>. ‘classesCode’ refers to the composed line of code that has the name of classes under test. The result of running this test runner should report the following:

```
Time: 1.618

OK (65 tests)

Finished. Result: Failures: 0. Ignored: 3. Tests run: 65. Time: 1618ms.
```

When the developer calls the method *getTestRunnerTemplateJUnitCode* and passes the list of all the class names to be included in the test runner, then they will be included as parameters for the junit.run call. Hence, all these classes will be invoked and the result will be reported as *resultReport* gets called at the end. Currently, we make this call from the *JUnitTestCaseGenerator* to get the test runner code generated as well.

Currently, we focus on the test runner for JUnit. However, the templates for RubyUnit and PhpUnit have been set up and can be integrated.

4.3 Grammar

The complete grammar for the Umple abstract test language is shown in Code Snippet 37.

A test model file can have an entity to start with, which represent in our case one modelDefinition or more in the same file. A model can have comments of two types, inline and multiline comments.

A definition of a test model starts with the keyword *test* followed by a model name and the content of the model which can be briefly summarized as, "Given …, "When …, Then …". The testcase component is the core element of the Umple test model, encapsulating key parts of the language. We have included a visual representation of the test case rammar using railroad diagrams to enhance the readability of the grammar as seen in Figure 22.

The Umple grammar language uses *OPEN_ROUND_BRACKET* and *CLOSED_ROUND_BRACKET* to represent certain terminals that would otherwise be interpreted as grammar metacharacters; however, have replaced these two in this thesis by the actual terminal value '(' in order to enhance readability and reduce the text wrapping within the code snippet box.

The *Generate* statement is a test directive that has a codeLang attribute set based on the chosen target unit test platform. The generate statement can also have zero or more suboptions with the indicator *-s or -subOption*.

```
File:../mbt_parsing.grammar
02: // The main top level elements to be found in an Umple file
03: entity- : [[modelDefinition]]
04:
05:
06: // test case are the most common elements in Umple Test Lanagueg.
07:
08: modelDefinition- : [[comment]]* [=isAbstract:abstract]? test [modelName] { [[modelContent]]
                      } [[comment]]*
09:
10: // The following items can be found inside the body of classes or association classes
11: modelContent- : [[generate]]? [[depend]]* [[givenCont]]? [[whenCont]]? [[thenCont]]?
12:
13: generate: generate [=codeLang:JUnit|PhpUnit|RubyUnit] ( [=subOptionIndicator:-s|-suboption]
     " [**subOption] " )* ;
14:
15: depend: depend [pValue] (, [pValue])* ;
16:
17:
18: givenCont: (GIVEN:) ( [[givenUmpleModel]] | [[comment]] ) *
19: givenUmpleModel : [~modelName].ump;
20:
21:
22: whenCont: (WHEN:) ( [[initialization]] | [[comment]] )*
23: thenCont: (THEN:) ( [[testCase]] | [[comment]] )*
24:
25:
26: testCase : [=abstract:abstract]? [=isConcrete:concrete]?
           [=isTargeted:JUnit|PhpUnit|RubyUnit]? test [~testCaseName] {( [[initialization]] |
           [[testAction]] | [[testInitAttWithMethodCall]] | [[assertion]] | [[testInitAtt]] |
           [[comment]]  )* }
27: initialization : [~identifier] [~objectName] '('[[parameter]]* ')'  ;
28:
29: testAction : [~objectName] (.) [**code] ;
30:
31:
32: testInitAtt : [~identifier]? [~attributeName] =  [[pValue]] ;
33: testInitAttWithMethodCall : [~identifier]? [~attributeName] =  [~object](.)[~call]
                               '(' [[parameter]] ')' ;
34:
35: testInitAtt : [~identifier] [~attributeName] =  [**code];
36: attMethodCall : [~object] (.)? OPEN_ROUND_BRACKET  CLOSE_ROUND_BRACKET
37:
38:
39:
40: parameter : ([pValue] | "[pValue]" )(,([pValue] | "[pValue]"))*
41: pValue- : ( [~name] | "[~name]" )
42:
43:
44: assertion : [=assertType:assertTrue|assertFalse|assertEqual|assertNull|assertMethod]
              '(' [[assertCode]] ')' ;
45:
46: assertEqualCode :  [~value1] , [~value2]
47:
48:
49:
50: assertCode : [[methodCall]] (== | != )? ( ( [**compValue] | ( "[**compValue]" ) )? |
               [[methodCall]] )?
51: methodCall : ([~objectName](.))?[~methodName]  ('(' [[assertParameter]]* ')')? (;)?
52: assertParameter :  ([pValue] | "[pValue]") (,([pValue] | "[pValue]"))*
53:
54:
55:
56: // Comments follow the same conventions as C-family languages. [*UmpleComments*]
57: comment- : [[inlineComment]] | [[multilineComment]] | [[annotationComment]]
58: inlineComment- : // [*inlineComment]
59: multilineComment- : /* [**multilineComment] */
```

Code Snippet 37: Current language grammar

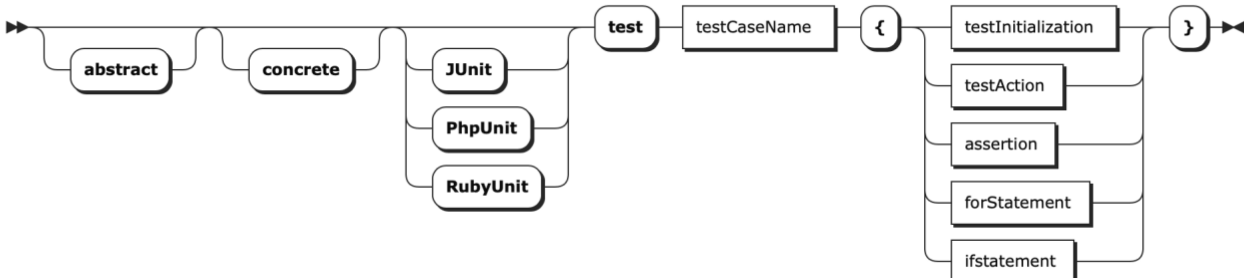

Figure 22: Visualizing the grammar for a single test case

4.3.1 Some Challenges in the Writing of the Grammar:

We encountered several issues when writing the grammar for the abstract testing language syntax.

For instance, when we first drafted the grammar we were planning have the *assertionCode* as an attribute of this type *[**assertionCode]* the way Umple handles this type of non-terminal by creating a value in the parent token with name 'assertionCode' and assigning the code to this value. The issue arises when the assertion code has brackets in its value. Brackets interfere with the next terminal in the grammar rule; which is the closing bracket defined for the preceding assertion. To handle this, we had to parse the assertion code to distinguish between the two brackets corresponding to the assertion. This required more grammar rules to be written.

There were several other options to consider. For instance, we could have handled the test code as one non-terminal. But since the code would then not be fully parsed it will not be checked for errors. Therefore, the tradeoff would be the lack of validation of the test code. In order to avoid this, we parsed the test code and assertion parameters to validate whether the syntax is correct and also to enhance the system notification by sending warning and error messages to the end user. In addition, we had to distinguish between the different assertion types, for instance, *assertEqual* had to be parsed differently than other assertions due to the fact that its sub-tokens had to be handled differently for each unit generator.

Another issue was the translation of an object's name and assertion code from the abstract model to concrete unit tests, this is discussed next.

4.3.2 Integrating the Test Language Syntax into Umple

In addition to the Umple Test Parser grammar shown above, we have also modified the core Umple grammar in order to parse the syntax for the user-defined test cases and assertions within the Umple model. The new testing rules were added to the *umple_classes.grammar* file where we have added a new rule for *testCase*, assertion and *testAction*; more grammar rules will be added as we expand the testing syntax for Umple models. We have also created several entities within the Umple core model such as *UmpleTestCase* , *UmpleAssertion,* and *TestAction* .

The Umple class content can have three test elements; these are: *testcase*, *testSequence* *testInit* and the *generic tests*. These have been integrated as part of the Umple class body. Note that the omitted code has already been introduced in Code Snippet 37 and therefore we are only presenting the main modification and changes to the Umple grammar The following modification had to be made:

```
classContent- :
    …omitted
    | [[testCase]]
    | [[genericTestCase]]
    | [[testSequence]]
    | [[testClassInit]]
    | … omitted

…omitted


genericTestCase : generic test [~testCaseName] '(' [[genericElement]] ')'
                  { ( [**code] )* }
genericElement- : [elementType] [=genericElement:attribute|method|association]
                  [[elementFix]]?  [[genericMethodParameters]]?
genericMethodParameters- : [~elementType] ( , [~elementType] )*
elementFix :  (.) [=fixType:prefix|suffix|regex] '(' [fixValue] ')'
```

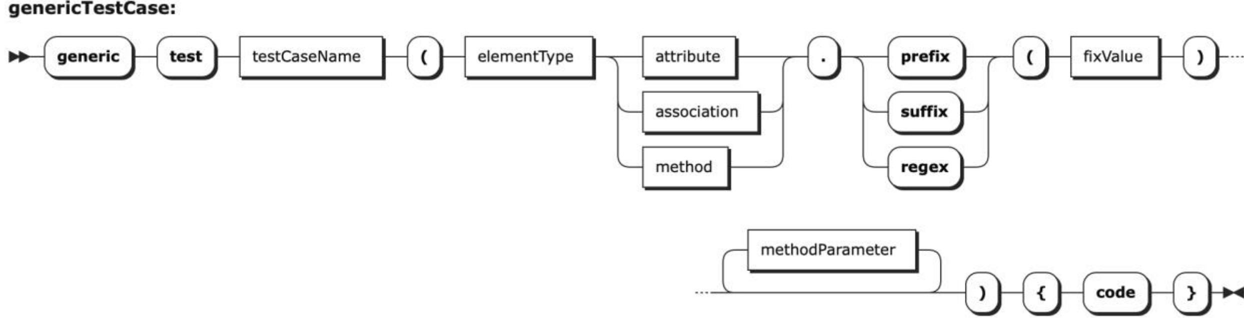

Figure 23: Visualizing generic test in Umple

TestModel is the core entity of the test metamodel, however, we do not define *TestModel* entity within the Umple metamodel since we consider *UmpleModel* as the main entity for tests when injecting tests within the Umple model. Hence, whenever test syntax is parsed within an Umple model, the tokens are generated and analyzed in order to create instances of testing elements within the Umple model. Eventually, when we run our abstract test generators *TestGenerator.ump* and *ModelTestGenerator* and *ClassTestGenerator*, test code is then integrated with the abstract test models.

4.4 Translating of Abstract Test Code to Different Unit Test Systems

One of the issues we had when analyzing the tokens and creating the instances for each element in the model (and based on the grammar we have written) was the translation of specific operators such as accessor operators for objects.

For instance, the way we handle accessors in the test model is using the '.' accessor. To access a set method in object *p1* we would write *p1.setId(..)* for example. This, however, does not work with all unit test systems such as PhpUnit which uses a different accessor similar to C++ '->'. Therefore, we had to handle these special cases before passing these instances to the template for generation by translating the accessor value based on the value of *codeLang* which represents the chosen unit test system in the 'generate' statement.

Another issue that required more processing before passing instances to the template was the translation of the object equals methods for RubyUnit. The way Ruby compares objects is different from Java and Php and therefore it is different from the one represented in the abstract test model syntax. To compare objects p1and p2 in Ruby, ideally, one would use the following syntax p1.equal?(p2) and in our case we wanted to compare two members of these objects we would generate p1.get_id.equal?(p2.get_id). It definitely required more processing in order to get the code generated correctly.

In addition to the equal operator for Ruby, the name of the method is different than other Umple generated systems. Umple generate Ruby code that respects the coding convention of the language, therefore, when generating Ruby's API, the naming is different than that of Umple (self), Java,

and Ruby. Therefore, a specific translation had to be done in order to make sure the names of the methods and attributes for Ruby matches the generated code from Umple.

4.5 Summary

In this chapter, we have explored the Umple testing language in terms of structure and syntax. We have presented the language elements as well as the test model. We also discussed the test generation process from Umple models as the first input to the Umple compiler, then the output of the Umple compiler as an input to our testing compiler (MBTCompiler). We addressed the different platforms the test generator supports and what kind of utilities to be generated as supporting classes for the test execution.

We have also included the grammar for the Umple test language. We have visualized some of the core elements of the testing language using railroad diagrams. The grammar discussion included two places where the grammar was used, the independent test language compiler and also the extension to the Umple grammar as well.

Chapter 5 User-Defined Model-Based Tests

In Chapter 3 we discussed what elements of the model we are testing, and in Chapter 4 we talked about the test language, which can be automatically generated. In this chapter we talk about situations where the developer wants to write their own tests using the language.

5.1 Basics of User-Defined Tests

User-defined tests are test cases written in our Umple abstract testing language that are defined by the developer in the model in order to test anything in the model that they want. The difference between user-defined tests and automatically generated tests is described in the following paragraphs.

First, automatically generated tests are generated based on pre-defined templates written using the Umple Template Language. In the case of automatic test generation, the user has no control over what the tests look like, but can control the coverage of these tests (which subset actually run). For instance, if the user has chosen to automatically generate tests for *lazy* attributes, then they cannot modify the template of the tests to be generated for that particular stereotype. On the other hand, a user-defined test might be written to test a lazy attribute in a specific way, perhaps in combination with other model aspects. When the compiler processes the model, it will generate the user-defined tests within the abstract test model where both user-defined and automatically generated tests are treated the same way.

Secondly, user-defined tests were intentionally integrated in Umple in order to cover areas where the automatic test generation mechanism might prove insufficient. This includes cases where the developer has added user-defined methods (arbitrary code); the automatic test generation mechanism will not be able to digest these, so user-defined tests can be a good tool to fill the gap, while still referring to model-level entities. The fact that the user can have the method code and the tests in the same model will make testing the method more comprehensible.

For tests embedded in methods, there are several challenges that we need to tackle in order to make sure testing code is not confused with the method code during parsing. Since Umple does not parse the method body's code and in fact it is output with only minor transformation in the final production system's code. The challenge came from writing the Umple grammar rules such

that they recognize the abstract test code distinctly from the method body, while making sure it does not appear in the final platform-specific code. The best way to handle this was to allow the user to write their own assertions only at the very top of the method body. We will discuss how this is handled in detail next.

5.1.1 Why Support Both Mixin Test Code and Model-Integrated Test Code?

Support for these two ways of organizing tests is valuable because it facilitates the aspects of test-driven modeling discussed by Antonia [9] in 2007 as a 'dream' for testing. She argues that tests should come from requirements rather than the model. Implementing this feature should allow the user to ensure their design meets the requirements by writing the test code in the model wherever needed this feature is demonstrated in more detail in Chapter 6 .

In addition, Umple supports several platforms for code generation, such as Java, Php, Ruby, etc. Writing tests for each platform for the same model is a redundant task and can be time consuming considering the size of typical systems. Providing our abstract test language can help in reducing the time and cost by allowing generation for different unit test systems from the same design model. However, for user-defined methods, the developer using Umple can provide platform-specific code based on the generator(s) used. In fact, Umple allows the user to provide the same method in more than one target language. The ability to embed tests in methods allows the user to write their abstract assertions within methods which will eventually be output for the desired platform.

The mixin strategy for organizing tests also can be of benefit: The developer can, for example, create three separate test files for a single model called *model.ump*. He or she would then choose the test file to include in the model based on which implementation language to be generated for that model. Code Snippet 38 shows how one would use a mixin in order to include test files based on the generated platform. Here, the targeted language is Java, hence we include the JUnit tests by adding the statement *use modelTest_JUnit.ump;*. In case a different platform is targeted such as Php, we would then comment out the 'use' statement and add one such as *use modelTest_PhpUnit.ump;* or the appropriate file could be included from the command line. Therefore, we can control the activation of the desired templates as needed in order to either target different platform tests or try a different set of test cases.

file: model.ump		

use modelTest_JUnit.ump;
generate Java;
 	
class Student {//class code}
//model code omitted		

modelTest_JUnit.ump	modelTest_PhpUnit.ump	modelTest_RubyUnit.ump

class Student {
//JUnit testcases
}
	class Student {
	//PhpUnit testcases
	}
		class Student {
		//RubyUnit testcases
		}

Code Snippet 38: Using a mixin to target different platforms

This will allow the user to separate the model code from test code. Hence, Umple now can allow the user to write the following options in the Umple file:

- Pure model: no method code or test code
- Embedded code in methods: platform-specific code such as Java
- Test code: which is considered as modelling code but still has different perspective than typical Umple model code.

5.1.2 Related Work

Related work has been done in Eiffel with *design by contract* as discussed by Jézéquel [88]. They proposed the notion of contract between the routine and the caller which requires some precondition/postcondition to be fulfilled. However, such contracts are evaluated at runtime with the final produced system. This is different from how Umple handles conditions and model assertions, Umple already has a mechanism to handle pre/postcondition and invariants which generate assertion statements in the final running system. However, this is not the same as testing, since tests are run in a separate testing phase, not every time a method is run as is the case for preconditions, postconditions and invariants. Full sets of tests also require construction, population of data, and working with cases from many equivalence classes.

An Umple user can define test assertions and testcases manually in two different ways:

- Add test assertions into user-defined methods
- Add testcases into Umple classes

5.2 *Handling User-Defined Methods in Umple Tests*

User defined methods are an aspect in Umple that allows the user to write their own methods that will be then generated essentially as-is in the platform-specific code, with only limited adjustments such as the addition of preconditions or tracing.

When it comes to testing user-defined methods, we would like the option of specifying model-level tests (i.e. tests that refer to model elements) that relate to specific methods. We would want to be able to specify these in the conventional manner, separate from methods. But it would also be good if we could actually embed the tests in methods, so a developer could see everything related to a method in one place.

Code Snippet 40 illustrates the generation of such test assertions, given the Umple model in Code Snippet 39.

```
01: File:<A.ump>
02: generate Test;
03:
04: class A {
05:
06:     Integer number = 321;
07:     Integer number2 = 10;
08:
09:     void calculateSomething ()
10:     {
11:             assertTrue (id == "something");
12:             assertFalse (number == 123);
13:             assertEqual (number, number2);
14:             assertMethod (getNumer());
15:
16:             // some extra code for the method
17:             statement;
18:             statement2;
19:     }
20: }
```

Code Snippet 39: Added assertion to user-defined method

Adding test assertions to user-defined methods will generate a testcases with the name of the method in the abstract test; which will be eventually processed by the Unit Test Parser/Generator and translated into platform-specific unit test code. When the unit test code is generated, it can be run against the model-generated code to make sure the tests are satisfied.

```
File: <ATest.umpt>
….
01: Test A {
02:
03:  test attribute_number {
04:          AssertTrue( a.setNumber(123)));
```

```
05:          AssertTrue( a.getNumber() == 123));
06:        }
07:
08:  test attribute_number2 {
09:
10:        AssertTrue( a.setNumber2(123)));
11:        AssertTrue( a.getNumber2() == 123));
12:
13:        }
14:
15:  test calculateSomething {
16:        assertTrue(id == "something");
17:        assertFalse(number == 123);
18:        assertEqual(number, number2);
19:        assertMethod(getNumber());
20:  }
21: }
```

Code Snippet 40: Generated abstract testcase for user-define method

As seen in the above code, *calculateSomething* is a test case that had been generated based on the name of the user-defined method in the Umple model and it has a list of assertions that can be translated into either JUnit, RubyUnit or PhpUnit using the *generate* statement in the top of the test model; *generate JUnit* for instance.

5.2.1 Handling User-Defined Testcases

User-defined test cases can be written either at the class level or the model level in the Umple model. The rationale behind this is to allow the user to define their own abstract test cases that can be injected in the abstract test model and eventually translated into different concrete unit testing platforms.

As described above, test cases consist of the three entities: initialization, action and assertion, the user can control the test scenario by managing the order of the statements. Umple respects the order of the entities as it considers the order of the statements as test sequence path. When a user defines a test case at the class level, it will generate a test case in the class specific file, however, if the testcase is defined at the model level (outside of the class) it will be injected in the model-specific test file with dependency on used classes. For instance, if we define a test case in class 'Student' it will be generated into 'StudentTest.umpt' file. On the other hand, if we define a test case outside of the class it will be generated into the file *'ModelFileName_ModelTest'*. The goal is:

- Allow developers to have the flexibility to write their own tests.

- Introducing the test-driven model perspective.
- Abstraction of tests within the Umple model artifact.

Code Snippet 41 illustrates creating a test case within a class and how it is generated in the abstract test model.

```
File<SomeClass.ump>
01: class SomeClass {
02:
03:   test calculateSomething
04:   {
05:     assertTrue (id == "something");
06:     assertFalse (number == 123);
07:     assertEqual (number, number2);
08:     assertFalse (number679, number56);
09:   }
10: }
```

Code Snippet 41: Umple model with test case

This will result in the test case being generated in the class-specific test file shown in Code Snippet 42:

```
File<SomeClassTest.umpt>
01: ////---- Tests for  SomeClass ----/////
02: Test SomeClassTest {
03:
04:       test calculateSomething {
05:           AssertTrue ( id == "something" ) ;
06:           AssertFalse ( number == 123 ) ;
07:           AssertEqual ( number, number2 ) ;
08:           AssertFalse ( number679, number56 ) ;
09:           }
10: }
```

Code Snippet 42: Generated Abstract test model with user-defined test case

5.2.2 Handling Testcase Order of Statements

As we have discussed, in Umple, the user can write their own model-level test cases. The test cases are composed using a mix of several test elements such as *test initialization*, *test action* or *test assertion*. When the compiler parses the test case code it will propagate the objects based on parsing order. However, it uses the parsing order after the compiler is done processing. The next step would be to pass model instance (including the test objects) to the two implementation classes *TestModelGenerator.java* or *TestClassGenerator.java*. These two will start to loop each element in order to construct the abstract test model code. The only problem is that we need to preserve the

order of the statements in the test body in order to keep it intact when test template generator is propagating the values.

This order must be respected in the final executable tests otherwise it may result in a wrong verdict. This is not the case for the majority of Umple template files since it is not important where one defines the association statement, for instance, it will always be processed correctly and the API will be generated despite the order of parsing. Test code is different since it is parsed internally. Therefore, whenever a test element is declared within the test code it will be given a number that represents its location within the test code. Given Code Snippet 43:

```
1: test testNO20 {
2:
3:    Person p7 ("Mat" , 345 , "bleekerst”); #1
4:    p7.setAddress(“regularStreet”);     // # 2
5:    assertTrue (p7.getId() == 345); // # 3
6:    p7.setAddress("somestreet”); // #4
7:  }
```

Code Snippet 43: Test case as defined in Umple model

The number represents the order of the statements as declared in the abstract test model. Without handling test sequence or the order of the statements, generating the code from the template resulted in broken test code:

```
1: @Test
2: public void testNO20 {
3:
4:    Person p7 = new Person ("Mat" , 345 , "bleekerst”); #1
5:    p7.setAddress (“regularStreet”);    // # 2
6:    p7.setAddress("somestreet”); // #4
7:    Assert.assertTrue (p7.getId() == 345); // # 3
8:
9:  }
```

Code Snippet 44: JUnit test case of testNO20 without preserved statement location

As seen above, statement #4 comes before statement #3. This sets the value of the *address* attribute before the intended assertion is supposed to be executed. Therefore, we had to implement an locOrder attribute, where each statement holds its position in the testcase. Hence, when writing the unit test templates we prioritize the generation based on the position of the statement in the test sequence. This results in the generation of the expected order of statements.

```
    1: @Test
    2:   public void testNO20()
    3:   {
```

```
    4:              Person p7 = new Person ("Mat" , 345 , "bleekerst”); #1
    5:              p7.setAddress “(regularStreet”);     // # 2
    6:              Assert.assertTrue (p7.getId()==345); //#3
    7:              p7.setAddress("somestreet”); // #4
    8:   }
```

Code Snippet 45: JUnit test case of NO20 when location is preserved

This issue often arises when we try to loop each element in order to propagate the values. If we say in the template language, *for each class* then *for each test in class* then there will be two options from there:

- Option 1: we call each test element consecutively. In this case, the code will be ordered based on test element type rather than parsing order.
- Option 2: we prioritize the parsing order and loop the test content using the location of statements starting from 0. Then based on the type of element detected in that location we propagate the template accordingly.

Option 2 logically is the correct way to handle such transformation between the Umple model and the abstract test model. However, option 2 requires more handling at the parsing level and at the template level. Eventually, however, although it is the correct option to consider it will increase the complexity of the code since it requires checking each statement type and look for the corresponding element from its list at each location.

5.3 Embedding Tests for Models With State Machines

In this section we discuss embedded tests in the model, and use state machines to illustrate this. Although in this thesis we don’t generate automated tests for state machines, we wanted to show how developers can nevertheless write model-level tests manually for state machines, and embed them directly in their model.

The example in Code Snippet 46 illustrates a model with user-defined testcase for targeting a state machine element called in the model [89]. The model has a state machine called *Bulb* with two main states: Off and On. The testcase is calling the Umple API that is generated for that particular state machine. Code Snippet 47 shows the tests merged with the model code in the same file , called *modelBulb.ump*, as the following:

```
File: <modelBulb.ump>
01: class House {
02:   name;
```

```
03:   test checkBulb{
04:         House h1 ("House1");
05:         String currentState = h1.getBulb();
06:         assertTrue(currentState == "Off");
07:         boolean didFlip = h1.flip();
08:         assertTrue(didFlip);
09:         currentState = h1.getBulb();
10:         assertFalse(currentState == "On");
11:         }
12:
13:   bulb
14:   {
15:     Off { flip -> On; }
16:     On { flip -> Off; }
17:   }
18: }
```

Code Snippet 46: Merging test with model

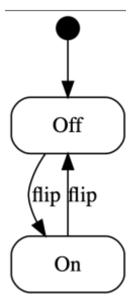

Figure 24: State machine for bulb

As seen in Code Snippet 46, we have a class called House that the has following elements:

- Attribute: name
- State machine: Bulb
    - On
    - Off
- Testcase: checkBulb

The same model file can be broken down into two files since we can rely on the Umple mixin features to separate test files from the model code to enhance the readability and structure of the system. For example, Code Snippet 46 can be rewritten in a different order as seen in Table 8. Hence, additional test elements can be added in separate files using the same format and including the necessary use statements.

Table 8: Model with separate test code

<table>
<tr><td>House.ump</td><td>HouseTest_Bulb.ump</td></tr>
<tr><td><pre>use HouseTest_Bulb.ump;
class House {
  name;
  bulb
  {
    Off { flip -> On; }
    On { flip -> Off; }
  }
}</pre></td><td><pre>class House {
  test checkState {
        House h1 ("House1");
        String currentState = h1.getBulb();
        assertTrue(currentState == "Off");
        boolean didFlip = h1.flip();
        assertTrue(didFlip);
        currentState = h1.getBulb();
        assertFalse(currentState == "On");
  }
}</pre></td></tr>
<tr><td>House.java</td><td>HouseTest.Java</td></tr>
<tr><td><pre>public Class House {
//implementation (omitted)
}</pre></td><td><pre>public class HouseTest {
//test code (ommited)
}</pre></td></tr>
</table>

We can embed testing in larger examples. Figure 25 represent a state machine for a garage door with several states. In Table 9 we show the state tables generated for this using the Umple online [90] state table generator; these represent a manual way of validating the state machine.

