University of Twente

Master Thesis

Domain-Specific Language Testing Framework

Author:
Robin A. ten Buuren

Examination Committee:
dr. Luís Ferreira Pires
dr. ir. Rom Langerak
Niek Hulsman

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Software Engineering Research Group
Department of Computer Science

October 2015

UNIVERSITY OF TWENTE

Abstract

Faculty of Electrical Engineering, Mathematics and Computer Science
Department of Computer Science

Master of Science

**Domain-Specific Language
Testing Framework**

by Robin A. ten Buuren

Domain-specific languages (DSLs) are languages developed to solve problems in a specific domain, which distinguishes them from general purpose languages (GPLs). One characteristic of DSLs is that they support a restricted set of concepts, limited to the domain. The benefits of using a DSL include improved readability, maintainability, flexibility and portability of software.

After a DSL is deployed, the user-developed artifacts (e.g., models and generated code) have to be tested to ensure correctness. Organizations spend up to 50% of their resources on testing. Testing is therefore important, expensive and time critical. The problem is that testing can also be error-prone, commonly experienced as unpopular or tedious work. Moreover, when using the conventional ways of testing (e.g., writing unit tests) the tester is required to have a thorough understanding of the system under test (SUT).

There are several testing techniques available that can be applied to domain-specific languages, each focused on a specific artifact or aspect of the DSL. However, to the best of our knowledge, no generic framework available allows the generation of tests for domain-specific artifacts or systems that use these artifacts using the domain-specific models.

In this report we present a framework for the generation of tests using domain-specific models. These tests can be used to verify the correctness of the artifacts generated using the domain-specific model and systems that use these artifacts. By generating the tests instead of manually developing them, development time is reduced while usability is improved.

The test generation process consists of three phases: generalization, generation and specification. In the generation phase the domain-specific model is transformed to an instance of a newly developed generic metamodel, to abstract away from language-specific features. In the generation phase, generic test cases are generated that achieve branch/condition coverage. In the specification phase the generic test cases are transformed to executable test code. By keeping the test cases generic, several types of tests can be generated using the same generic test case.

We show that the developed framework supports a multitude of languages and resulting test types. We also discuss several areas where the framework can be extended with addition features. An important lesson learned during the research and development is that a DSL (testing) framework should be setup modular, extensible and small.

Acknowledgements

This thesis presents the results of the final assignment in order to obtain the degree Master of Science. Now that I have completed my thesis, I would like to thank a couple of people:

First, I would like to thank Luís Ferreira Pires and Rom Langerak for being my supervisors on behalf of the university. Thank you for the meetings we had and the provided support. Your feedback significantly helped me to improve this research.

Second, I want to thank Niek Hulsman and Ramon Ankersmit for supervising on behalf of Topicus Finance. Thank you and the team for the (stand-up) meetings we had during the last eight months. Our discussions helped me to obtain the result I was aiming for.

Third, I would like to thank Topicus Finance for letting me perform my final assignment at their office in Zwolle. I want to thank all the colleagues and fellow students in the company for their support and the great time we had.

Last but not least, I want to thank my family and friends for their continuous support over the last couple of months. I could not have achieved my goals without all the support I received.

Robin Alexander ten Buuren

Enschede, October 2015

Contents

Abstract ii

Acknowledgements v

Contents viii

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Objectives 3
- 1.3 Approach 4
- 1.4 Report structure 4

2 Testing techniques 6
- 2.1 Black-box testing 6
- 2.2 White-box testing 8
- 2.3 Model-based testing 10
  - 2.3.1 Domain-specific modeling 11
  - 2.3.2 FSM testing 12
  - 2.3.3 UML testing 12
- 2.4 Behavior-driven development 14
- 2.5 Domain-specific testing language 16
- 2.6 Automated web testing 18
- 2.7 Testing levels 19
- 2.8 Application to domain-specific languages 20

3 Approach 22
- 3.1 General development goals 22
- 3.2 Transformation chain 24
- 3.3 Common programming elements 25
  - 3.3.1 Common language specification 26
  - 3.3.2 Common operands 26
  - 3.3.3 Common operators 27
  - 3.3.4 Common conditionals 29
- 3.4 Domain-specific elements 29
- 3.5 Solution overview 30

4 Common elements metamodel **33**
4.1 Model definition 33
4.2 Mapping 42
4.2.1 The mapping DSL 43
4.2.2 Mapping domain-specific elements 46
4.2.3 Example artifacts 47
4.3 Framework options 47

5 Case generation **50**
5.1 Expression evaluation 50
5.1.1 Expression trees 51
5.1.2 String transformation 53
5.1.3 String evaluation 56
5.2 Value generation 56
5.3 Variable assignment 62
5.4 Example 63

6 Test generation **67**
6.1 Test case generation 67
6.2 JBehave 69
6.2.1 Generation 69
6.2.2 Execution 71
6.3 Selenium 72
6.3.1 Generation 72
6.3.2 Execution 73

7 Case study **74**
7.1 Finan Financial Language 74
7.2 Generalization 76
7.3 Generation 81
7.4 Specification 81
7.4.1 JBehave 82
7.4.2 Selenium 85
7.5 Conclusion 89

8 Final remarks **90**
8.1 Conclusions 90
8.2 Research answers 92
8.3 Future work 93

A Expression mapping grammar **95**

B Precedence grammar **99**

C Precedence transformer **103**

D Generated JBehave story **111**

E Generated Selenium test **113**

F Selenium functions **117**

Bibliography **119**

Chapter 1

Introduction

This chapter is structured as follows: Section 1.1 presents the motivation for this work. Section 1.2 defines the research objectives. Section 1.3 describes the approach taken to achieve the objectives. Section 1.4 presents the structure of this report.

1.1 Motivation

For this research we use the definition of domain-specific language given in van Deursen et al. [1]:

"A *domain-specific language* (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."

Domain-specific languages (DSLs) are languages developed to solve problems in a specific domain, which distinguishes them from general purpose languages (GPLs). One characteristic of DSLs is that they support a restricted set of concepts, limited to the domain. DSL can be developed from scratch, but also by extending a general purpose language. Domain-specific languages are diverse because they are developed for a specific domain and can be developed using several methods. Another key characteristic is that DSLs are often declarative, meaning that the code describes *what* should be computed instead of *how* it should be computed. The benefits of using a DSL include improved readability, maintainability, flexibility and portability of software [2].

According to Kurtev et al. [3], a DSL is "a set of coordinated models". The domain knowledge (e.g., concepts of the domain and their relations) can be represented as a model to which the DSL models must validate, therefore being a metamodel. This

metamodel is called the domain definition metamodel (DDMM), and is used as the abstract syntax of the DSL. Next to an abstract syntax, DSLs can have multiple concrete syntaxes. A concrete syntax can be acquired by developing a transformation from the abstract syntax to a specific language, e.g., UML notation, or by developing a concrete syntax independent of any other language. The concrete syntax defines the notation used to express models [4]. The execution semantics of DSLs can be achieved by developing a transformation model that converts the DDMM to a (executable) language, called the target language, like, for example, Java or some formalism, e.g., in case of mathematical models. The semantics describe the function of the model in the target language.

DSLs can be developed using Model-Driven Engineering (MDE), which is based on the Object Management Group (OMG) Model-Driven Architecture (MDA) approach of using (meta)models as cornerstones for the construction of systems. MDE has a broader scope than MDA as it combines process and analysis with architecture [5]. The benefits of using MDE are that the models can be made platform-independent, reusable and easily adaptable. The drawbacks of using MDE are that the development of the models used to generate code results in extra upfront costs and companies have to change their development methods to adhere to MDE [6].

After a DSL is deployed, the user-developed artifacts (e.g., models and generated code) have to be tested to ensure correctness. Organizations spend up to 50% of their resources on testing [7]. Testing is therefore important, expensive and time critical. The problem is that testing can also be error-prone, commonly experienced as unpopular or tedious work and when using the conventional ways of testing (e.g., writing unit tests) the tester is required to have a thorough understanding of the system under test (SUT) [8]. There are several testing techniques available, e.g., model-based testing, that can be applied to domain-specific languages, each focused on a specific artifact or aspect of the DSL. However, to the best of our knowledge, no generic framework available allows the generation of tests for domain-specific artifacts or systems that use these artifacts using the domain-specific models.

1.2 Objectives

In this research we work towards a framework, in which different types of tests can be generated using the domain-specific models. These tests can be used to test the generated model artifacts and systems that use the model artifacts.

The main research objective of this thesis is:

To improve the quality of testing of generated artifacts from domain-specific language models and systems that use these artifacts

To achieve this objective, we developed a framework in which the generated artifacts can be tested using generated tests, e.g., validating the generated code using the developed models. This testing can be done on different levels, e.g., testing the generated artifacts using unit tests, testing an engine or application that uses the artifacts or testing the output of a model provided with input. Since the system under test using the artifacts can be diverse, several testing approaches like, for example, behavior-driven development and automated web testing could be applied.

We consider three quality aspects of the framework:

1. Effectiveness: by applying the framework, test development should take less time compared to manual test development. Instead of manually developing tests, tests are generated using the model.
2. Usability: by applying the framework, test development should be more user-friendly compared to manual test development. By supporting several generators, different types of test can be generated using the same model. All these tests would otherwise be developed manually.
3. Correctness: by applying the framework, the system under test should contain fewer bugs compared to non-tested systems.

During the development of the framework, the following research questions were considered:

RQ1. Which testing techniques are available and what is their coverage?

RQ2. How can these testing techniques be applied to domain-specific languages?

RQ3. How to deal with different language constructs and syntax?

RQ4. How to assess the reusability and verify the quality of the testing framework?

1.3 Approach

In order to achieve the main objective of this research and answer the research questions, we took the following steps:

1. Perform a literature study on testing techniques, their coverage and their applicability to DSLs
2. Define and implement a framework for the generation of tests using domain-specific models
3. Test the framework by performing a case study
4. Validate the uses of the testing framework for two domain-specific languages and identify its limitations
5. Validate that the framework improves the quality of testing by detecting introduced bugs in the system under test and questioning stakeholders about their experience with the framework.

1.4 Report structure

This report is further structured as follows:

Chapter 2 discusses some available testing techniques and their coverage. The testing levels of software systems are analyzed and the testing techniques are examined with regards to their applicability to domain-specific languages.

Chapter 3 discusses our approach and structure of the testing framework. The chapter describes the general goals of the framework and gives an overview of the transformation chain. Common elements as well as domain-specific elements are discussed. An overview of the developed framework is also given.

Chapter 4 discusses the generic metamodel used by the framework as well as the mapping from domain-specific metamodels to the generic metamodel. It also gives an overview of the options the framework provides.

Chapter 5 discusses how the framework generates test values using the generic models.

Chapter 6 discusses how these test values can be used to create generic test cases. It also shows how these test cases can be transformed to JBehave stories and Selenium tests.

Chapter 7 discusses how the testing framework can be used in a work environment by analyzing its application on a case study.

Chapter 8 gives our conclusions, answers the research questions, analyzes the reusability and quality of the framework and discusses future work.

Chapter 2

Testing techniques

This chapter discusses a number of testing techniques, often used to check the functionality of a software system and improve its quality. Section 2.1 explains black-box techniques, while Section 2.2 discusses white-box techniques. Section 2.3 identifies different model-based testing techniques (e.g., FSM and UML testing) and Section 2.4 discusses behavioral-driven development. Section 2.5 describes testing using a domain-specific testing language. Section 2.6 discusses automated web testing and the Selenium tool. Section 2.7 describes the different testing levels of software systems. Section 2.8 analyzes the application of the discussed testing techniques to models and code developed using domain-specific languages and concludes this chapter.

2.1 Black-box testing

Black-box testing, also called function testing, is based on the idea that the tester does not know how the system under test works internally, i.e., the source code is unknown. To apply black-box testing the user provides inputs and checks the outputs based on the requirements specification of the system under test (SUT).

A benefit of black-box testing is that non-programmers can perform it since no internal program knowledge is required. A downside of black-box testing is that exhaustive input testing, testing all the possible inputs, is impossible since the source code is unknown. The tester usually cannot deduct all the possible inputs based on the requirements alone.

There are several techniques for black-box testing, like equivalence partitioning, boundary value testing, cause-effect graphing and random testing [9].

- **Equivalence partitioning testing**

  In equivalence partitioning testing, the input is divided into partitions under the assumption that testing one element of that partition is equivalent to testing the whole partition. An example would be choosing the number 34 if the input range is an integer from 0 to 100.

- **Boundary value testing**

  In boundary value testing, the boundaries of inputs and outputs are tested because these are prone to errors. Since a program can have a large number of inputs and outputs, boundary value testing can result in a large number of tests. For example: the test cases for an integer input range from 0 to 10 are:

  - -1 (below lower boundary)
  - 0 (start lower boundary)
  - 10 (end upper boundary)
  - 11 (above upper boundary)

- **Cause-effect graphing**

  Cause-effect graphing can be applied to testing by converting the requirements specification of the SUT into causes and effects. For example, the functional requirement specifying that to save a file 'the file name length must consist of six characters otherwise an error is displayed', can be converted into one cause and two effects: the cause is *The length of the file name consists of six characters*, while the effects are 1. *The file is saved* and 2. *Error message "invalid file name length" is displayed.* After the conversion, each cause and effect receives a unique number.

  The requirements specification also needs to be converted into a Boolean graph thereby linking causes and effects. A decision table should be created using this graph, which is then used to construct test cases.

  The benefit of using this technique is a clear representation between causes and effects in terms of a Boolean graph and a decision table. This technique also reduces the time necessary to search for the cause of an error since the error can be related to an effect, which is directly linked to its possible causes in the cause-effect graph.

- **Random testing**

  Random testing is a technique in which the tester tries a random subset of inputs. Although the chance of actually finding an error is low, by randomly choosing input some unlikely error might be detected. This technique should therefore be seen as an add-on technique, only to be used in combination with other testing techniques [9].

2.2 White-box testing

White-box testing, also called structural testing, is based on the internal structure of the SUT. Instead of focusing on the software requirements specification, the source code itself is analyzed using logic.

A benefit of white-box testing is that the source code is checked, thereby increasing the probability of finding bugs introduced by the programmer. Since the specification is not consulted during white-box testing, this approach does not consider the conformance to the requirements.

Examples of white-box techniques are: statement coverage, branch coverage, condition coverage and branch/condition coverage [9].

- **Statement coverage**

  The goal of statement coverage testing is to make sure every statement (line of code) is executed at least once. Although the whole program can be tested, branches can still be missed. Statement coverage is therefore a weak code coverage approach [9].

- **Branch coverage testing**

  In branch coverage testing, also called decision coverage testing, all the program branches are tested. A branch is a choice point of the program, like an *if-then(-else)* construction, so the if-clause as well as the else-clause must be executed during the test run [9]. This technique does however have some flaws as illustrated by the example below:

  $if(A \;\&\&\; (B \;||\; FooBar()))$
  $Foo();$
  $else$
  $Bar();$

Test cases:

1. A = True, B = True
2. A = False

The test cases make sure full branch coverage is achieved since both the if-clause as well as the else-clause is executed. However, function *FooBar()* is never called due to lazy evaluation resulting in an untested possible cause of errors.

- **Condition coverage testing**

  Condition coverage is similar to branch coverage, since it is also based on branches. Instead of each branch having to evaluate to true and false, each condition within a branch must now be covered [9]. However, branches may still be missed by the test, as illustrated in the following case:

  $if\ (A\ \&\&\ B)$
  $Foo();$
  $else$
  $Bar();$

  Test cases:

  1. A = True, B = False
  2. A = False, B = True

  The test cases make sure full condition coverage is achieved since both conditions A and B are tested with the values *True* and *False*. However, *Bar()* is never executed, resulting in an untested possible cause of errors.

- **Branch/condition coverage testing**

  To completely test all branches, a combination of branch and condition coverage, called branch/condition coverage testing should be performed. The goal is to test not only every branch but also every condition in every branch. This is a strong method of coverage testing since it combines several techniques and makes up for the missed coverage of each individual technique [9].

Gray-box testing

Where black-box testing focuses on the requirement specifications and functionality of the SUT, and white-box testing on the source code and logic, gray-box testing combines these two testing techniques. Using the knowledge of the internal program structure (white-box style), can influence the development of requirements specification-based test cases (black-box style).

For example, parameters that should be tested for each combination using black-box techniques, can be tested separately if the tester knows these parameters are only used separately. This knowledge can be gained by inspecting the source code [9].

2.3 Model-based testing

Model-based testing (MBT) is a testing technique that uses a model of the system under test thereby testing at a higher abstraction level. Based on this model, test cases can be generated. Although models can also be seen as black boxes, since the source code could be unknown and only input and output are observed, thereby making MBT a kind of black-box testing, this specific field of testing is discussed separately due to its relation to Model-Driven Engineering (MDE). Both MBT and MDE use models as basis, and generate artifacts using these models.

The MDA abstraction levels (e.g., platform-independent and platform-specific) can also be applied to testing, which is graphically displayed in Figure 2.1. It shows that the developers can take several paths to develop test code. Two examples are:

1. Starting from a platform-independent system model → transforming to a platform-independent test model → transforming to test code
2. Starting from a platform-independent system model → transforming to a platform-specific system model → transforming to a platform-specific test model → transforming to test code.

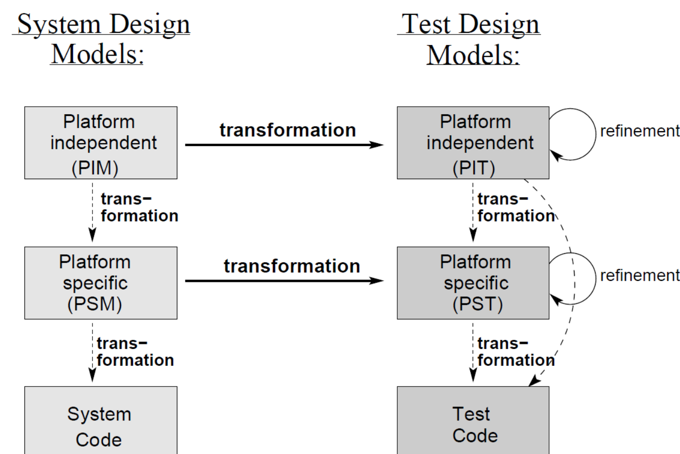

FIGURE 2.1: The relation between system design models and test design models [10]

2.3.1 Domain-specific modeling

In Puolitaival and Kanstrén [11] an experiment is explained in which domain-specific modeling (DSM) has been used as a basis for MBT. The authors use DSM to denote DSL development using the MDE approach. First a metamodel and modeling language are defined that capture the domain concepts. The metamodel and language are then used to create models, which are transformed to other forms (e.g., application code) in a later phase.

The test approach described in Puolitaival and Kanstrén [11] starts by defining a modeling language used to create models of the SUT. Based on these models, test models can be generated using transformations. These test models can then be used as input for the test generators of different MBT tools. By executing these test generators, test cases are generated, which can be run by the test environment to test the SUT. This process is displayed in Figure 2.2.

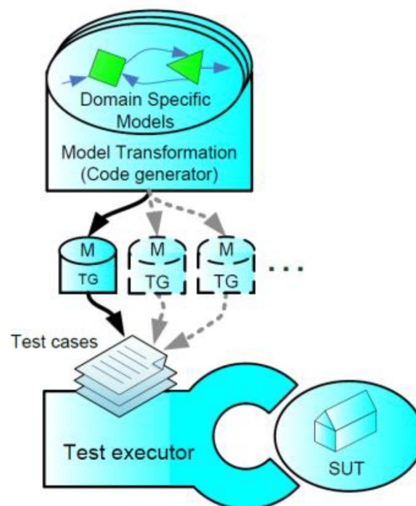

FIGURE 2.2: Model-based testing with domain-specific modeling [11]

The benefits of using testing models are the reusability of these models (different mappings results in different tests), lower maintenance since only the domain-specific models have to be maintained, and independence of testing environment. A possible drawback is that the development of the DSM language and test generators result in extra costs.

2.3.2 FSM testing

In Lee and Yannakakis [12] the principles and methods of Finite State Machine (FSM) testing are discussed. There are several ways in which FSMs can be used for testing, but only conformance testing is discussed here. Conformance testing, also called fault detection or machine verification, uses two FSMs. The first FSM is specification machine A of which we have complete information on the behavior of the SUT, such as state transitions and output functions. The second FSM is implementation machine B, which acts as a black-box of the system, so only input and output can be observed. The goal of the test is to determine whether machine B is a correct implementation of machine A by providing machine B with inputs and observing the returned outputs.

For this to be possible, four assumptions concerning machine A and B are necessary:

1. Specification machine A is strongly connected, i.e., every state is reachable from every other state.
2. Machine A is reduced, i.e., A is modeled using a minimal number of states.
3. Implementation machine B does not change during the experiment and has the same input alphabet as A.
4. Machine B has no more states than A.

For the test to be decisive, the test sequence must be a checking sequence. Let A be a specification FSM with n states and initial state s_0. A checking sequence for A is an input sequence x that distinguishes A from all other FSM with n states. If the sequence is not a checking sequence, the difference between the FSMs could be missed. This difference could contain an error. However, the checking sequence could take a large amount of time thereby rendering this approach unfeasible. By limiting the execution time, this approach could become applicable, yet this also decreases the reliability of the testing procedure.

2.3.3 UML testing

In Dai [10] the UML 2.0 Testing Profile (U2TP) is discussed, which provides a number of concepts that can be used to develop test specifications and test models for black-box testing. The U2TP concepts are divided into four groups: test architecture, test behavior, test data and time. An overview of the concepts per group is given in Table 2.1.

Table 2.1: Overview of the UML 2.0 Testing Profile concepts, divided into four groups [10]

Test architecture concepts:	
System under test	The system to be tested using the models
Test components	Objects that interact with the SUT or required for the test
Test context	Used to categorize test cases
Test configuration	Defines the relation between test components and the SUT
Test control	The order in which the test cases should be executed
Arbiter	Used to define how the overall verdict should be interpreted
Scheduler	Schedules the test cases by creating objects and starting/stopping the cases
Test behavior concepts:	
Test objectives	Goals of a test
Test case	The specifications of expected test behavior (i.e., how the test components should interact with the SUT to achieve the test objective)
Defaults	The specifications of unexpected behavior
Validation action	Action performed by a test component to be interpreted by the arbiter
Verdict	Result of a finished test
Test data concepts:	
Wildcards	Used to deal with unexpected events
Data pools	Contain concrete test data used in test context
Data selectors	Interact with data pools
Coding rules	Used to define encoding and decoding of test data
Time concepts:	
Timers	Used to change and manage test behavior and to make sure tests terminate
Time zones	Used to group test components, allowing the comparison of time events within the same time zone

The test design model is developed by extending the system design model defined in UML 2.0 using U2TP concepts. The whole process is described in Dai [10], but only a summary is given here.

After importing the classes and interfaces, the test architecture and test behavior specifications should be defined. For both the test architecture and test behavior there are two sorts of requirements (called issues), namely mandatory and optional. The mandatory issues (test components) have to be resolved for a correct test design model, while the optional issues are not always required (e.g., timers).

After the specifications are defined, the metamodel-based transformations can be developed using, for example, QVT. An overview of the transformation process is given in Figure 2.3. The transformation transforms the UML model to the U2TP model.

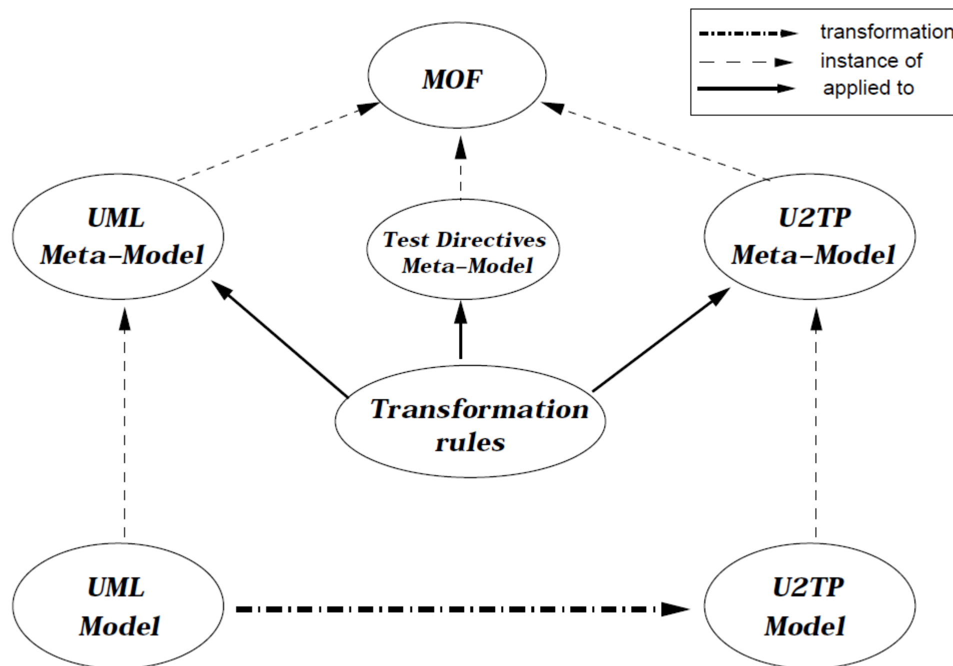

Figure 2.3: Overview of the transformation process using three metamodels [10]

As the transformation does not allow the tester to group and remove UML elements, for example classes, objects, instances, etc., required for creating test components and SUT, the authors introduced mechanisms called 'test directives', which are defined in the Test Directives metamodel. All three metamodels used during the transformation process are described using the MOF.

2.4 Behavior-driven development

In Solis and Wang [13], behavior-driven development (BDD) and its characteristics are discussed. BDD is often seen as the future of Test-Driven Development and Acceptance Test-Driven Development. Using specifications, the behavior of the system is modeled, which can, for example, be used to automatically generate test cases. Since the specifications can be written down in a natural language, a non-programming language, BDD enables domain experts and developers to understand each other and the test specifications.

The following six characteristics of BDD are discussed in Solis and Wang [13]:

1. **Ubiquitous language**

   The language used to define the testing procedures should be specifically designed for a specific domain, to among others increase productivity and simplify the

learning process. It is crucial to incorporate the knowledge of the domain experts and developers into the language, since both stakeholders will use the language to communicate.

2. **Iterative decomposition process**

   The process of using BDD should be iterative. In the early stages, expected system behavior should be collected, which can then be converted to a set of features (similar to requirements). Based on the features, user stories are created that describe the interaction between the user and system as well as the benefit the user gains if the system provides the feature. These user stories can then be divided into scenarios that describe different contexts and outcomes of the stories.

3. **Plain text description with user story and scenario templates**

   These features, user stories and scenarios must be written down in a specific format using a ubiquitous language to facilitate inspection and transformation to test cases.

4. **Automated acceptance testing with mapping rules**

   Mapping the scenarios to test cases creates an appropriate test set for the functionality of the system under test. A key requirement here is that the scenarios should be run automatically, thereby reducing time waste.

5. **Readable behavior-oriented specification code**

   Since the scenarios are written in plain text and mapped to test cases, the code is readable and can act as documentation together with the specifications. This means that the mappings improve readability.

6. **Behavior driven at different phases**

   BDD should not only be used during the implementation phase (e.g., testing), but also during the planning and analysis phase (e.g., setup feature list). This improves the common understanding of the system for the domain experts and the developers.

The researchers also investigated several BDD toolkits and checked which characteristics were supported by each toolkit. The results are shown in Table 2.2.

Table 2.2 shows that, although none of the toolkits provide all the six described characteristics, the xBehave Family and SpecFlow toolkit provide the most functionality. The authors state that, for future work purposes, a toolkit could be developed or extended to enable the application of BDD techniques during the planning and analysis phase.

Table 2.2: BDD toolkits and the supported characteristics [13]

Support of the BDD Characteristics		xBehave Family		xSpec Family		StoryQ	Cucumber	SpecFlow
		JBehave	NBehave	RSpec	MSpec			
Ubiquitous language definition		×	×	×	×	×	×	×
Iterative decomposition process		×	×	×	×	×	×	×
Editing plain text based on	User story template	√	√	×	×	×	√	√
	Scenario template	√	√	×	×	×	√	√
Automated acceptance testing with mapping rules		√	√	×	×	×	√	√
Readable behaviour oriented specification code		√	√	√	√	√	×	√
Behaviour driven at different phases	Planning	×	×	×	×	×	×	×
	Analysis	√	√	×	×	×	√	√
	Implementation	√	√	√	√	√	x	√

Note: √ – The toolkit supports the BDD characteristic, × – The toolkit does not support the BDD characteristic

Another limitation of the researched toolkits is that they only provide mapping rules for transforming user stories and scenarios to code. The toolkits could be extended with support to map to namespaces and packages resulting in the option to group stories based on their test feature.

2.5 Domain-specific testing language

In Santiago et al. [14] an approach is described that leverages MDE, by describing several domain and platform models, to improve the specification, execution, and debugging of functional tests for cloud applications with several domain-specific elements. As a result, a test case specification language is defined that can be used to develop automated tests for a particular application domain. An overview of the domain-specific testing language proposed in Santiago et al. [14] is given in Figure 2.4.

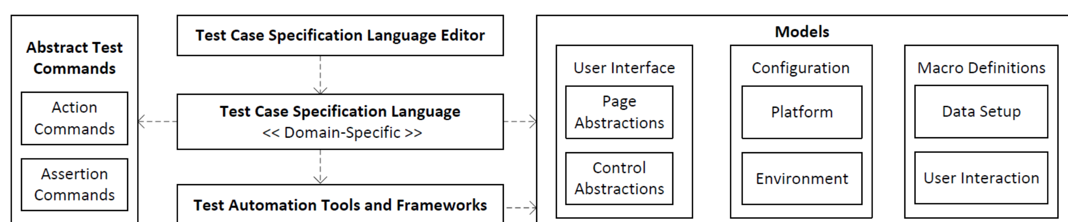

Figure 2.4: Structure of the domain-specific testing language [14]

The key elements of the language are the action commands (used to apply inputs to the system under test) and the assertion commands (used to check outputs of the system under test).

The authors created models (abstractions) of the elements of the system under test, such as the user interface. By abstracting away from the domain, the tests can reference the models and are independent of domain changes, thereby improving maintenance.

The authors also provide users with the option to define macros, which are patterns that specify how a sequence of inputs is mapped to a replacement input sequence. This means the users can reduce repetitive tasks and error-prone activities, thereby improving productivity and usability.

An example of a test case developed using the domain-specific testing language is given in Figure 2.5. The test case consists of four parts: summary, declarations, setup and tests. The summary specifies the purpose, authors and configuration. The used applications and data are specified in the declarations part. The setup part enumerates the preconditions using behavioral-driven development, while the tests part defines the tests to be executed.

```
1  Summary
2     Purpose   Validate Run Payroll Feature
3     Authors   Dionny Santiago, Adam Cando
4     Configs   .NET, Payroll-14
5
6  Declarations
7     UltiPro UltiPro
8
9  Setup
10    Given I Setup a Payroll (...)
11    And launch UltiPro
12    And login as a Payroll Administrator
13      By setting the UserNameTextBox to "admin",
14                     PasswordTextBox to "ulti"
15      And clicking on the LoginButton
16
17 Tests
18    Scenario: Payroll Should Run to Completion
19    Given I Navigate to the Payroll Overview
20    When I click the StartPayrollButton
21    Then the ProcessStatusLabel is "Completed."
22    And the CreateBatchesButton is enabled
```

FIGURE 2.5: Example of a test case [14]

According to the authors, their main factor for success was to have a robust, highly extensible and configurable underlying testing framework. This is in line with Kent [5], who states that highly extensible and configurable tools are preferred. Since the framework has a high usability, even non-technical users can develop test cases using the domain-specific testing language of Santiago et al. [14].

Using this domain-specific testing language, the authors of King et al. [15] developed a DSL-based testing toolset called Legend. It provides the user with story linking, so

that the business requirements and user stories are linked to a specific test suite. It also generates a test template for a new test, thereby improving the usability of testing. The toolset supports high-level test steps so the user can define commands, thereby reducing errors and code duplication. The test suites can be stored using several kinds of third-party software, resulting in a central inventory of tests. Lastly, Legend provides a centralized test tagging system, so that tests can be filtered efficiently.

2.6 Automated web testing

Web application usage has increased in the last decades, with the Internet user numbers growing over 700% over the last 15 years [16]. Web applications have a number of benefits, for example, continuous and ubiquitous availability. As users rely upon the idea that web applications should be online all the time, downtime should be minimized to improve user experience and loyalty. To achieve a high quality and reliable web application, the application should be tested thoroughly to ensure correct functionality. By testing whether the code is consistent with the requirements specification of the web application for acceptance, the user experience should be improved.

Testing web applications is however more complicated than testing traditional applications since these applications are heterogeneous, distributed and concurrent. Web applications can be tested using black-box (functional) testing tools, yet this would not ensure the reliability of the application. White-box (structural) testing should also be applied [17]. To save time and money, web applications should be tested automatically e.g., by using an automated test tool or framework.

Selenium

One tool for automated testing of web applications is Selenium, which is open source and supports several programming languages. Selenium is “a suite of tools specifically for automating web browsers”[1], but we only discuss it in regards to automating web applications for testing purposes. It was originally developed by ThoughtWorks and now has an active community of developers and users. Selenium runs in several browsers and operating systems, while being compatible with a number of programming languages and testing frameworks. By using Selenium, browser incompatibility can be determined easily by running the same tests on different browsers [18].

In Bruns et al. [19] Selenium version 1.0 is discussed by first examining Selenium Core. The Core tool allows the user to interact with the web application by running a JavaScript application in a host browser used to manipulate the web application under

[1] http://www.seleniumhq.org/about/

test. This is done by sending commands in Selenese (Selenium's DSL) to change and evaluate elements. There is also the Selenium RC tool which allows programmers to use the supported programming languages (Java, C#, Ruby, etc.) instead of Selenese to interact with the web application.

The Selenium Project [20] also describes Selenium IDE and Selenium Grid. Selenium IDE is a prototyping tool used for building test scripts by allowing users to record their actions and export recorded actions as scripts. Selenium Grid allows the user to group Selenium RC tests into suites and run them in multiple environments. It also speeds up the process by allowing the tests to run in parallel. In the latest update, Selenium WebDriver (previously Core) and Selenium RC are combined into Selenium 2.

In Holmes and Kellogg [21] advice is given on how to use Selenium. The first advice is to keep the test self-contained, thereby improving flexibility and maintainability. Although Selenium supports the definition of test suites and grouping of tests, the authors tried to keep the tests as independent and self-contained as possible, so that changes and refactoring could be applied rapidly. The second advice is to exploit the open source property of Selenium by writing extensions when required. As Selenium tests are easy to write, users can apply the Test Driven Development approach of writing the tests before developing the application. The last advice is to use IDs for the identification of website elements instead of XPath expressions since this improves the speed of locating these elements.

2.7 Testing levels

Software systems and artifacts can be tested on different levels. In Bourque and Fairley [22] three testing levels are defined:

- **Unit testing:** test the functionality of software components that are testable in isolation.

  Depending on the DSL and artifact generator, there can be one or more generated model artifacts (e.g., Java classes). Each artifact could be seen as a unit and tested individually using unit testing. If an artifact depends on another artifact, a mock object can be used to emulate the other artifact thereby ensuring that artifacts are tested individually in their intended working environment.

- **Integration testing:** test the interaction of the individual elements. The perspective switches from low-level to integration level.

  The different generated artifacts might have to work together and this integration could be tested using integration testing. One way of testing the integration is by testing a software system that uses multiply artifacts in combination, for example, a website displaying several model elements. To exclude errors from the individual artifacts while performing integration testing, the individual artifacts should already be unit tested.

- **System testing:** test the behavior of the entire system. Usually used for testing non-functional requirement, e.g., performance and reliability.

  Using the tested integrated software components, the whole system can be tested using system testing. This does not only test the components but also the hardware that the software runs on.

2.8 Application to domain-specific languages

In this section we discuss some available testing techniques based on their application to test models and code developed using domain-specific languages.