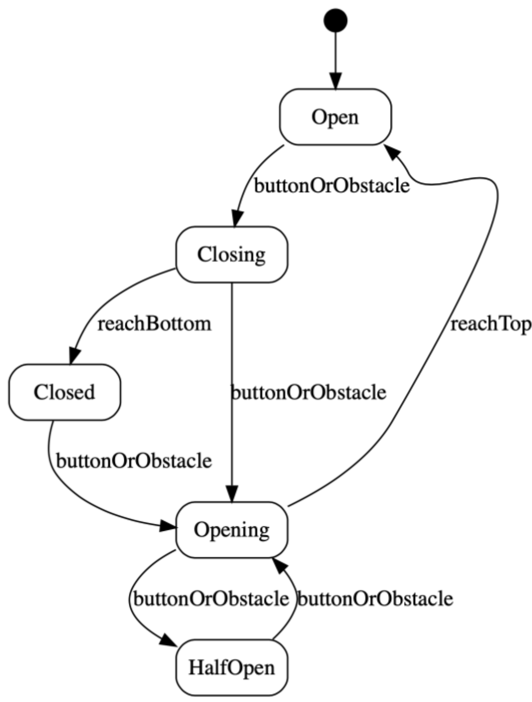

Figure 25: State machine for garage door

Table 9: GarageDoor State Tables Generated Using UmpleOnline

Class GarageDoor state machine status

State-event table

	buttonOrObstacle	reachBottom	reachTop
Open	Closing		
Closing	Opening	Closed	
Closed	Opening		
Opening	HalfOpen		Open
HalfOpen	Opening		

State-state table

	Open	Closing	Closed	Opening	HalfOpen
Open		buttonOrObstacle			
Closing			reachBottom	buttonOrObstacle	
Closed				buttonOrObstacle	
Opening	reachTop				buttonOrObstacle
HalfOpen				buttonOrObstacle	

In Umple, we can model the garage door state machine as shown in Code Snippet 47.

```
Umple model file for state machine
File: <model.ump>
01: class GarageDoor
02: {
03:    status {
04:           Open { buttonOrObstacle -> Closing;  }
05:
06:   Closing {
07:            buttonOrObstacle -> Opening;
08:            reachBottom -> Closed;
09:           }
10:           Closed { buttonOrObstacle -> Opening; }
11:           Opening {
12:            buttonOrObstacle -> HalfOpen;
13:            reachTop -> Open;
14:           }
15:           HalfOpen { buttonOrObstacle -> Opening; }
16:   }
17: }
18:
```

Code Snippet 47: Garage door state machine

We can write a list of test cases that check any sequence of event in this model or calls that we want to check within this model using the test syntax. As mentioned above, this test code is not automatically generated. This example demonstrates how the developer can embed Umple tests for state machine in the same model (not necessarily the same file since a single model can be

written using several files). We can, for instance, create another file that holds the tests for this state machine. Assuming that we want to check that when we are in the state *Open* and the button was clicked then the system should switch state to closing.

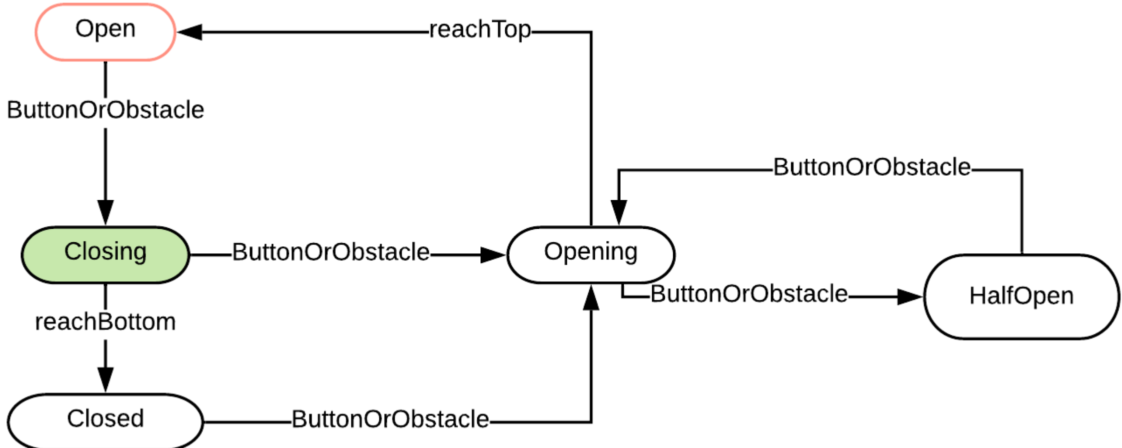

Figure 26: Garage door test

State *closing* is actually the only state that the state machine can go to from state *Open*. Therefore, we can verify that this is always correct by injecting some testing code in the model. Figure 26 shows that the test case will only pass successfully if we end up in *Closing* while we are in *Open* when the event ButtonOrObstacle had been triggered once and only once; the green color represents the accepted state. In order add the previously mentioned test into the model, we must first implement the test sequence shown in Figure 27 and then represent it in the model using the testing language.

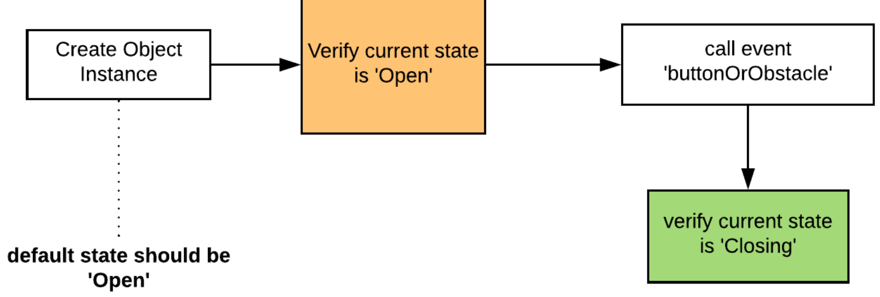

Figure 27: Test sequence for state testing

In order to implement the above sequence, we initially create a test model file. Call it modelTest.ump, as shown in Code Snippet 48: Test model skeleton for a test embedded directly in the model.

```
Umple test file
File: <modelTest.ump>
1: class GarageDoor
2: {
3:   // tests to be added here
4: }
```

Code Snippet 48: Test model skeleton for a test embedded directly in the model

Then we start adding test cases in the model to answer the test objectives that we have discussed earlier, as shown in Code Snippet 49.

```
Umple test file
File: <modelTest.ump>
01:  class GarageDoor {
02:
03:    test checkStateWhileInOpen {
04:       GarageDoor gDoor ();
05:       String currentState = gDoor.getStatus();
06:       assertTrue(currentState == "Open");
07:       gDoor.buttonOrObstacle();
08:       currentState = gDoor.getStatus();
09:       assertTrue("Closing");
10:     }
11:
12:   }
```

Code Snippet 49: Test embedded in the model

We can add more tests to reach the desired level of adequacy if we have a specific coverage criterion that we want to check for state machine. Assume that we want to verify that the garage door will end up in the state *HalfOpen* if we have executed the following sequence of events:

1- buttonOrObstacle
2- buttonOrObstacle
3- buttonOrObstacle

This can be interpreted as such that the user had pressed the button while the garage door is *Open* and while it is *Closing* the user pressed the button again. Then, the garage door should switch to *Opening* and finally when the user presses the button a third time the garage door should stop at *HalfOpen*. This sequence of events is highlighted in Figure 28, which shows the *HalfOpen* in

green as the accepted final state for this test case after the event *ButtonOrObstacle* had been triggered three times in a row.

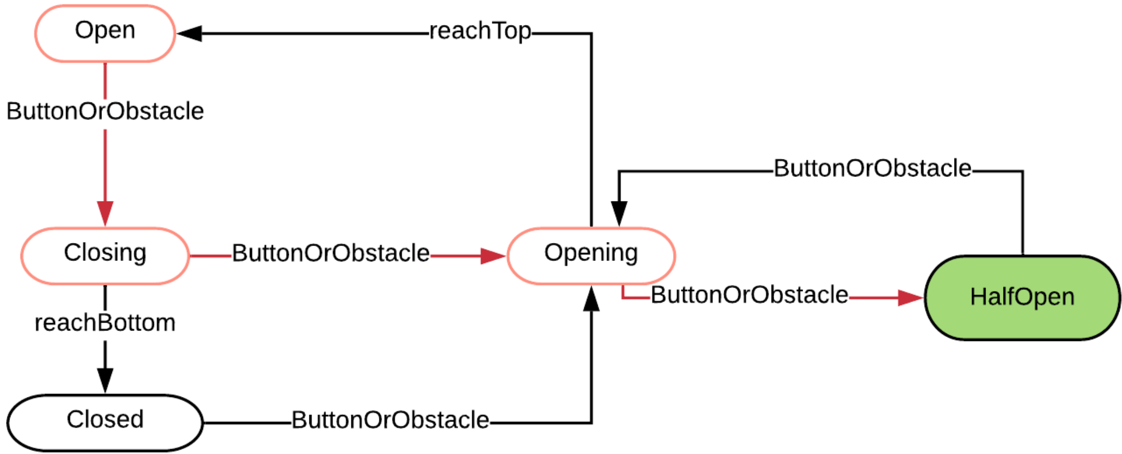

Figure 28: State machine with a specific sequence

Our goal for such case is to be able to write test cases using the Umple testing language that should verify the final state for such sequence of events. Hence, we have two options

1- create another test model file: modelTest_OpenToHalfOpen.ump

```
Umple test file
File: < modelTest_OpenToHalfOpen.ump >
01:  class GarageDoor {
02:    // add new test case for this requirement
03:   }
```

2- add another test case in the same previous Umple test model file that we called *modelTest.ump*.

Both are valid options are considered viable, it comes down to the convention the developer prefers to organize their tests for the system. For the sake of demonstration, we will go with the second option by adding another test case in the previously created model file, with result as in Code Snippet 50.

```
Umple test file
File: <modelTest.ump>
01:  class GarageDoor {
02:
```

```
03:   test checkStateHalfOpenFromOpen {
04:     GarageDoor gDoor ();
05:     String currentState = gDoor.getStatus();
06:     assertTrue(currentState == "Open");  // ensure we are in the right state
07:     gDoor.buttonOrObstacle();   // first attempt
08:     gDoor.buttonOrObstacle();   // second attempt
09:     gDoor.buttonOrObstacle();   // third attempt
10:     currentState = gDoor.getStatus();
11:     assertTrue("HalfOpen");
12:   }
13:
14: }
```

Code Snippet 50: Test file with three attempts to press buttonOrObstacle

5.4 Support for Flow Control in the Test Language

When the developer embeds his/her tests in the Umple model, they have the option to use some flow control within the test code. Flow control are statements used to point the compiler to which block of code to execute based on standard programming logic. In Umple, we do not execute the logic at the Umple level but we provide the representation of the logic of the control flow. This done by allowing the user to use the *if* statements . Consider Umple model as in Code Snippet 51. In such a model, we want to have a different verdict based on a derived value in the model. For instance, if the *Person:id* is “1324” then the job must be *Salesman*, otherwise if it is “980” then the job must *Cashier*. This example is given for demonstration of the if statement.

```
class Person {
  name;
  job;
  Integer id;

  test checkIfElse {

    if (tempId == 1324 )
    {
      assertTrue (job == "Salesman");
    }

    if else (tempId == 980)
    {
      asserTrue (job == "Cashier");
    }
  }
}
```

Code Snippet 51: Flow control within a test case

The language supports the following statements: if/if else/else. Any combination of the first two and only one of the latter can be present in one single chain of if statements.

5.5 Test Sequence Control in Umple Test Models

In Umple, the developer has the option to change the order of the tests. This can be useful in combination with global initialization, in cases where the order with which changes occur will affect the result. We can declare our object at the class level, then compose a list of test methods. Then we can change the order to try out different sequences. We can define the test sequence using the following statement in at the class level:

```
testSequence ts1 { testName1 -> testName2 -> …  ;  }
```

For example, we can separate our test method as the following: *checkRegistration* , *checkDropCourse*, *checkAddCourse*. We can create these three separately but using the same set of objects. Then we can choose the order of executing these tests.

```
Class Student {
  name;
  id;
  0..1 -- * Course;
  test checkRegistration { //register for course}
  test checkDropCourse {//drop course}
  test chckAddCourse {// add course }
  test checkStudentSequence_01 {//list of assertions}

  testSequence   studentSequence_01   {   checkRegistration   ->   checkDropCourse->
checkAddCourse -> checkStudentSequence_01;}
}
```

Code Snippet 52: Test sequence

The compiler has a number of error messages to support test sequence. For instance, if a developer declares a test sequence when the class has no test cases, it will raise a warning indicating there are no tests. Also, it will give a warning if there are tests not included in the test sequence.

5.6 Summary

In this chapter, we have demonstrated how the developer can utilize the Umple testing language in order to embed tests within the model. We have discussed how Umple is different from other approaches that embed a checking mechanism in the language used. We have also discussed the syntax and the scope of the user-defined test cases within each model and presented examples of how they can be defined. In addition, we addressed the convention used by Umple

developers in order to separate modelling concerns from testing using the Umple *mixin* feature where an Umple system can be split arbitrarily into several files, where each file has a specification of different aspects for the same class which will be weaved together at compilation time. Later in the chapter, we demonstrated how the Umple developer can embed tests in a model that has state machines.

Chapter 6 Test-Driven Modelling Using Umple

A testing process currently gaining increasing interest is *test-driven modelling*, which was discussed by [18] in 2004. This concept aims to allow the developer to derive their model from tests. The process is to strengthen the model design by making sure it incrementally satisfies the requirements based on tests derived from those requirements.

6.1 Test Driven Modeling Using Umple as a Modelling Language

Since our tool is separate from Umple, the developer has the option to use any modelling language. For the demonstration of this approach, we are using Umple to model the system for the given set of tests. To summarize the process of test-driven modelling Figure 29 illustrates the process:

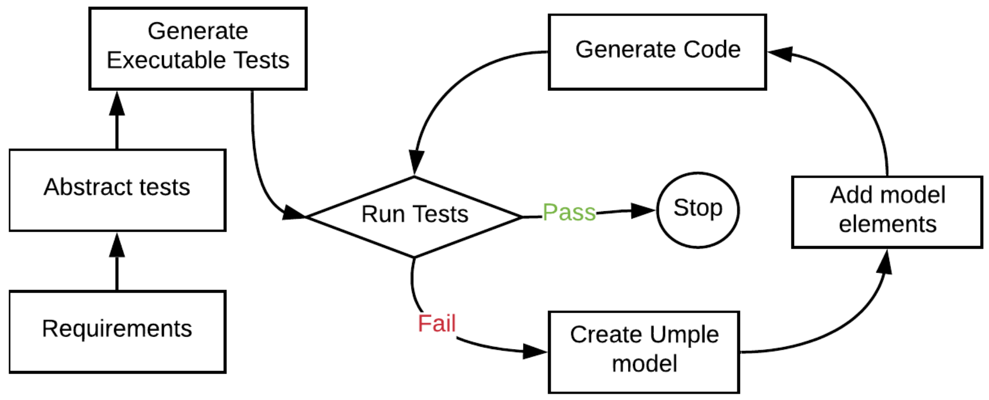

Figure 29: Workflow for test-driven modelling in Umple

The process starts by writing abstract tests using the Umple Testing Language based on given requirements then generating the executable tests. The tests will fail because classes and methods are not there. The developer will then start creating the model that should satisfy the generated unit tests until all tests passes and that should indicate that the model satisfies the given requirements. This is very similar to the process of test-driven development (TDD) by Kent Beck [19]. Whenever a new requirement is added to the system, the developer starts by writing the abstract tests that should satisfy that requirement and follow the process above to end up with a model representation that conforms to the requirement based on the tests provided.

Several studies discussed deriving tests from requirements such as the work by Urdhwareshe [91]. It is important to mention that we do not focus on the process of deriving these abstract tests from the requirements provided. We rely on the developer to provide these tests as how he/she finds it fit. Using this technique, user begin the process by having the description of the tests as part of the requirement of the system as an input (or a start point). Then, the user begins automating the process by incrementally adding tests to the abstract test file as needed. This technique can adapt to changes in the requirement. The user would add elements to the test model as the requirements changes, and where the verdict of the execution should dictate whether more elements are to be added to the design model (Umple/UML model) or not. The extension of this work to support automatic test derivation from requirement using NLP and POS methodologies are discussed in future work.

Assume that one was given the testcases shown in Table 10 based on a set of requirements, and one wanted to start using the test-driven modelling approach:

Table 10:Test cases

TC #	Test Case Description	Data	P/F
1	Check student is registered for courses	Student: name = ”John” id = 4231 age = 21	Pass
2	Student under 18 cannot register	Student: name = ”John” id = 4231 age = 21 Course: “Calculus” “12”	Pass
3	Check if stuent is underage	Name ‘Jane’ id = 5678 age = 14	pass

In this case, one would start writing the abstract tests that represent the test case in the table as the shown in Code Snippet 53: Test model based on requirements:

```
01: File: <testModel.umpt>
02: test Student{
03:  generate JUnit; // the x-unit system
04: depend Course;
05: WHEN:
06:   Student s1("john", 4231, 21);  //init data
07:   Course c1 ("Calculus", 12);
08: THEN:
09:  test isRegistered {
10:   assertTrue(s1.addCourse(c1));
11:   assertTrue(s1.getCourses().size() > 0);
12: }
13: test age{
14:
15: assertTrue(s1.getAge() >= 18);
16: }
17: }
```

Code Snippet 53: Test model based on requirements

Compiling the above model with an xUnit generator tool as the following: *java -jar umple.unit-test.jar -generate test/testModel.umpt -l JUnit* will generate the Junit test file StudentTest.java. The file should have errors of missing classes such as Student.java, and Course.java since they are used in the test file and should be present for the test to compile successfully.

Eventually, tests will fail for several types of errors. Hence, the developer must deal with these errors, such as compilation errors which require the test case to import the depended-on classes; such as Student.java. The modeler, therefore, then creates the Umple model incrementally adding model elements as needed in order to make sure these tests pass. Other errors require the modeler to add user-defined methods or certain attributes to have the right Umple API generated.

For the example above, The model should at least have a class Student and class Course as in Code Snippet 54 in order to fix the compilation errors in the generated unit tests:

```
File: <systemModel.ump>
1: class Student {
2:   String name;
3:   Integer id;
4:   int age;
5:   0..1 -- * Course; // association
6: }
7: class Course {
8:   String name;
9:   String Code;
10: }
```

Code Snippet 54: Test-driven Umple model

This approach requires the developer to be familiar with the Umple API in order to avoid method duplication. For instance, in order to respond to the above test case, the developer would first need to ensure there is a setter/getter for each attribute such as *getAge getId* …etc. Umple already generates a list of methods to allow access for these attributes, plus other methods for handling the elements in the model in general. With some knowledge about Umple and what API it generates, the developer can therefore add the necessary attributes and also add any necessary methods that are not generated by the Umple API in order to make the test cases pass successfully.

There are various ways of approaching the implementation of the model in order to pass the tests. For example, the developer can create a method called *isUnderAge* which returns Boolean if the student is not above the age of 18. Hence, when writing the test cases based on the requirements, we can use such methods in the testing (although they do not yet exist) such as seen in Code Snippet 55:

```
test issUnderAge{
  Student s2(”Jane”, 5678,14);
  assertTrue(s2.isUnderAge());
}
```

Code Snippet 55: Test-driven modelling

We can see that the method *isUnderAge* is not part of Umple API. It is acceptable to have methods that are not automatically generated as long as the developer provides an implementation for it. We do not really focus on how such methods are implemented as long as they provide the expected result. In the model from Code Snippet 54, we can add the method *isUnderAge* within the class Student as given in Code Snippet 56:

```
class Student {
  String name;
  String id;
  int age;
  0..1 -- * Course; // association

  boolean isUnderAge()
  {
   if (this.age <18)
    {
      Return true;
    }
   Return false;
  }
}

class Course {
  String name;
  String Code;
}
```

Code Snippet 56: Adding methods not in Umple API to support testing

The first time we compile the tests for class *StudentTest* we will get an initialization error. This error is due to the fact that class *Student.java* does not yet exist and we have to add it to the model. This is shown in Figure 30.

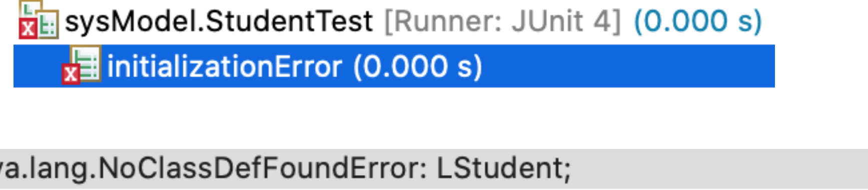

Figure 30: First initial error from running tests

We then add the necessary classes to the model – *Student* in this case, in order to ensure the first phase of errors are removed. The result is shown in Figure 31.

```
I ~/u/umple    sysModel/src/sysModel  java -jar Umple.jar  -g Java sysModel.ump
Processing -> sysModel.ump
Success! Processed sysModel.ump.
```

Figure 31: compiling the model after adding class Student

After compiling the model, test *Student* is accessible, and the initializations error has been resolved. Now we can see the other errors in the tests and we can solve them one by one. Figure 32 shows what appears next.

sysModel.StudentTest [Runner: JUnit 4] (0.000 s)
isUnderAge (0.000 s)
age (0.000 s)
isRegistered (0.000 s)

Figure 32: Requirements tests after passing initialization errors

In Figure 33 have run the first test case *age*, which check whether the Student is above 18. The test passes for the first student 'john' who had been initialized with the age of *21*.

sysModel.StudentTest [Runner: JUnit 4] (0.001 s)
isUnderAge (0.000 s)
age (0.000 s)
isRegistered (0.001 s)

Figure 33: Incrementally passing test cases

Next, we try to satisfy the second test which is *isRegistered*. This requires another class called *Course*, which we have not yet created. An error *Course cannot be resolved as a type* is returned. We go back to the model and add an Umple class called *Course* and add two attributes, *name* and *id*. We can infer the type of these attributes from the test case initialization, see Code Snippet 53:line 07. After adding the necessary elements to the model, this includes an association *0..1 -- * Course* in class students to get the Umple API for processing the set, we run the test cases again to get the results as seen in Figure 34.

sysModel.StudentTest [Runner: JUnit 4] (0.000 s)
isUnderAge (0.000 s)
age (0.000 s)
isRegistered (0.000 s)

Figure 34: Passing *isRegister* test case

The JUnit code generated for the above test cases is the following:

```
public class StudentTest {
        Student s1 = new Student("John", 4231, 21);
        Course c1 = new Course ("Calculus", 12);

        @Test
```

```
        public void isRegistered()
        {
                assertTrue(s1.addCourse(c1));
                assertTrue(s1.getCourses().size()>0);
        }

        @Test
        public void age ()
        {
                assertTrue(s1.getAge() >= 18);
        }
…omitted
```

Figure 35: the JUnit code for the test-driven modelling

Finally, we want to satisfy the last requirement which requires us to implement a method to check whether the student is underage or not called *isUnderAge*. We do this by adding the method as discussed before in in Code Snippet 56. Finally, we get the result where all three test cases are passing. Our model has evolved as we try to satisfy these tests case and now conforms to the requirements given. the end result of the model code is shown previously in Code Snippet 56.

sysModel.StudentTest [Runner: JUnit 4] (0.001 s)
isUnderAge (0.001 s)
age (0.000 s)
isRegistered (0.000 s)

```
@Test
        public void isUnderAge()
        {
                Student s2 = new Student("Jane",5678, 14);
                assertTrue(s2.isUnderAge());
        }
```

Code Snippet 57: JUnit code for isUnderAge

We have demonstrated so far how we can incrementally add elements in the model in order to respond and satisfy the test cases that the developer writes based on given requirements. We did not focus on which type of requirements are present because we left this open for the user to choose. Our focus is the tests those are derived from the requirements. We do not provide any mapping between requirements and tests, neither do we provide an automatic generation mechanism. However, it would be a good extension of the work to provide a tool to fill such a gap. The example provided above is for demonstration and is scaled down to give the idea of how this approach can be adapted in order to derive the model based on the tests written first hand.

6.2 Challenges in Test-driven Modeling

There are several challenges that need further investigation and more focused studies to answer. For instance, we assume that the developer will be able to write the tests based on requirements first. This requires the developer to come up with the names for some of the entities and relations to be addressed. Hence, the developer ends up creating a domain model as he is writing the abstract tests which might be challenging and might in fact make it better to create initial domain model before writing those tests. There are different choices that can be taken going from requirements to the abstract test modelling. One is to have an initial domain model with the entities and associations in addition to the given requirements. The user in this situation is basing his tests on requirements and partially on the model where he expects the model to evolve to answer the tests as they expand to answer further requirements of the system. Hence, it is important at this stage of this approach to have an additional link between the model and the abstract tests as can be seen in Figure 36.

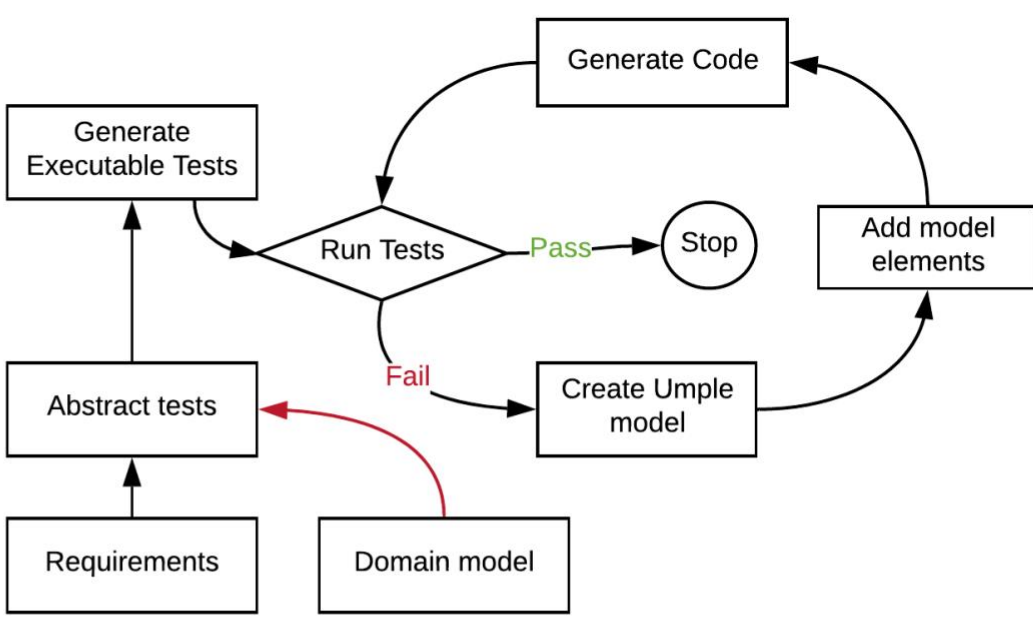

Figure 36: Improved test-driven modeling workflow

This *Domain Model* link can be skipped if the developer is very familiar with associations and class relationships and is able to infer them directly from requirements. However, this remains a challenge without having a domain model provided. A developer who is familiar with the Umple API can use the previous approach without facing difficulties inferring method names and the associations-related interface. As mentioned above, more empirical studies are required to determine the effectiveness of this approach.