Black-box testing is based on the idea that the code is unknown, while focusing on the specifications. An application to domain-specific languages is to use the models developed using the DSL as specifications to test the correctness of the code generated from these models. This technique can be combined with the UML 2.0 Testing Profile discussed in Dai [10], which specifies concepts for the development of test specifications. White-box testing techniques on the other hand use logic to analyze the source code and can, for example, be applied to test code generated from the model.

When the DSL is developed using metamodeling, a metamodel of the domain and its concepts is already available. The domain-specific modeling testing technique of Puolitaival and Kanstrén [11] can therefore be applied to this metamodel to develop a testing language used to model the system under test. Based on the system models, test models can be generated using transformations, which are used as input for test generators of different MBT tools. By applying this technique, the models developed using the DSL can be tested.

One application of the FSM testing technique to DSLs is the development of a specification FSM of the model (using the DSL) and an implementation FSM of the code generated from this model. The specification FSM can then be used for correctness

testing of the implementation FSM, thereby testing if the code is correctly generated from the model and is consistent with the specifications of the model.

For techniques using the MDE approach, models have to be created (e.g., FSMs), which results in upfront effort but also better maintainability and reusability.

Behavior-driven development gives the domain expert the option to write test for their programs, which means the domain knowledge can be directly applied in the testing process. Several toolkits are already available that support this testing technique. One way DSLs can apply BDD techniques is by allowing the domain expert to develop scenarios (tests) that can be executed to test the domain models.

Web acceptance testing provides the end user with the option to automatically test web applications, which saves time and money while also improving the maintainability of tests. A number of tools are already available that support this technique. This technique can, for example, be applied to DSLs by testing a website that uses DSL artifacts.

By developing a domain-specific testing language using the concepts of the DSL, this language can be used to produce, for example, behavior-driven or automated web test cases. These test cases can then be used to test the code generated from the models developed using the DSL.

The testing techniques can be applied in different ways to test a domain-specific language. A reoccurring application is to test the code generated using the models developed with the DSL for correctness according to these models. This can be done either in isolation (unit testing) or by testing the system that uses the generated code (integration testing). By applying several testing techniques, higher test coverage can be achieved.

Chapter 3

Approach

This chapter introduces the approach taken in this research project and gives an overview of the developed framework. Section 3.1 discusses the general goals of our testing framework. Section 3.2 explains the transformation chain used to generate the tests from domain-specific models. Section 3.3 analyses common programming language elements used by the framework's generic metamodel in order to support multiple languages. Section 3.4 explains how domain-specific elements can be incorporated in the generic metamodel. Section 3.5 gives an overview of the developed framework.

3.1 General development goals

Currently artifacts generated using domain models are often tested using manually developed tests. These tests are developed by a tester that analyzes the use cases and the artifacts or the software specification. Manually developing these tests is a time-consuming activity that requires the tester to have domain and test knowledge. These tests could be, for example, unit tests. If the tester wants to develop tests of a different type, for instance, automated web tests, these new tests again have to be written manually.

To the best of our knowledge, there is currently no testing framework available that automatically generates tests for generated artifacts using domain-specific models. Our goal is to develop this framework, while supporting several languages and different types of tests. An example: based on the domain-specific model, several Java classes (the artifacts) are generated. Our framework then generates tests using the same domain-specific model to verify the correctness of these Java classes. Several types of tests can be generated using the same domain-specific model such as behavior-driven development tests or automated web tests.

To reach acceptable test coverage, the framework applies several techniques described in Chapter 2. Since the internal structure and source code of the models is known, white-box testing techniques are applied. The tests generated by the framework achieve branch/condition coverage, because this is the strongest technique as described in Section 2.2. To successfully achieve branch/condition coverage, test values are generated based on the model data. The framework also applies model-based testing techniques by using a developed generic metamodel, the *common elements metamodel*, to increase maintainability and flexibility. Next to that, using the metamodel separates the test generation functionality from the DSL-specific properties. This metamodel elements are described in more detail in Section 4.1.

The common elements metamodel was developed as an Ecore metamodel, as well as an implementation consisting of a set of Java classes. Java was used since it is platform-independent and prior experience was available for this language. One drawback of using the common elements metamodel in the framework is that the supported types of domain-specific models and languages is limited. Since the framework uses the branch/-condition theory to generate tests, it depends on expressions thereby requiring the domain-specific language to be expression based. Next to that, the majority of the domain elements have to be transformable to the generic metamodel elements so their data (e.g., values) can be used for the generation of test values and test cases. Another requirement is that the domain-specific language should generate some artifacts that can be tested using the information extracted from the model.

We have aimed to support several language types such as data structure and process definition languages by focusing on common programming language elements, for instance variables, expressions, arrays and basic primitive types. The properties of these common elements can however differ depending on the domain, and we focused on data storage models for the common elements metamodel, since this best matched the models of the case study.

For the framework to be applied to domain-specific models, domain-specific models should be mapped to instances of the common elements metamodel. Based on the assumption that the domain-specific language contains both common language elements and domain-specific elements, the mapping should support the transformation of both types of elements. Using this mapping, the domain-specific models can be transformed to common elements models to be used by the framework to generate test cases. By keeping the test cases generic, several types of tests, for example, behavior-driven development tests, can be generated from the same test case.

3.2 Transformation chain

Using the common elements metamodel for the test case generation has one main benefit and one main drawback:

Benefit: the domain knowledge in the domain models is defined using a domain-specific language. By transforming this domain model into a common elements model, the framework components can focus solely on test case generation independent of the language, e.g., its constructs and syntax. This also ensures the test case generator components are reusable and easily modifiable.

Drawback: the drawback of using a generic representation is that every domain model has to be transformed to a common elements model and the result of the generated generic test cases have to be transformed to executable tests for the domain model artifacts. Although this allows the generation of different types of tests, it also requires the user to develop the transformer and test generator. An example of this transformation is the generation of JBehave tests using the generic test cases, so these tests can be executed to test the system under test.

Since we rely on a generic metamodel, the usage of the test framework consists of three phases: generalization, generation and specification. An overview of these three phases is given in Figure 3.1. The framework components are combined and abstracted into six components to provide a better overview of the transformation chain.

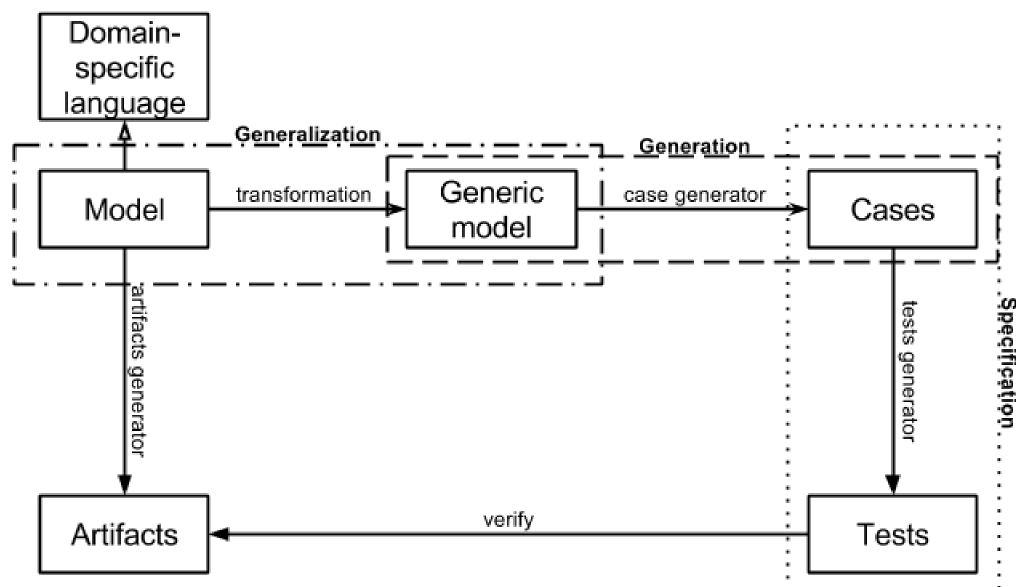

FIGURE 3.1: Overview of the transformation chain

Generalization

In the generalization phase, the domain-specific model is transformed to an instance of the generic metamodel. This is done by recursively traversing the model tree and transforming each element in the tree. The main requirement of this process is to ensure all information, like operator precedence, is transformed correctly.

Generation

In the generation phase the framework uses the generic model to generate a set of generic test cases. Since the test case generator components are based on information in the generic model, these are independent of the DSL metamodel and therefore reusable.

Specification

In the specification phase, the generic test cases are transformed to executable tests depending on the choice of test generator. For example, if a JBehave test framework is used, the generic test cases are transformed to JBehave tests. An important element here is that, the transformation from generic test cases to tests should result in tests that refer to the same elements as the domain model.

To summarize, the transformation from domain-specific model to generic model and from generic test cases to executable tests should not alter or lose any model information.

3.3 Common programming elements

As the testing framework should be usable for a multitude of domain-specific languages, our framework focuses on supporting elements that are common in programming languages such as expressions and variables. Domain-specific elements, such as specialized functions, are also supported, which is achieved using mock functions. Mapping domain-specific elements is explained in more detail in Section 4.2.2.

Abelson and Sussman [23] define a number of common programming elements:

- **Expressions:** expressions are used in a large number of programming languages and consist of operands, for instance numbers, and operators, such as + and *.
- **Variables:** variables are used to save values and improve readability, which is useful in many languages.
- **Evaluating combinations:** most languages can combine different expressions to build complex structures, thereby improving expressiveness.

- **Compound procedures:** a common technique for languages to improve modularity and reusability is the ability to define procedures (methods).
- **Conditional expressions and predicates:** to branch during execution, most languages support conditional expressions and predicates (if-else, switch, etc.).

Even though variables and compound procedures are not essential for a language as they can be replaced with duplicate code, they do improve attributes like reusability and maintainability. Variables, expression combinations and compound procedures are abstract concepts, but for (conditional) expressions, consisting of operands and operators, we had to determine a set of common constructs to support.

3.3.1 Common language specification

The set of common (conditional) expressions must be rich enough to support several domain-specific languages. According to ECMA International [24], the Common Type System "establishes a framework that enables cross-language integration, type safety, and high performance code execution". There is also the Common Language Specification (CLS) which is defined as "a set of rules intended to promote language interoperability". It specifies a subset of the CTS type system and a number of usage conventions. According to ECMA International [24], "frameworks will be most widely used if their publicly exposed aspects (classes, interfaces, methods, fields, etc.) use only types that are part of the CLS and adhere to the CLS conventions". The testing framework supports a subset of the CLS since the framework focuses on domain-specific languages used for data storage.

3.3.2 Common operands

The defined CLS data types are given in Table 3.1. An additional column is added that specifies whether the test framework supports the data type.

In the generic metamodel, numeric values are represented using a Java double (double-precision 64-bit IEEE 754 floating point[1]). When a domain-specific language defines several numeric types, e.g., float, integer or double, these types are mapped to a Java double. An double was used since it is a primitive Java type meaning it is a predefined Java type with a specific keyword. Primitive types are not dependent on third-party software, e.g., libraries and they are often supported in other software (for example

[1]https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

model transformation languages like ATL[2]), thereby increasing the reusability of the metamodel elements. Characters are represented as Strings in the framework for the same reason. Since the supported data types are setup in a modular way, new types can be added without much effort or affecting existing types.

Table 3.1: Common Language Specification data types

Data Type	Description	Support
bool	True/false value	Supported
char	Unicode 16-bit char.	As String
object	Object or boxed value type	Unsupported
string	Unicode string	Supported
float32	IEC 60559:1989 32-bit float	As double
float64	IEC 60559:1989 64-bit float	As double
int16	Signed 16-bit integer	As double
int32	Signed 32-bit integer	As double
int64	Signed 64-bit integer	As double
native int	Signed integer, native size	As double
unsigned int8	Unsigned 8-bit integer	Unsupported

3.3.3 Common operators

The CLS unary operators are given in Table 3.2, while the CLS binary operators are given in Table 3.3. Again an additional column is added that specifies whether the test framework supports the operator. When an operator has the 'Supported indirectly' tag, its function can be achieved by a workaround. The framework focuses on higher level operations due to the scope of supported languages, thereby excluding byte operations. However, these elements can also be added without much effort due to the modular setup of the framework.

Table 3.2: Common Language Specification unary operators

Name	ISO C++ operator symbol	Support
op_Decrement	Similar to - -	Supported indirectly
op_Increment	Similar to++	Supported indirectly
op_UnaryNegation	- (unary)	Supported
op_UnaryPlus	+ (unary)	Supported
op_LogicalNot	!	Supported
op_True	Not defined	Unsupported
op_False	Not defined	Unsupported
op_AddressOf	& (unary)	Unsupported
op_OnesComplement	~	Unsupported
op_PointerDereference	* (unary)	Unsupported

[2]https://eclipse.org/atl/

TABLE 3.3: Common Language Specification binary operators

<table>
<tr><th>Name</th><th>ISO C++ operator symbol</th><th>Support</th></tr>
<tr><td>op_Addition</td><td>+ (binary)</td><td>Supported</td></tr>
<tr><td>op_Subtraction</td><td>- (binary)</td><td>Supported</td></tr>
<tr><td>op_Multiply</td><td>* (binary)</td><td>Supported</td></tr>
<tr><td>op_Division</td><td>/</td><td>Supported</td></tr>
<tr><td>op_Modulus</td><td>%</td><td>Supported</td></tr>
<tr><td>op_ExclusiveOr</td><td>^</td><td>Unsupported</td></tr>
<tr><td>op_BitwiseAnd</td><td>& (binary)</td><td>Unsupported</td></tr>
<tr><td>op_BitwiseOr</td><td>|</td><td>Unsupported</td></tr>
<tr><td>op_LogicalAnd</td><td>&&</td><td>Supported</td></tr>
<tr><td>op_LogicalOr</td><td>||</td><td>Supported</td></tr>
<tr><td>op_Assign</td><td>=</td><td>Supported</td></tr>
<tr><td>op_LeftShift</td><td>«</td><td>Unsupported</td></tr>
<tr><td>op_RightShift</td><td>»</td><td>Unsupported</td></tr>
<tr><td>op_SignedRightShift</td><td>Not defined</td><td>Unsupported</td></tr>
<tr><td>op_UnsignedRightShift</td><td>Not defined</td><td>Unsupported</td></tr>
<tr><td>op_Equality</td><td>==</td><td>Supported</td></tr>
<tr><td>op_GreaterThan</td><td>></td><td>Supported</td></tr>
<tr><td>op_LessThan</td><td><</td><td>Supported</td></tr>
<tr><td>op_Inequality</td><td>!=</td><td>Supported</td></tr>
<tr><td>op_GreaterThanOrEqual</td><td>>=</td><td>Supported</td></tr>
<tr><td>op_LessThanOrEqual</td><td><=</td><td>Supported</td></tr>
<tr><td>op_UnsignedRightShiftAssignment</td><td>Not defined</td><td>Unsupported</td></tr>
<tr><td>op_MemberSelection</td><td>-></td><td>Unsupported</td></tr>
<tr><td>op_RightShiftAssignment</td><td>»=</td><td>Unsupported</td></tr>
<tr><td>op_MultiplicationAssignment</td><td>*=</td><td>Supported indirectly</td></tr>
<tr><td>op_PointerToMemberSelection</td><td>->*</td><td>Unsupported</td></tr>
<tr><td>op_SubtractionAssignment</td><td>-=</td><td>Supported indirectly</td></tr>
<tr><td>op_ExclusiveOrAssignment</td><td>^=</td><td>Unsupported</td></tr>
<tr><td>op_LeftShiftAssignment</td><td>«=</td><td>Unsupported</td></tr>
<tr><td>op_ModulusAssignment</td><td>%=</td><td>Unsupported</td></tr>
<tr><td>op_AdditionAssignment</td><td>+=</td><td>Supported indirectly</td></tr>
<tr><td>op_BitwiseAndAssignment</td><td>&=</td><td>Unsupported</td></tr>
<tr><td>op_BitwiseOrAssignment</td><td>|=</td><td>Unsupported</td></tr>
<tr><td>op_Comma</td><td>,</td><td>Unsupported</td></tr>
<tr><td>op_DivisionAssignment</td><td>/=</td><td>Supported indirectly</td></tr>
</table>

3.3.4 Common conditionals

There are several conditional expressions and predicates available. Some common constructs:

- If–then(–else)
- Else-if
- Case statements
- Switch statements

The framework currently only supports the *if–then(–else)* construct, since it achieves the branching functionality. Although the other constructs do not provide new functionality, these can be added to the framework without major efforts if the user requires them.

3.4 Domain-specific elements

Since DSLs can have domain-specific elements and users would want to test these elements, the testing framework should support some way of mapping the domain-specific elements to generic metamodel elements. The problem is however that since the elements are domain-specific, their nature and properties can be diverse making the mapping to the supported generic metamodel element (discussed in Section 4.1) problematic and complex. To provide the user with the option of testing domain-specific elements, while keeping the framework generic, the GenericFunction element was defined.

This element is a collection with entries consisting of inputs and their corresponding outputs, thereby mocking/simulating a common function/method structure supporting multiple inputs and one output. This structure was chosen since it provides a frequently used functionality that can serve a broad range of purposes. The inputs act as keys for the outputs and are defined as lists of the generic metamodel base type, called GenericAbstractElement. Outputs are defined as singular elements, also of type GenericAbstractElement. The user can define their GenericFunction element in the transformer component of the framework, which maps the domain-specific elements to the generic elements. If, for example, the domain model contains a complex and CPU intensive formula that takes two numbers and returns one number, this formula can be mocked using a GenericFunction element with a list entry of two inputs and one output. Using the GenericFunction element also simplifies the testing process since the formula is seen as a black-box.

When the domain-specific elements cannot be mapped to any of the supported generic elements, the user can extend the generic metamodel by defining and implementing their own elements, as long as they inherit from GenericAbstractElement. This does however require the user to also extend the other framework components to cope with the new type of element.

3.5 Solution overview

A graphical overview of the developed framework and its components is given in Figure 3.2. External (user or third-party) components are filled gray, while the internal framework components are not filled. Components with a gray border are input and output components, while components with black border are framework functionality components. Next, each component of the framework is discussed shortly. The framework components are explained in more detail in the remainder of this report.

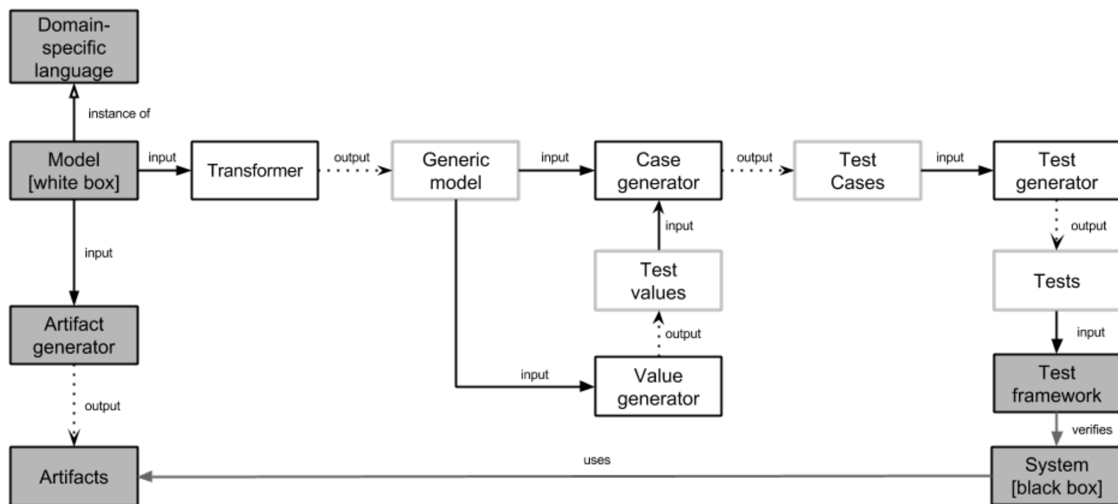

FIGURE 3.2: Overview of the testing framework

- **Domain-specific language**

  The domain-specific language used to develop the models.

- **Model**

  The model is an instance of the language. It describes the domain-specific elements used to generate the domain artifacts and tests. These elements are mapped to the generic model elements.

- **Artifact generator**

  The artifact generator is a user-defined component that transforms the model elements to artifacts. One possible function of an artifact generator is the generation of Java classes based on the model elements.

- **Artifacts**

  The artifacts generated by the artifact generator can be text files (e.g., Java classes), images, etc., depending on the chosen generator.

- **System**

  The system component describes a system under test. This system uses the generated artifacts, thereby indirectly testing the model. This system can be, for example, an engine or website. Based on the system, the testing level can be chosen. An engine could test the domain artifacts on a unit level, while a website could test the artifacts on an integration level.

- **Transformer**

  The transformer component transforms the domain model to a common elements model. This is done by transforming the domain model elements to generic model elements.

  Our framework focuses on supporting the common language elements, described in Section 3.3. Domain-specific elements similar to these elements can easily be mapped and transformed to the generic elements. A transformation language was developed for defining a mapping from domain elements to generic elements for a subset of the supported common elements. The result of this mapping is a simple transformer class, which can then be extended for the remaining elements.

  For domain-specific elements that are not captured by the generic metamodel, the user can extend the generated transformer class by defining their own mapping to the GenericFunction element. Another option is to add elements to the generic metamodel and update the transformer class to use this new element.

- **Generic model**

  The generic common elements model is used as the basis for the generation of test cases. By using this generic model, the value generator and case generator could be developed independent of the used language. This improves the flexibility and maintainability of the framework.

- **Value generator**

  The value generator is used to generate test values for the unassigned variables (variables without a value) in the generic model, thereby creating the different

test cases. This component traverses the generic model elements and calculates the possible values for the unassigned variables to create test cases that accomplish branch/condition coverage.

- **Test values**

  Based on the elements in the model, the value generator outputs the test values for the unassigned variables in the model. These values are then provided as input for the case generator.

- **Case generator**

  The case generator takes the generic model and map of possible values for the unassigned variables and combines these two items into a list of generic test cases. A generic test case represents a unique combination of values for the unassigned variables.

- **Test cases**

  A test case consists of two list describing the preconditions and postconditions. By using a generic representation for the conditions, different kinds of test can be generated using the same test case. A precondition consists of a variable and a value, thereby describing the assignment of the test value to the variable. A postcondition describes a variable or function and a value, thereby describing the assertion of a variable/function and its value/result.

- **Test generator**

  The test generator uses the generic test cases to develop tests. Different test generators can be used to generate different kinds of tests. Examples of tests that can be generated are: unit tests, behavior-driven development tests and web acceptance tests.

- **Tests**

  The output of the test generator is the set of tests that can be executed by the testing framework.

- **Test framework**

  The testing framework executes the generated tests to test the system under test, which uses the artifacts generated by the artifact generator. The choice of testing framework depends on the chosen test generator. Examples of frameworks are JBehave and Selenium.

Chapter 4

Common elements metamodel

This chapter discusses the common elements metamodel, the mapping process and the framework options. Section 4.1 discusses the generic metamodel developed as language-independent component used for test case generation. Section 4.2 explains the mapping from domain-specific model to generic model. Section 4.3 identifies the options provided by the framework, for example, to generate test cases and JBehave stories.

4.1 Model definition

Figure 4.1 shows an overview of the elements and the relations in the common elements metamodel.

To represent domain-specific model elements independent of properties like syntax or operator precedence, these elements are transformed to common element instances so they can be further processed. Each common element is shortly discussed below, using the developed Java class that serves as the implementation of the metamodel. Every class has a *toString()* method to print its value, a *clone()* method to duplicate itself, a *hashCode()* method and an *equals()* method that are self-explanatory and will therefore not be discussed here.

- **GenericAbstractElement:** an interface used as a generalization for the types of elements.
- **GenericNumber:** used to represent numeric values. This class contains a private double that is set at initialization. This value can be retrieved and altered.
- **GenericString:** used to represent text values. This class contains a private String that is set at initialization. This value can be retrieved and altered.

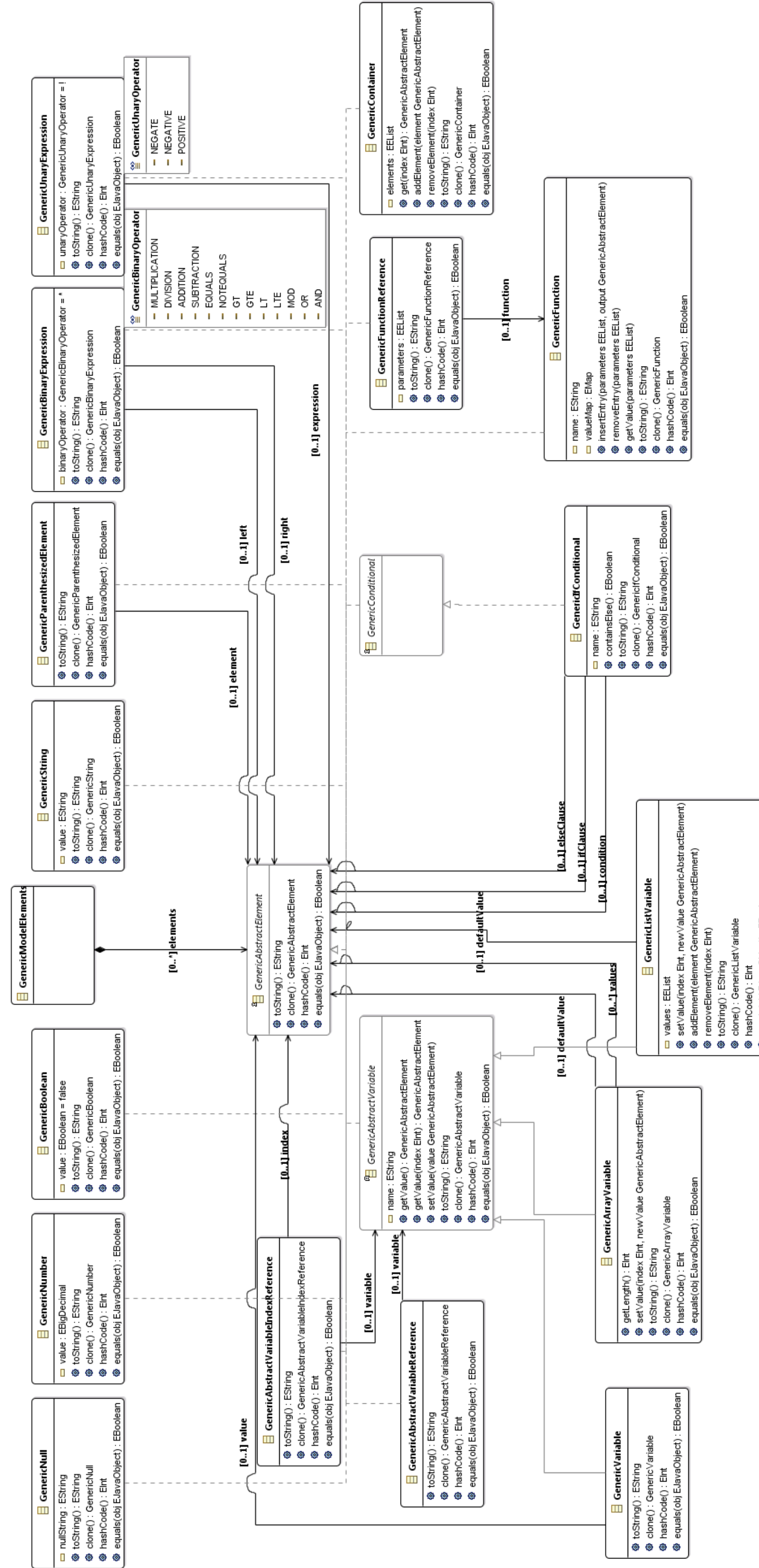

FIGURE 4.1: Overview of the common element metamodel

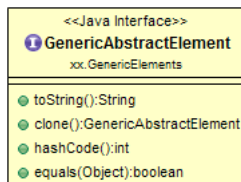

FIGURE 4.2: GenericAbstractElement interface

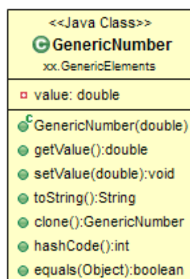

FIGURE 4.3: GenericNumber class

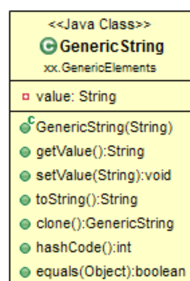

FIGURE 4.4: GenericString class

- **GenericBoolean:** used to represent Boolean values. This class contains a private Boolean that is set at initialization. This value can be retrieved and altered.

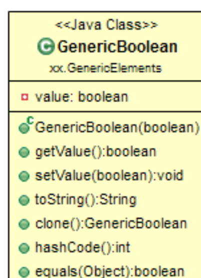

FIGURE 4.5: GenericBoolean class

- **GenericNull:** used to represent null values. This class contains only a String attribute to define the textual representation of the GenericNull.

- **GenericParenthesizedElement:** used to represent parenthesized elements. The class contains a private GenericAbstractElement that is set at initialization and can only be retrieved.

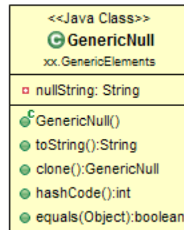

Figure 4.6: GenericNull class

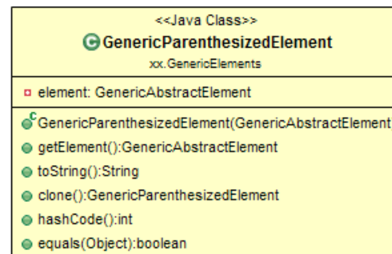

Figure 4.7: GenericParenthesizedElement class

- **GenericAbstractVariable:** abstraction used to represent different types of variables. The class contains a private String to represent the name of the variable. Next to that, several abstract methods are defined to get and set the variable value, to be implemented using the specific requirements of the different types of variables.

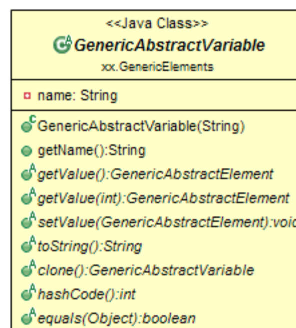

Figure 4.8: GenericAbstractVariable class

- **GenericVariable:** subtype of GenericAbstractVariable used to represent variables with a single value. The class contains a private GenericAbstractElement to represent its value. This value is set at initialization and can be retrieved. When no value is given, the variable is initialized using a GenericNull.

- **GenericArrayVariable:** subtype of GenericAbstractVariable used to represent a variable consisting of an array of values. This class contains a private GenericAbstractElement array to hold the values of the array and a GenericAbstractElement to represent the default value. The array, the default value and the length of the array are set at initialization and can be retrieved. Elements can also be retrieved

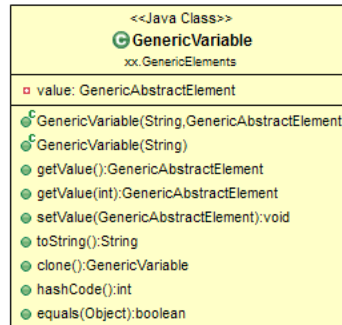

FIGURE 4.9: GenericVariable class

and set by index. A new default value can be set, which only alters the values in the array that equal the current default value. If no default value is given at initialization, a GenericNull is used.

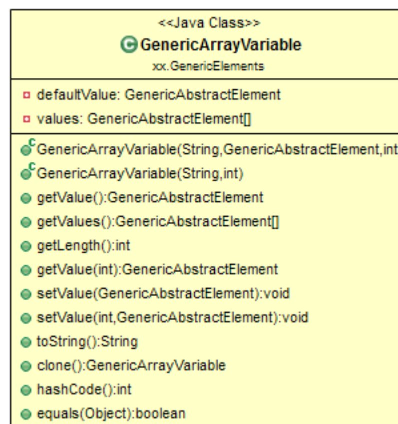

FIGURE 4.10: GenericArrayVariable class

- **GenericListVariable:** subtype of GenericAbstractVariable used to represent variable consisting of a list of values. This class contains a private GenericAbstractElement list to hold the values of the list and a GenericAbstractElement to represent the default value. The list, the default value and the initial length of the list are set at initialization and can be retrieved. Elements can also be retrieved and set by index. A new default value can be set, which only alters the values in the list that equal the current default value. If no default value is given at initialization, a GenericNull is used. The GenericListVariable has a variable length and provides methods to add and remove an element to/from the list of values, thereby providing additional functionality relative to the GenericArrayVariable.

- **GenericAbstractVariableReference:** used to represent variable references. This class contains a private GenericAbstractVariable that is set at initialization and can be retrieved.

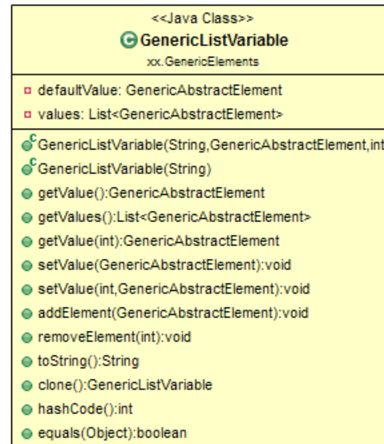

FIGURE 4.11: GenericListVariable class

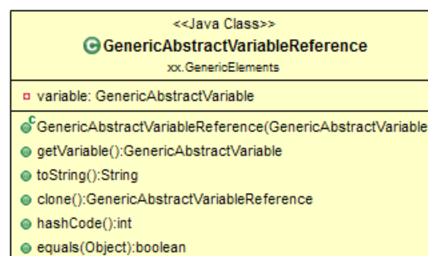

FIGURE 4.12: GenericAbstractVariableReference class

- **GenericAbstractVariableIndexReference:** used to represent a reference to a specific element of an abstract variable that contains multiple values. The specific value is selected using an index. This class contains a private GenericAbstract-Variable and GenericAbstractElement index that is set at initialization and can be retrieved.

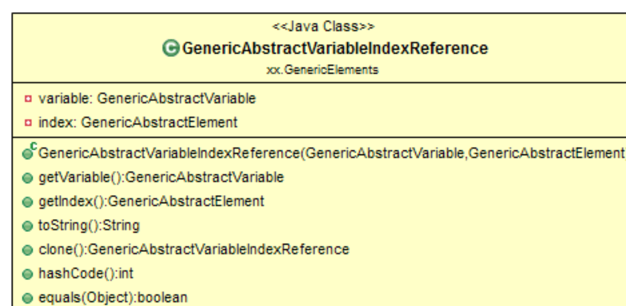

FIGURE 4.13: GenericAbstractVariableIndexReference class

- **GenericContainer:** used to represent containers. This class contains a private list with elements of type GenericAbstractElement. This list is set at initialization and can be retrieved. When no list is given in the constructor, an empty list is initialized. Next to that, there are methods to add an element to the list, remove an element using an index and retrieve an element using an index.

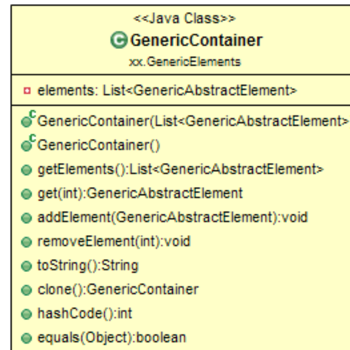

FIGURE 4.14: GenericContainer class

- **GenericUnaryExpression:** used to represent unary expressions. This class contains a private GenericAbstractElement to represent the expression and a GenericUnaryOperator to represent the unary operation to be applied to the expression. These values are set at initialization and can be retrieved.

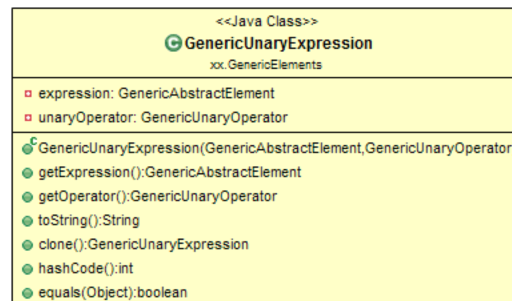

FIGURE 4.15: GenericUnaryExpression class

- **GenericUnaryOperator:** enumeration used to define the supported unary operators used by the GenericUnaryExpression elements. Each enumeration literal consists of a name, description and representation symbol. Supported unary operators are: negation (!), positive (+), negative(−).