6.3 Summary

In this chapter, we have demonstrated how we can utilize the Umple testing language in order to apply the test-driven modelling method. We have shown based on given simple requirements how we can incrementally add model features in order to satisfy the tests. Hopefully, following an approach like this will eventually improve quality of models since they would be more likely to conform to the requirements of the system.

Chapter 7 Advanced Issues in Model-driven Testing In Umple

In this chapter we discuss some features that enhance the power of our work. These include the use of traits to contain tests that can be injected in multiple classes, mixsets to allow switching on or off of both model and test elements, inheritance and polymorphism with tests, platform-specific tests, as well as before/after tests to allow setup and teardown of a test environment.

7.1 Testing with Traits

Traits is a feature in Umple implemented by Abdelzad, [45]. It allows an Umple class to inherit traits of a class without actually using target-language specific inheritance – the text of the trait is simply copied into the relevant classes. The inherited elements are woven into the Umple model before the code is generated. When it comes to testing, there are two ways to look into traits in Umple. First, using the Umple testing language to test traits. Secondly, use traits to promote reusability for the test cases used redundantly with different classes. We will discuss the two perspective briefly.

First, testing traits in the Umple model using the abstract test representation can be challenging due to the fact there is not a clear mapping between the generated code and the abstract traits. This makes it really difficult to determine the inherited elements by looking at the generated code. Since traits are technically a model transformation from a higher level to a lower level at the model level (Umple-to-Umple), when we process the Umple model instance for testing, it should already have been woven with the elements inherited using traits. For instance, looking at the example in Code Snippet 58:

```
File: <model.ump>
01: generate Test;
02:
03: trait Identifiable{
04:  name;
05:  id;
06:
07: }
08:
09: class Person{
10:   isA Identifiable;
11: }
```

Code Snippet 58: Umple model with trait

We have a trait named Identifiable and a class Person inheriting *name* and *id*. When we run the test generator, we get the result in Code Snippet 59.

```
File: <modelTest.umpt>
01: ////---- Tests for  methodAssertion ----/////
02: Test Person {
03:   //tests for :name
04:
05:       test attribute_name {
06:     AssertTrue( person.setName("RandomString1"));
07:     AssertTrue( person.getName() == "RandomString1");
08:     AssertTrue( person.setName("RandomString123"));
09:     AssertTrue( person.getName() =="RandomString123");
10:   }
11:
12:  //tests for :id
13:
14:       test attribute_id {
15:       AssertTrue( person.setId(123)));
16:       AssertTrue( person.getId() == 123));
17:
18:   }
19: }
```

Code Snippet 59: Generated tests for class using trait

As seen in Code Snippet 58 the test generator is considering the inherited elements from trait. This is done automatically since we are dealing with a woven Umple instance when we are generating the abstract test. Therefore, when we look for attributes in class Person, it should have the inherited attribute added to it already by the trait processor.

If our goal is to use traits to reuse test cases, then the test cases defined in the trait should be inherited to all clients (classes or other traits using a given trait). The test generator does not produce trait code in the abstract test model but will process the trait to look for test cases to inherit or assertions within user-defined methods and then pass them down to the inheritors. For instance, consider the case if the trait had a test case as in Code Snippet 60.

```
File: <model.ump>
01: generate Test;
02:
03: trait Identifiable{
04:  name;
05:  id;
06:
07:  test idMustBeLargeNumber{
08:      assertTrue(id>100);
09:  }
10:
11: }
12: class Person{
```

```
13:   isA Identifiable;
14: }
```

Code Snippet 60: Trait with testcase

Note the this contains a test case that should be inherited by Person. When the test token is detected, an instance of UmpleTestCase (with UmpleAssertions and TestActions if needed) is created and then added to each child. Hence, when a test generator is processing such a case, it will include the test case from the trait into the Person test model. This is handled within the Umple Internal Parser only since the Test Templates already recognizes test cases in classes and will eventually process them regardless whether they were inherited or natively defined within the class.

7.2 Test Generation for the Mixset Feature

Umple mixsets is a feature presented by Algablan in [47]. It allows for reusability of code and enabling of features to create different members of a product line. The example shown in Code Snippet 61 illustrates a system with two alternative special versions.

```
File: <mixsetModel.ump>
01: class X {
02:  a;
03: }
04: mixset specialVersion {
05:   class X {
06:     c;
07:   }
08: }
09: mixset specialVersion2 {
10:  class X {
11:    y;
12:  }
13: }
14: use specialVersion;
15: use specialVersion2;
```

Code Snippet 61: Basic Mixsets

This model represents a mixset called *specialVersion* which adds the attribute *c* when activated. The feature can be activated by using the statement *use specialVersion*. When we are doing automatic test generation for mixsets, we want to make sure that the attribute *c*, which comes from the mixset, is also included in the analysis. This does not require extensive analysis since the Umple compiler processes mixset ahead of testing. Therefore, when we get an instance of the Umple metamodel for analysis, all features will have been weaved into the original model. The

testcase simply has to also be included in mixset *specialVersion*. Hence, when we check the class elements for the attribute *c* in this case, it will be present.

7.3 Test Inheritance and Polymorphism

Inheritance and polymorphism is object-oriented programing languages are considered core concepts [92]. Test inheritance in Umple is supported through class inheritance and interfaces. For instance, test cases can be declared as an element of the class and can be inherited to child classes if there are any [91].

7.3.1 Inheriting Tests Defined in a Parent Class

Assume that we have a model where several child classes are extending others. The child class should inherit the elements in the parent class, such as attributes, methods, and state machines. In Umple, we also support *test* inheritance. If a parent class has a list of user-defined test cases, then these test cases will be passed down to the child class. Additionally, the child class has the following options to perform on the inherited tests:

- **Extend the parent test code:** When this is done, more test code can be added to the parent test in the child class by declaring the test again. The child test code will be appended and executed after the parent test code. (as long it is not declared as an *override* test.
- **Override the parent test:** In this case, the child can completely replace the parent test case by declaring the same test with the keyword *override*. If a test that is in the parent class is declared this way in the child class. The whole test code will be replaced by code specified in the child class.
- **Do nothing:** When nothing is done to the test case, it will be inherited and executed as how it was declared in the parent class.

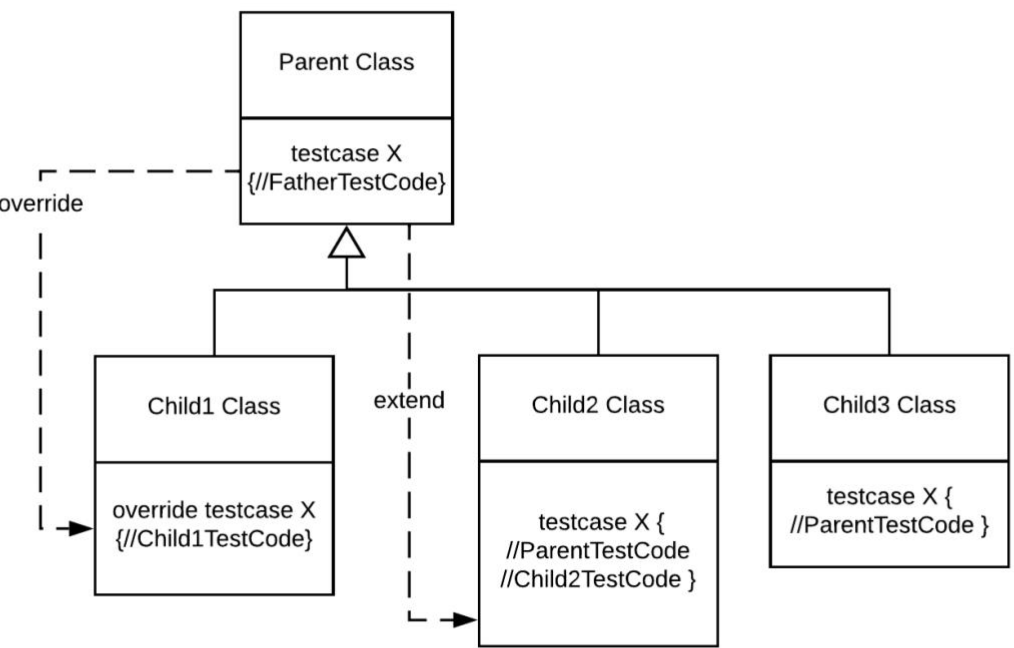

Figure 37: Class hierarchy showing tests

The model in Figure 38 shows an example of several classes has a number of cases of inheritance defined.

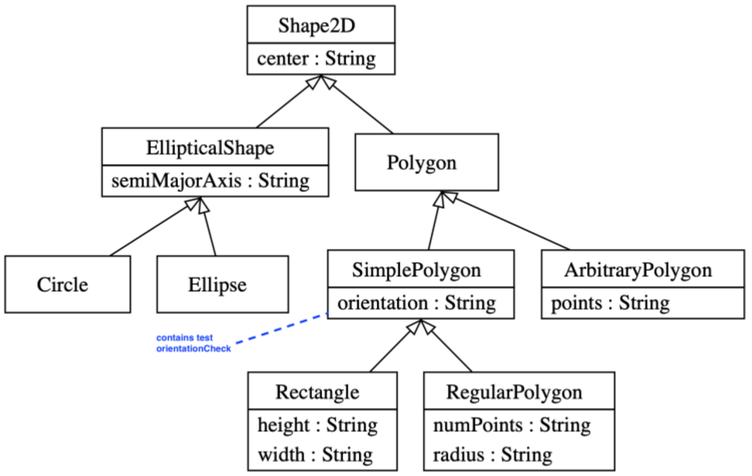

Figure 38: Example inheritance model illustrating testing

We can see in Figure 38 that various attributes inherit among the different shape classes. We want to focus on tests that are defined by the user. Hence, we will create another file in order to test this model and will demonstrate how test inheritance is done with different options listed above. The model in Figure 38 can be modelled in Umple as per Code Snippet 62.

```
Model.ump: Umple Model for 2D shapes
namespace Shapes.core;
class Shape2D {
 center;
}
//Abstract
class EllipticalShape {
 isA Shape2D;
 semiMajorAxis;
}
//Abstract
class Polygon {
 isA Shape2D;
}
class Circle {
 isA EllipticalShape;
}
class Ellipse{
 isA EllipticalShape;
}
class SimplePolygon {
 orientation;
 isA Polygon;
}
class ArbitraryPolygon {
 points;
 isA Polygon;
}
class Rectangle {
 isA SimplePolygon;
 height;
 width;
}
class RegularPolygon {
 numPoints;
 radius;
 isA SimplePolygon;
}
```

Code Snippet 62: Umple 2D Shape Model

We want to inject a test in class *SimplePolygon* which is called *checkOrientation*. Therefore, we will create a file to test the model as shown in Code Snippet 63 and we will call it *modelTest.ump*. We will be using the Umple *mixin* feature to weave the test with the classes in order to separate concerns.

```
ModelTest.ump : Umple model for testing
Class SimplePolygon {
test checkOrientation {
    SimplePolygon sp (“5”,"West");
    sp.setOrientation("North");
    tempOr = sp.getOrientation();
    assertTrue(tempOr == "North");
  }
}
```

Code Snippet 63: Check orientation test

We have added a test to the class SimplePolygon that checks that the orientation is settable and the value of the orientation is changed as expected. When we process the model, we want to make sure that this test will be also generated in the children classes; these are Rectangle and RegularPolygon, as shown in Code Snippet 64.

RectangleTest.umpt	RegularPolygonTest.umpt

`//------------------`
` //User-defined Tests`
` //------------------`
`      test checkOrientation {`
`        SimplePolygon sp ("5","West");`
`        sp.setOrientation("North");`
`         tempOr = sp.getOrientation();`
`        assertTrue(tempOr == "North");`
`        }`
	`//------------------`
	` //User-defined Tests`
	` //------------------`
	`      test checkOrientation {`
	`        SimplePolygon sp ("5", "West");`
	`        sp.setOrientation("North");`
	`         tempOr = sp.getOrientation();`
	`        assertTrue(tempOr == "North");`
	`        }`

Code Snippet 64: Test code injected in subclasses

If we want to replace the test in the child class, we use the *override* keyword when declaring the test at the child level. This can be done by adding the test as shown in the right-hand side of Code Snippet 65 in a child class (Rectangle in this case).

Class SimplePolygon	Class Rectangle

`Class SimplePolygon {`
`test checkOrientation {`
`    SimplePolygon sp ("5","West");`
`    sp.setOrientation("North");`
`    tempOr = sp.getOrientation();`
`    assertTrue(tempOr == "North");`
`  }`
`}`
	`Class Rectangle {`
	`override test checkOrientation {`
	`    Rectangle rec ("5","West","10","20");`
	`    rec.setOrientation("South");`
	`    rec.setHeight("20");`
	`    tempOr = rec.getOrientation();`
	`    rempHeight = rec.getHeight();`
	`    assertTrue(tempOr == "South");`
	`    assertTrue(tempHeight == "20");`
	`  }`
	`}`

Code Snippet 65: Overriding testcases

By overriding the parent tests, we can inherit all applicable properties of the parent class and also have the option to replace any inherited test if it does not fit with the test goals. The above example shows that we needed to test that the orientation is to settable to South and also the height is at 20. Allowing the user to override inherited tests gives flexibility to not be forced into tests that might not fit if the child has properties that may change the way we test the elements that have been inherited.

We have discussed how to override the test above, now we want to discuss the case where the developer wants to add more test code to the test inherited from the parent class. Assume that the

tests coming from the class *SimplePolygon* fits with the test goals for the child class *RegularRectangle*, therefore, we want to add more to that tests without replacing the code already in the parent class. We can always write another test case and avoid the confusion; however, we want to reuse as much code as possible and avoid redundancy. Hence, we can do this by declaring the test again in the child code. This declaration will not create a new test case, in fact, the Umple compiler checks if the test case (using the test case name) already exists in the parent class. In the case where this is true, the compiler calls a merging method that will add the parent test code followed by the child test code. Then the merged test is added to the child. This process is done after the model had been analyzed. Code Snippet 66 show the result of the extension of the parent tests.

Parent test : SimplePolygon	Child test : RegularPolygon

class SimplePolygon {
  test checkOrientation {
    SimplePolygon sp (“5”,"West");
    sp.setOrientation("North");
    tempOr = sp.getOrientation();
    assertTrue(tempOr == "North");
  }
}
	class RegularPolygon {
	  test checkOrientation {
	  RegularPolygon regPoly (“5”, “West”,
	  “6”, “4”);
	  tempPoint = regPoly.getNumPoints();
	  assertTrue(tempPoint == “6”);
	}
	}

test checkOrientation {
    SimplePolygon sp (“5”,"West");
    sp.setOrientation("North");
    tempOr = sp.getOrientation();
    assertTrue(tempOr == "North");
  }
	test checkOrientation {
	    SimplePolygon sp (“5”,"West");
	    sp.setOrientation("North");
	    tempOr = sp.getOrientation();
	    assertTrue(tempOr == "North");
	    RegularPolygon regPoly (“5”, “West”,
	    “6”, “4”);
	    tempPoint = regPoly.getNumPoints();
	    assertTrue(tempPoint == “6”);
	  }

Code Snippet 66: Merging new test code in a subclass

7.3.2 Implementing Tests in Interfaces

Interfaces are a type of element in Umple and UML that encapsulate a set of methods that must be implemented by the classes that use the interface. Integrating tests within interfaces in Umple is done by declaring the test name in the interface. When declared in the interface, any class that implements the interface *must* have the tests implemented. The Umple compiler forces the implementation by raising list of errors and warnings in the case where these tests have not been implemented. When implementing interface tests, the compiler does not check the implementation

itself but rather only detects the existence of the tests within the class. The following example illustrates this. Given the following model.

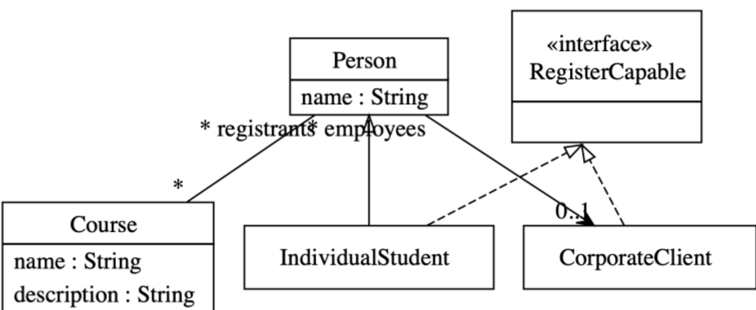

Figure 39: A model with an interface

In Figure 39, a model that has abstraction elements and an interface. The two classes IndividualStudent and CorporateClient inherits some attributes from class Person. The interface *RegisterCapable* has a method *registerForCourse*. The Umple code for the interface *RegisterCapable* is seen in Code Snippet 67.

```
File: <interfaceModel.ump>
01: interface RegisterCapable
02: {
03:   depend school.util.*;
04:   boolean registerForCourse(Course aCourse);
05: }
06:
07: class Person {
08:   name;
09: }
10:
11: class CorporateClient {
12:   isA RegisterCapable;
13:   boolean registerForCourse(Course aCourse) {
14:      // write code here
15:   }
16:   0..1 <- * Person employees;
17: }
18:
19: class IndividualStudent {
20:   isA Person, RegisterCapable;
21:   boolean registerForCourse(Course aCourse) {
22:      // write code here
23:   }
24: }
25:
26: class Course
27: {
28:   name;
29:   description;
30:   * -- * Person registrants;
31: }
```

Code Snippet 67: Umple code with an interface

Our goal here is to define tests in *RegisterCapable* that should be implemented by each interface implementer. We want to the developer to be able to verify the following:

- Check registration for each class that is *Register Capable*.
- Force developer to write specific tests for each class implementing the interface.

We can do this by adding the tests names in the interface. For such interface, we want to write tests that check the registration of each Person. Therefore, if we assumed that we want to enforce *IndividualStudent* and *CorporateClient* to test the method *registerForCourse* in *RegisterCapable*, we can do this by writing the following test declarations in the interface.

Table 11: Interface tests

#	Tests	Description
T01	personMustBeRegistered	Every person must be registered for at least 1 course.
T02	noDuplicateRegistrationInPerson	Each instance of Person cannot register twice for the same course.

Table 11 contains two tests that we want to declare in the interface so that each implementer must write a test for at the model-level. Test *T01* requires the developer implementing *RegisterCapable* to write a test to make sure each implementer (in this case every child of Person) must be registered for at least one course. The second test *T02* enforces the developer to write a test to make sure that each instance object *IndiviualStudent* or *CorporateClient* is not registered for the same course twice.

In Umple, we can write the previously mentioned tests within the interface *RegisterCapable* as per Code Snippet 68.

```
Umple model < interfaceModel.ump > : class RegisterCapable
01: interface RegisterCapable
02: {
03:   depend school.util.*;
04:   boolean registerForCourse(Course aCourse);
05:   test personMustBeRegistered;
06:   test noDuplicateRegistrationInPerson;
07: }
```

Code Snippet 68: Umple Interface with tests

If one of the implementers did not implement any of these tests, it is considered a violation of the model specifications and the Umple compiler will raise a warning regarding this. Figure 40 shows the Umple messages when dealing with unimplemented tests.

```
I  ~/u/demo  java -jar umple.jar -g Test InterfaceModel.ump                                   1369ms
Processing -> InterfaceModel.ump
Warning 6001 on line 13 of file 'InterfaceModel.ump':
Missing implementation for tests found in interface. Class CorporateClient must implement : RegisterCapable tests
Warning 6001 on line 21 of file 'InterfaceModel.ump':
Missing implementation for tests found in interface. Class IndividualStudent must implement : RegisterCapable tests
```

Figure 40: Result from model analysis of unimplemented tests

7.4 Platform-Specific Tests

Tests in Umple can be declared to be specific to a certain platform. When the abstract test is to be concretized, it will be targeted to one of the three supported unit systems. These are JUnit, RubyUnit and PhpUnit. As much as these platforms represent unit testing systems, there are cases when one test logic would work on JUnit but not on the other platform and this due to the mechanics of the programming language to be tested. Hence, in cases where a developer wants to write separate tests for a particular unit test system then they can declare the test case with the name of the targeted platform. If the test is declared *JUnit* then it is considered as a platform specific test that will only be generated if the unit-test generator chosen was JUnit. Otherwise, the test will be omitted if the unit-test generator chosen was either PhpUnit or RubyUnit. This checking is done at the template level. When the Umple parser processes the test syntax, it will tag the test case with the concrete language, if there is any. The syntax is the following:

```
JUnit test checkRegistration { //code
}
```

Another way to declare platform-specific tests is to declare them as *concrete* tests. This means that the content of this test is *not* written using the abstract Umple test language but rather the unit-test system language such as JUnit. This can be done by adding the keyword *concrete*.

```
concrete JUnit test checkRegistration { //code
}
```

The resulting test is tagged as concrete code and the Umple compiler emits it in the final production as is. Therefore, we can say that it is a way to mix concrete code and model and the convention in such case is to separate the definition of such tests into a different file.

7.5 Before/After Keywords in Testing Elements

Unit test systems have the notion of *before* and *after* methods [93]. These methods are invoked in specific order when the test class is invoked [94]. In Umple, we aim to create a testing language that is intuitive for unit-test users. Hence, it is important to map the majority of the important and widely used notations.

Umple syntax supports before and after by allowing the developer to declare the test case using *before* or after keywords. When the test is declared before, it will be mapped and, ultimately, generated as a test method with the required *before* notation that matches the platform used. For instance, if the platform used was JUnit, then the test tagged with *before* in Umple will be generated as a test method in the corresponding Java class and will be tagged as *@Before*. This method will run first and developer can use it to initially execute a number of actions such as initiating some instances. The same logic applies for the notation *after* which runs at the end of the test class execution; the developer may use this to destroy test files or run final checks such as verifying a list of invariants in the model that should be unaffected after the other tests have been executed. Declaring an Umple before and after is illustrated in Code Snippet 69 and Code Snippet 70.

```
Umple before test method
before test setUp {
          //initalize enviroment
}
```

Code Snippet 69: Before test method

```
Umple after test method
after test teardown {
          //delete files
}
```

Code Snippet 70: After test method

The before/after keyword can also be declared for assertions. If the developer writes a list of assertions within a test method, they can indicate a list of assertion to be executed at the beginning of the test method's execution; or at the end of it in the case the *after* keyword was used. Given a

model where we have a test case that checks job promotion, we want to write test to check that the value of object Job.name is always what we are expecting before we run the test code and is changed to the new job after the promotion is applied. This is similar to preconditions and post conditions except that these assertions are invoked only during execution of generated tests.

```
test checkCashierPromotion {
        Job job1 ("Cashier");

        job1.promote();

        before assertTrue (jobName == "Cashier");
        after assertTrue (jobName == "Manager");

        }
```

Code Snippet 71: Before and after assertions in model

```
test checkCashierPromotion {

        Job job1 ("Cashier");

        //----------------
        //before assertions
        //----------------
        assertTrue(job1.getName()  == "Cashier");
        //End of before assertions

        job1.promote();

        //----------------
        //after assertions
        //----------------
        assertTrue(job1.getName() == "Manager");
        //End of after assertions
        }
```

Code Snippet 72: Generated before/after assertions

This is useful in specifications where we want to know what should be true before running the test and what should be true also after we run the sequence of events. As seen in Code Snippet 72: Generated before/after assertions, the assertion that checks the user has to be *Cashier* before applying the promotion is injected at the beginning of the test method in the generated model. The same applies for checking the that the job name has been set to manager after the promotion has been applied. These assertions can be injected anywhere in the test code and will be placed according to their declaration when the test model is generated.

7.6 Test Backward Traceability

Traceability is important when it comes to the analysis of any transformation result. It provides a reference back to the original model element. We can look at model testing traceability from two different perspective. 1) We want to have a reference between Umple model elements and the abstract test element; and 2) we want to trace back the result of test execution to the corresponding model elements in the Umple model.

The first item is a convention that is used in most Umple generators but is not yet incorporated into the test generator. Umple injects a comment regarding which model file and line-of-code was used to generate the subsequent code block. Such as seen in Code Snippet 73.

```
// line 671 "../../../../ump/mbt_parser_src.ump"
   private void analyzeTestInitAttribute (Token token, int analysisStep,
TestCase aTestCase, int locOrder){
    …. omitted
```

Code Snippet 73: Source model tagging of generated code blocks

We do this similarly in the generated abstract test model file by stating which elements a particular testcase is targeting. For attributes, we include a comment before the test case that indicates this is for a particular attribute. On the other hand, the association is done by first indicating which class these tests are for and also which related class is being tested for the other end of the association; the line number in the original Umple file is not yet included.

Regarding the second point, we do report the result of the test case, however, we do not trace back the result to the model currently. Such a feature would be highly beneficial for Umple since it will allow for executability of the model and eventually help in case we wanted to implement an *online-testing* mechanism in future.

7.7 Summary

In this chapter, we have discussed several advanced issues that were evident in the domain of model-based testing while implementing our work separately and also as part of Umple. Some of these issues related to Umple-related features like *traits* and *mixsets*. Others issues were more general issues deriving from the general concepts of model-based testing such as inheritance, test embedding within models, dependency handling, object instantiation, and backward traceability.

We have discussed each of these issues under the umbrella of our work and have shown how we handle them from technical point of view.

Chapter 8 Generic Model Tests

Writing tests for a model requires the developer to inject test artifacts in the model where needed. One of our top goals is to cut time and costs of testing. Hence, in some projects that adopt the model-driven development approach, the model or the metamodel of the system can get very large. Several studies have addressed this problem before, the work by Tonella discusses a list of algorithms in order to deal with such a problem – she uses the metaphor of genes and chromosomes to work around the dependency between classes and constructors in order to compose the dependencies needed then inject them into the constructors [95]. Umple itself is an example project that has very large metamodel and it can be navigated on the following link: *metamodel.umple.org* [96]. Therefore, in order to maximize the utilization of the testing framework for such models, a mechanism that generically generates tests based on a query result can be helpful to produce tests for a large number of elements in the model.