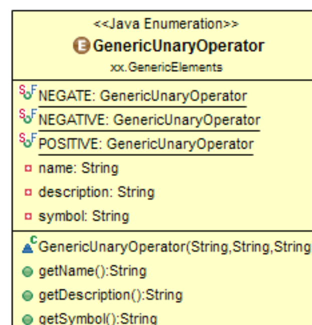

FIGURE 4.16: GenericUnaryOperator enumeration

- **GenericBinaryExpression:** used to represent binary expressions. This class contains two private GenericAbstractElements to represent the child expressions and a GenericBinaryOperator to represent the binary operation to be applied to the expressions. These values are set at initialization and can be retrieved.

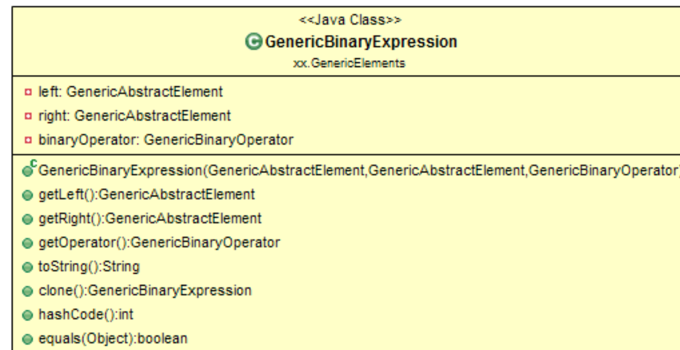

FIGURE 4.17: GenericBinaryExpression class

- **GenericBinaryOperator:** enumeration used to define the supported binary operators used by the GenericBinaryExpression elements. Each enumeration literal consists of a name, description and representation symbol. Supported binary operators are: multiplication $(*, /, \%)$, addition $(+, -)$, comparison $(<=, <, >, >=)$, equality $(==, !=)$, logical AND (&&), logical OR (||) and the modulo operator (%).

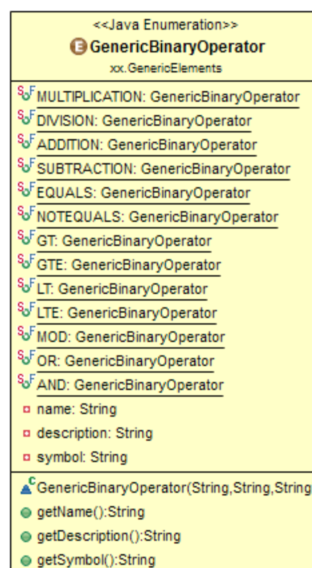

FIGURE 4.18: GenericBinaryOperator enumeration

- **GenericConditional:** an interface used as a generalization for the types of conditionals. It extends the GenericAbstractElement interface.

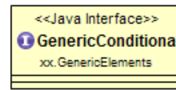

FIGURE 4.19: GenericConditional class

- **GenericIfConditional:** used to represent *if–then(–else)* constructs. This class contains a String to represent the name of the construct, and three GenericAbstractElements to represent the condition, if-clause and else-clause. These values are set at initialization and can be retrieved. When no else-clause is given, a GenericNull is used for this element. The class also contains a method to check whether the construct has an else-clause.

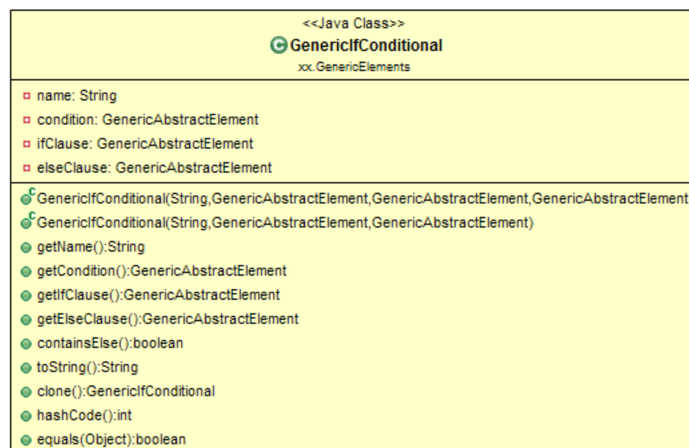

FIGURE 4.20: GenericIfConditional class

- **GenericFunction:** used to represent functions and mock domain-specific elements. This class contains a private String to represents the name of the function and a map where the keys consists of lists with type GenericAbstractElement to allow multiple inputs. The values are of type GenericAbstractElement, so only singular return values are allowed. These values are set at initialization and can be retrieved. A new map can be set, the current map can be updated with entries and values can be retrieved and removed given a key.

  Additional functions can be implemented by extending this base class and overriding the *getValue(List)* method.

- **GenericFunctionReference:** used to represent a reference to a specific input choice of a function. This class contains a private GenericFunction and a list of GenericAbstractElements acting as key for the GenericFunction. Both elements are set at initialization and can be retrieved.

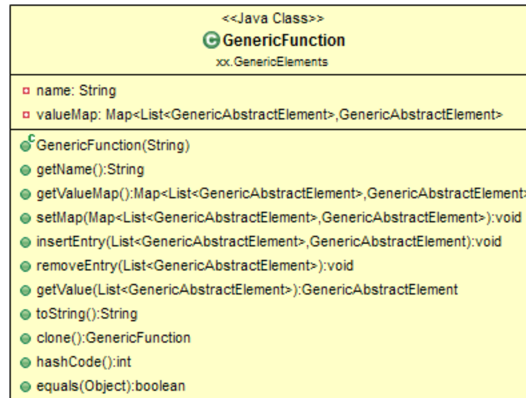

Figure 4.21: GenericFunction class

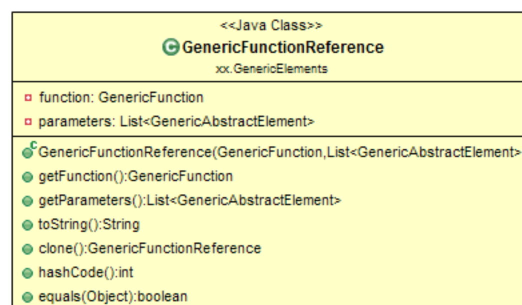

Figure 4.22: GenericFunctionReference class

4.2 Mapping

Since test case generation is based on a generic model, the domain-specific elements have to the transformed to the discussed supported elements. Since this framework was developed using the Xtext framework[1], a Xtext conformable method of transforming the domain elements is described. This is done using a transformer Xtend[2] class. Xtend is a dialect of Java, which compiles into readable Java 5 compatible source code. A Java class could also be generated, but this class would contain more boilerplate code. The user can also choose for another way of using a transformer class to obtain the generic elements given the domain elements, or exclude the transformer component and obtain the generic elements using model transformations.

When a domain-specific model file developed using the Xtext framework is saved the *doGenerate(Resource, IFileSystemAccess)* method of a generated Xtend class is called using an Eclipse Modeling Framework (EMF) Resource object[3] containing the domain-specific elements. The generated Xtend class can be extended with the functionality to forward the elements of the EMF Resource object to the developed transformer, which

[1]https://eclipse.org/Xtext/
[2]https://eclipse.org/xtend/
[3]org.eclipse.emf.ecore.resource.Resource

then transforms the elements to generic elements and return them. The generated Xtend class can then use these generic elements to generate different types of tests by calling the desired testing framework method.

To reduce the workload of developing a transformer class, a DSL was developed that allows the user to define a mapping from the domain-specific elements to a subset of the generic elements, resulting in a generated transformer Xtend class. If the mapping DSL does not support specific elements or structures of the domain model, the users have the option to define their own transformation methods for these elements by extending the generated transformer class or write a transformer class from scratch. Since the elements of domain-specific languages can be diverse, the mapping DSL focuses on supporting unary and binary expressions and common literals as these elements are consistent among languages.

4.2.1 The mapping DSL

Mapping

The documented mapping DSL grammar is given in Appendix A. To illustrate the function of the mapping DSL, a short snippet of a demo mapping is discussed here. The result of each mapping model is a generated transformer Xtend class that transforms domain-specific elements to generic elements. The package declaration of the resulting transformer class and the required imports, such as domain-specific elements, can be defined in the mapping model and are copied to the resulting transformer class (DemoTransformer).

Listing 4.1 gives an example mapping that maps multiplicative expressions and numbers to their generic counterpart, GenericBinaryExpression and GenericNumber respectively.

```
1  Grammar Demo

3  MainType DemoExpression

5  Type BinaryExpression

7  Type IntConstant

9  MAP Binary Expression BinaryExpression with leftChildMethod = left
   operatorMethod op returns BinaryOperator.MULTIPLY
11 rightChildMethod = right
   to type Multiplication;
13
   MAP Literal IntConstant to type NUMBER using operation value;
```

Listing 4.1: DemoMapping defining the mapping that generates the DemoTransformer Xtend class

The first line of the mapping defines the name of the grammar (Demo), which is used to name the resulting Xtend class.

In line 3 the MainType of the elements is defined, based on the assumption that a generalization is used to define the elements. This MainType is mapped to the GenericAbstractElement type.

After defining the main type, the different subtypes are defined in lines 5 and 7, using ‘Type’ plus the name of the subtype. These subtypes are used as the target subtypes provided during the definition of subsequent element mappings.

Line 9 till 12 describes an example binary mapping. After the keyword ‘MAP Binary Expression’ the name of the subtype should be given and, as stated, only defined subtypes are accepted. Since binary expressions have two child expressions and an operator, these elements have to be defined. The child expressions are defined by specifying the methods to get the children from the binary expression, while the operator is defined by specifying the method to get the operator and the resulting binary operation. In this case, *op()* is the method to retrieve the operator from the binary expression and BinaryOperator.MULTIPLY is the binary operation. The binary operations of the example DSL are defined in the DSL using an enumeration. Finally the user specifies one of the supported GenericBinaryOperators to be used as the operator in the GenericBinaryExpression. The supported GenericBinaryOperators are defined as an enumeration in the grammar.

Line 14 describes an example literal mapping. The name of the target type is again one of the subtypes defined in the mapping. The target type is a LiteralTarget, which is again defined in the grammar as an enumeration, and the operation should specify the method to extract the value from the DSL literal object.

This mapping is then used to generate a transformer Xtend class file, which transforms the domain-specific elements to the generic elements. To illustrate this functionality a DemoTransformer is discussed next.

Transformer

Listing 4.2 describes the DemoTransformer Xtend class, which transforms Demo elements (DemoExpression objects) to generic elements (GenericAbstractElement objects), so these elements can be used in the framework to generate test cases.

```
1  public class DemoTransformer {
2
3      def GenericAbstractElement transform(DemoExpression element){
4          if(element instanceof BinaryExpression){
5              transform(element as BinaryExpression)
6          }
7          else if(element instanceof IntConstant){
8              transform(element as IntConstant)
9          }
10         else {
11             throw new Exception("Transformer encountered unsupported Expression
       type: " + expression);
12         }
13     }
14
15     def GenericBinaryExpression transform(BinaryExpression expression){
16         if(expression.op.equals(BinaryOperator.MULTIPLY)){
17             return new GenericBinaryExpression(transform(expression.left),
       transform(expression.right), GenericBinaryOperator.MULTIPLICATION)
18         }
19         else {
20             throw new Exception("Transformer encountered unsupported
       BinaryOperator type: " + expression.operator)
21         }
22     }
23
24     def GenericNumber transform(IntConstant expression){
25         return new GenericNumber(expression.value);
26     }
27
28 }
```

Listing 4.2: Generated DemoTransformer Xtend class which transforms DemoExpression objects to GenericAbstractElement objects

The class starts with a method (line 3 till 13) to transform the different types of demo expressions (e.g., binary expressions) by exploiting polymorphism, based on the types defined in the mapping file. When an unsupported type is encountered, an exception is thrown.

Since the demo mapping specifies one binary expression operation (multiplication), the method to transform binary expressions (line 15 till 22) only has one if-clause and no else-if clauses. The method's function is to transform the BinaryExpression to a GenericBinaryExpression while also transforming the child expressions. When an unsupported binary operator is encountered, an exception is thrown.

IntConstants are transformed (line 24 till 27) by creating a GenericNumber with the value of the expression.

4.2.2 Mapping domain-specific elements

As previously addressed, domain-specific elements can either be mapped to the generic elements, simulated using the GenericFunction element or mapped to a newly developed element. If the generic metamodel is extended with a new element, the transformer can also use this new element as mapping target type. To illustrate the latter two options, the following code snippet is given in Listing 4.3. This snippet is defined in the transformer Xtend class.

```
def void instantiateFunctions(){
2
    //Define new function
4   var GenericFunction demoFunction = new GenericFunction("demoFunction");

6   //Define input sequence one
    var List<GenericAbstractElement> input1 = new ArrayList<
    GenericAbstractElement>();
8   input1.add(new GenericString("Foo"));
    input1.add(new GenericNumber(20));
10
    //Define input sequence two
12  var List<GenericAbstractElement> input2 = new ArrayList<
    GenericAbstractElement>();
    input2.add(new GenericString("Bar"));
14  input2.add(new GenericNumber(30));

16  //Add input sequences and their result to the function
    demoFunction.insertEntry(input1, new GenericBoolean(true));
18  demoFunction.insertEntry(input2, new GenericBoolean(false));

20  //Add the function to the list of supported functions
    functions.put("demoFunction", demoFunction);
22  //Add an externally define function type to the list of supported functions
    functions.put("Abs", AbsFunction());
24 }
```

LISTING 4.3: Example of defining functions in the transformer Xtend class

The GenericFunction element with name *demoFunction*, instantiated in line 3 and filled with entries in line 14 and 15, simulates a domain element that returns *true* when the inputs *Foo* and *20* are given, and returns *false* when the inputs *Bar* and *30* are given.

This functionality is achieved by creating two input lists and filling these list with the parameters of type GenericAbstractElements (or subtype). These input lists are then used in the *insertEntry(List, GenericAbstractElement)* method that adds a key-value pair to the GenericFunction element, where the value represents the output for the specific input.

The user can also define their own functions by extending the GenericFunction class. In the snippet the function *AbsFunction* was developed, which returns the absolute value of a number. This new function can then be added to the map of functions in the transformer using an String key that acts as the identifier.

4.2.3 Example artifacts

In Appendix B, a grammar is given of a developed DSL called *Precedence*. This language was originally developed to show that models and languages with different operator precedence could be transformed, but was later extended to act as a testing language for the framework. Models of this language can be successfully transformed to common element models using the transformer given in Appendix C. The grammar and transformer can be used as examples for the development of future transformers.

4.3 Framework options

Having achieved a generic representation for the language elements, these generic elements can be used in the framework to create test cases and executable tests. To provide the user with a clear overview of all the options available in the framework, the GenericOptions Java class was developed. A visual representation of this class is shown in Figure 4.23.

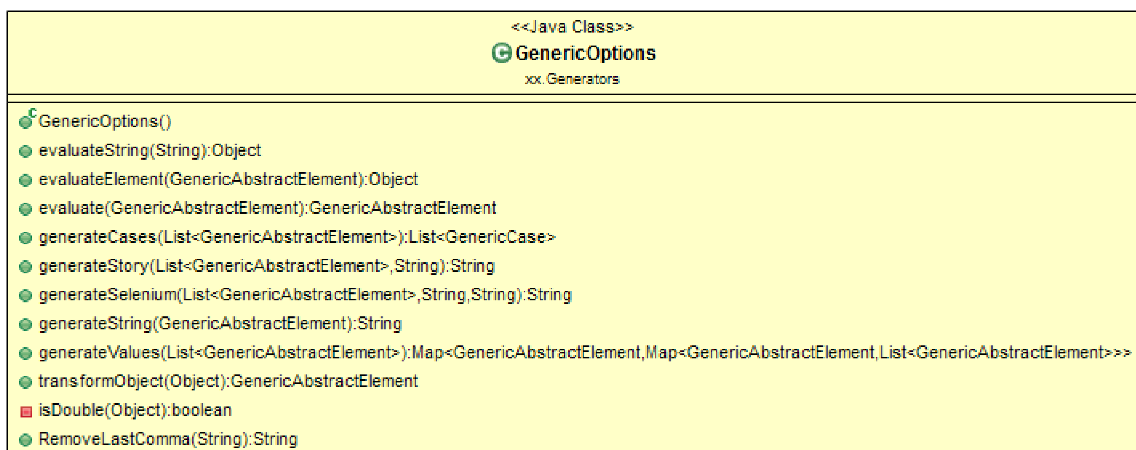

FIGURE 4.23: GenericOptions Java class

It provides the following methods:

- *transformObject(Object)* transforms a Java Object to a GenericBoolean, GenericNumber, GenericString, GenericAbstractElement array or GenericNull

  For the evaluation of expressions, the JavaScript engine *eval(String)* method is used, which is explained in more detail in Chapter 5. The result of this *eval(String)* method is a Java Object. The *transformObject(Object)* method tries to parse the Object to a GenericBoolean, GenericNumber, GenericString or GenericAbstractElement array so it can again be used by the framework components. If this fails, the method returns a GenericNull. Additional result types can be added without major efforts.

- *generateString(GenericAbstractElement)* generates a String representation for a GenericAbstractElement

  Transforms a GenericAbstractElement to a String so it can be evaluated by the *eval(String)* method of the JavaScript engine.

- *evaluateString(String)* evaluates a String using the *eval(String)* method of the JavaScript engine and return the Java Object

  Before the expressions in the generic model can be evaluated by the *eval(String)* method of the JavaScript engine, these expressions first need to be translated to String format. The JavaScript method returns a Java Object with the evaluated value.

- *evaluateElement(GenericAbstractElement)* evaluates a GenericAbstractElement and return the Java Object

  This method transforms the GenericAbstractElement to a String using the method *generateString(GenericAbstractElement)* and then calls the method *evaluateString(String)*.

- *evaluate(GenericAbstractElement)* evaluates a GenericAbstractElement and return a GenericAbstractElement

  This method does the same as the method described before but also parses the resulting Java Object using the *transformObject(Object)* method.

- *generateValues(List)* generates a map of variables and values for a list of GenericAbstractElements

  This map defines the values of the variables that are used to generate test cases. The values are determined so that branch/condition coverage is achieved with the resulting set of test cases.

- *generateCases(List)* generates test cases for a list of GenericAbstractElements

  Using the generic model and the map of values, this options generates a list of GenericCase elements that define the pre- and postconditions of the tests. These are setup generically so they can easily be converted to different kinds of tests.

- *generateStory(List, String)* generates a JBehave story in String format for a list of test cases and the DSL name

  Transforms a list of generic test cases to a JBehave story. The DSL name is used to choose the correct test case to executable test transformer.

- *generateSelenium(List, String, String)* generates a Selenium test in String format for a list of test cases, the DSL name and the file name

  Transforms a list of generic test cases to a Selenium test. The DSL name is used to choose the correct test case to executable test transformer.

Chapter 5

Case generation

This chapter discusses the evaluation of expressions, and how this is used in test case generation. Section 5.1 discusses how expressions are evaluated in the framework. Section 5.2 discusses how expression evaluation is used to generate values for test cases. Section 5.3 discusses how these values are used to develop cases. Section 5.4 describes an example to illustrate the process from domain model to value and case generation.

5.1 Expression evaluation

After the expressions and elements defined in the domain-specific models are transformed to generic elements, these elements can be used by the framework to generate test cases. To achieve branch/condition coverage, test values need to be generated to construct all test cases. For the generation of test values, the generic elements, such as expressions, need to be evaluated. Since the values used in the expressions and the evaluated expression values are determined during run-time, the framework requires dynamic evaluation of expressions. There are several ways to dynamically evaluate expressions in Java, of which three are discussed here.

1. Dynamic compilation, instantiation and execution of Java classes

   A flexible, yet complicated approach is to wait with the compilation of the expression classes until the required values are determined. After assigning the values, compile and load the class to evaluate the expression.

2. Evaluate expressions using Java libraries

   There are numerous libraries available that evaluate expressions in Java. Examples are: JEXL[1] and exp4j[2]. Each library has its own approach to evaluate expressions and supports different operators.

3. Evaluate expressions using scripting engines

   Instead of using a library, expressions can also be evaluated using a scripting engine. The JavaScript engine is by default included in the Java Virtual Machine, but other scripting engines are also supported. For the JavaScript engine approach, the expressions are transformed to a String representation and used as input in the JavaScript *eval(String)* method. This method evaluates the String and returns a Java Object with the evaluated value.

We have chosen to transform the expressions into Strings and evaluating these Strings using the *eval(String)* method of the JavaScript engine, since this method does not require third-party libraries and is straightforward to implement. One important aspect of this approach is that the String representation must correctly represent the GenericAbstractElement object and the original (domain-specific) element.

5.1.1 Expression trees

Algebraic expressions have an inherent tree-like structure, and can be represented as trees, called expression trees. The operators of the expression form the non-terminal nodes of the tree, while the variables and constants form the terminal nodes, also called leafs. Parentheses presented in the expression can be omitted in the tree, since the operator precedence can be derived from the structure of the tree. The child nodes are executed first and therefore have a higher precedence than the parents.

As domain-specific models are often converted to an Abstract Syntax Tree (AST), thereby representing the expressions as trees, one way of transforming the expressions to a String is by traversing the tree while transforming each node. An important element to note here is that the (implicit) parenthesis can be omitted in the AST since the tree structure already defines the order. This can cause incorrect operator precedence after conversion, resulting in incorrect outputs for the tests. To cope with this problem, parentheses are added to each expression in the tree during the transformation process, thereby separating operator precedence from the evaluation process. This makes

[1]http://commons.apache.org/proper/commons-jexl/
[2]http://www.objecthunter.net/exp4j/

sure the expressions are evaluated in the right order, thereby maintaining the correct precedence. An example is given next.

Given the expression

$$2 * 3 + 4$$

and two languages, language A and language B, with A prioritizing multiplication over addition while B prioritizes addition over multiplication.

In this example the tree of language A is structured as shown in Figure 5.1.

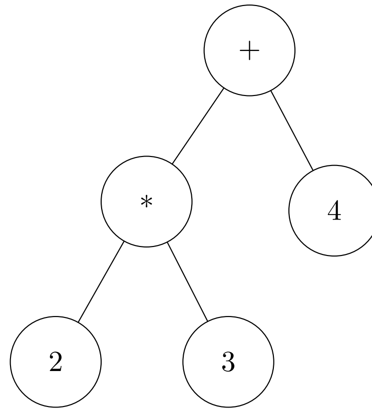

FIGURE 5.1: Binary tree of the expression $2 * 3 + 4$ with priority on multiplication

In contrast, the tree of language B is structured as shown in Figure 5.2.

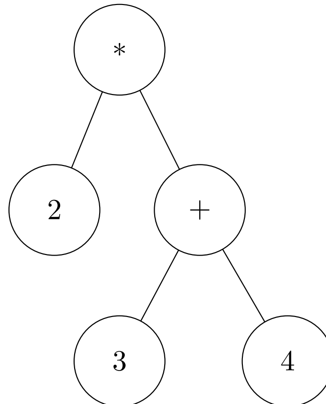

FIGURE 5.2: Binary tree of the expression $2 * 3 + 4$ with priority on addition

Since we omitted the parenthesis in these expression trees, these have to be added when transforming the tree to the String representation to correctly preserve operator precedence. To retrieve a String representation of the expression trees we use an *inorder* traversal of the expression described in Preiss [25] as:

- When encountering a terminal node (leaf), return the String representation.
- When encountering a non-terminal node, do the following:
    1. Return a left parenthesis; and then
    2. traverse the left subtree; and then
    3. return the root String representation; and then
    4. traverse the right subtree; and then
    5. return a right parenthesis.

Using this method, the expression of language A is represented in String format as:

$$((2.0 * 3.0) + 4.0)$$

which results in 10 when evaluated.

And the expression of language B is represented in String format as:

$$(2.0 * (3.0 + 4.0))$$

which results in 14 when evaluated.

The example shows that this method can evaluate expressions correctly taking into consideration the operator precedence of the language. The method is used to setup a generic expression representation that can support different language operator precedences.

5.1.2 String transformation

To transform the GenericAbstractElements to a String, the class GenericStringGenerator was developed. The objective of the class is to transform a GenericAbstractElement to a String representation, ready to be evaluated by the *eval(String)* method of JavaScript. The result is achieved by using polymorphism and transforming each element recursively to its String representation. The goal of the transformation is that the result String can be correctly evaluated using the JavaScript *eval(String)* method, while still representing the same information as the original expression. The class GenericStringGenerator is shown in Figure 5.3.

The string generator algorithm is given as pseudo code in Algorithm 1, and a snippet of the GenericStringGenerator code is shown in Listing 5.1.

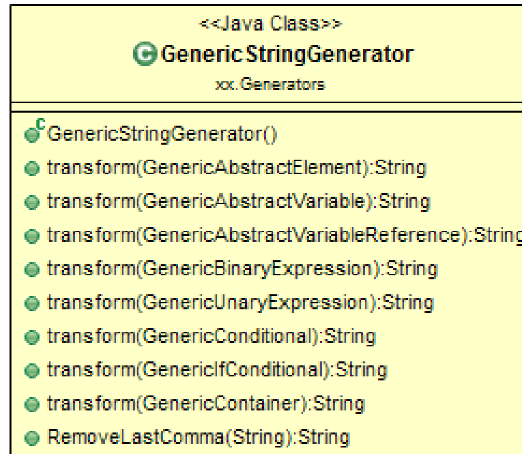

FIGURE 5.3: GenericStringGenerator class

Algorithm 1 String generator algorithm

1: **procedure** TRANSFORM(*GenericAbstractElement* element)
2: Use polymorphism to forward element
3: **if** element *instanceof* literal **then** ▷ literal = Number, String, Boolean, etc.
4: **return** element.*toString*()
5: **else**
6: **return** syntax + TRANSFORM(element.*children*()) ▷ syntax = (,), -, !, etc.
7: **end if**
8: **end procedure**

```java
public class GenericStringGenerator {

    public String transform(GenericAbstractElement element) {
        String value = "String generator encountered unsupported type: " +
    element;
        if (element instanceof GenericBinaryExpression) {
            return transform((GenericBinaryExpression)element);
        }
        else if (element instanceof GenericNumber) {
            return "" + ((GenericNumber) element).getValue();
        }
        return value;
    }

    public String transform(GenericBinaryExpression bE) {
        return "(" + transform(bE.getLeft()) + bE.getOperator().getSymbol() +
    transform(bE.getRight()) + ")";
    }
}
```

LISTING 5.1: GenericStringGenerator used to transform the GenericAbstractElements to Strings

The class uses polymorphism to differentiate between the subtypes of GenericAbstractElement. GenericBinaryExpressions are transformed to Strings using the method *transform(GenericBinaryExpression bE)* starting at line 14. The String representation is obtained using the same method described earlier.

1. Start with a left parenthesis; and then
2. parse the left child expression; and then
3. add the binary operator; and then
4. parse the right child expression; and then
5. add a right parenthesis.

As previously mentioned, parenthesis are added to maintain the correct operator precedence. An example that shows that precedence is correctly maintained.

Given again the two languages A and B, with A prioritizing multiplication over addition while B prioritizes addition over multiplication and the expression:

$$if(30 > (4 + 4 * 5), 2, 3)$$

.

The generator produces the following String for language A:

$$if(30 > ((4 + (4 * 5))))2\ else\ 3$$

which results in 3 when evaluated.

And for language B:

$$if(30 > (((4 + 4) * 5)))2\ else\ 3$$

which results in 2 when evaluated.

With this example, we show that the original expression and operator precedences are maintained.

5.1.3 String evaluation

As explained in Section 5.1, we have chosen to evaluate the expressions as Strings using the *eval(String)* method of the JavaScript engine. The GenericOptions class provides the user with several options/methods to evaluate a GenericAbstractElement. Listing 5.2 shows the default method, which accepts String parameters and returns a Java Object.

```
public Object evaluateString(String element){
    //Setup engine manager
    ScriptEngineManager manager = new ScriptEngineManager();
    //Setup JavaScript engine
    ScriptEngine engine = manager.getEngineByName("js");
    //Try to evaluate the String object
    try {
        return engine.eval(element);
    } catch (ScriptException e) {
        System.err.println("Exception while evaluating expression");
        e.printStackTrace();
    }
    return "Evaluation Exception";
}
```

Listing 5.2: *evaluateString(String)* method used to evaluate transformed GenericAbstractElements using the JavaScript engine

There are also methods that have a GenericAbstractElement as parameter and return a Java Object or a GenericAbstractElement. In the latter case, instead of returning a Java Object, we parse the Java Object using the *transformObject(Object)* method to a GenericAbstractElement (currently supported types are: GenericBoolean, Generic-Number, GenericString, GenericAbstractElement array or GenericNull) and return this element.

5.2 Value generation

Having the option to evaluate expressions, the value generator could be developed. Its purpose is to generate values for the variables in an expression that do not have a value assigned yet, so that when the expression is tested with all generated values, branch/-condition coverage is achieved. The class GenericValueGenerator was developed for this functionality. To generate the values, the user should call the *generateValues(List)* method of the GenericOptions class, which takes a list of generic elements (the transformed model elements) as parameter. This method returns a map consisting of variables and their possible values. Its only function is to call the GenericValueGenerator for each element in the list of generic elements and add that result to the result list. The class GenericValueGenerator is shown in Figure 5.4.

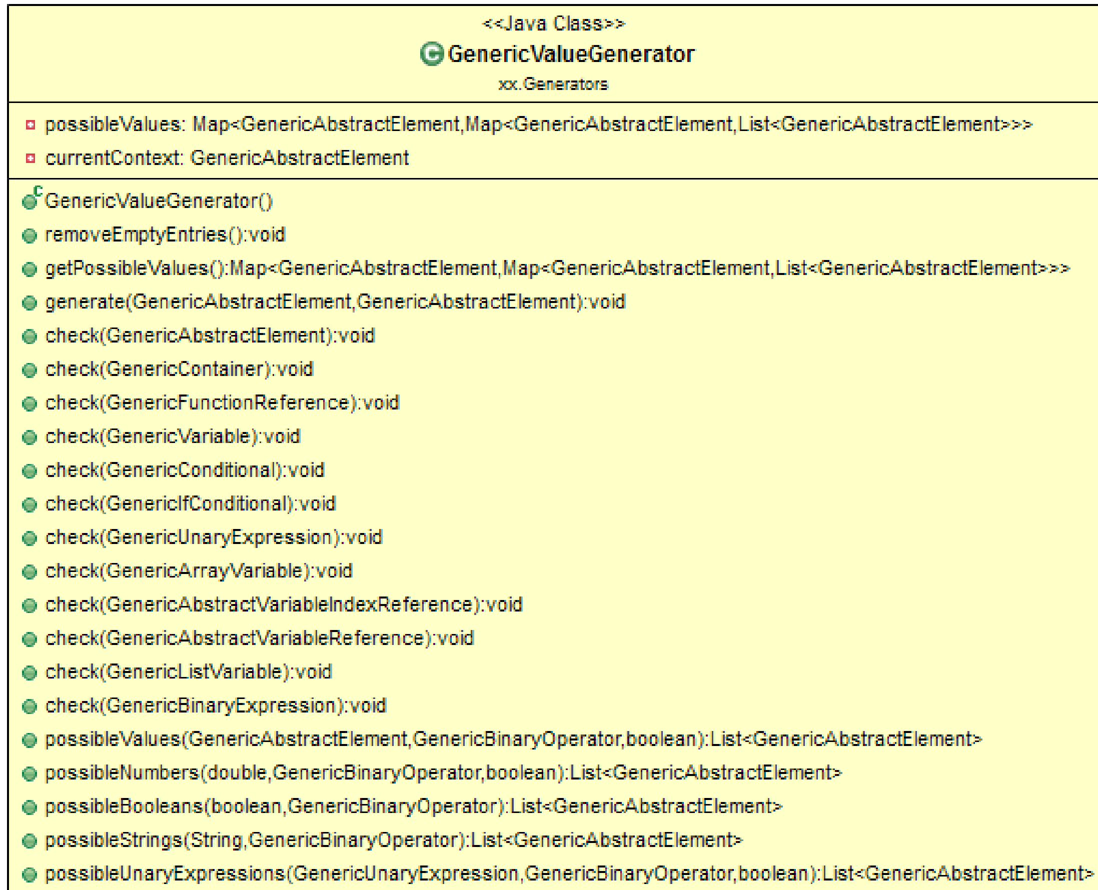

FIGURE 5.4: GenericValueGenerator class

To reach branch/condition coverage, each branch and condition needs to be evaluated. Applying this theory to the supported elements means the *if* and *else* clause of each *if-then(-else)* construct has to be evaluated and each conditional expression has to be evaluated to *true* and *false*. Since the *if* and *else* clause are evaluated when the condition of the *if-then(-else)* construct is evaluated to *true* and *false*, the framework can focus on conditional expressions. The variable conditional expressions, expressions being able to evaluate in *true* and *false*, supported by the framework are binary expressions with a comparison or equality operator. When the binary expression contains a comparison operator the data type of both sub-expressions must be number or unary expression for it to be a valid expression (evaluable by the JavaScript engine), while the equality operator can be applied regardless of sub-expression data type.

The *check(GenericAbstractElement)* method of the generator checks each element for its data type and uses polymorphism to forward the element until a binary expression is encountered. If this happens, the operator of the binary expression is compared to the supported comparison and equality operators. If there is a match, the child expressions of the binary expressions are checked for unassigned variables, i.e. variables with value GenericNull. For these variables, the possible values are generated. If a

different element is found, the generator continues checking the right and left child of the binary expression. This process is shown as pseudo code in Algorithm 2.

Algorithm 2 Value generator algorithm

1: **procedure** CHECK(*GenericAbstractElement* element)
2:     Use polymorphism to forward element
3:     **if** element *instanceof* GenericBinaryExpression **then**
4:         **if** binaryOperator *instanceof* <=,<,>,>=,== or != **then**
5:             **if** left child *instanceof* GenericAbstractVariable && value = Null **then**
6:                 POSSIBLEVALUES(right child value, binaryOperator, true)
7:             **else if** right child *instanceof* GenericAbstractVariable && value = Null **then**
8:                 POSSIBLEVALUES(left child value, binaryOperator, false)
9:             **else**
10:                 CHECK(element.*children*())
11:             **end if**
12:         **end if**
13:     **else**
14:         CHECK(element.*children*())
15:     **end if**
16: **end procedure**

For the generation of possible values, four cases have been identified, depending on the data type of the other sub-expression (not the unassigned variable). Currently the following four data types are supported: GenericBoolean, GenericNumber, GenericString and GenericUnaryExpression. If the data type is not supported, the generic element is passed to the *evaluate(GenericAbstractElement)* method of the framework, thereby evaluating the expression using the JavaScript engine and parsing the return value. The return value is either a supported parser generic element or GenericNull. This procedure could cause an exception thrown by the JavaScript engine since the expression might contain unassigned variables. More support could be added in the future to facilitate more user freedom when defining expressions. An overview of the valid supported combinations of binary operators and data types of sub-expressions is given in Figure 5.5. For these combinations, values can be generated by the framework. The algorithm for generating values is given in Algorithm 3.