In Umple, we have created a syntax for generic tests. The goal of these tests is to work similarly to template languages, where developers create a test template of their own that can generate elements with specific features. This syntax can be written at the class level by declaring a test as generic using the prefix keyword *generic*. In order to accomplish this, we need to give the developer the ability to refer to some of the generic meta-elements. For instance, the developer can access the element *attribute* name by injecting <<attribute>> in their generic tests; this is illustrated in Code Snippet 74. This is different from normal user-written tests that would refer to specific attributes at a lower-level in terms of abstraction. Working with generic tests requires the developer to deal with higher level of abstraction than the level we are usually working with when writing our regular tests cases in an Umple model.

Another rationale behind the integration of this feature is to overcome any limitation in the language. This may occur when a developer attempts to adapt the technology, but they want to add more than just the test templates we support for automatic test generation. Therefore, generic tests will allow them to add templates of their own that can be automatically generated for each of the specified elements in the test. The generic testing mechanism should enhance the flexibility of model-based testing in Umple.

8.1 Generic Tests for Attributes

Code Snippet 74 is an example that demonstrates tests that are written to be generated for each attribute in the class with the type *String*. In this context, we want to make sure that each attribute value of type *String* had been logged.

```
File: <checkLoggingForEachString>
01: class Person {
02:     Integer id;
03:     name;
04:     address;
05:
06:   generic test checkifLogged(String attribute){
07:         Person p1 ( "S1425", "John", "Ottawa") ;
08:         String valueToCheck = p1.get<<attribute>>();
09:         ps1.getValue(<<attribute>>);
10:         boolean isLogged =  p1.checkIsLogged(valueToCheck);
11:         assertTrue(logged == "true");
12:         }
13:
14:         String  returnSumOfValues (int x, int y) {
15:          return x+y;
16:          }
17:
18:          boolean checkIsLogged (String value) {
19:          // ommitted code
20:          }
21: }
```

Code Snippet 74: Attribute generic test in Umple

As seen in Code Snippet 74, the test case here is declared as *generic*. This means that the compiler will replicate the test, replacing each <<attribute>> by the actual attribute name but only if it is of type *String* in the case of the above model. So, in the above tests will only be generated for the attributes *name* and *address* since their type is *String*. Other templates can be added as needed.

This is a brief abstracted version of the grammar to demonstrate what the compiler expects the syntax of generic tests to be.

generic test [testCaseName] ([elementType] [elementName] [elementParameters]) {
[test code]
}

8.2 Generic Tests for Methods

Support for generic tests covers methods as well. The developer has the option to write tests for each method using its parameters and return value for the filtering phase.

For instance, we can write a test that will be generated in each class that has a method that matches the following signature : *String <<method>> (Integer <<parameter>> , String <<parameter>>)* . This is done by creating a list that contains a list of parameters with their types. The Umple compiler looks into this list and compares the order of each method in the class with the list derived from the generic test with respect to order of parameters. This means a test with the signatures on the left and right of Figure 41 are not considered the same although they have the same parameters type in the list.

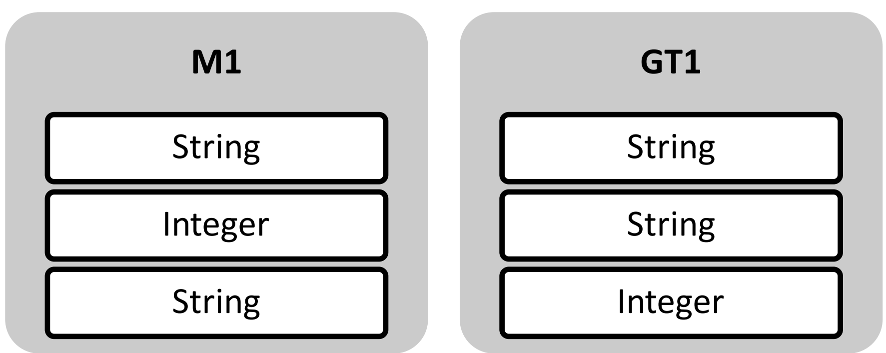

Figure 41: Umple Method parameters matching

Given the above two lists with respect to order, assume that we have method M1 in a class and GT1 stands for the list created by the compiler based on the generic test definition. The compiler will consider M1 as an unmatched method due to the order of parameters. The first element in the list always refers to the return value of the method while the rest of the elements in the list refers to the parameter list in order.

Code Snippet 75 demonstrates generic tests for methods.

```
File: <genericMethodTests.ump>
01: class Calculator{
02:
03:     Integer x;
04:     Integer y;
05:
06:   Integer  returnInteger (Integer x) { return x+y;}
07:
08:   String  returnStirng (Integer x) { return (x+y).toString();}
09:
10:   generic test checkifLogged(Integer method Integer){
11:         Calculator c1 ( 4, 5) ;
12:         String valueToCheck = p1.get<<method>>();
13:         ps1.getValue(<<method>>);
14:         boolean isLogged =  p1.checkIsLogged(valueToCheck);
15:         assertTrue(logged == "true");
```

```
16:         }
17:    generic test checkifLogged(String method Integer){
18:         // code omitted
19:      }
20: }
```
Code Snippet 75: Umple Method Generic Testing

This type of generic testing for methods is very useful when the model has a large number of methods. It allows the user to systematically search for specific method and have that template propagated with the method values. This feature should help the developer achieve the following goals:

- Enhance test coverage based on method types. This is done by apply a certain test template on every method in the class (or model) that matches the query.
- Cut time of testing. This is achieved by automatically generating the template test for the matched method in the abstract model instead of writing the test individually for each method using manual inspection.
- Increase flexibility to the developer in case they want to expand the automatic test generation for method using their own test templates.

8.3 Generic Tests for Associations

In terms of associations, we provide query of element of associations based on their type. For instance, one can generate particular tests for each association of type “bi-directional one to many”. The developer can use <<association>> in order to access the association variable. Then the developer can call the Umple API based on that value. For instance. One can *use get<<association>>();* which it will be translated into something like *getStudents();*.

One of the main issues is that based on the type of the association, Umple determines the type of the association variable. It will decide whether the association variable is a list or single value. Hence, we can always use the template to call the association variable name with no problem.

A minor issue occurs if we want to call a method in the Umple API that refers to a singular form of the association variable. These methods are supposed to process one value in association variable list; such as getStudent(int index) , indexOfStudent(Student aStudent), addStudent(…). Calling the template variable <<association>> on these methods will result in having the

previously mentioned methods translated with the list variable name which is students or Students (if it is in the middle of a method call) and this does not match the singular names on the Umple API.

This subset of the API needs to be handled specifically. Therefore, when it comes to associations, the developer can call <<association.toSingular>> in which the compiler will translate the association variable name (if it was a list) in its singular form in order to allow the developer to call the corresponding Umple API if applicable. Table 12 shows the corresponding call for each method in Umple API

Table 12: Umple Generic Association Testing

Description	Umple Method	Umple Test Template
Association variable	students	<<association>>
AV within method call	getStudents()	get<<association>>()
Call AV in singular form	getStudentAt(index)	get<<association.toSingular>>At(index)
Call AV in singular form	addStudent(someStudent);	Add<<association.toSingular>> (some<<association.toSingular>>) ;

The other challenge is how to tell the compiler that the target element is *association* and also how to apply filtering (query) on the different types of this particular element. In order to specify the target element is association, one must to declare the generic test as the following:

```
generic test <testName> ( association <associationType> ) { <testCode> }
```

If we want to test every association that is of a type one-to-many, then we would write the generic test as the following. We will call the test *checkAssociationList*, This means that this test will be generated for each association of type one-to-many:

```
generic test checkAssociationList ( association 1--* ) { //test code }
```

8.4 Discussion

One of the important issues that developers need to pay attention to when they write generic tests for associations is the initialization of objects. This can be done within the test if there are no dependencies between classes and this is not likely the case in many large models. Since generic tests target large models, it is extremely likely that there will be nested dependencies among classes. Therefore, in order to ensure the all objects are initialized correctly in the test class for the model, the developer can initialize the objects using the *test init* feature; which allows the developer to write initializations that will be inserted in the global level of the test class. Then, the developer can refer to each object from the generic test method according to how it had been initialized in the *test init* code.

```
test init {
   Student someStudent ( … ) ;
   Mentor mentor1 (….);
}
```

The query statements for attributes, methods and associations can be enhanced by introducing more query statements. For instance, we can add the following query statements on the elements name: attribute.suffix(Id) . The compiler will look for every attribute that has the word Id at the end of its name. This type of query will only match attributes such as studentId , cardId .. etc. The opposite type of query works too, such as attribute.prefix(employee) . This type of query will match each attribute that has the word employee at the beginning of the attribute s name; such as employeeId , employeeName etc.. The developer can use some code convention while naming the attributes in order to build a more testable model. For instance, all attributes that are related to class Student can be better tested using this approach if its attributes contained student- at the beginning. The same type of query can be applied on associations if the association contains a role name.

It is important to mention that this type of generic testing shines when it is applied to large models. If the model size is small, then it might not be as effective as if it was applied to large model. In order to have an idea about how testing can be time consuming when it comes to large models, we have to look into some metrics. Let us consider the Umple metamodel for instance,

the model contains a large number of elements in general. Table 13 shows the metrics generated using the Umple compiler for the Umple meta model. The metrics were generated using the SimpleMetric generator in Umple [97].

Table 13: Umple metamodel metrics

	State Machine	**Associations**	**Attributes**	**Methods**
TOTAL	19	546	1672	15707
AVERAGE	0.0	1.1	3.5	32.8
MAX	2	32	51	1350
MIN	0	0	0	0

As seen in the table, the number of methods within the model is very high – 15707 methods in total. If we made the assumption that it takes 10 minutes (in average) in order to write a test for a single method, then it will take 157070 minutes to write tests in order to reach full methods coverage in the model; which is around ~2617 hours. This approach we are proposing in this thesis will significantly reduce the number of tests to be written eventually. Writing a single generic test may result in the generation of a large number of tests. We cannot claim that every method is tested using the same template. This assumption will cause us to fall behind in standards logically. However, with such query mechanism, we can claim that we can reduce the amount of testing resources required; especially when we are handling a large number of elements in a model.

8.5 Summary

In this chapter, we have discussed generic testing in Umple and demonstrated how to apply this for attributes, associations and methods using generic test declarations with the help of query-like syntax called fixes. We have also addressed some of the domains where such mechanism could help reduce resource consumption. Further studies are required in order to motivate this work to make it more practical and usable.

Chapter 9 Dependency Handling for Instantiation of Test Objects

In this chapter we discuss a key challenge that we encountered during the development of this work: Instantiating models when there are multiplicity constraints that must be adhered to, which result in chains of dependencies regarding the order of instantiation of the classes. We had to overcome this when developing the language and integrating our testing syntax into Umple.

9.1 Object Model Instantiation

One of the issues in model-based testing in general is managing dependencies [98][99]. In object-oriented programming, we cannot test a class without first instantiating it. The problem arises when the class has a number of dependencies to be instantiated first [100]. This issue was partly discussed in Chapter 2 when we presented the automatic test generation for different types of associations. The issue is related to the problem discussed in Section 3.5.2 where the association under test is of type One-To-Many. In UML class diagrams, models have no start point, if the model is to be traversed to create a set of instantiated objects consistent with the model, then any given class can be an acceptable start point. Hence, we have to process the model and prepare a dependency analysis that will allow for instantiation that conforms to the model. The types of dependencies between classes can be categorized as the following:

- Direct dependencies: These represent classes whose instances that have to be fed to the constructor a second class in order to instantiate the target class.
- Indirect Dependencies: Instances of these are indirectly linked to instances of the target class via direct dependencies. If A has a direct dependency on B, and B has a direct dependency on C, then A has an indirect dependency on C, and the C must be instantiated first.

Ultimately, classes with no dependencies on other classes must be instantiated first, then classes that depend directly on them, and so on. But the numbers of instantiations must also be sufficient to allow for all multiplicity combinations to be satisfied during testing.

Consider Figure 42 as an example. We are given an Umple model with a list of classes and associations. We need to initialize a number of objects based on the multiplicities in order to

execute the model. The issue of dependency in the figure needs to be addressed and processed. In the figure, we can note that class *A* has a direct dependency on class *B*. This is due to the fact that the association between *A* and *B* is of a type *Many-to-One*. This type of association can be interpreted as the following:

"*each object of class A requires at least 1 and only object of class B in order to be initialized. Also, each object of class B can have zero or more objects of class A.*"

We need to understand how Umple handles associations in order to get a deeper look into the final production of such an association.

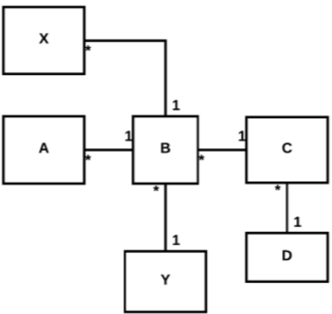

Figure 42: A model with a number of associations

For such an association A * -- 1 B, Umple generates the constructor for each class as shown in Code Snippet 76

<table>
<tr><th>Class A</th><th>Class B</th></tr>
<tr><td><pre>//------------------------
// CONSTRUCTOR
//------------------------
public A(B aB)
{
   boolean didAddB = setB(aB);
   if (!didAddB)
   {
    // exception thrown
   }
}</pre></td><td><pre>//------------------------
// CONSTRUCTOR
//------------------------
public B()
{
  as = new ArrayList<A>();
}</pre></td></tr>
</table>

Code Snippet 76: Constructors generated by Umple

We can see that class A requires an object of *B* yet class of *B* does not require an instance of A. Therefore, *B* needs to be initialized first. In order to initialize *B*, we need to process *B* separately and resolve any dependency in that class too.

9.1.1 Problem 1: Dealing with Dependency Level

Objects need to be initialized with respect to dependency level. This is necessary since each may be dependent on other classes in order to be initialized. Therefore, classes with no dependencies have to be instantiated first before we call the constructor of the other dependent classes.

9.1.2 Problem 2: Starting Point and the Possibility of Cycles

Class diagram models are arbitrary graphs and have no start point as in trees. Therefore, in order to traverse the model, a starting point has to be determined. In such graphs, several issues may occur such as cycles where a class has an association that can be traversed back to the same starting point.

9.1.3 Observation: Multiplicity Lower Bound of One Is a Key Constraint

Looking at the Umple-generated code and inspecting the constructors of each class, we have noticed that Umple only requires an instance when the association end has a multiplicity of type *One*. This is the case as in *A * -- 1 B* . All the other possibilities do not require passing the object instance of the other class to the constructor. The other case where Umple does not require the object instance of the other class for the multiplicity of 'One' is when the association is directional such as in Figure 43.

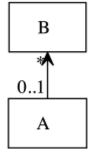

Figure 43: Umple optional association

In this case, if the user declares an association of type *A 1 - > * B*, the Umple compiler will raise an error since it cannot be handled logically (each B's does not know about the A so the constraint can't be guaranteed to hold). The issue occurs when we want a directional association

where *B* does not know about 'A' but with multiplicity of *One* then *B* requires an instance object of *A*. Therefore, this causes a conflict of specification. However, this can be done by declaring the association *0..1* -- * and this will generate a class *B* that does not have a reference to *A* as shown in Code Snippet 77.

Class A	Class B

`//------------------------`
`// MEMBER VARIABLES`
`//------------------------`
 	
`//A Associations`
`private List<B> bs;`
 	
`//------------------------`
`// CONSTRUCTOR`
`//------------------------`
 	
`public A()`
`{`
`  bs = new ArrayList<B>();`
`}`
	`public class B`
	`{`
	
	`//------------------------`
	`// MEMBER VARIABLES`
	`//------------------------`
	
	`//------------------------`
	`// CONSTRUCTOR`
	`//------------------------`
	
	`public B()`
	`{}`
	
	`…`

Code Snippet 77: Constructors for directed associations

9.1.4 Solution: Traversal Algorithm

In order to handle the previously mentioned problem there are several ways of approaching the problems. These are:

- Traversing the model as a graph by determining a start point and counting the number of direct and indirect dependency between classes. This is done using an algorithm.
- Traverse the model classes counting only direct dependencies and then increase the level of the class based on each nested indirect dependency.

Figure 44 illustrates how we handle dependency between classes. First, we check for cycles of dependency, as seen in the figure, A depends on B, B depends on C, and C depends on A. This type of model is impossible to instantiate since there is a cycle of dependencies. In order to initialize A we need an object of B. Also, in order to initialize B we need an object of C. However,

the paradox occurs when C requires A. Hence, we cannot initialize A because we indirectly need C in order to initialize A.

Our goal is to detect such a cycle before we proceed, in order to do this, we create a stack when traversing dependencies. Whenever a class appears twice in the stack then this is considered a cycle and therefore we cannot proceed and only raise a warning in the compiler asking the developer to resolve such a design issue. We use each class as a start point for a single traversal and only traverse links that have dependencies. Hence, if we have an association A * -- 1 B, it means A needs at least 1 B in order to initialize, when this is detected we apply the following:

- Add B as dependency in A
- Add B to the stack
- Look For dependencies in B

Algorithm 1: Umple Resolving Dependency

```
Input: model.ump
Output: dependency lists
currentClass
 processedLinks ← empty
 stack ← {}
 visited ← {}
 dependencyList  Function checkDependencies(model):
   foreach class in model.getClasses do
      stack.push(class);

      foreach as ∈ class.getAssociations do
         if as.otherEnd is 1 then
            dependencyList.add(as.otherEndClass)
             if stack.contains(cs) then
                // Cycle Detected
            else
                // no Cycle
            end
         else
            // next association
         end
      end
   end
return true
```

In the case where B has a dependency (C for example), then we traverse that link and add C to the stack and apply the same previous steps. We always check if the dependency is in the stack

already or not. When we check for the dependency in C we will find A, then before adding A to the stack we check whether it is already in the stack (which is true) then we stop the analysis and declare the design issue until resolved.

In the case where there is no cyclical issue, C will be added to B and we are assuming that C does not have any dependency and start to construct dependency lists. In B we create a list where only C is listed as dependency. Also, when we finish processing B, we create a dependency list in A and add B to the list. When we attempt to initialize A, we ask whether it has dependency in the list, then ask to initialize all of them first by calling a recursive method that will add initialization code to a string buffer in order. When B is to be instantiated, we also check if it has any dependent instantiations and resolve those first. This way, we end up creating an instance object of C first, then B and finally A, in which A depends on the other two directly and indirectly.

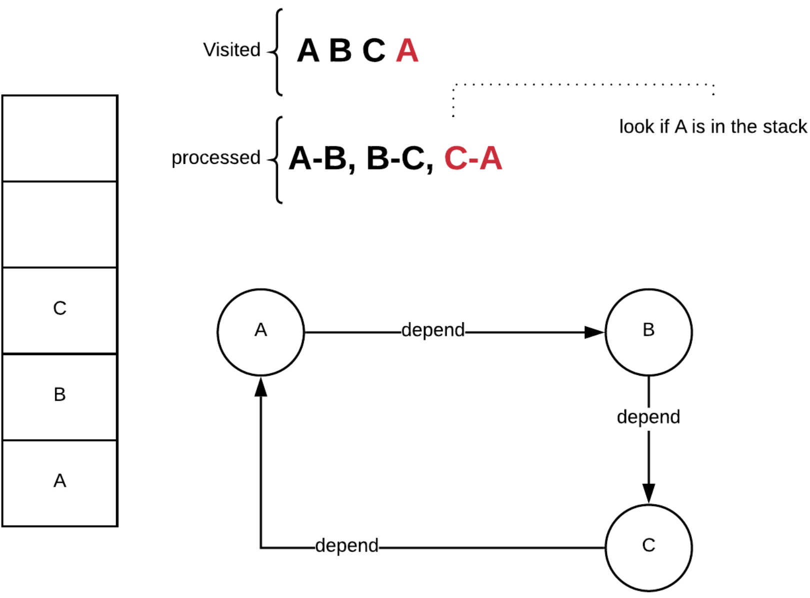

Figure 44: Model Traversing and Dependency Check

9.2 Dependency Injection and Object Instantiation for the Airline Example

Code Snippet 78 shows the automatically-generated dependency objects for the whole model. In the test generator, this can be done by choosing either to initialize the whole model or a single class.

```
Airline aAirline()
PassengerRole aPassengerRole() //lvl0
EmployeeRole aEmployeeRole()  //lvl0
RegularFlight aRegularFlight("17:03:00",123,aAirline) //lvl1
Person aPerson("RandomString1",123,aAirline) //lvl1
SpecificFlight aSpecificFlight("2020-12-06",aRegularFlight) //lvl2
PersonRole aPersonRole(aPerson)  //lvl2
Booking aBooking("RandomString1",aSpecificFlight,aPassengerRole) //lvl3
```

Code Snippet 78: Airline dependency handling

Initiating a single class in Umple test template can done as in Code Snippet 79. We are automatically initiating class *specificFlight*.

```
Airline aAirline()
RegularFlight aRegularFlight("17:03:00",123,aAirline)
SpecificFlight aSpecificFlight("2020-12-06",aRegularFlight)
```

Code Snippet 79: Class initiation in Umple test template and dependency injection

9.3 Summary

In this chapter we discussed object instantiation and the need to do dependency analysis. We presented an algorithm to ensure this can be done. Sometimes that can result inability to instantiate, and hence inability to generate tests.

Chapter 10 Implementation of the Approach

We want to implement Umple test generator within the current architecture of Umple. This can be done by introducing the following components in the system:

- Umple Test Generator.
- Umple Abstract Test Model (Test Language)
- Umple Test Model Parser
- xUnit Test Generator

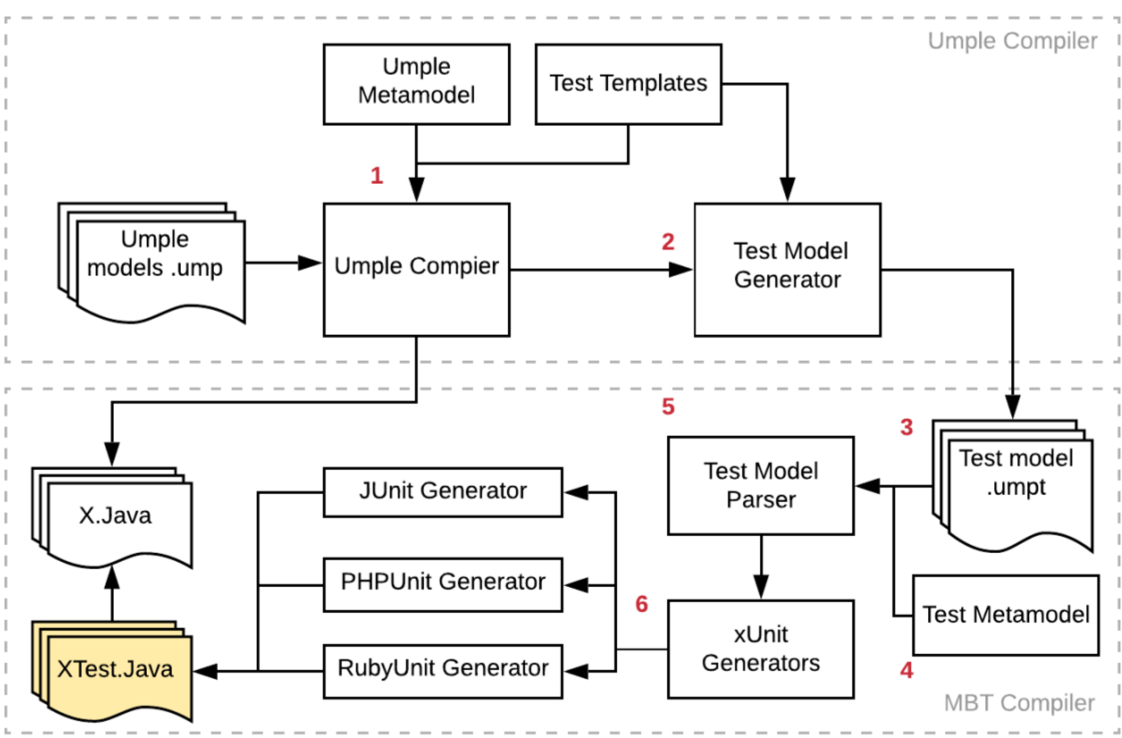

Figure 45: Test generation in Umple

10.1 MBTParser: Umple Abstract Test Parser

In order to allow for separation of concerns in Umple [101], we provide a test parser that will parse the generated abstract test model in order to create an instance of the test metamodel that can be transformed into executable concrete unit tests for different platforms. There are several goals behind this, as discussed earlier in this thesis. However, we mainly aim at the following goals:

- We want to allow the user to explicitly write custom unit tests from the abstract test model without relying on the Umple compiler to do that.
- We want to allow third party tools to adapt the language and use it to generate concrete tests.

This parser simply calls upon two main files in order to create an instance of the metamodel with the values on the abstract model being propagated; these files are:

- Grammar file
- Abstract test model to be parsed.

This parser was created using UmpleParser [102], a stand-alone Umple application that can be used to create parsers for any language. This work allowed us to avoid relying on any third party tool such as ANTLR [103] [104]or XTEXT [105] [106], that were considered earlier in the development. Benefits of relying on our own technology include 1) avoidance of dependencies that consume effort to frequently manage as new versions are released; 2) our own tool serves as a test case for itself; 3) if we run into bugs or challenges, we can manage them ourselves.

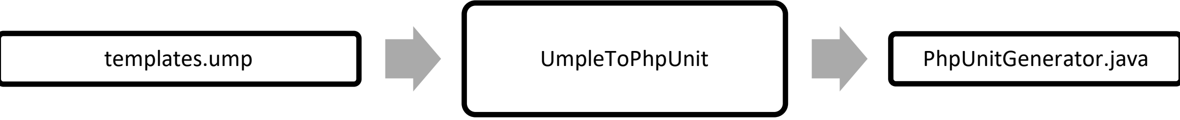

Figure 46: Workflow for creating a unit generator using Umple

The parser has a list of analyzers that will take the root token and then delegate to each analyzer based on the Token type in order to populate the test metamodel instance. Table 14 shows the command-line tool for unit test generation and the parameters that can be used for test generation.

Table 14: Command-line parameters for unit-test generator

Option	Input
-generate	Takes Umple model file: model.umpt
-s	A general option that can be passed to the tool, for instance: -s 'HtmlReport' and it will specify the report to be generated as HTML.
-h	Will print a help message to guide the user.
-l	The unit test generator to be used 'JUnit','RubyUnit' or 'PhpUnit'.

10.2 Generators

We created xUnit generators for the following unit test systems:

- JUnitGenerator: This is responsible for generating unit tests for Java using JUnit
- PhpUnitGenerator: This is responsible for generating unit tests for Php using PhpUnit
- RubyUnitGenerator: This is responsible for generating unit tests for Ruby using RubyUnit (Test::Unit library)

The workflow of creating an xUnit generator is Umple is as follows: We first create a project with the name "UmpleTToXUnit". This project contains a list of Umple files that contain the templates for the targeted xUnit tests written using the Umple Template Language (UTL) [72][107] . Figure 46 shows the output of the project "UmpleTToPhpUnit" as the unit generator class. The unit generator class is called by UmpleToTest to generate the concrete unit tests for the system. When an abstract model is passed to the parser, an instance of *TestModel* is created with instances of all elements provided in the abstract model. The model instance is then passed to the unit generator where the model is then transformed into Php, Java or Ruby unit code for execution.