To achieve both binary expression results, *true* or *false*, the unassigned variable must at least get assigned two values. The first value is equal to the value of the other sub-expression in the binary expression, resulting in either *true* or *false* depending on the binary operator. The second value is determined per case. The generated values are ordered so that the evaluated Boolean expression always first results in *true* and then in *false*. When the data type of the other sub-expression is a GenericBoolean or GenericString, the second value is a negated version of the other sub-expression (*true*

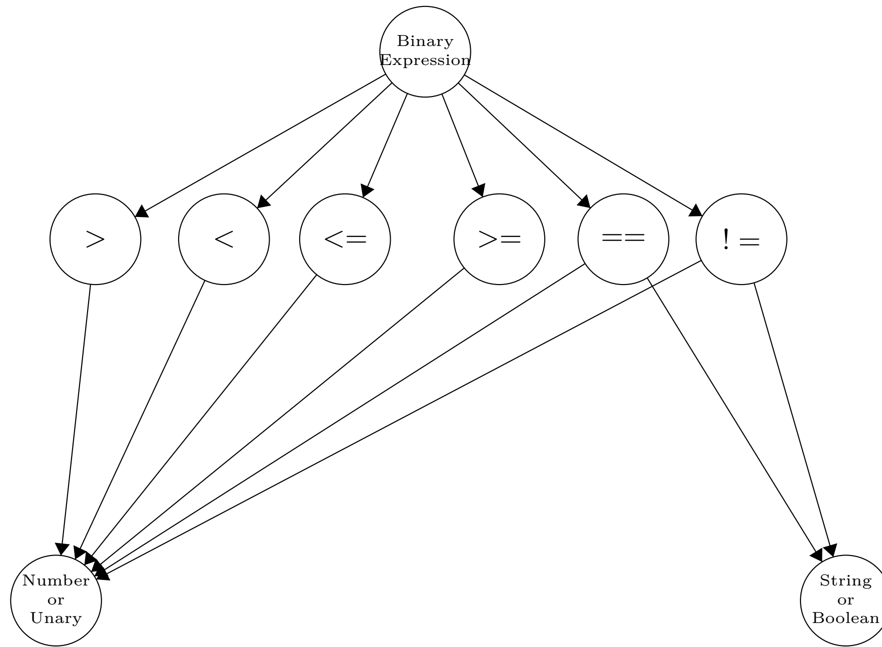

FIGURE 5.5: Overview of the valid supported combinations of binary operators and types of expressions for value generation

for *false* and vice versa for Booleans and a "Not " prefix for Strings). This is visually represented in Figure 5.6 and 5.7. To illustrate these cases an example is given next.

The binary expressions (var A ! = "test") and (var B == true) result in the generation of four values. For variable *A* the values *"Not test"* and *"test"* are generated, while for variable *B* the values *true* and *false* are generated. These four values make sure both binary expressions evaluate to *true* for the first generated value, and to *false* for the second generated value.

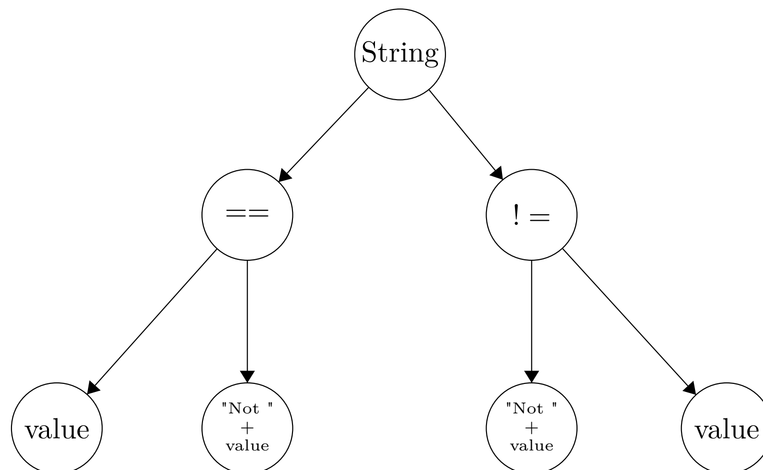

FIGURE 5.6: Overview of the generated Strings depending on the binary operator

Algorithm 3 Possible values generator algorithm

```
1: procedure possibleValues(GenericAbstractElement element, BinaryOperator
   binaryOp, Boolean left)
2:     if element instanceof GenericString then
3:         Return possibleStrings(element, binaryOp)                ▷ Figure 5.6
4:     else if element instanceof GenericBoolean then
5:         Return possibleBooleans(element, binaryOp)               ▷ Figure 5.7
6:     else if element instanceof GenericNumber then
7:         Return possibleNumbers(element, binaryOp, left)          ▷ Figure 5.8
8:     else if element instanceof GenericUnaryExpression then
9:         if unaryOperator instanceof NegateOperator then
10:            if unaryElement instanceof GenericBoolean then
11:                Return new UnaryExpressions using possibleBooleans(element,
   binaryOp)
12:            end if
13:        else if unaryOperator instanceof PositiveOperator then
14:            if unaryElement instanceof GenericNumber then
15:                Return new UnaryExpressions using possibleNumbers(element, bi-
   naryOp, left)
16:            end if
17:        else unaryOperator instanceof NegativeOperator
18:            if unaryElement instanceof GenericNumber then
19:                Return new UnaryExpressions using possibleNumbers(element, bi-
   naryOp, left.negate())
20:            end if
21:        end if
22:    else
23:        Return possibleValues(Evaluate(element), binaryOp, left)
24:    end if
25: end procedure
```

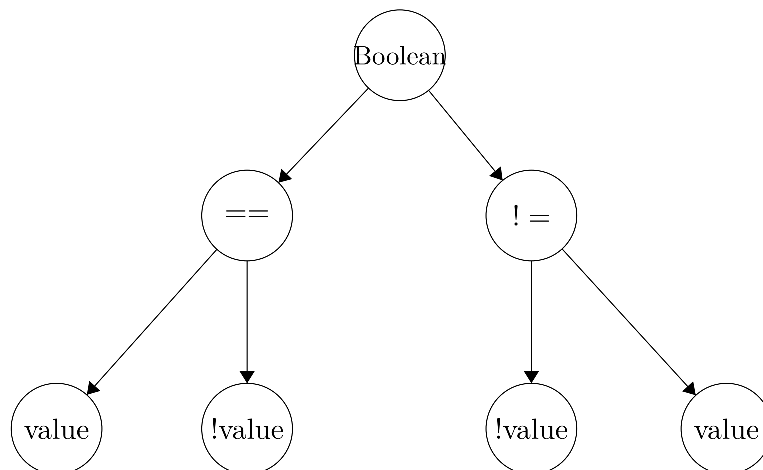

FIGURE 5.7: Overview of the generated Booleans depending on the binary operator

If an equality operator (==, !=) is applied to an unassigned variable and a GenericNumber, the second value is determined by taking the value of the other sub-expression

and increment it with the value 1. When the comparison operator > or <= is applied, the second value is again equal to the value of the other sub-expression incremented with the value 1, while the comparison operators < and >= return the value of the other sub-expression decremented with the value 1. All the value generation paths for an unassigned variable and a GenericNumber are visually represented in Figure 5.8. The nodes *Left* and *Right* represent whether the left or right sub-expression of the binary expression is the unassigned variable. This information is used during the generation process to make sure the binary expressions evaluate to *true* for the first generated value and to *false* for the second generated value.

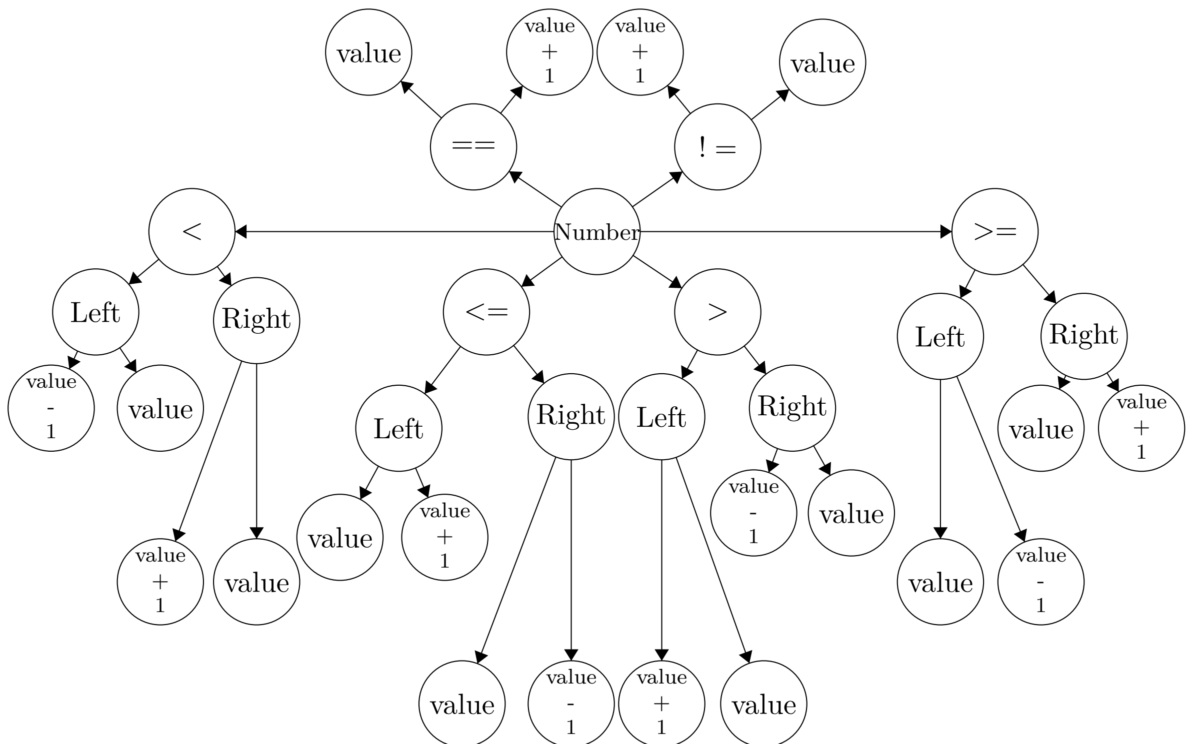

FIGURE 5.8: Overview of the generated numbers depending on the binary operator and binary sub-expressions position

If an equality operator is applied to an unassigned variable and a GenericUnaryExpression, the values are determined depending on the unary operator. If the unary operator *negate* is detected, the unary sub-expression must be of type GenericBoolean since the Boolean negate operator cannot be applied to numbers. The values for the unassigned variable are therefore generated using the *possibleBooleans(Boolean, BinaryOperator)* method and two new unary expressions are returned, representing the negated version of the generated Boolean values. When the unary operator is *positive* or *negative*, the unary sub-expression must be a of type GenericNumber since these unary operators cannot be applied to Booleans. In case of the positive operator, the *possibleNumbers(double, BinaryOperator, Boolean)* method can be reused since the method already assumes two

positive numbers. Two unary expressions are returned, containing the generated numbers and a positive unary operator. The same holds for the negative operator, however the generation of values is changed. Where the generator would normally increment the value, the value is decremented and vice versa to ensure the first value results in the expression evaluating to true and the second value results in the expression evaluating to false. This is achieved by calling the number value generation process using a negated Boolean (used to determine the position of the unassigned variable). So given binary expression $A > -3$, the number value generation process is called with binary expression $3 > A$, so the numeric values 2 and 3 are generated. These number values are then used to create unary expression values -2 and -3. Resulting in binary expressions $-2 > -3$ (true) and $-3 > -3$ (false).

5.3 Variable assignment

Having generated the variable values, these values can be used to generate the cases to achieve branch/condition coverage by assigning the generated values to the unassigned variables. For this the helper class GenericVariableAssigner was developed which is called by the case generator (discussed in detail in Section 6.1), using a generic element, unassigned variable name and the value to be assigned as parameters. The variable assigner will then return the element where the value of the variable is set to the value in the parameter. The GenericVariableAssigner class is graphically represented in Figure 5.9.

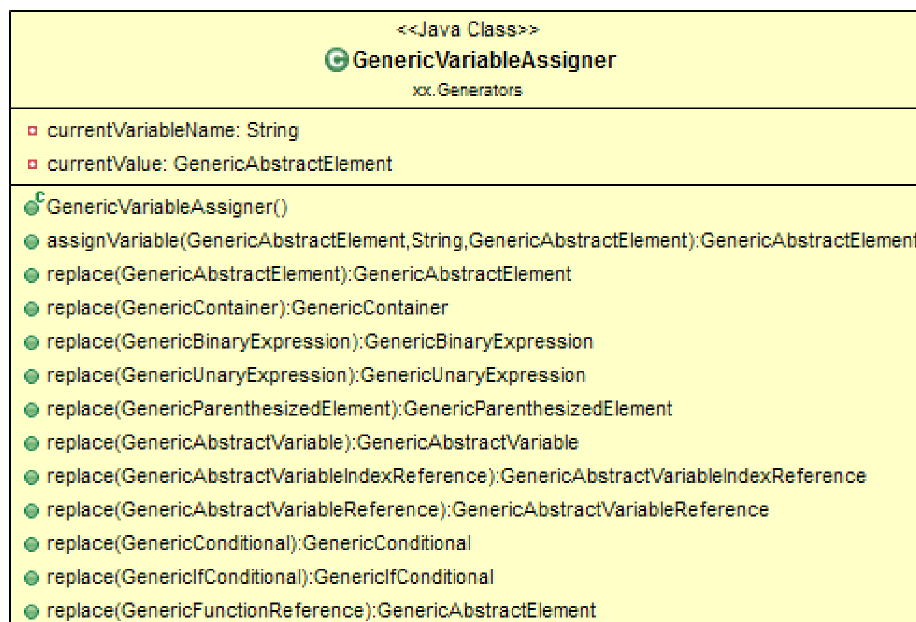

FIGURE 5.9: GenericVariableAssigner class

The assignment process traverses the original element until a variable with value GenericNull is encountered. The variables are matched on name and context, thereby requiring the model to have unique variable names. The context object is the expression that uses the variable, which is taken into account to ensure multiple expression can use the same variable. When a match is found, the value of the variable is set to the new value from the variable values map which is passed as a parameter. If the element does not contain the variable, its children are traversed. If the element does not have children, it is returned in its original form. This process is shown as pseudo code in Algorithm 4.

Algorithm 4 Value assigner algorithm

```
1: procedure Replace(GenericAbstractElement element)
2:     Use polymorphism to forward element
3:     if element instanceof GenericAbstractVariable then
4:         if replaceVariableName = element.name() then        ▷ Compare on name
5:             element.setValue(replaceValue)                  ▷ Replace the value
6:         else
7:             Replace(element.children())
8:         end if
9:     else
10:        Replace(element.children())
11:    end if
12: end procedure
```

5.4 Example

An example of the model transformation and value generation process for the Precedence test language is discussed next. The example Precedence model is given in Listing 5.3 and visually represented in figure 5.10. The model contains three unassigned variables and a function that uses these three variables. We will first transform the domain-specific model to a generic model and then show that branch/condition coverage is achieved by generating values and cases for the generic model.

```
variable double A

variable bool B

variable String C

function String Foobar if((var A > 3) && (var B != true) && (var C == "test"), "
    true clause", "false clause")
```

Listing 5.3: Precedence example model code

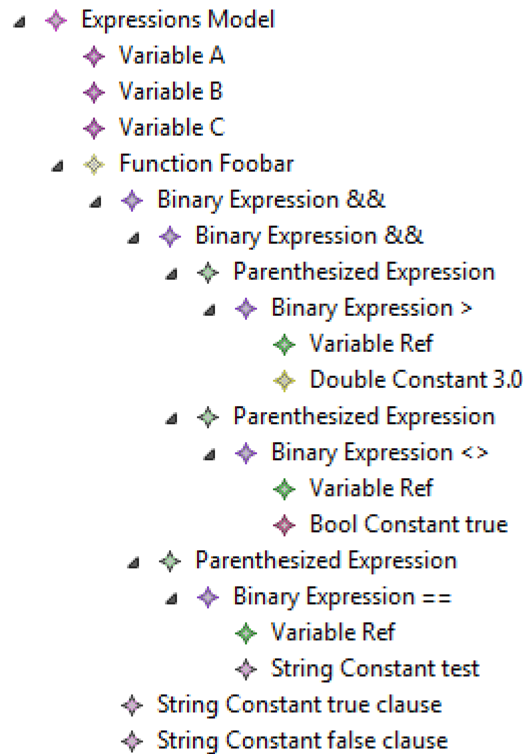

FIGURE 5.10: Precedence example model

For the generation of test values and generic test cases, this domain-specific model has to be transformed to a generic model. This can be achieved in several ways, for example, using the PrecedenceTransformer class in Appendix C. Another way of achieving the generic model is by model transformations. In figure 5.11 a generic model is shown obtained using an ATL model transformation, the Precedence metamodel, the common elements metamodel and the domain-specific model. For the remainder of this example, the result of the developed PrecedenceTransformer was used.

Now that the model is an instance of the common elements metamodel, the value generation process can start. The model contains three unassigned values used in a binary expression with a supported operator:

1. Numeric variable A used in the binary expression: *A* is greater than *3*
2. Boolean variable B used in the binary expression: *B* is not equal to *true*
3. String variable C used in the binary expression: *C* is equal to *"test"*

To achieve branch/condition coverage for the function *Foobar* that uses all three binary expression, all the expressions must evaluate in *true* and *false* (condition coverage) and both the if-clause and else-clause of the *Foobar* function have to be evaluated.

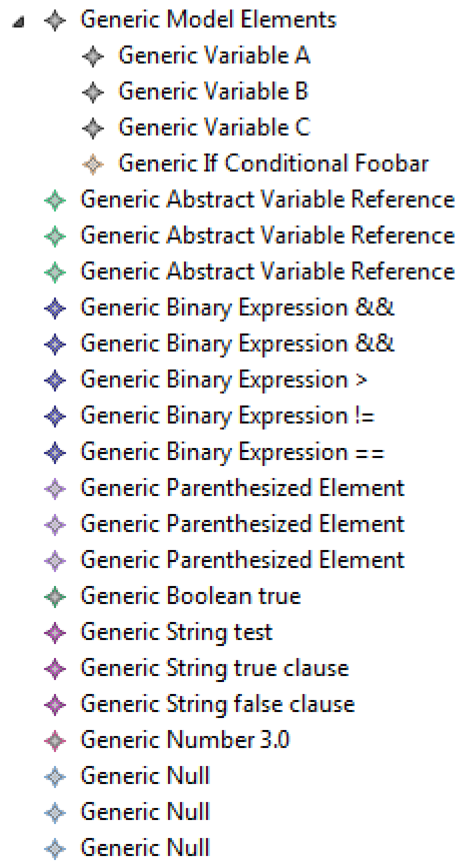

FIGURE 5.11: Generic example model

The value generator therefore generates the following map of values:

1. Variable A with values 4 and 3
2. Variable B with values false and true
3. Variable C with values "test" and "Not test"

Using these variable values, all three binary expressions evaluate in *true* for the first value and *false* for the second value. Combining all these cases, 8 test cases can be constructed using the variable assigner:

1. if((4>3) && (false!=true) && ("test"=="test"), "true clause", "false clause")
2. if((4>3) && (false!=true) && ("Not test"=="test"), "true clause", "false clause")
3. if((4>3) && (true!=true) && ("test"=="test"), "true clause", "false clause")
4. if((4>3) && (true!=true) && ("Not test"=="test"), "true clause", "false clause")

5. if((3>3) && (false!=true) && ("test"=="test"), "true clause", "false clause")
6. if((3>3) && (false!=true) && ("Not test"=="test"), "true clause", "false clause")
7. if((3>3) && (true!=true) && ("test"=="test"), "true clause", "false clause")
8. if((3>3) && (true!=true) && ("Not test"=="test"), "true clause", "false clause")

These cases makes sure that every binary expression in function *Foobar* is condition covered. It also ensures that the function is branch covered, although this example only has two branches. Using these two properties, branch/condition coverage is achieved for the function. In Chapter 6, test generation for this example is discussed.