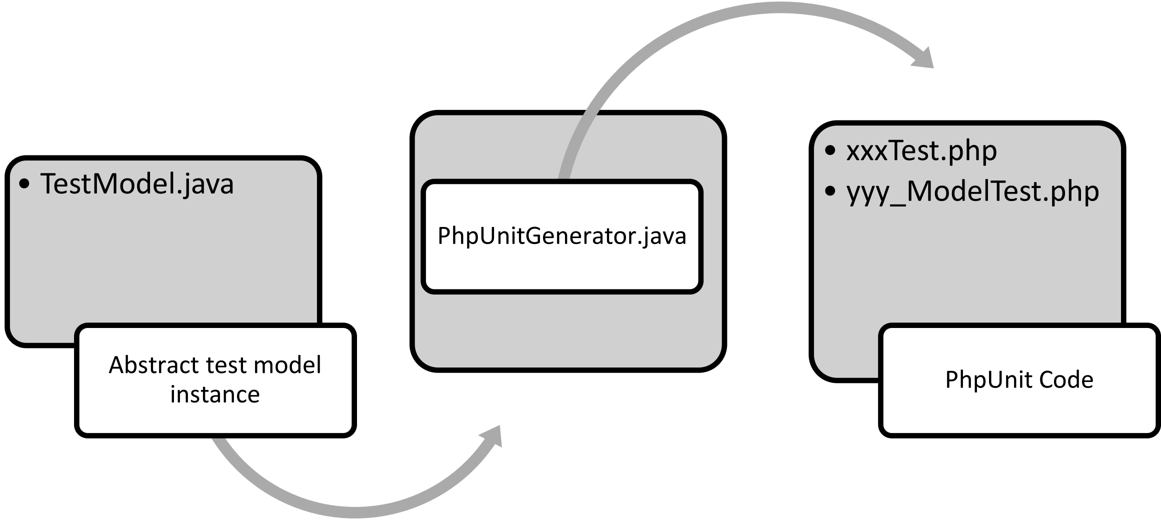

Figure 47: Unit test generator with Php as an example

Code Snippet 80 shows an abstract model for the airline system. Code Snippet 81 shows some code generated for this in JUnit. Code Snippet 82 shows the generated PHPUnit code, and Code Snippet 83 shows the RubyUnit code.

```
File: <Airline.umpt>
01: test Airline {
02:
03:   generate JUnit;
04:
05:   depend Airline, Passenger ;
06:
07:   GIVEN:
08:    Airline.ump;
09:   WHEN:
10:    Person p1 ("John", 123) ;
11:   THEN:
12:
13:   test someTest {
14:    Person p1 ("John", "123","someAddrss" );
15:    Person p2("Jane", "321", "street4");
16:    assertTrue (p1.setId ("1234") ) ;
17:    p2.setName("Roger");
18:    assertTrue(p1.getId() == null);
19:    assertFalse ( p1.getId() == 1324 ) ;
20:    assertEqual(p1.getId(),p2.resetId());
21:    }
22:
23:   test test2 {
24:     Person p2 ("Jane", 456, "anotherstreet");
25:     Person p1 ("Jane", 456, "anotherstreet");
26:     p2.setId("anId");
27:     assertTrue ( p2.resetAddress() != null);
28:     assertTrue ( p2.setAddress("321 another st")) ;
29:     assertEqual( p2,p1.setId());
30:   }
```

```
31:
32:   test sometest3 {
33:     Person p4 ("Joiana", 123, "bleekerst");
34:     assertTrue(p4.getId() == 123);
35:   }
36:
37:   test tesNO10{
38:     Person p7 ("Aida", 321, "Mannst");
39:     Person p6 ("Mark", 123, "bleekerst");
40:     assertEqual (p7.getId(), p6.getId());
41:   }
42:
43:   test testNO20 {
44:     Person p7 ("Mat" , 345 , "bleekerst");
45:     p7.setAddress("regularStreet");
46:     assertTrue (p7.getId()==345);
47:     p7.setAddress("someStreet");
48:   }
49:
50: }
```

Code Snippet 80: An Umple abstract test model example

```
File: <AirlineTest.java>
01: public class AirlineTest {
02:
03:   @Before
04:   public void setup(){}
05:
06:   @After
07:   public void teardown(){}
08:
09:   @Test
10:   public void someTest()
11:   {
12:           Person p1 = new Person ("John","123","someAddrss");
13:           Person p2 = new Person ("Jane","321","street4");
14:           Assert.assertTrue (p1.setId("1234"));
15:
16:
17:           p2.setName("Roger");
18:           Assert.assertTrue (p1.getId()==null);
19:
20:
21:           Assert.assertFalse (p1.getId()==1324);
22:
23:           Assert.assertEqual (p1.getId(),p2.resetId());
24:   }
25:
26:   @Test
27:   public void test2()
28:   {
29:           Person p2 = new Person ("Jane",456,"anotherstreet");
30:           Person p1 = new Person ("Jane",456,"anotherstreet");
31:           p2.setId("anId");
32:           Assert.assertTrue (p2.resetAddress()!=null);
33:
34:
35:           Assert.assertTrue (p2.setAddress("321anotherst"));
36:
37:
38:           Assert.assertEqual (p2,p1.setId());
39:   }
40:
41:   @Test
42:   public void sometest3()
```

```
43:   {
44:             Person p4 = new Person ("Joiana",123,"bleekerst");
45:             Assert.assertTrue (p4.getId()==123);
46:
47:   }
48:
49:   @Test
50:   public void tesNO10()
51:   {
52:             Person p7 = new Person ("Aida",321,"Mannst");
53:             Person p6 = new Person ("Mark",123,"bleekerst");
54:             Assert.assertEqual (p7.getId(),p6.getId());
55:   }
56:
57:   @Test
58:   public void testNO20()
59:   {
60:             Person p7 = new Person ("Mat",345,"bleekerst");
61:             p7.setAddress("regularStreet");
62:             Assert.assertTrue (p7.getId()==345);
63:
64:
65:             p7.setAddress("someStreet");  }
66: }
Â
```

Code Snippet 81: Generated JUnit code

```
File: <AirlineTest.php>
01:
02: <?php
03: class AirlineTest extends UnitTestCase{
04:   public function setup(){ }
05:
06:   public function teardown(){ }
07:
08:   public function someTest()
09:   {
10:             $p1 = new Person ("John","123","someAddrss");
11:             $p2 = new Person ("Jane","321","street4");
12:             $this->assertTrue($p1->setId("1234"));
13:             $p2->setName("Roger");
14:             $this->assertTrue($p1->getId()==null);
15:             $this->assertFalse($p1->getId()==1324);
16:             $this->assertEqual($p1->getId(),$p2->resetId());
17:   }
18:
19:   public function test2()
20:   {
21:             $p2 = new Person ("Jane",456,"anotherstreet");
22:             $p1 = new Person ("Jane",456,"anotherstreet");
23:             $p2->setId("anId");
24:             $this->assertTrue($p2->resetAddress()!=null);
25:             $this->assertTrue($p2->setAddress("321anotherst"));
26:             $this->assertEqual($p2,$p1->setId());
27:   }
28:
29:   public function sometest3()
30:   {
31:             $p4 = new Person ("Joiana",123,"bleekerst");
32:             $this->assertTrue($p4->getId()==123);
33:   }
34:
35:   public function tesNO10()
36:   {
37:             $p7 = new Person ("Aida",321,"Mannst");
```

```
38:             $p6 = new Person ("Mark",123,"bleekerst");
39:             $this->assertEqual($p7->getId(),$p6->getId());
40:    }
41:
42:
43:    public function testNO20()
44:    {
45:             $p7 = new Person ("Mat",345,"bleekerst");
46:             $p7->setAddress("regularStreet");
47:             $this->assertTrue($p7->getId()==345);
48:             $p7->setAddress("someStreet");
49:    }
50: }
51:
```

Code Snippet 82: Generated PHPUnit code

```
File: <AirlineTest.rb>
01: module AirlineModule
02:
03:   class AirlineTest
04:
05:     def setup
06:         #Setup Method
07:     end
08:
09:     def teardown
10:       #Tear Down
11:     end
12:
13:   def someTest < Test::Unit::TestCase
14:             p1 = Person.new("John","123","someAddrss")
15:             p2 = Person.new("Jane","321","street4")
16:             assert(p1.set_id"1234")
17:             p2.set_name("Roger")
18:             assert(p1.get_id==nil)
19:             assert(!p1.get_id==1324)
20:             assert_equal (p1.get_id.equal?(p2.reset_id))
21:   end
22:
23:
24:   def test2 < Test::Unit::TestCase
25:             p2 = Person.new("Jane",456,"anotherstreet")
26:             p1 = Person.new("Jane",456,"anotherstreet")
27:             p2.set_id("anId")
28:             assert(p2.reset_address!=nil)
29:             assert(p2.set_address"321anotherst")
30:             assert_equal (p2.equal?(p1.set_id))
31:   end
32:
33:   def sometest3 < Test::Unit::TestCase
34:             p4 = Person.new("Joiana",123,"bleekerst")
35:             assert(p4.get_id==123)
36:   end
37:
38:   def tesNO10 < Test::Unit::TestCase
39:             p7 = Person.new("Aida",321,"Mannst")
40:             p6 = Person.new("Mark",123,"bleekerst")
41:             assert_equal (p7.get_id.equal?(p6.get_id))
```

```
42:     end
43:
44:
45:   def testNO20 < Test::Unit::TestCase
46:             p7 = Person.new("Mat",345,"bleekerst")
47:             p7.set_address("regularStreet")
48:             assert(p7.get_id==345)
49:
50:             p7.set_address("someStreet")
51:
52:   end
53: end
```

Code Snippet 83: Generated RubyUnit code for model

10.3 Summary

In this chapter, we have demonstrated the implementation architecture of the Umple Test Language compiler. We have discussed the unit-test generator as a tool and given examples of the different unit-test generators that are currently supported by the tool. We have also shown how the workflow of the templates and implementation classes of the unit tests are done using UmpleTL. The implementation of the Umple test language (separately) does not have a static semantics such as OCL to add a layer of constraints between the metamodel and compiler. This is currently a limitation and can be implemented in future to enhance the quality of the language compiler.

Chapter 11 Evaluation and Testing of the Approach Itself

In this chapter we describe how we have evaluated the model-based testing language and technology we have presented in this thesis. In the following we discuss the evaluation objectives, the testing infrastructure and the mutation testing approach we followed.

11.1 Evaluation Objectives

The framework we are providing has three main outputs that must be evaluated in order to ensure the quality of the tests. Every output of the testing framework is developed through test-driven development, this is explained in details, this is discussed further in Section 11.3. The outputs are shown in Figure 48. The figure has red boxes around the three main outputs that need to be evaluated and tested, these are summarised in the following subsections.

11.1.1 Evaluation Objective 1: Output of TestGenerator

TestGenerator generates the abstract Umple test model as seen in Figure 48. The objective is to test the output of the TestGenerator and is done within the Umple compiler. The goal of this testing phase is to evaluate the correctness of the test templates. Test templates are created in UmpleToTest and we have two main generators being compiled as implementation classes for the generation of model test generator and class test generator.

The way the Umple testing infrastructure is designed enforces a list of features to be covered by any newly added generator. This is done by extending the testing template class. In addition to these essential features in Umple, we add tests that are specifically related to test generation. We are interested in evaluating the following points:

- First, we want to make sure that for each particular model, we get the number of test model files we are expecting.
- Secondly, we want to make sure the content of the test model is generated exactly as we expect them to be.
- Lastly, we want to make sure the Umple Internal Parser is interpretting the test code correctly as well as is generating the correct tokens based on the tasting grammar rule that had been added to Umple.

In order to facilitate these items, we have created several test-related methods in order to make the testing code reusable. We have about 122 test files in order to check this outout of the TestGenerator in the Umple compiler. More details on Section 10.1.

11.1.2 Evaluation Objective 2: The Output of TestModelParser

This is responsible for reading the Umple test model and then generating the tokens and analyzing them.

In order to test the output of the test parser, we have created a list of tests that check whether the test parser is creating the correct tokens based on the given grammar rules. Hence, we want to evaluate the following: First, that test model files are correctly parsed. Secondly, that the parser is generating the correct tokens.

11.1.3 Evaluation Objective 3: The Output Coming From the xUnitGenerator

This is abstracted in the figure. However, in practice each platform should have its own xUnit Generator such as JUnitGenerator, PhpUnitGenerator, RubyUnitGenerator etc. These generators are responsible for generating the unit tests that should be testing a particular Umple system.

In this phase, for each particular feature, we have to fork the output into three main platforms (the currently supported unit systems) JUnit, PhpUnit and RubyUnit. The output of these generators is based on the unit-test templates that had been created. These are given in Table 15:

Table 15: Unit templates

Project	Main Template	NO. of templates	Output
UmpleTToJUnit	JUnitGenerator.ump	23 Templates	JUnitGenerator.java
UmpleTToPhpUnit	PhpUnitGenerator.ump	23 Templates	PhpUnitGenerator.java
UmpleTToRubyUnit	RubyUnit Generator .ump	23 Templates	RubyUnitGenerator.java

For each template, we have the corresponding implementation class. These implementation classes are used by MBTCompiler ‘TestCaseGenerator’ in order to generate the final production. When it comes to templates, we want to make sure that we check the following aspects: First, we want to make sure that the platform-specific files have been generated successfully. We do this by

checking the existence of every expected file. Secondly, we want to make sure the content of the generated files matches what we are expecting.

In order to make this process easier and to enhance the reusability of the test code, we have created several methods that helps in testing the unit test generation. We have created *createUmpleTestSystem* which takes an Umple test model, unit test language and the path of the file and then generates the necessary final production. After that, we can run the necessary checks and validate the correctness of these files.

11.2 The Infrastructure for Testing in Umple

Umple test generation runs through several layers of testing. This includes testing the Umple compiler, the syntactic testing of the generated code and ultimately applying mutation testing for the generated test code. The following describes the whole spectrum of the testing architecture:

We start with a list of tests in order to syntactically verify that the generated abstract tests are what we are expecting. This layer is necessary to align with other generators within the test architecture of the Umple compiler for all implementations. In order to set this up for TestGeneration, we created a test class called TestTemplateTest which extends ClassTemplateTest in Umple. ClassTemplateTest was designed by Andrew Forward, one of the early developers of Umple, as testing template for Umple class which should be inherited by any generator that is supposed to work with an Umple Class. TestTemplatetest extends it in order to get all of the enforced test cases by in ClassTemplateTest. Therefore, we end up with set of test files, those have ClassTemplateTest_* as a prefix that derives from the parent class and those defined on top of the Class Template tests; usually test cases that are not covered by the class test template.

Ultimately, the whole test suite should cover a decent number of test cases that verify the generated code is the same as expected by the developer.

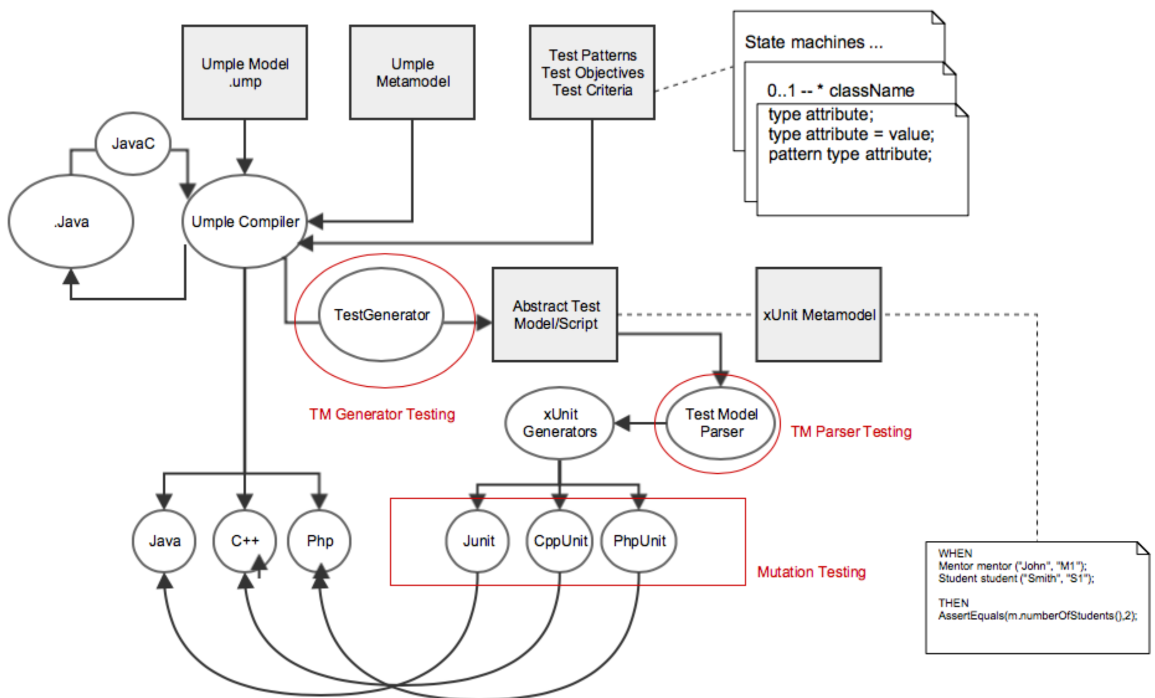

Figure 48: Evaluation of Umple outputs regarding tests

Each of the outputs must then be tested separately to ensure quality of code. For the *TestGenerator*, we wrote a list of syntactic tests in the Umple compiler to evaluate the correctness of the generated output. For 'TestModel Parser Testing', we wrote tokenization tests for the MBTParser output. Finally, the generated executable 'XUnit Code' is evaluated and tested using mutation testing mechanism discussed later.

Ultimately, we end up with xUnit test cases, either Junit, RubyUnit or PhpUnit. In order to evaluate the quality of the test code, we use mutation testing to make sure tests are working.

11.3 Test-Driven Development of Umple MBT

As discussed before, the work in this thesis was developed following the test-driven development method. We adapt agile methods in order to strengthen the quality of the code. Test-driven development has proven to result in systems that have fewer regression [108]–[110] [111]. The workflow for a single feature implementation in Umple MBT is done according to Figure 49.

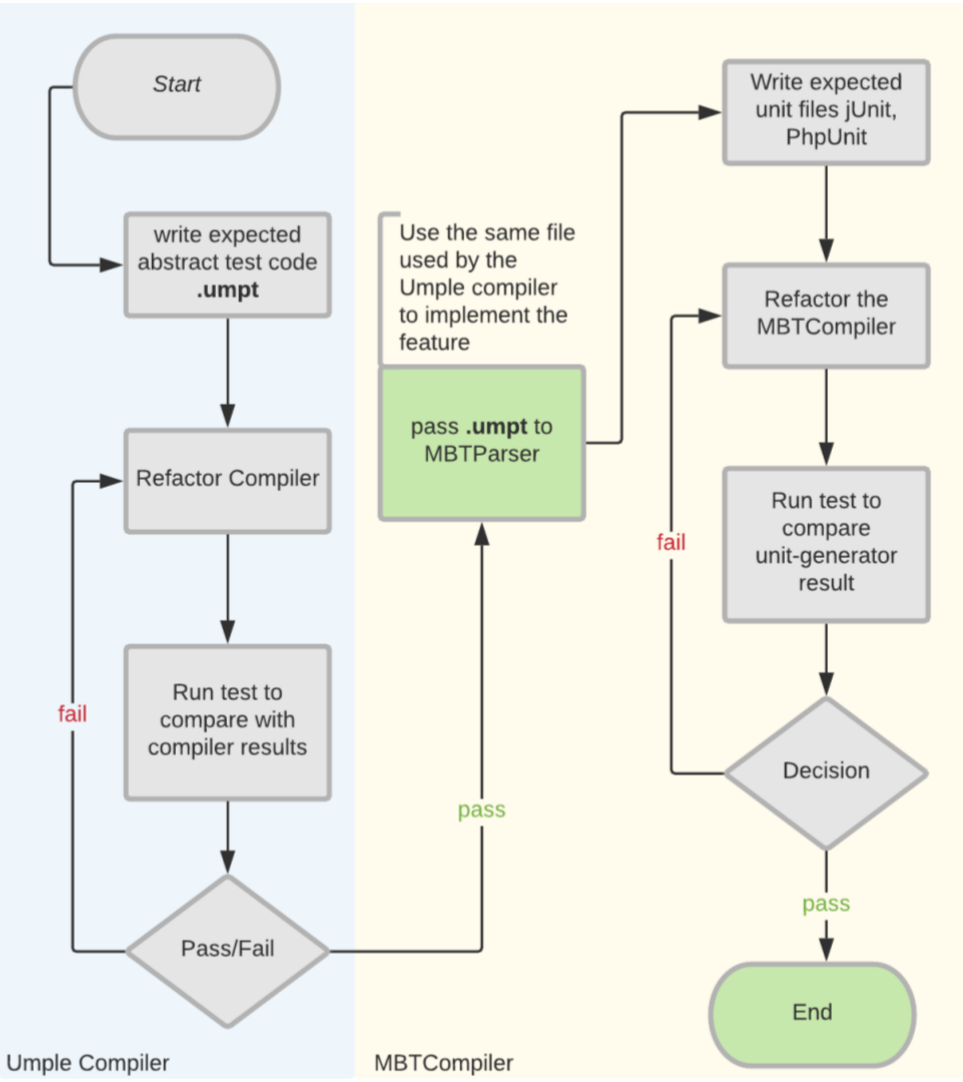

Figure 49: Test-driven development of a single feature

We start the development by determining a single feature to implement. For instance, assume we planned to integrate the *concrete* test concept into the language syntax. In order to do this, we have to implement the feature as part of Umple compiler all the way to the concrete test generation. We start in the Umple compiler by writing a test that compiles an Umple model with the newly added keyword *JUnit*. The parser will not recognize the syntax and will raise a syntax error. We refactor the compiler by adding the necessary grammar rules. The tests now pass, but the semantics generated are not what we are expecting. The test will still fail for error such as seen in Figure 50.

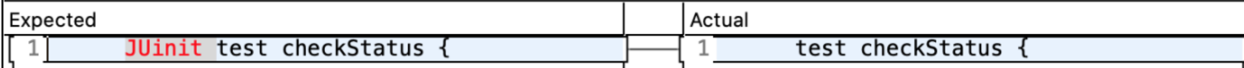

Figure 50: TDD in Umple Compiler

We conduct the necessary refactoring on the Umple compiler by adding the necessary token analysis and processing. This requires we modify the model in order to add new attributes within the test case in order to hold the value of whether it is a concrete test case or not. After several iterations of refactoring, the test case will pass as the feature is now implemented in the Umple compiler and tested as well. The output in this case would be a '.umpt' that contains the abstract test case that has the new feature which is not recognized by the MBTCompiler at this stage. We switch to the MBTCompiler, add a test that targets this feature. We eventually receive a parsing error indicating the newly added keyword is not recognized by the MBTCompiler, we modify the grammar rules and apply same steps we executed at the Umple compiler. We create the necessary token analyzers and make sure the new keyword is detected by the compiler. Finally, for a single feature added by the MBT compiler, we create tests in 3 different places:

cruise.umple.testgenerator.jUnit
cruise.umple.testgenerator.phpUnit
cruise.umple.testgenerator.rubyUnit

Figure 51: Tests for unit test generators

Because the newly added feature will only include such a concrete test case in JUnit generator, then the other two will not be affected by the change. Hence, we will get a failure in the JUnit generator test suite as the following:

cruise.umple.testgenerator.phpUnit.PhpUnitTemplateTest [F
cruise.umple.testgenerator.rubyUnit.RubyUnitTemplateTest
umple.testParser.test.Mbt_parserTest [Runner: JUnit 4] (0.0
cruise.umple.testgenerator.UnitTemplateTest [Runner: JUnit
cruise.umple.testgenerator.jUnit.JUnitTemplateTest [Runner

Figure 52: TDD Result

We then run the necessary refactoring in order to process the generators in order to accommodate the new change, which will only generate the *JUnit* tagged test case if the generator used was the *JUnitGenerator*. When the final production is generated as expected, we then consider this feature implemented. This TDD development allows us to add features that end up tested. In some cases, we end up adding more tests to cover regression. For the list of features that we implement across the two compilers, we go through a series of iterations as seen in Figure 49.

It is Important to mention that there is a difference between the way we handle the test language syntax in the Umple compiler and the way we handle it in the *MBTCompiler*. Some language syntax items are Umple related and they should not be implemented in the MBTCompiler in order to maintain the separation of concerns. For instance, the generic test syntax is Umple related and will not be recognized by the MBTCompiler. When Umple processes the generic test syntax, the result will be a group of tests being injected in the abstract test model '.umpt'. Hence, this type of transformation is horizontal and such a feature should not be expressed in the test-specific language the *MBTCompiler* is parsing.

Another difference is that the MBTCompiler needs grammar rules in order to parse the test model structure such as: 'When', 'Then' , 'Depend' statement and more while the Umple compiler does not need to parse the whole test model structure. The Umple compiler needs the grammar rules for parsing 'test cases' inclusively among other features that are related to Umple. Also, defining test sequences is an Umple model-related feature. When a user defines a test sequence within the Umple model, the order of the test cases in the abstract model will change accordingly. Therefore, the MBTCompiler does not require the implementation of such feature. In case the developer wants to control the test sequence in the abstract test mode, then they can rely on the parsing order in order to express such logic.

As seen in Figure 49 and Figure 48, our final production is testing code. Hence, we still need to evaluate the quality of the final production. The third phase of our evaluation of the testing output checks the correctness of the template generation and the correctness of the MBT compiler. However, we still need a mechanism that checks the quality of such output. The final production for phase 3 is test code and therefore in order to check the quality of test code we are using mutation testing [112] which is explained in the next Section.

11.4 Umple Mutation Testing

Neglecting tests at early stages of the development can dramatically increase the cost of software maintenance [113] .Therefore, writing tests for the system is of paramount importance to the quality of the system. The challenge rises when attempting to evaluate the quality of the test code. Since the test code was written in order to verify the behavior of the system code in the first place, then, logically, writing tests for the code intended for testing will only result in a perpetual testing cycle. Therefore, a fault-based mechanism is often used to verify the quality of tests. One of these mechanisms is a technique that generates several mutated clones of the system called *mutation tests* [114]. Mutation testing is an approach that achieved by imitating user errors when writing the code, these mutated systems are then executed against the original set of test cases. If the tests provided are comprehensive enough, they will detect the intentional errors injected in the mutant. Several issues had been observed in this approach: according to Aichernig [53] , injected errors are not always detected depending on which part of the code had been mutated.

In our approach, we focus on mutating the abstract model of the system as written using Umple. Our work is distinct from earlier work since we are focusing on a set of elements at the model level that represents the system. We cover Umple/UML elements such as associations, attributes and other elements of the model. Therefore, the approach is considered model-oriented mutation testing as we do not mutate the concrete code of the system; such as Java or C++.

We can picture mutation testing as in biology, if we mutate one gene only then it would be possible to observe resulting phenotype difference and hence conclude that our understanding of the gene function is as expected. However, making more than one mutation at the same time would make it hard to determine which mutation is the cause of any defect. Therefore we clone the original system multiple times and apply a single mutation into each copy, at critical part of the system.

The impacts of mutations are expected to vary greatly. If we look at the biological metaphor, there is a big difference between two clones of an animal where one has a different eye color while the other has just one leg. A clone with different eye color can perform most of the basic functionality that the original animal does. On the other hand, the clone with one leg is severely

malfunctional and will not be able to perform the majority of the required tasks. The same concept applies into software system models: mutating a multiplicity to allow ‘any number’ of links rather than a fixed maximum (e.g. 5) is not as likely to have the same kind of negative results as changing an association to be optional (e.g. 0..1) when it should be mandatory.

A major challenge of mutation testing as a concept stems from the number of possible faults that could be present in the system. Without criteria of coverage, the number of mutants could dramatically increase based on the size of the system. Therefore, we have to define coverage criteria to avoid costly execution of a large number of test cases.

The first serious discussion and analysis of mutation testing emerged in 1971; it was written by Lipton as his student report in Carnegie Mellon University [115]. It was then discussed further by Budd [116], and later in depth by Offut in 1991 [117]. The later authors expanded on the number of mutation operators used to determine the type of mutants to be considered.

Jia and Harmen conducted a survey on mutation testing in 2010 [118] covering essential aspects of the methodology. They listed what operators the developer should use for mutation. Jia has also discussed first-order mutation. First order mutation is the process of applying fault injection in the source code once [119]. He also further discusses another type of mutation called *Higher-Order* mutation, which is a process that applies a series of first-order mutations in order to achieve better results in time by cutting the number of mutants to be generated and as a result cutting the computational time.

The majority of studies in the literature focus on the creation of mutants using class and method operators from a concrete and platform-specific perspective. This makes the process resource consuming and increases the computational time if not approached properly. Although a large number of studies are available in the literature, we were able to find only a few papers that tackle the subject from a model-oriented perspective. The work by Krenn in MoMut::UML [53] [120] discussed the analysis of mutation operators for UML class elements. The tool applies its mutation on UML class diagram models in order to generate a list of tests. Despite its exploratory nature, this study offers insight different from the previously discussed work, since it provides observation about the problem from a higher level of abstraction [121].

The finding of Choi and DeMillo in [116] are consistent with what we are trying to establish in this paper. There are also definite similarities to the work done by Krenn and Granda in [122]. Both discuss UML mutation operators for class diagrams. Krenn's work is different than the previous work, since it lists a number of association-related mutations by injecting faults in the multiplicity of the association.

A key motive behind our work is to deliver an evaluation mechanism to check the quality of the automatically-generated tests for the Umple models. Hence, our mutation testing fills such a gap by focusing on mutating the model at different levels but focusing on associations as the main target. This is due to the fact that associations account for a lot of what UML modellers actually do, and because we have focused on them in this thesis. We also use a random-based fault injection mechanism based on the element type. This has not been addressed by Krenn. Tools such as LittleDarwin [123] specifies exactly what faults are to be injected for a particular mutation operator. Although the tool's aim is to mutate Java systems, there are similarities in the way mutation of the source code is approached.

11.4.1 Statement of the Evaluation Problem

There two main motivations for the development of this tool. First, the problem of dealing with a large number of mutants, which can be very time consuming. Hence, a tool for automatic mutant generation is needed. Although there are several tools that target test automation such as PIT [124], such tools only apply mutations at the platform-specific code level which does not tackle the design of the model at a higher level. Having the model mutated instead of (just) the concrete code can detect important types of errors earlier in the development cycle.

The second rationale is that Umple allows for test generation from its models. This feature requires an evaluation method and since the final production of such an approach is test code, then we needed an approach to measure the quality of the generated tests. Thus, the generation of mutants should produce a final score that indicates the level of quality of the generated tests using the following function to calculate the final score [114]:

$$\text{score} = \frac{number\ of\ killed\ mutants}{number\ of\ mutants} \times\ 100$$

In Umple, we want to mutate a part of the model code that is of interest to our testing goals. We have developed a tool that reads Umple files and based on syntactic analysis the tool will capture the targeted code for the mutation process. After processing the model file, the tool will use several algorithms to decide which element to replace the targeted element with. It is important to note that a single Umple model, including its individual classes, can be divided into several files using a feature called 'Mixins' which processes these separate files and finally weaves together the instance of the model in order to generate the desired files. One can also write the whole model code into one file, however, the common practice in the language is to separate model concerns to enhance readability and file structure. For instance, one can write the association and attribute declaration in one while writing the methods and testing code in a separate file. This feature is often used to structure the system [42]. This was discussed earlier in Section 5.1.1.

There are several approaches to generate mutants from an Umple model to be considered. These are:

- **Mutating the metamodel instance**:

  This can be done by modifying the Umple compiler in order to pass the instance of the metamodel to our mutation component. This would require creating a mutation generation as part of the Umple compiler.

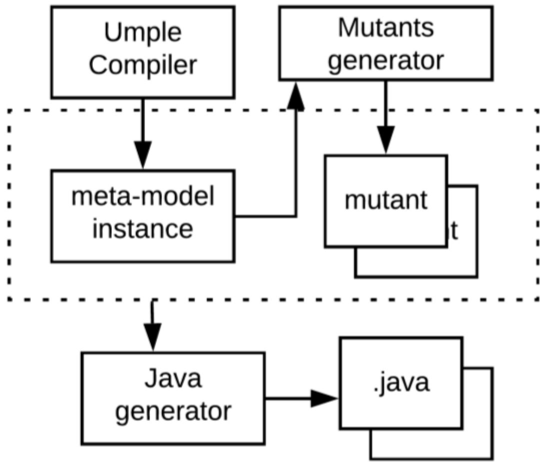

Figure 53: Generating mutants based on the metamodel instance

- **Mutating the model file code:**

This approach requires syntactic analysis of the model since the input in this case is an Umple file in textual format that has all the model specification we aim to mutate.

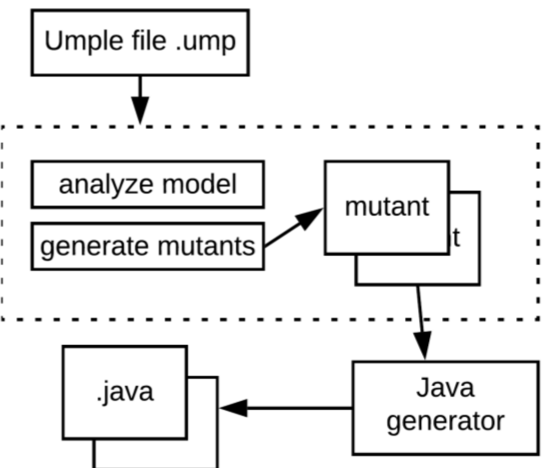

Figure 54: Second approach to process the Umple model for mutation

- **Modifying the generated code:**

  This approach is the least of interest to our objectives. We want to cut time and cost and avoid combinatorial explosion by making modifications at a higher level of abstraction. Modifying the generated code (language code such as Java or Php) is against the philosophy of Umple. Every time Umple compiles its model files, the generated files are overwritten and, therefore, it is not recommended to make a direct modification to the generated code.

The approach we considered for the work in this thesis is the second, we take an Umple file as an input and we process it syntactically.

When we target a single model file for mutation, we inspect the model first to see if it is written in a single file or is divided into multiple files. We do this by processing the model and looking for the 'use' statements which refer to the linked files in the model. Our input for the tool is an Umple model file that is written in a textual form. Hence, we first look for these linked files and include them in the mutation process. If a model has linked files, we add all our candidate files for mutation (including the linked ones) in a list then we randomly select one of these files to be mutated. This single mutation is then considered the target mutation for the whole model.

The following algorithm recursively examines the model file to look for linked files.

Algorithm 2: Mutating Umple model with linked files

Input: model.ump
Output: model_{mutationOperator}.ump
$filesToMutate.add(model)$
$searchForLinkedFiles(model)$
Function `searchForLinkedFiles(`$file$`):`
  if $\exists model \ni linkedFiles$ **then**
    foreach $k \in linkedFiles$ **do**
      if $k \ni linkedFiles$ **then**
        $searchForLinkedFiles(k)$
        $filesToMutate.add(k)$
      else
        $filesToMutate.add(k)$
      end
    end
  else
    `// model has no linked files`
  end
  $R \in filesToMutate$
  $mutate(R)$
  return $mutant$

The example in Figure 55 shows a telephone system modelled in Umple. It contains a list of classes and uses a varied type of associations. This example was taken from the Umple online website that hosts large number of examples[33]. We will be performing the mutation on this model using different inputs and targeting the generation of different mutants.

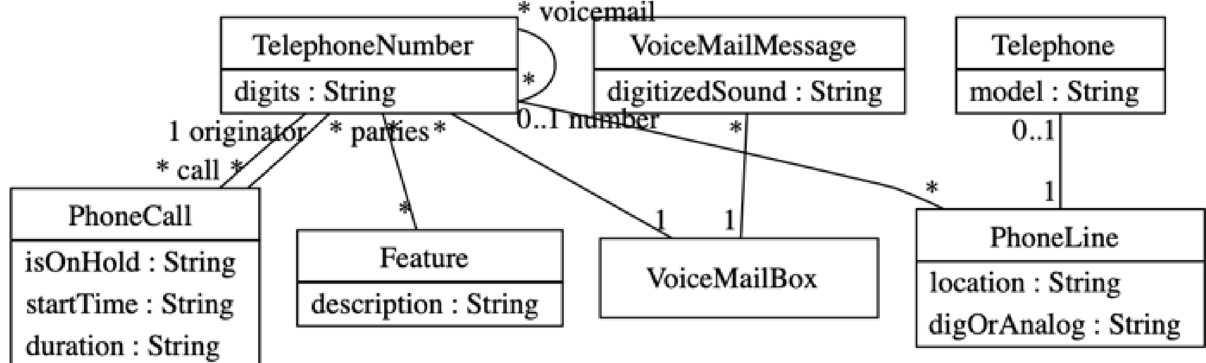

Figure 55: Telephone example in Umple

```
File: <TelephoneSystem.ump>
01: namespace telesystem.core;
```

```
02: class PhoneCall{
03:    isOnHold;
04:    startTime;
05:    duration;
06:    * call -- 1 TelephoneNumber originator;
07:    * -- * TelephoneNumber parties;
08: }
09: class TelephoneNumber{
10:    digits;
11:    0..1 number -- * TelephoneNumber voicemail;
12: }
13: class VoiceMailBox{
14:    1 -- * VoiceMailMessage;
15:    1 -- * TelephoneNumber;
16: }
17: class VoiceMailMessage {
18:    digitizedSound;
19: }
20: class Feature {
21:    description;
22:    * -- * TelephoneNumber;
23: }
24: class Telephone {
25:    model;
26: }
27: class PhoneLine {
28:    location;
29:    digOrAnalog;
30:    * -- * TelephoneNumber;
31:    1 -- 0..1 Telephone;
32: }
```

Code Snippet 84: Telephone system model

Given the previous Umple model 'telephone.system' as seen in Code Snippet 45 we would run the tool in order to generate the mutant system as the following:

Java -jar umple.mutation.jar -generate model.ump -t random -p OneToMany -l Java

The above command will take model.ump as an input and will search for 'OneToMany' associations in the file and randomly choose one of these OneToMany associations and replaced it with a random applicable replacement. We ran this on the commandline and the chosen replacement was 'OptionalOneToMany'. We have also ran a diff file in order to spot the change that had occurred on the model and the result was:

```
1c1
< namespace paper;
---
> namespace paper.OneToMany;
15c15
<  1 -- * TelephoneNumber;
---
>  0..1 -- * TelephoneNumber;
```

Code Snippet 85: Diff result on the model mutation

Algorithm 2 shows how the tool handles the inputs from the command line. The tool allows the user to choose the element to be mutated in the model and also allows for the choosing the replacement. The goal was to generate mutants with random defects; however, we needed the option to allow for passing mutation targets and replacement to test the mutation process itself is working.

Our tool is developed through test-driven development where small tests are created before a feature is implemented; then the necessary features are added to make the tests pass. In order to implement a mutation method for *OneToMany*, we start by creating a mutant that has the mutated code already injected by hand. Then we try to make the tool generate the expected mutant by sending the right inputs. When the tests pass, we are certain that the tool is generating the mutant that we are expecting.

The mutation methods developed in the tools are named after each targeted element. Therefore, if we are targeting *OptionalOneToMany* association in the model, we invoke the method *mutate_optionalOneToMany* and pass the model file as an input and the replacement type. Replacement types are determined by either *Random* or specific. If we are invoking a mutation method targeting random element and also using a random replacement, that means we want our framework to choose a random element for mutation and replace it by a random element. We want to make sure that the replacement is not equal to the target, otherwise it will be meaningless. Algorithm 2 shows how the tool handles random inputs and invoking the needed mutation method. *collection* contains all possible element type for associations for targeting and for replacements.

```
Algorithm 3: Mutating Umple element
  Input: model.ump
  Output: model_{mutationOperator}.ump
  muOp ← mutationOperator
   repEl ← replacementElement
   colletion ← {OneToMany, OptionalOneToMany, OptionalOneToOne, ...}
   Function mutateCode(model.ump, muOp, repEl):
     if muOp = "random" then
       collection.shuffle
        if repEl = "random" then
          collection.shuffle
           repEl = collection.get
           while muOp = muEl do
             collection.shuffle
              repEl = collection.get
          end
          mutate_{muOp}(model, repEl)
       else
          mutate_{muOp}(model, repEl)
       end
     else
       while muOp = muEl do
          collection.shuffle
           repEl = collection.get
       end
       mutate_{muOp}(model, repEl)
     end
     return mutant
```

It is imperative to note that we do not target the generated code for the mutation but rather we mutate the Umple model. This is done for several reasons: First, we want to reduce the number of equivalent mutants.

Equivalent mutant*: Is a mutant that produces the same output as the original program* [125].

When we mutate the Umple model, a single modification we make will scatter among the generated code causing several changes to occur in the generated code. All these changes are related to the source and mostly come from the generated API of Umple; the analysis is based on a manual inspection done by the developer. For instance, changing the association from *OneToMany* to *OptionalOneToMany* will cause changes in the constructor and also several changes in the API, such as *setter/getter* methods. Mutating the model will result in fewer equivalent mutants, which is highly desirable when applying mutation testing and a very common practice [126].

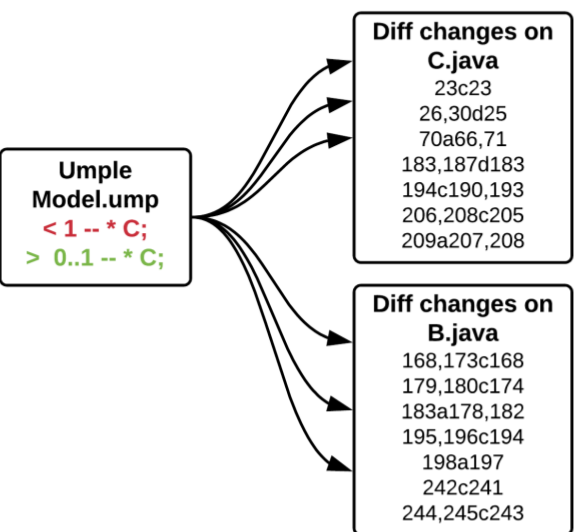

Figure 56: Model scattered mutations

As seen in Figure 56, mutating a single line of code in the model will result in a number of changes occurring in the generated files as a result; as seen from the figure the number of changes occurring in the two classes *C.Java* [3] and *B.java*[4].

Since our mutation framework focuses on models instead of generated code, we believe it will result in more efficient mutation. When a model is mutated, the changes that occur on the generated code is derived from a specific single line of code in the model. Therefore, the by-product mutation in the generated code is representative of a single intended property change.

11.4.2 Umple Mutation Test Tool

The tool is developed as a command line tool that takes an Umple model file as an input with other parameters as indicated in Table 16:

3 http://sultaneid.com/files/C.diff

4 http://sultaneid.com/files/B.diff

Table 16: Umple Mutation Test Tool Parameters

Option	Input
-generate	Takes Umple model file to mutate : model.ump
-t	Indicate the type of replacement for the mutation : random, specific element: OptionalOneToMany, OneToMany, OptionalOneToOne, ManyToMany, direction, Integer, String, Double, Float
-p	Indicates the target element for mutation one can specify 'OptionalOneToMany' for example or use 'full' to generate all possible mutation types.
-l	The language chosen to generate the mutated system: Java, Php .. etc.
-n	Represents the number of mutants to be generated, requires random values

If we set the mutation parameters to double-random, then we can summarize the steps in the following points and also in the given flow chart:

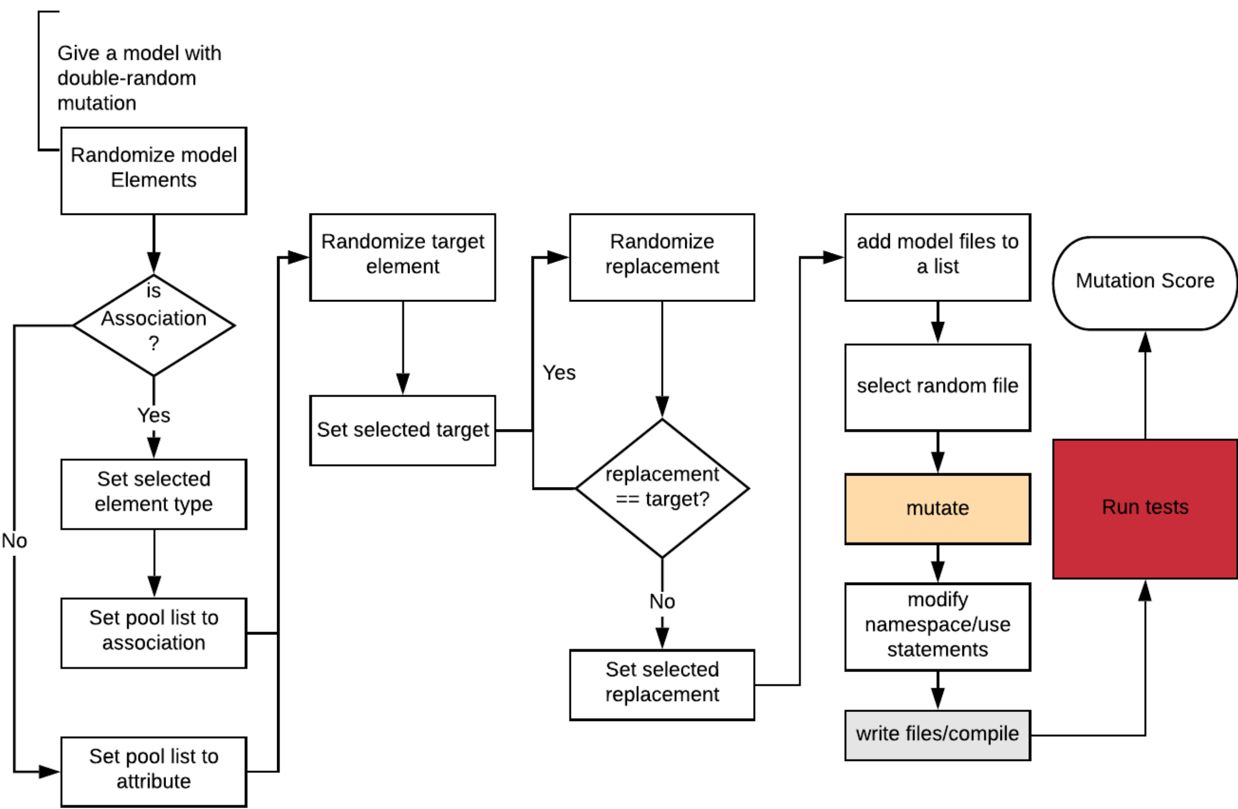

Figure 57: Model mutation using double-random

Mutation testing is a technique where several copies of the same original program are created with small modification to imitate certain errors. Each modified copy is called a mutant, these mutants are supposed to fail when executed. The failure of a mutant is an indication that the original tests (which pass) are validly testing the underlying system. Mutation testing is often used to evaluate the quality of test cases. Whenever a mutant fails it is considered killed, if every mutant is killed, a score is calculated based on the number of killed mutants. Scoring 100% should be the ideal result. Given a list of tests, for each given test we would generate mutants as seen in the following figures

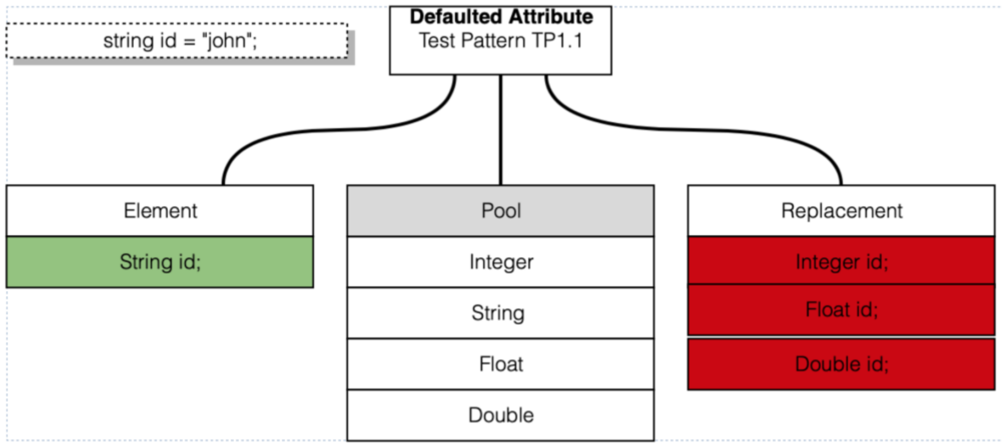

Figure 58: Generated mutants for the generated tests for a regular

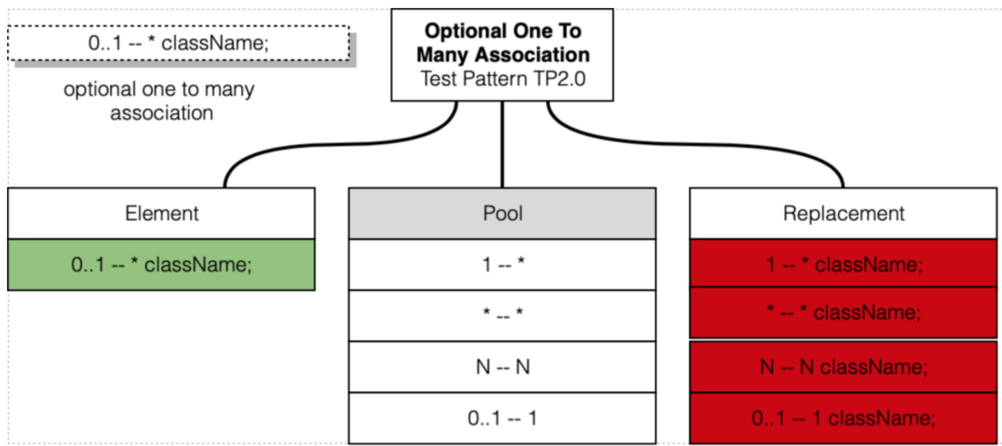

Figure 59: Generated mutants for generated tests for defaulted attribute in Umple.

The generated mutant is then executed against the unit tests generated from the original model. The result is a score calculated based on how many mutants had failed to pass the original generated tests. The higher the score, the better the quality of the tests are.

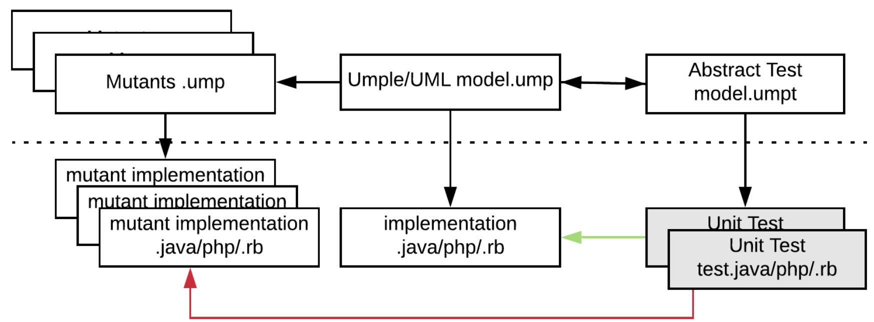

Figure 60: Model mutation testing

Umple also allows the user to add a list of patterns such as lazy attributes, auto-unique and immutable attributes. In this paper, we take aim at association and attribute since they are heavily used in modeling. The following is a set of elements we are covering in Umple currently: Association, attribute, patterns.

Table 17: List of mutation operators in Umple

Mutation Operator **Id**	Mutation Target Element	Syntax	Replacement applicable
UAS1TM	One To Many	1 -- * A	UASO1TM, UAS1TO1, UASMTM
UASO1TM	OptionalOne To Many	0..1 -- * A;	UAS1TO1, UASMTM, UAS1TM
UAS1TO1	OneTo Optional One	1 – 0..1 A;	UASMTM, UAS1TM, UASO1TM
UASMTM	ManyToMany	* -- * A;	UAS1TM, UASO1TM, UASMTM
UASDIR	Directionality	-> A; <- A;	Bi/Uni directional
UATTINT	Integer	Integer name;	UATT*
UATTSTR	String	String name; Or name;	UATT*
UATTFL	Float	float number	UATT*
UATTDO	Double	Double number;	UATT*
UATPLA	Pattern LAZY	lazy String id;	On/Off
UATPIMM	Pattern Immutable	Immutable String id;	On/Off

The way we are approaching mutation testing in Umple is by providing a set of classes that should handle the mutation process:

- Model Mutator: This is responsible for changing part of the models based on the mutation type provided.
- Mutation Checker: This checks the result of the generated test cases against the mutated version of the generated code and returns the result.
- Score Calculator: This calculates the percentage of mutants that were killed by the process.

The goal of Umple mutation testing is to ensure the quality of the generated unit tests. Therefore, in order to achieve this, we first generate the original unit tests, make sure they all pass and have no errors. Then we verify the quality of them by mutating the model and generating a mutated system (in Java for instance), running the tests against the mutated system and finally reporting the results. The things we try to mutate in the Umple model vary based on what type of error we expect an Umple modeler to make. These range from mistakes in multiplicity to mistakes in attribute values. Prior studies had investigated the mutation of UML models, the work by Granda and Voset [122] discussed several categorization for mutations area in the class diagram. The work by Aichering, Brandl, Jobstl, Krenn Schlik in MoMuT:UML [53] discusses mutation operators for models with behavior aspects as they include the generation of mutant as part of their test code generation. In Umple, we the following is a list we are considering on covering for the mutations:

- Multiplicity value: Multiplicity is one of the easiest thing modeler can make mistakes about when writing code. It is important to be able to show that such mistakes can be detected by tests. We would mutate the value between zero, 1, many (*), or a specific upper or lower bound.
- Association: Associations can be mutated in terms of direction and in terms of the target class. We may mutate the association to make it directional instead of bidirectional or vice-versa. Also we can change the name of one of the connected classes to eliminate the association between two classes which would cause the tests to fail. Finally, we can change role names of an association.
- Attributes: We can change the value of attribute at the initialization or change one of the applied stereotypes such as adding or removing *immutable*.
- Constraints: We can modify the logic or constant values in constraints on attributes or method preconditions. We can also change values so they violate constraints.

11.4.3 Case Study: Umple Java Testbed

Code Snippet 86 shows the test harness model which contains a list of files, each supposed to test a particular aspect in Umple. We have excluded the files that are focused on aspects outside of the scope of thesis such as *distributed class* and *composite structure*. Hence, the files included in the analysis are:

```
generate Java "../testharness";

use TestHarnessAttributes.ump;
use TestHarnessAssociations.ump;
use TestHarnessAssociationClass.ump;
use TestHarnessAssociation0_1_mMultiplicity.ump;
use TestHarnessAssociationSpecializations.ump;
```

Code Snippet 86: Test harness model

We run the tool using the using parameters as seen in Code Snippet 87, the input model file *TestHarness.ump*, the parameter *-t random* indicates the type of replacement to be considered for the target element selected using the parameter *-p random*. We call this double-random selection which is interpreted by the compiler as:

"Select a random element in the model and mutate it using a random applicable replacement"

When the compiler randomly selects an association, for instance, then it will only mutate it with some other type of association and the same applies for attributes, directionality and constraints. The parameters are listed in Table 16.

```
-generate TestHarness.ump -t random -p random -l Java -n 25
```

Code Snippet 87: Command for the Umple mutant generator

The result from processing the model will generate a log of the mutated files including file name, related files (using the *use statements*) and also the value of randomized selections.

```
Replacement type used: random
Targeted mutation operator : random
FileName::: TestHarness.ump
Number of linked model files: 5
uFileNAME:::2
uFileNAME:::TestHarnessAssociations.ump
RandomElement Selected: Float
FileName::: TestHarnessAttributes.ump
related file generated : TestHarnessAttributes.ump
```

```
FileName::: TestHarnessAssociations.ump
FileName::: TestHarnessAssociationClass.ump
related file generated : TestHarnessAssociationClass.ump
FileName::: TestHarnessAssociation0_1_mMultiplicity.ump
related file generated : TestHarnessAssociation0_1_mMultiplicity.ump
FileName::: TestHarnessAssociationSpecializations.ump
related file generated : TestHarnessAssociationSpecializations.ump
FileName::: TestHarness.ump
related file generated : TestHarness.ump
Writing file:
/Users/sultaneid/umple20/umple/cruise.umple.mutation/1_Float/FloatMutation_TestHarness.ump
Writing file:
/Users/sultaneid/umple20/umple/cruise.umple.mutation/1_Float/FloatMutation_TestHarnessAssociati
ons.ump
Generating files for:
/Users/sultaneid/umple20/umple/cruise.umple.mutation/1_Float/FloatMutation_TestHarness.ump
Generating files for:
/Users/sultaneid/umple20/umple/cruise.umple.mutation/1_Float/FloatMutation_TestHarnessAssociati
ons.ump
…omitted
generated 25 mutants
```

Code Snippet 88: Mutant generation result for TestHarness model file

The compiler does not randomize the selection between the related files and does not differentiate between the meaning of the names of the files. This means if we have an attribute defined in file *TestHarnessAssociations* then it might be selected randomly for mutation. This happened in the mutation that occurred in line:668 in that file, see Table 18. We are presenting the result of running only 5 mutants for the demonstration. However, the number of generated mutants can be increased as desired. The LOC represents the line of code where the mutation is occuring.

Table 18: Fault injection in the model file: TestHarness.ump

LOC	Mutated File	Before	After
33	OneToManyMutation_TestHarnessAssociations.ump	0..1 mentor -- * StudentC students;	1 mentor -- * StudentC students;
142	OptionalOneToManyMutation_ TestHarnessAssociation.ump	1 mentor -- * StudentJ students;	0..1 mentor -- * StudentJ students;
103	StringMutation_TestHarnessAttributes.ump	Integer g = 42;	String g = 42;
588	OneToManyMutation_TestHarnessAssociations.ump	0..1 mentor -- * StudentAM students;	1 mentor -- * StudentAM students;

668	DoubleMutation_TestHarnessAssociations.ump	Integer id;	Double id;

11.4.4 Case Study: Advisor Example

This example is based on the class diagram shown in Figure 61.

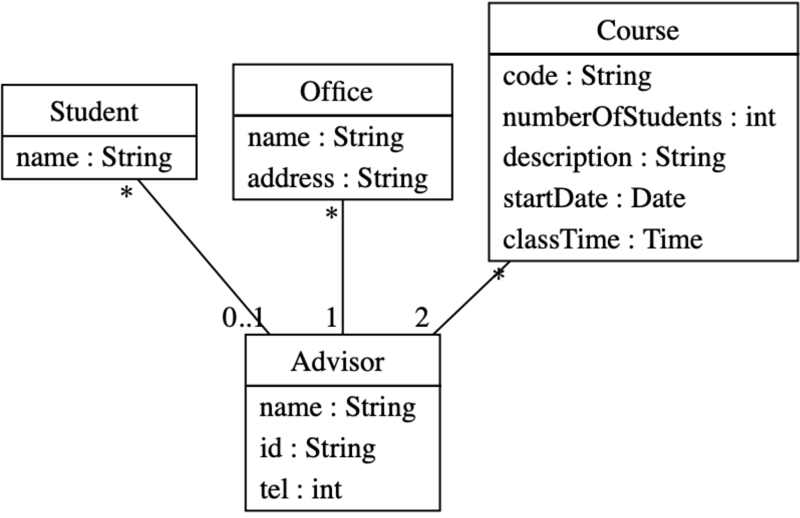

Figure 61: Advisor example model

The following is the result of the test execution

```
StudentTest ✔
└─ attribute_name() ✔
OfficeTest ✔
├─ attribute_name() ✔
└─ attribute_lazy_address() ✔
CourseTest ✔
├─ attribute_code() ✔
├─ attribute_startDate() ✔
├─ attribute_description() ✔
├─ attribute_numberOfStudents() ✔
└─ attribute_classTime() ✔
```

```
├─ AssociationTest ✔
│  ├─ replaceAdvisorInOffice() ✔
│  ├─ CreateAdvisorWithoutStudent() ✔
│  ├─ ReplaceAdvisorInStudent() ✔
│  ├─ createCourseWithoutAdvisor() ✔
│  ├─ createOfficeWithAdvisor() ✔
│  ├─ replaceAdvisor() ✔
│  ├─ removeOfficeInAdvisor() ✔
│  ├─ createAdvisorWithoutOffice() ✔
│  ├─ removeAdvisor() ✔
│  ├─ belowBoundaryInCourseForAdvisor() ✔
│  ├─ overFlowInCourseForAdvisor() ✔
│  ├─ deleteStudentFromAdvisor() ✔
│  ├─ AddStudentToNewAdvisor() ✔
│  └─ AddOfficeToNewAdvisor() ✔
```

Figure 62: Result of unit tests

The result is executed using JUnit5-console-launcher which detects tests cases automatically in the folder and there is no need to generate the test runner in this case. As seen in Figure 62, our

test cases pass successfully. We can modify the input data to verify that the test can catch a fault input. We will modify the value of the Course test in the generated abstract model.

```
CourseTest ✔
├─ attribute_code() ✘ expected: <RandomString1> but was: <CSI9997>
├─ attribute_startDate() ✔
├─ attribute_description() ✔
├─ attribute_numberOfStudents() ✔
└─ attribute_classTime() ✔
```

Figure 63: Failure caught by trying a fault input

Now we have verified that our generated tests work and successfully pass. We then generate a list of mutants in order to evaluate the tests against these generated mutants. We generated 20 mutants and ran each of these mutants against the original tests as we described in the workflow of the evaluation above. Table 19 shows the results of the mutation.

Looking at the results in Table 19, we have executed 20 mutants against the generated testbed. 90% of the mutants were killed and 10% was not detected by the generated testbed. The 10% are the two mutants numbered 4 and 13 in the table. Both of these mutants relate to the lazy attribute definition in Umple. Although these two have lived, we noticed that the Umple compiler raises a warning about them instead of an error. This occurs when an attribute is defined as lazy and also is initialized. This is a bad practice, since the lazy pattern is supposed to do the initialization for the user by assigning a null value. Our testbed checks that the initial value was null but failed since the attribute was already initialized in the model. This means that our testbed failed to capture such an error, and thus, we need to redesign our test templates for lazy attributes in order to handle all possible cases in conformity with the Umple compiler standards.

Table 19: Result of mutation tests for Advisor example

	LOC	Class	Before	After	Result	Failure
1	10	Advisor	0..1 -- * Student;	1 -- * Student;	Killed	Error compiling
2	14	Advisor	2 -- * Course;	3 -- * Course;	Killed	Failure
3	12	Advisor	1 -- * Office;	1 -- 2 Office;	Killed	Failure
4	39	Course	Date startDate = "2020-12-26";	Lazy Date startDate = "2020-12-26";	Lived	Passed
5	41	Course	lazy Time classTime = "12:59:59";	lazy Time classTime;	Killed	Failure
6	20	Student	Name;	immutable name = "John";	Killed	Error Compiling
7	36	Course	Int numberOfStudents	String numberOfStudents	Killed	Error Compiling
8	10	Advisor	0..1 -- * Student;	0..2 -- * Student;	Killed	Error Compiling
9	10	Advisor	0..1 -- * Student;	0..1 -- 1 Student;	Killed	Error Compiling
10	12	Advisor	1 -- * Office	1 -> * Office	killed	Error
11	12	Advisor	1 -- * Office;	* -- * Office;	Killed	Error
12	10	Advisor	0..1 -- * Student;	* -- * Student;	Killed	Error
13	27	Office	lazy address;	lazy immutable address;	Lived	Passed
14	10	Advisor	0..1 -- * Student;	immutable 0..1 -- * Student;	Killed	Umple Error
15	8	Advisor	Int tel;	int tel = 123456789;	Killed	Error Compiling
16	12	Advisor	1 -- * Office;	1 -- 1 Office;	Killed	Error Compiling
17	14	Advisor	2 -- * Course;	2 -- 5 Course;	Killed	Error Compiling
18	14	Advisor	2 -- * Course;	2 -- 0..4 Course;	Killed	Error Compiling
19	14	Advisor	2 -- * Course;	2 -- 1 Course;	Killed	Error Compiling
20	35	Course	description;	defaulted description;	Killed	Umple Error
		Result	90% passed	10% failed		

11.4.5 Categories of Killed Mutants

Mutant behavior can be mainly divided into two categories of output. Either a mutant lives or is killed.

Lives: In the first case, we consider the mutant to be equivalent to the original according to the tests and thus we have to determine whether it is a tolerable behavior or is it a fault that has to be

addressed and caught by the test. In the latter case, we have to refactor the tests accordingly in the majority of the cases.

Killed: This type of mutant has caused the tests to proceed unsuccessfully. Mutants go through stages: Umple compiler, platform-compiler, test execution. The resulting error can be at any of these stages as follows:

- **Umple Error**: These mutants failed to by compiled by the Umple compiler (the first stage). This happens when a mutant violates the syntax of the Umple model, or in some cases generates a mutant that is logically unacceptable to the Umple compiler. An example case is where an immutable association has to be directional and optional one-to-many.
- **Compilation Error:** This type of error is caused when the platform compiler (javac for example) fails to compile the generated mutant. In this case, the mutant is killed at the second stage.
- **Failure:** Failure refers to a mutant that fails at the third stage. This means the mutant was compiled successfully using Umple and was compiled successfully using the target platform (Java) but fails during test execution. These types of mutants are the most effective ones since they make it to the testbed.

Given the second case study, the results show a high portion of the failures occur at the target platform compiler stage (stage 2) and are mostly related to instantiation errors and incompatible constructors. This occurs due to the fact that some dependencies were modified causing the Umple compiler to automatically generate constructors not compatible with those present in the originally generated tests.

11.5 Summary

In this chapter, we have presented the two ways to evaluate the work in this thesis. We have discussed the different generated outputs of the system and how we can verify each part. We have also demonstrated the test-driven development approach we have used to develop the implementation of the system in the Umple compiler and also in the MBTCompiler. Later in this

chapter, we presented the model-oriented test mutation tool that we developed to check the quality of the test suite.

The described evaluation method could be significantly improved by combining model-level mutation and lower-level mutation that targets operators, conditional expressions and other elements in the body of the generated API.

We have concluded that the method works best with pure models that do not embed any arbitrary methods.

Chapter 12 Conclusion

In this thesis, we presented a model-oriented testing language, an approach to use this for model-based testing, along with an implementation in Umple to demonstrate the approach. Our technology includes the ability to automatically generate tests for associations, and to translate our language to multiple xUnit languages. Our language can be written stand-alone or can be embedded in Umple methods. Also, it will work with Umple's other features such as traits and mixsets to allow separation of concerns. This approach raises the level of abstraction of tests and has promise to improve the efficiency and effectiveness of the quality assurance of systems.

We followed a design-science approach, as outlined in Section 1.3. As discussed in that section, the key criteria for a design science activity to be considered a research contribution are that an approach works, is an improvement over the stage of the art, and is not merely routine design.

To prove our approach works, we have provided an open-source implementation, and have given execution results in this thesis. Our approach is an improvement that goes beyond routine design because it has a number of capabilities and features for model-level testing that we could not find in any tool in the literature. Such features include embedding tests class model using a test-specific language, the ability to support feature-based testing, as well as integrating tests with interfaces to enforce implementation of these tests. Other tools do have similarities to these features, but our approach combines and enhances them. Some form of feature-based testing can be done using low-level aspect orientation, but those tools pose challenges when it comes to ease of use.

12.1 Summary of the Contributions

The follow summarizes the main contributions of the thesis.

- **A test generator in the Umple compiler**. Umple now can generate an abstract test model from the Umple class model's associations. This contains the test cases, necessary to test that model, but not in a concrete test language such as JUnit, but rather at an abstract language that can be used to generate *multiple concrete test languages* (Umple currently supports JUnit, PhpUnit and RubyUnit) and can be extended to support any test language. For each construct in Umple,

this generates an appropriate list of tests. This generator uses a tagging mechanism which allows the developer to choose between different criteria for coverage. The generator also uses a test data generator that enables boundary testing and random values testing. This is a general scientific and engineering contribution that can be used by developers, or by researchers who want to further explore model-driven testing.

- **An overall architecture for model-based testing**: that can be used by anyone. The architecture of the separate test language compiler can be re-created, used as is, or extended.
- **Details about how to test several model elements.** These include attributes, associations and other elements. We provide test templates for these problems.
- **A language for the above**. This testing language specifies tests at the abstract level which are then concretized into desired testing platform. This language is the target of the automatic test generator, and also can be used by developers to write their own tests. This is parsed by Umple, but could be re-implemented in other tools, hence it is a general contribution. We did not conduct an empirical study to measure the effectiveness and ease of use of the language. We do not claim the language is optimal at its current state. However, it is effective since we have shown that it works and can be used as described.
- **A parser for the testing language**. This is called MBTParser and is based on Umple Parser. The goal of this parser is to process the generated test model and build a test model that will then be passed to the XUnit Generator (either JUnit, RubyUnit or PhpUnit).
- **A modelling environment for test-driven modelling:** This allows the Umple user to generate their executable tests from an abstract test model that conforms to requirements while incrementally adding model constructs to satisfy those tests. This paradigm should increase the quality and confidence of the system.
- **An infrastructure for the evaluation of the test outputs using mutation testing**. This is done by mutating the Umple model that runs against the original set of tests. A mutation score is calculated as a percent which indicates how adequate the tests are; if most tests fail this indicates that the tests are effective as they are catching potential modeling errors.
- **Flexibility of the testing approach**: We have shown that tests can be embedded within models or specified separately as an independent test-specific language that comes with its own compiler.

- **The ability to conduct test-driven modeling using this language**. Although there are several areas in our approach that can be further researched, we have described how it is achievable with little knowledge of Umple and domain modeling. We have provided a running example to demonstrate that it works.

12.2 Summary of Evaluation

We have evaluated our work in in three ways. We have demonstrated by example that each feature works as advertised; this includes numerous examples that form the test suite for our implementation, produced as a result of our test-driven development process. We have carefully explained how our work provides benefits to developers, including seven capabilities that work synergistically together, and are listed in Section 0. Finally, we showed how mutation testing has been employed, and self-checks that our approach works.

Our infrastructure for test-driven development of the system includes a set of components that to verify the syntactic correctness of the tests. The infrastructure is separate from the one used to test the Umple compiler, this is specifically created to test the test parser and test generators, and includes: 1) *CreateUmpleTestSystem*; 2) *AssertUmpleTestFile*; 3) *AssertTestFileGeneration*; 4) *xUnitTestTemplateTest* (extended by classes for the testing specific unit test systems), and finally 5) *ParserTest*, a set of tests to verify the parser output and token analysis is done correctly.

12.3 Summary of Answers to Research Questions

In Section 1.5 we posed several research questions. Here we outline the answers to these that we have learned while conducting this research

RQ1 asked "*What would be an effective syntax and semantics for an abstract model-level test language for testing UML class models?*" As we progressed through our research, we explored a variety of options for the testing language, and eventually arrived at the language we have described in this thesis. Key features of the language include:

- **Separate sections:** The language separates concerns with sections entitled *Given When* and *Then* to enhance readability.

- **Reference to model elements:** The language targets UML classes and declares tests for each class. The language refers to model elements related to classes, currently including attributes, associations and methods.
- **Separate initialization:** The language separates initializations from the encapsulated test methods.
- **Flexibility of semantics:** The language allows the developer to include concrete test cases as part of the test model. These can to be targeted toward specific platforms. This improves flexibility by giving the developer the option to directly dictate to the compiler that this test is to be injected as-is and its content is arbitrary code.
- **Language consistency:** Since the Umple test language can be processed separately using test-specific language compiler, we want to maintain consistency when integrating the test language syntax as part of an Umple model.

RQ1.1 asked, "*What kind of defects should be detected by this language*", in response to this question, we have presented an automatic test generation mechanism that looks for a number of defects in the model. These defects include the following:

- **Violation of model semantics:** This can happen on different levels. This occurs when the generated code does not behave as shown in the model, and when considering UML/Umple semantics. The list of tests we generate are specific toward elements, therefore, these tests will be compiled using unit system such as JUnit and will check the correctness the generated code. We have created a number of test templates under *UmpleToTest*. These templates target each element in the design model. For instance, if it is an attribute, we check whether it is settable/gettable and whether the initialized value in the model is reflected in the instantiated objects. The same applies for associations and other aspects in the model.
- **Violation of Stereotypes:** Stereotypes serve as constraints, and govern how code is generated and behaves. Examples include *singleton* and i*mmutable*. Semantic assertions for these are abstract assertions that check whether the Umple generator injected specific statement in response to the stereotypes. For instance, if a class is declared using the pattern *singleton*, we check whether the generated code has the attribute that captures the number of objects that was already instantiated from that class. This is done using special syntax *assertAttribute* or *assertMethod*. These two are model-oriented and will be translated into a number of concrete

assertions. Hence, we can check the generated semantics using these abstract assertions where we detect the presence of model elements.

- **Impossible Instantiation Dependency Cycle:** The compiler itself does not incorporate a way to check dependency cyclical issues in the model. However, in the TestGenerator in Umple, we do run a dependency analysis to check whether there are any cycles in the model, as we need to create instantiations for testing. This is very important because not handling this at an early stage of the generation will result in the generation of code that cannot run. Umple already has formal checking tools created by another developer to do something similar but this still requires the generated formal logic to be compiled using a third-party tool called Alloy.

The Umple test language is designed to allow the developer to embed their own tests. Hence, they have the option to create their manual test to target whatever defect they desire to investigate.

RQ1.2 asked, "*How can test generation using this language best be integrated within Umple to allow testing of code automatically generated by the Umple compiler?*". The answer to this question was iteratively derived as case was discovered, and appears in the form of a list of test templates for the model elements. Chapter 3 discussed this in detail. We have integrated the Umple testing language within the Umple compiler in two ways:

1. **Integrating an Automatic Test Generation Mechanism:** This is done by adding a test generator as part of the Umple compiler. This defined a list of test templates that covers a subset of the Umple elements.
2. **Integrating Test Language Syntax:** This allows the developer to embed their manual tests within the Umple model. We have done this by extending the Umple grammar and providing the necessary token analysis within the Umple internal parser. This integration includes the capability of adding test cases within a class, defining assertions within methods and add new Umple-related test syntax that will generically generate test templates (for large models).

In addition, the integration of testing included the conformity with other features in Umple such as, traits, mixsets and mixins. Test can be embedded within a mixset to allow for feature-based testing, can be integrated as part of a trait in order to enhance reusability of tests, and can be

specified as required in an interface in order to enforce the implementation of tests on implementer classes.

RQ1.3 asked "*How can this language be designed to facilitate test-driven modelling in Umple*", To answer this question we have demonstrated how the implemented solution can be used to derive the model according to test specifications that were written based on requirements. This is addressed in Chapter 6 in detail. This was promoted when we originally developed the test language separately. The ability to write the test separately using our test-specific language compiler allows us to be able to generate the concrete unit test firsthand where we then can respond to each unit test and make sure they pass. The design architecture we decided to choose made the process smoother and improved the approach toward test-driven modelling using the Umple test language.

RQ1.4 asked, "*What should the testing architecture be for the use of this language*?". We found a good approach was to initially develop a separate compiler for the test language, written using Umple and the Umple Template Language. The details of the architecture were addressed in Chapter 9. We also discussed the flow of test inputs between the Umple compiler and the test compiler. The modification made to the Umple architecture itself included adding a *TestGenerator* as part of the Umple compiler and also the two template implementation classes *ClassTestGenerator* and *ModelTestGenerator*. We also designed a metamodel for our testing language within the Umple MBT compiler. The MBT compiler uses a similar convention to the architecture of Umple. We implemented *TestCaseGenerator* which determines the language defined in the test model, and then calls the corresponding unit test generator; which has to be one of the three: *JUnitTestcaseGenerator* , *PhpUnitTestcaseGenerator* and *RbyUnitTestcaseGenerator*. We demonstrated the architecture of this part in detail in Section 9.1.

RQ2 asked, "*How can we evaluate the quality of the test specifications and the automatically generated tests?*" The answer to this question is discussed in detail in Chapter 10. In order to evaluate the quality of the generated tests, we have adapted a test-driven development method in order to ensure our test artifacts: the parser, the generator and analyzers are producing the output we are expecting. This is done through intensive testing at different levels. The outputs we are testing are the following:

- **The test model generator output:** We have added a list of tests to check the output of the TestGeneratorin order to make sure every test template we add is generated correctly.
- **The unit test generators output:** This is done within the MBT compiler, we have added a number of tests to the output for each unit generator we have currently: JUnit, PhpUnit and RubyUnit. Each generator's output is checked to make sure the unit test templates are propagated correctly.

We have looked in the literature for the best methodology to evaluate the quality of the tests. Mutation testing is well-accepted mechanism for such a purpose. In order to improve our evaluation, we have created a separate tool called Umple Mutation Test, which is a tool that will take an Umple model as input and then generate a number of mutants in which each mutant has a fault injected. This fault injection method is used to imitate the errors and mistakes a developer may make while writing the target code. Hence, our tool does the same thing. It targets elements in the model and changes the declaration of an element. The mutation can include attribute type and stereotypes, or association multiplicity and directionality. We focus the most on attributes and associations. The tool can generate any number of mutants by running an algorithm that randomly looks for fault injection targets. We use these mutants to evaluate the quality of the automatically generated tests. This evaluation technique had been discussed in detail in Section 10.3.

To respond to RQ2.1, which asked "*What types of evaluation techniques should be applied in order to ensure the quality of the tests using the language from RQ1?*" we look at the mutation testing, which we discussed earlier, as the evaluation technique to approach this. As stated above, we have developed a mutation tool that mutate any Umple model file and generates a list of mutants.

12.4 Summary Comparison of Our Work To Other Research Tools

In Section 2.4, we described several existing technologies for model-based testing. Table 1 and Table 2 provide a comparison of those tools. The following describes how our work compares to these tools, using the columns of these tables:

OS Supported: *Independent*. Our tool works on any platform that supports JVM, as do three of the existing tools.

Languages Supported: *UML-derived Umple, Java, PHP, Ruby*. Our work compares favorably. Most tools only support one modeling language or one programming language.

DSL: *Yes, both internal and external*. Our tool incorporates its own Domain Specific Language, as does QML. Our DSL is internal when it is being hosted by the Umple modelling language and also external (an independent language) when compiled separately with the MBT compiler

Customizable criteria: *Yes*. Our tool allows selection of testing coverage criteria for a specific testing session, writing of customized model-based tests, creation of generic tests, and incorporation of platform-specific tests. Our tool goes far beyond the other tools in this regard.

UML coverage: *Class diagrams*. Our tool generates tests for class diagrams (attributes and associations only) at the current time, although test templates and customized tests can be written for other features (based on the API for those UML features generated by Umple).Most of the compared tools support only a single UML diagram type, particularly state machines. Our tool is therefore similar to other tools in this respect.

Type: *Open Source*. About half the tools are open source.

Test Data: *Yes*. Our tool generates randomized test data.

Constraints: *Limited*. Our tool handles multiplicity constraints. Umple does support a subset of OCL, but we do not support it currently.

Model Driven: *Yes*. Our tool is designed for model-driven testing. Only one other tool is oriented towards that.

A couple of other ways that our tool stands out are:

Support for Test-Driven Modelling: Our tool can support test-driven modeling by generating unit test from any test model written in its language.

Large Model Testing: Our tool provides a mechanism to test very large models. The ability to query elements from a model or class using signatures allows us to search among large number of elements; this allows us to test more with less code. The tool also allows test elements based on elements names that matches a specific query, such as attributes that starts with a particular prefix or end with a special suffix. We have not found a tool in the literature that supports such a feature.

The conclusion is that our tool has many advantages overall as compared to the other tools with which we have compared it, but that the other tools also have their own strengths. No tool duplicates what our tool does, so our tool does provide many advances.

12.5 Further Work

Several adaptations and feature were left for the future to ensure the thesis is focused. The future work we discuss here is either extensions that could bring enhancement to the work presented in this thesis or simply ideas that emerged through curiosity to try to approach a particular issue in a different manner.

12.5.1 Expanding The Current Implementation

Umple has a large number of features. We were able to cover a subset of Umple elements for our test generation, focusing on associations. We can expand the implementation of the tool to cover a larger number of Umple features. Possible elements that can be covered by our testing framework in future are :

- Automatic test generation for tracing. Currently, tracing is used in Umple [127] [128] to introduce monitoring capabilities to the model. We can detect these trace directives in Umple and generate the necessary test cases. Also, a possibility would be to allow tracing of test cases in the Umple model where the developer can collect information from running these test classes.
- Automatic test generation for Umple distributable classes [129] . Umple classes can be *distributable* meaning that multiple runtime VMs with the same Umple code can operate as a distributed system. When a class is declared as such, we can generate tests for distributable classes, for instance, using *assertMethod* to check whether the generated code always have the distribution-related methods; such as *setRealObj* and

setRemoteObject. These methods are always generated for distributable classes and running *assertMethod(setRemoteObj)* will always check the presence of the method in the generated code. Beyond this, would be testing that distribution works properly. We can also investigate further into what is the best template to test different scenarios of distributed classes in Umple.

- Umple automatic test generation for traits [45] is currently limited to testing attributes in traits. Writing test cases for associations in traits requires abstracting the objects name which had not been investigated thoroughly. We hypothesize that initialization of objects should have naming related to the trait name such as *someIdentifiable1* which will be replaced (or translated) by the client name such as *somePerson1*. This helps in very simple cases of object initialization. This still requires further investigation. Also, currently, traits support user-defined test cases at one level and does not inherit elements that had been inherited from a nested trait. This issue was noticed when we created the analyzers for Umple testcase within traits and will be fixed in the future.
- Allow for aspect-oriented programming for the test semantics. Currently, code injection into the generated test code is disabled. We can enhance the test templates by allowing the user to inject code before or after a particular test case. In Umple, if AOP added to the model using syntax like that shown in Code Snippet 89 (assuming we have class X with attribute a), this will inject the AO code at the end of the *setA* method since the keyword used was *after*. The result is shown in Code Snippet 90.

```
Class X
{
  a;
  whenWasASet;
  after setA {
     setWhenWasASet(getA() + System.currentTimeMillis());
  }
}
```

Code Snippet 89**: AOP in Umple**

```
public boolean setA(String aA)
{
 boolean wasSet = false;
 a = aA;
 wasSet = true;
 // line 9 "model.ump"
 setWhenWasASet(getA() + System.currentTimeMillis());
```

```
// END OF UMPLE AFTER INJECTION
return wasSet;
}
```

Code Snippet 90: Result of AOP on setA

When it comes to testing, we can apply AOP to a test case where we can inject some assertions after/before the test case, as shown in Code Snippet 91.

```
Class X
{
  a;
  whenWasASet;
  test checkSetA {
   X a1("a", "10 PM");
   assertTrue(a1.getA() == "a");
  }

  after checkSetA {
     assertTrue (a1.getWhenWasSet() == "10 PM");
  }
 }
```

Code Snippet 91: AOP injection in Umple model for test

Adding the AOP for a test case will result in the test case including the code injection within the production code as seen in Code Snippet 91.

```
Test checkSetA {
   X a1("a", "10 PM");
   assertTrue(a1.getA() == "a");
   // lin .. "model.ump"
   assertTrue (a1.getWhenWasSet() == "10 PM");
  // END OF UMPLE AFTER INJECTION
   }
```

Code Snippet 92: Aspect-Oriented Programming for Testcase

Using AO for test cases has some similarities to the before/after feature discussed in Section 7.5. They are similar in the fact that we can inject some test code at the beginning or at the end of the method. However, they differ in the syntax and location of declaration. AO code can be injected anywhere in the class while the before/after syntax for testing can only be used within the declaration statements of the either the test case or the assertions. Also, AOP is broader, it can target joint point to match elements such as method calls.

- Covering state machines for automatic test generation. This requires finishing the state machine test templates, and implementing of a list of coverage criteria for state machines such as: Edge-coverage, Structural coverage, round-tripping, etc. Hence, it requires state-based graph coverage to check the flow of the model and also structural control coverage.

12.5.2 Conducting a Empirical Study of the Test Language and Technology

This future work item concerns producing will assess the expressiveness and comprehensibility of the test language We have designed a series of experiments that target a group of participants but we could not obtain the ethics approval and also conduct the experiment by the deadline for the thesis, while also delivering the complete technology and the written thesis.

The participants will be screened to only include people with basic knowledge of software modeling and existing unit testing languages.

In one experiment, we would give some participants a simple model in Umple and ask them to write Unit tests in our test language. We would give a different group of participants the same system in Java and ask them to write tests in Junit. The dependent variables we would measure would include the time taken and the number of errors made

The group working with Umple would be given basic training about Umple in general and be specific training into the Umple testing language. We will provide equivalent training in Junit for the non-Umple group to avoid bias.

The group using Umple and our test language will also, subsequent to the experiment, be given substantive samples of the testing language to comment on, and will be asked for their feedback about the extent to which they understand the language, whether they think it is sufficiently expressive, and whether they think they would use it if given the opportunity.

12.5.3 Mutation Enhancements

There are several enhancements that could be done to the mutation system, as future work. One of these would be to mutate the Umple compiler itself and test against its own testbed. This is currently not feasible since Umple compiler has a very large number of embedded methods; some of these adjust the semantics of the models and are not covered by the mutation system. In

addition, our approach as we discussed is semi-automatic. The current system does not execute the generated mutants automatically against the original tests. Implementing this requires a test runner that will point the testbed to the mutants instead of the original system. There are different paths to do the automation. For instance, we can create a test runner that compiles the mutant and overwrites the original system and then run the test container. This can be done repeatedly to cover all mutants and reduce the load of the test execution.

12.5.4 More support for Test-Driven Modelling

Test-driven modelling requires more attention in order to be more viable for developers. For instance, we can add a transformation mechanism that will analyze given software requirements and then automatically generate the corresponding test model. Such tool requires analysis text and applying a part of speech tagging mechanism. Such an NLP tool can help in deriving tests directly from requirements and also reduce the time and effort spent into mapping the requirements into tests.

12.5.5 Graphical Representation

Our Umple test language is currently only textual. One of the possible extensions that we can consider adding to it is a graphical representation of the test model. There are several challenges, however, that we would need to look into. For instance, we would need to determine comprehensible shapes to represent a *test class* while differentiating test classes from design classes.

In addition, a graphical representation of the different test sequences and the flow of events would enhance the analysis and readability of the tests. Just like we do with Umple, we can create a duality view where the user can see the test code and the test graphical representation to better grasp the logic of it.

References

[1] B. Beizer and O. Vinter, "Software testing techniques: Bug Taxonomy and Statistics (Amended Appendix)," *Appendix*, 1990.

[2] A. a Sawant, P. H. Bari, and P. M. Chawan, "Software Testing Techniques and Strategies," *J. Eng. Res. Appl.*, vol. 2, no. 3, pp. 980–986, 2012.

[3] M. Sarma, P. V. R. Murthy, S. Jell, and A. Ulrich, "Model-based testing in industry," in *Proceedings of the 5th Workshop on Automation of Software Test - AST '10*, 2010, pp. 87–90.

[4] Q. Xie, "Developing cost-effective model-based techniques for GUI testing," in *Proceeding of the 28th international conference on Software engineering - ICSE '06*, 2006, p. 997.

[5] R. Baillargeon and R. Flores, "Model Driven Testing," *SAE Tech. Pap.*, pp. 01–0743, 2008.

[6] O. Badreddin, "Umple," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, 2010, vol. 2, p. 337.

[7] O. M. G. A. Specification and C. Bars, "OMG Unified Modeling Language (OMG UML)," *Language (Baltim).*, no. November, pp. 1–212, 2007.

[8] M. Fowler, *Domain-Specific Languages*, vol. 5658. 2010.

[9] A. Bertolino, "Software Testing Research : Achievements , Challenges , Dreams," *Futur. Softw. Eng. FOSE '07*, no. September, pp. 85–103, 2007.

[10] S. T. March and G. F. Smith, "Design and natural science research on information technology," *Decis. Support Syst.*, 1995.

[11] S. Gregor and A. R. Hevner, "Positioning and presenting design science research for maximum impact," *MIS Quarterly: Management Information Systems*. 2013.

[12] P. Ammann and J. Offutt, *Introduction to Software Testing*, vol. 54. 2008.

[13] F. Fleurey, J. Steel, and B. Baudry, "Validation in model-driven engineering: Testing model

transformations," in *Proceedings - 2004 1st International Workshop on Model, Design and Validation, SIVOES - MoDeVa 2004*, 2004.

[14] A. Andrews, R. France, S. Ghosh, and G. Craig, "Test adequacy criteria for UML design models," *Softw. Testing, Verif. Reliab.*, vol. 13, no. 2, pp. 95–127, 2003.

[15] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *Proceedings - International Conference on Software Engineering*, 2015.

[16] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Softw. Eng.*, 2015.

[17] D. Cohen, M. Lindvall, and P. Costa, "An Introduction to Agile Methods," *Advances in Computers*, vol. 62, no. C. pp. 1–66, 2004.

[18] Y. Zhang, "Test-driven modeling for model-driven development," *IEEE Softw.*, vol. 21, no. 5, pp. 80–86, 2004.

[19] K. Beck, "Test-Driven Development By Example," *Rivers*, vol. 2, no. c, p. 176, 2003.

[20] S. Schulz, J. Honkola, and A. Huima, "Towards model-based testing with architecture models," in *Proceedings of the International Symposium and Workshop on Engineering of Computer Based Systems*, 2007, pp. 495–502.

[21] S. Fraser, K. Beck, B. Caputo, T. Mackinnon, J. Newkirk, and C. Poole, "Test driven development (TDD)," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2003.

[22] A. Mishra and A. Mishra, "Introduction to Test-Driven Development," in *iOS Code Testing*, 2017.

[23] M. Utting and B. Legeard, *Practical Model-Based Testing*. 2007.

[24] S. R. Dalal *et al.*, "Model-based testing in practice," *Proc. 1999 Int. Conf. Softw. Eng. (IEEE Cat. No.99CB37002)*, pp. 285–294, 1999.

[25] S. Schulz, "Automating Functional Test Design with Model-Based Testing," *SAE Int. J. Passeng. Cars - Electron. Electr. Syst.*, vol. 5, no. 1, pp. 27–33, 2012.

[26] M. Blackburn, R. Busser, and A. Nauman, "Why Model-Based Test Automation is Different and What You Should Know to Get Started," *Int. Conf. Pract. Softw. Qual. Test.*, p. 16, 2004.

[27] M. Mussa, S. Ouchani, W. Al Sammane, and A. Hamou-Lhadj, "A survey of model-driven testing techniques," in *Proceedings - International Conference on Quality Software*, 2009.

[28] M. Utting, B. Legeard, A. Pretschner, and B. Legeard, "A Taxonomy of Model-Based Testing," *Softw. Testing, Verif. Reliab.*, vol. 22, no. April, pp. 297–312, 2006.

[29] O. O. Adesina, S. S. Somé, and T. C. Lethbridge, "Modeling state diagrams with and-cross transitions," in *CEUR Workshop Proceedings*, 2016.

[30] O. O. Adesina, T. C. Lethbridge, S. S. Somé, V. Abdelzad, and A. B. Belle, "Improving formal analysis of state machines with particular emphasis on and-cross transitions," *Comput. Lang. Syst. Struct.*, 2018.

[31] H. D. Patel and S. K. Shukla, "Model-driven validation of systemC designs," in *Proceedings - Design Automation Conference*, 2007, pp. 29–34.

[32] A. Huima, "Implementing Conformiq Qtronic," *Test. Softw. Commun. Syst.*, vol. 4581, pp. 1-12–12, 2007.

[33] T. C. Lethbridge, G. Mussbacher, A. Forward, and O. Badreddin, "Teaching UML using umple: Applying model-oriented programming in the classroom," in *2011 24th IEEE-CS Conference on Software Engineering Education and Training, CSEE and T 2011 - Proceedings*, 2011.

[34] A. Hartman, M. Katara, and S. Olvovsky, "Choosing a test modeling language: A survey," *Hardw. Software, Verif. Test.*, 2007.

[35] G. P. S. A. for TTCN-3, "General Purpose SUT Adapter for TTCN-3," *Master Thesis*, 2004.

[36] G. M. Kapfhammer, "Software testing," in *Computer Science Handbook, Second Edition*, 2004.

[37] "Introduction to software testing," *Choice Rev. Online*, 2008.

[38] V. Garousi and J. Zhi, "A survey of software testing practices in Canada," *J. Syst. Softw.*, 2013.

[39] I. Hooda and R. Singh Chhillar, "Software Test Process, Testing Types and Techniques," *Int. J. Comput. Appl.*, 2015.

[40] O. Badreddin, "Umple: a model-oriented programming language," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, 2010, vol. 2, pp. 337–338.

[41] M. H. Orabi, A. H. Orabi, and T. Lethbridge, "Umple as a component-based language for the development of real-time and embedded applications," in *MODELSWARD 2016 - Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development*, 2016.

[42] T. C. Lethbridge, V. Abdelzad, M. Husseini Orabi, A. Husseini Orabi, and O. Adesina, "Merging modeling and programming using umple," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016, vol. 9953 LNCS, pp. 187–197.

[43] M. a. Garzon, H. Aljamaan, and T. C. Lethbridge, "Umple: A framework for Model Driven Development of Object-Oriented Systems," *2015 IEEE 22nd Int. Conf. Softw. Anal. Evol. Reengineering*, no. September, pp. 494–498, 2015.

[44] H. Aljamaan, T. Lethbridge, M. Garzón, and A. Forward, "UmpleRun: A dynamic analysis tool for textually modeled state machines using umple," in *CEUR Workshop Proceedings*, 2015.

[45] V. Abdelzad and T. C. Lethbridge, "Promoting traits into model-driven development," *Softw. Syst. Model.*, 2017.

[46] V. Abdelzad, “Extended traits for Model-Driven software development,” in *CEUR Workshop Proceedings*, 2015.

[47] T. C. Lethbridge and A. Algablan, “Applying Umple to the rover control challenge problem: A case study in model-driven engineering,” in *CEUR Workshop Proceedings*, 2018.

[48] H. Aljamaan and T. C. Lethbridge, “Towards tracing at the model level,” in *Proceedings - Working Conference on Reverse Engineering, WCRE*, 2012.

[49] M. Utting, “Position Paper : Model-Based Testing Overview of Model-Based Testing,” in *Verified Software: Theories, Tools, Experiments. ETH Zürich, IFIP WG 2*, 2005, pp. 1–9.

[50] S. Schulz, “Automating functional test design with model-based testing,” *SAE Tech. Pap.*, 2012.

[51] P. Saubert, S. Bevanda, and M. Beißer, “Model-based Testing Strategy in Vehicle Testing Using the Example of car2go,” *22nd Aachen Colloq. Automob. Engine Technol.*, pp. 1603–1610, 2013.

[52] P. Saubert, S. Bevanda, and M. Beißer, “Model-based Testing Strategy in Vehicle Testing Using the Example of car2go,” *22nd Aachen Colloq. Automob. Engine Technol.*, 2013.

[53] W. Krenn, R. Schlick, S. Tiran, B. Aichernig, E. Jöbstl, and H. Brandl, “MoMut::UML model-based mutation testing for UML,” in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings*, 2015.

[54] J. Peleska, “Industrial-Strength Model-Based Testing - State of the Art and Current Challenges,” *Electron. Proc. Theor. Comput. Sci.*, 2013.

[55] J. Grabowski, A. Wiles, C. Willcock, and D. Hogrefe, “On the design of the new testing language TTCN-3,” *Test. Commun. …*, pp. 1–16, 2000.

[56] J. Grabowski, D. Hogrefe, G. R??thy, I. Schieferdecker, A. Wiles, and C. Willcock, “An introduction to the testing and test control notation (TTCN-3),” in *Computer Networks*, 2003, vol. 42, no. 3, pp. 375–403.

[57] J. Zander, Z. R. Dai, I. K. Schieferdecker, and G. Din, “From {U2TP} Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing,” in *Testing of Communicating Systems*, 2005, pp. 289–303.

[58] A. Ulrich, “The ETSI Test Description Language TDL and its application,” 2014.

[59] D. Arnold, J.-P. Corriveau, and V. Radonjic, “Open Framework for Conformance Testing via Scenarios,” *Companion to 22Nd ACM SIGPLAN Conf. Object-oriented Program. Syst. Appl. Companion*, pp. 775–776, 2007.

[60] D. Arnold, J. P. Corriveau, and W. Shi, “Modeling and validating requirements using executable contracts and scenarios,” in *8th ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2010*, 2010, pp. 311–320.

[61] A. Guiotto, B. Acquaroli, and A. Martelli, “MaTeLo: Automated Testing Suite for Software Validation,” in *European Space Agency, (Special Publication) ESA SP*, 2003, no. 532, pp. 253–261.

[62] V. Panthi and D. P. Mohapatra, “Automatic test case generation using sequence diagram,” *Adv. Intell. Syst. Comput.*, vol. 174 AISC, pp. 277–284, 2013.

[63] L. Bergmans and C. V. Lopes, “Aspect-oriented programming,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1999.

[64] P. Kumar and T. Baar, “Using AOP for discovering and defining executable test cases,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010.

[65] S. Benz, “AspectT: Aspect-oriented test case instantiation,” in *7th International Conference on Aspect-Oriented Software Development, AOSD.08 - Research Track Proceedings*, 2008.

[66] M. Boussaa, O. Barais, B. Baudry, and G. Sunýe, “Automatic non-functional testing of code generators families,” in *GPCE 2016 - Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, co-*

located with SPLASH 2016, 2016.

[67] J. Offutt and A. Abdurazik, “Generating Tests from UML Specifications,” *«UML»’99 — Unified Model. Lang. SE*, pp. 416–429, 1999.

[68] R. Heckel and M. Lohmann, “Towards model-driven testing,” *Electron. Notes Theor. Comput. Sci.*, vol. 82, no. 6, pp. 37–47, 2003.

[69] G. Meszaros, *xUnit Test Patterns*. 2004.

[70] R. Binder, “Design for Testability in Object-Oriented Systems,” *Commun. ACM*, 1994.

[71] O. Badreddin, A. Forward, and T. C. Lethbridge, “Exploring a model-oriented and executable syntax for UML attributes,” in *Studies in Computational Intelligence*, 2014.

[72] M. Husseini Orabi, A. Husseini Orabi, and T. C. Lethbridge, “Umple as a Template Language (umplE-TL),” in *MODELSWARD 2019 - Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development*, 2019.

[73] O. Badreddin, A. Forward, and T. C. Lethbridge, “Improving code generation for associations: Enforcing multiplicity constraints and ensuring referential integrity,” in *Studies in Computational Intelligence*, 2014.

[74] M. Fowler, “UML Distilled: A Brief Guide to the Standard Object Modeling Language,” *Pearson Paravia Bruno Mondad*, 2004.

[75] J. Hartmann, M. Vieira, H. Foster, and A. Ruder, “A UML-based approach to system testing,” *Innov. Syst. Softw. Eng.*, 2005.

[76] O. M. Group, “OMG Unified Modeling Language TM (OMG UML), Version 2.5,” *InformatikSpektrum*, 2017.

[77] C. Doungsa-ard, K. Dahal, A. Hossain, and T. Suwannasart, “Test data generation from UML state machine diagrams using GAs,” in *2nd International Conference on Software Engineering Advances - ICSEA 2007*, 2007.

[78] K. T. Cheng and A. S. Krishnakumar, “Automatic functional test generation using the extended finite state machine model,” in *Proceedings - Design Automation Conference*, 1993.

[79] M. Aggarwal and S. Sabharwal, “Test case generation from UML state machine diagram: A survey,” in *Proceedings of the 2012 3rd International Conference on Computer and Communication Technology, ICCCT 2012*, 2012.

[80] H. Zhu, P. A. V. Hall, and J. H. R. May, “Software unit test coverage and adequacy,” *ACM Comput. Surv.*, 1997.

[81] R. W. Selby, “Comparing the Effectiveness of Software Testing Strategies,” *IEEE Trans. Softw. Eng.*, 1987.

[82] A. Eriksson and B. Lindström, “UML Associations - Reducing the Gap in Test Coverage between Model and Code,” in *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development*, 2016.

[83] M. Friske, B. H. Schlingloff, and S. Weißleder, “Composition of Model-based Test Coverage Criteria,” *MBEES’08 Model. Dev. Embed. Syst.*, 2008.

[84] D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, B. Helmkamp, and D. North, *The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends*. 2010.

[85] H. Bünder and H. Kuchen, “A model-driven approach for behavior-driven GUI testing,” in *Proceedings of the ACM Symposium on Applied Computing*, 2019.

[86] P. Baker, Z. R. Dai, J. Grabowski, Ø. Haugen, I. Schieferdecker, and C. Williams, “Model-driven Testing: Using the UML testing profile,” *Model. Test. Using UML Test. Profile*, pp. 1–183, 2008.

[87] Junit.org, “JUnit 5.” [Online]. Available: http://junit.org.

[88] J.-M. Jazequel and B. Meyer, “Design by contract: the lessons of Ariane,” *Computer (Long. Beach. Calif).*, 1997.

[89] O. O. Adesina, T. C. Lethbridge, and S. S. Some, “A fully automated approach to discovering nondeterminism in state machine diagrams,” in *Proceedings - 2016 10th International Conference on the Quality of Information and Communications Technology, QUATIC 2016*, 2017.

[90] T. Lethbridge, “Umple Online,” 2020. [Online] Available: try.umple.org.

[91] A. Urdhwareshe, “Object-Oriented Programming and its Concepts,” *Int. J. Innov. Sci. Res.*, 2016.

[92] N. Liberman, C. Beeri, and Y. B. D. Kolikant, “Difficulties in learning inheritance and polymorphism,” *ACM Trans. Comput. Educ.*, 2011.

[93] N. Hunt, “Unit testing,” *JOOP - J. Object-Oriented Program.*, 1996.

[94] T. Hauser, *the Art of Unit Testing*. 2009.

[95] P. Tonella, “Evolutionary testing of classes,” in *ISSTA 2004 - Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2004.

[96] T. C. Lethbridge, “Umple Metamodel,” 2019. [Online]. Available: metamodel.umple.org.

[97] O. Badreddin, “Umple: A model-oriented programming language,” in *Proceedings - International Conference on Software Engineering*, 2010.

[98] I. Schieferdecker, “Model-Based Testing,” *IEEE Softw.*, 2012.

[99] M. R. Blackburn, R. D. Busser, A. M. Nauman, and T. R. Morgan, “Model-based testing in practice,” in *INFORMATIK 2006 - Informatik fur Menschen, Beitrage der 36. Jahrestagung der Gesellschaft fur Informatik e.V. (GI)*, 2006.

[100] M. Sarma and R. Mall, “Automatic test case generation from UML models,” in *Proceedings - 10th International Conference on Information Technology, ICIT 2007*, 2007.

[101] O. Badreddin, T. C. Lethbridge, A. Forward, M. Elaasar, H. Aljamaan, and M. A. Garzon, “Enhanced code generation from UML composite state machines,” in *MODELSWARD*

2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, 2014.

[102] Umple Team, "Umple Parser." [Online]. Available: https://github.com/umple/umple/tree/master/UmpleParser.

[103] T. J. Parr and R. W. Quong, "ANTLR: A predicated-LL(k) parser generator," *Softw. Pract. Exp.*, 1995.

[104] J. Bovet and T. Parr, "ANTLRWorks: An ANTLR grammar development environment," *Softw. - Pract. Exp.*, 2008.

[105] S. Efftinge and M. Völter, "oAW xText: A Framework for Textual DSLs," *Proc. Work. Model. Symp. Eclipse Summit*, 2006.

[106] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*. 2016.

[107] M. Husseini Orabi, A. Husseini Orabi, and T. C. Lethbridge, "A textual notation for modeling and generating code for composite structure," in *Communications in Computer and Information Science*, 2019.

[108] K. Beck, "Test Driven Development: By Example," *AddisonWesley Longman*. 2002.

[109] H. Erdogmus, G. Melnik, and R. Jeffries, "Test-Driven Development," in *Encyclopedia of Software Engineering*, 2010.

[110] D. Janzen and H. Saiedian, "Test-driven development: Concepts, taxonomy, and future direction," *Computer (Long. Beach. Calif).*, 2005.

[111] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 2010.

[112] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2014.

[113] M. J. Harrold, "Testing: A Roadmap," *Proc. Conf. Futur. Softw. Eng. - ICSE '00*, no. May 2001, pp. 61–72, 2000.

[114] D. Hamlet, "Foundations of software testing," *ACM SIGSOFT Softw. Eng. Notes*, 2004.

[115] R. Lipton, "Fault Diagnosis of Computer Programs" Carnegie Mellon University1971.

[116] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Theoretical and empirical studies on using program mutation to test the functional correctness of programs," 2003.

[117] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing Android apps," *Inf. Softw. Technol.*, 2017.

[118] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*. 2011.

[119] Y. Jia and M. Harman, "Higher Order Mutation Testing," *Inf. Softw. Technol.*, 2009.

[120] W. Krenn and R. Schlick, "Mutation-driven Test Case Generation Using Short-lived Concurrent Mutants -- First Results," *CoRR*, 2016.

[121] B. K. Aichernig *et al.*, "Model-based mutation testing of an industrial measurement device," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014.

[122] M. F. Granda, N. Condori-Fernández, T. E. J. Vos, and O. Pastor, "Mutation operators for UML class diagrams," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016.

[123] A. Parsai, A. Murgia, and S. Demeyer, "LittleDarwin: A Feature-Rich and Extensible Mutation Testing Framework for Large and Complex Java Systems," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017.

[124] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: A practical mutation testing tool for Java (Demo)," in *ISSTA 2016 - Proceedings of the 25th*

International Symposium on Software Testing and Analysis, 2016.

[125] A. Parsai, A. Murgia, and S. Demeyer, "A Model to Estimate First-Order Mutation Coverage from Higher-Order Mutation Coverage," in *Proceedings - 2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016*, 2016.

[126] A. Parsai and S. Demeyer, "Dynamic mutant subsumption analysis using LittleDarwin," 2017.

[127] H. Aljamaan, T. C. Lethbridge, and M. A. Garzón, "MOTL: a textual language for trace specification of state machines and associations," in *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, 2015.

[128] H. Aljamaan, T. C. Lethbridge, O. Badreddin, G. Guest, and A. Forward, "Specifying trace directives for UML attributes and state machines," in *MODELSWARD 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, 2014.

[129] M. H. Orabi, A. H. Orabi, and T. C. Lethbridge, "Concurrent programming using umple," in *MODELSWARD 2018 - Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*, 2018.