Chapter 6

Test generation

Using the variable values generator and variable assigner, branch/condition coverage cases can be generated, which form the basis for test case generation. This chapter discusses case generation, JBehave and Selenium test generation and execution. Section 6.1 discusses the generation of generic test cases. Section 6.2 discusses the JBehave story generation and execution. Section 6.3 discusses the Selenium test generation and execution.

6.1 Test case generation

For the generation of generic test cases, the case generator and generic case classes were developed. The case generator class provides the user with the option to generate a list of generic test cases given a list of generic elements and map of variable values. Each case contains two list of generic elements representing the preconditions and postconditions. These cases can then be used to generate different types of tests, e.g., JBehave stories and Selenium tests. The classes GenericCaseGenerator and GenericCase are shown in Figure 6.1 and 6.2.

```
<<Java Class>>
GenericCaseGenerator
xx.Generators
result: List<GenericCase>
currentCase: GenericCase
GenericCaseGenerator()
generateCases(List<GenericAbstractElement>,Map<GenericAbstractElement,List<GenericAbstractElement>>):List<GenericCase>
generateCases(List<GenericAbstractElement>,Map<GenericAbstractElement,List<GenericAbstractElement>>,GenericCase):List<GenericCase>
generateCase(List<GenericAbstractElement>,Map<GenericAbstractElement,List<GenericAbstractElement>>,GenericAbstractElement,GenericAbstractElement,GenericCase):void
replaceElementForList(List<GenericAbstractElement>,GenericAbstractElement,GenericAbstractElement):List<GenericAbstractElement>
addElements(List<GenericAbstractElement>):void
```

FIGURE 6.1: GenericCaseGenerator class

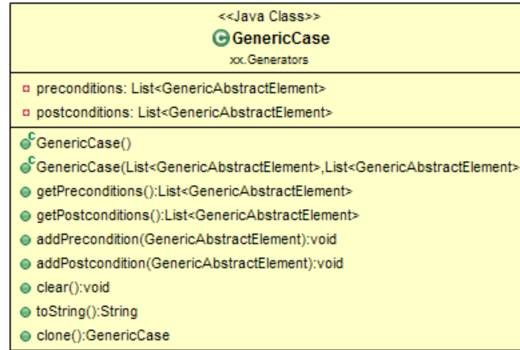

FIGURE 6.2: GenericCase class

The list of generic case objects is generated by traversing the list of variables and their values and generating a case for each value using the first element of the list until the variable list is empty. The case generator is given a clone of the generic elements, the map of variable values without the first element, the unassigned variable, its new value and the current case. This is the base case which is filled with the precondition stating that the unassigned variable gets the new value. The variable assigner is applied to the generic elements parameter, thereby assigning the new value to every instance of the unassigned value resulting in a list of generic elements were the variable is set to the new value. After that, two options remain.

If there are not more variables in the list of variable values, the postconditions are added to the case, the case is finished and added to the result. The postconditions consist of each variable or conditional (e.g., *if-then(-else)* constructs) and their current value. If the list of variables values is however not empty, the current case is used as base case for the new call to the generator with the replaced elements (elements where the variable is assigned the value) and the current map of variable values. This algorithm ensures every case is generated recursively and added to the result list. This process is shown as pseudo code in Algorithm 5 and 6.

Algorithm 5 Case generator main algorithm

1: **procedure** GENERATECASES(List elements, Map values, GenericCase case)
2: **if** values not *empty*() **then**
3: variable $\leftarrow$ values.$firstKey$()
4: **for all** value : values.*get*(variable) **do** GENERATECASE(elements.*clone*(), values.*remove*(variable), variable, value, case)
5: **end for**
6: **end if**
7: **end procedure**

Algorithm 6 Case generator sub algorithm

```
1: procedure GenerateCase(List elements, Map values, GenericAbstractVariable
   variable, GenericAbstractElement value, GenericCase case)
2:     currentCase ← new GenericCase
3:     replacedElements ← assignValue(elements, variable, value)      ▷ ValueAssigner
4:     for all precondition : case.preconditions() do
   currentCase.addPrecondition(precondition)
5:     end for
6:     currentCase.addPrecondition(variable)
7:     if values is empty() then
8:         for all element : replacedElements do
9:             if element instanceof GenericAbstractVariable || GenericConditional
   then
10:                currentCase.addPostcondition(element)
11:                result.add(currentCase.clone())
12:            end if
13:        end for
14:    else
15:        newGenerator ← new GenericCaseGenerator
16:        newGenerator.generateCases(replacedElements, values, currentCase)
17:    end if
18: end procedure
```

6.2 JBehave

JBehave is "a Java-based framework supporting Behavior-Driven Development (BDD), an evolution of Test-Driven Development (TDD) and Acceptance-Test Driven Development (ATDD)."[1] As mentioned in Section 2.4, by generating tests for this framework the tests can be read and edited by the domain experts. Although the BDD frameworks are not perfect according to Solis and Wang [13], we chose to generate JBehave stories, since this framework provides the most functionality as described in Section 2.4, is Java based and is being used in the organization of the case study.

6.2.1 Generation

For the generation of Precedence JBehave stories, the PrecedenceStoryGenerator class was developed. This class provides the user with the option to generate the String representation for a JBehave story to test Precedence artifacts given a list of generic test cases. The list of cases is traversed and each case is transformed to a scenario for the resulting story. The PrecedenceStoryGenerator class is shown in Figure 6.3.

[1]http://jbehave.org/introduction.html

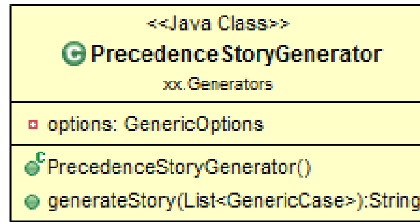

Figure 6.3: PrecedenceStoryGenerator class

For each precondition of type variable, a *When* case is added to the scenario stating which value is set for the variable. This value is computed using the *evaluateElement(GenericAbstractElement)* method of the GenericOptions class. For each postcondition of type variable or conditional, a *Then* case is added to the scenario stating that the variable or conditional must return the value specified in the postcondition. This value is again computed using the JavaScript *eval(String)* method by calling the *evaluateElement(GenericAbstractElement)* method of the GenericOptions class. This completes the run for the case. The generic test case to test transformation is shown as pseudo code in Algorithm 7.

Algorithm 7 Generic test generator algorithm

1: **procedure** GENERATETEST(List cases)
2: *String* result ← empty String
3: *Integer* caseCounter ← 1
4: **for all** case : cases **do**
5: *String* caseResult ← "case" + caseCounter
6: **for all** precondition : case.*preconditions*() **do**
7: caseResult.*add*(precondition.*toTestString*()) ▷ Convert preconditions
8: **end for**
9: **for all** postcondition : case.*postconditions*() **do**
10: caseResult.*add*(postcondition.*toTestString*()) ▷ Convert postconditions
11: **end for**
12: caseCounter.*increment*()
13: result.*add*(caseResult)
14: **end for**
15: **return** result
16: **end procedure**

Using the Precedence language example model from Chapter 5 as shown in Listing 6.1.

```
variable double A

variable bool B

variable String C

function String Foobar if((var A > 3) \&\& (var B != true) \&\& (var C == "test")
    , "true clause", "false clause")
```

Listing 6.1: Precedence example model code

This model is transformed to the JBehave story shown in Appendix D and one case is shown in Listing 6.2.

```
Scenario: case 1
When variable A gets value 4
When variable C gets value 'test'
When variable B gets value false

Then variable A should return value 4.0
Then variable B should return value false
Then variable C should return value test
Then function Foobar should return value true clause
```

LISTING 6.2: Case 1 of the JBehave story generated from the Precedence example model

6.2.2 Execution

To execute the generated JBehave stories, JBehave steps have to be created that map the story code to executable code. For example, the steps can map the story code to engine calls, so JUnit can be used to test the engine and the generated artifacts. Some example steps and mappings for the model described in Listing 6.1 are given in Listing 6.3.

```
@When("variable $variable gets value $value")
public void setVariableValue(String variable, String value){
    switch(parseObject(value)){
    case 0 : mt.setStringValue(variable, value);
    break;
    case 1 : mt.setDoubleValue(variable, Double.parseDouble(value));
    break;
    case 2 : mt.setBoolValue(variable, Boolean.parseBoolean(value));
    break;
    }
}

@Then("variable $variable should return value $value")
public void assertVariableValue(String variable, String value){
    switch(parseObject(value)){
    case 0 : mt.assertStringValue(variable, value);
    break;
    case 1 : mt.assertDoubleValue(variable, Double.parseDouble(value));
    break;
    case 2 : mt.assertBoolValue(variable, Boolean.parseBoolean(value));
    break;
    }
}
```

LISTING 6.3: Example steps defining the mapping from the story elements to Java code

The first step translates the *When* statements of setting the variable value to an engine call that parses the value and sets the value of the variable. The variable identification is again done using its name, thereby requiring unique variable names. The second steps maps the *Then* statements to assert calls of the engine, thereby checking whether the value defined in the story matches the value of the actual object. When there is an error in the engine, for example, the value of C is set when the value of B should be set, this error is detected by executing the test.

6.3 Selenium

As mentioned in Section 2.6, Selenium is a tool suite used for automated web testing and supports Java. Again we chose this framework since it supports Java and it is being used in the organization of the case study.

6.3.1 Generation

For the generation of Selenium tests, the PrecedenceSeleniumGenerator class was developed. This class provides the user with the option to generate the String representation of a Selenium test to test Precedence artifacts given a list of generic test cases, similar to the PrecedenceStoryGenerator class. The class also traverses the cases and generates a test method for each case. The class PrecedenceSeleniumGenerator is shown in Figure 6.4.

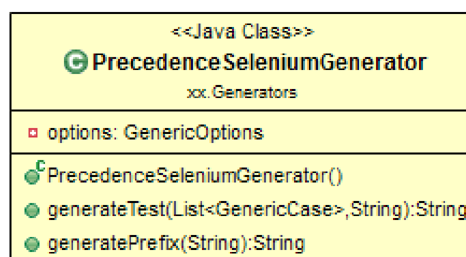

FIGURE 6.4: PrecedenceSeleniumGenerator class

Selenium tests are Java classes and must therefore validate to the Java syntax. The generator provides each test class with a method that initializes the drivers (Firefox or Chrome) and a *setUp()* method that sets up a test environment by, for example, loading the desired website.

For each test case, a *testCase()* method is added to the file. After that, each precondition is transformed in a valid Selenium command that sends a String, the calculated value, to a specific WebElement, the variable, which is located using an XPath expression.

Each postcondition is transformed into an *AssertEquals* statement that checks whether the postcondition value is equal to the value in the WebElement. The postcondition value is again calculated using the JavaScript *eval(String)* function by calling the *evaluateElement(GenericAbstractElement)* method of the GenericOptions class. This completes the run for the case.

Using the same example model as used in the JBehave example (Listing 6.1), the generated Selenium test is shown in Appendix E and one case is shown in Listing 6.4.

```
1  @Test
   public void testCase1() throws Exception {
3      driver.findElement(By.xpath("//*[@example-id[='A']/input[@example-id='value']
       ")).sendKeys("4.0");
       driver.findElement(By.xpath("//*[@example-id[='C']/input[@example-id='value']
       ")).sendKeys("test");
5      driver.findElement(By.xpath("//*[@example-id[='B']/input[@example-id='value']
       ")).sendKeys("false");

7      Assert.assertEquals("4.0", driver.findElement(By.xpath("//*[@example-id[='A
       ']//*[@example-id='value']")));
       Assert.assertEquals("false", driver.findElement(By.xpath("//*[@example-id[='B
       ']//*[@example-id='value']")));
9      Assert.assertEquals("test", driver.findElement(By.xpath("//*[@example-id[='C
       ']//*[@example-id='value']")));
       Assert.assertEquals("true clause", driver.findElement(By.xpath("//*[@example-
       id[='Foobar']//*[@example-id='value']")));
11 }
```

Listing 6.4: Case 1 of the Selenium test generated from the Precedence example model

6.3.2 Execution

These Selenium tests can be run as JUnit tests, if the required libraries are added to the project. By running these tests, inconsistencies between the online model and offline model can be detected. Since the example language is not extended with an online feature due to time constraints, the shown test case could not be executed, although it has correct Java and Selenium syntax. In Chapter 7 an executable generated test case is described.

Chapter 7

Case study

In this chapter we asses our test generation framework with a case study to show it supports multiple domain-specific languages (not only the Precedence language) and improves the quality of testing domain-specific artifacts or systems that use these artifacts. For the case study we use models developed using the Finan Financial Language (FFL). Section 7.1 introduces the Finan Financial Language. Section 7.2 discusses the generalization phase of the transformation chain. Section 7.3 discusses the generation phase of the transformation chain and Section 7.4 discusses the specification phase. Section 7.5 concludes the chapter.

7.1 Finan Financial Language

The organization in which the framework was tested is Topicus Finance[1]. Topicus has developed a domain-specific language, the Finan Financial Language (FFL), to construct Finan models. These models contain the business rules and logic for (online) applications used in the bank and accountant sector. The models consist of several variables and formulas with specific properties used, among others, for financial forecast calculations. When customers, for example, want to get a forecast of their future financial status, they enter their historic and/or current financial data into an Finan model. This data is then used for several calculations with a prediction of their future financial status as result.

FFL was developed based on a legacy language called FIN. The new language was developed using Xtext and Xtend. A Finan model consists of two types of files, namely the model files and the context files both instances of the FFL metamodel [26]. "The

[1] http://www.topicus.nl/finance/

model files define the economic model consistent of modules, variables, tuples, include statements, expressions and properties. The context files define the properties, annotations, functions, property blocks and property types. These context definitions are used to provide validation and documentation during model development." To initialize, manage and execute the model, Topicus developed the Finan Execution Service (FES) which can be interacted with by the front-end developers through an API developed by Topicus.

The Finan Financial Language architecture consists of two modular Xtext projects, namely the expression and financial project, both developed using an Xtext grammar. The expression project was developed for the processing of expressions, and has been kept modular for possible reuse. The financial project extends the expression project and defines the grammar for the development of Finan models. The financial grammar is used to generate the FFL metamodel. Topicus also developed a generator using Xtext and Xtend that generates Java artifacts of the model elements. These Java artifacts can be used by the FES and, therefore, by the customer applications.

Applying the theory of the DSL development phases described in Mernik et al. [27], the Finan Financial Language is formally designed using a grammar. Since the model is then transformed to Java code using an artifact generator written in Xtend and complete static analysis can be performed, it uses the application generator pattern for the implementation phase. The FFL language is developed using the legacy language (FIN) as inspiration by adapting the FIN supported expressions used in the model calculations one-to-one. Thereby a piggyback subpattern of the language exploitation pattern was used, i.e., the DSL partially uses an existing GPL or DSL. The structure of FFL is inspired by the hierarchical structure of XML so the same pattern is used here. Finally the language is influenced by the syntax and constructions of the programming languages Java and Pascal.

Using the models defined with the FFL language, the Java artifacts are generated. These are currently tested with manually developed TestNG, JBehave and Selenium tests. The goal is to show that Selenium tests as well as JBehave tests can be generated using Finan model information. These tests can then be executed to test the generated artifacts. Due to confidentiality reasons, the FFL grammar and Finan models cannot be completely shown in this research. To cope with this, only some syntactically correct examples of Finan models are discussed to give an impression of what the language allows and how the framework generates tests for this language. Next we will describe the three phases (generalization, generation and specification) of the framework for one example Finan model.

7.2 Generalization

The first phase of the framework's transformation chain is the generalization phase, where the Finan model is transformed to a generic common elements model so the test case generation algorithm can be applied to it. In Listing 7.1 we show a snippet of an Finan model to illustrate its syntax.

```
import FinanMath.*;
import BaseModel.*;
model TEST uses BaseModel
{
    variable A{
        datatype : number;
        frequency : document;
    }
    variable B{
        datatype : number;
        frequency : document;
    }
    variable V1 {
        formula : 10;
        datatype : number;
        frequency : column;
    }
    variable V2 {
        formula : V1[(A-1)];
        datatype : number;
        frequency : column;
    }
    variable V3 {
        formula : 15;
        datatype : number;
        frequency : column;
    }
    variable V4 {
        formula : V3[B];
        datatype : number;
        frequency : column;
    }
    variable V5 {
        formula: if((4=A) AND (B<>7), 0, 1);
        datatype : number;
        frequency : document;
    }
}
```

LISTING 7.1: Finan example model

In Listing 7.1, the variables A and B do not have a value assigned yet, since the formula property is not defined. Variable V1 is an array variable with default value 10. Users can then enter different values depending on an index using a web interface, but for testing

purposes we simulated this effect by hard-coding it in the FFL transformer component using Xtend code. The value of V1[3] is set to 30, V1[4] to 40 and V1[5] to 50. V2 is an array variable with a formula consisting of the value of V1 using index A-1, so its value depends on the value of A and the value array of V1. Variable V3 has default value 15, and the simulated values V3[7], V3[8] and V3[9] result in 40, 20 and 15, respectively. Variable V4 is the value of V3[B] and the formula of V5 is an expression, that checks whether the (unassigned) variable A equals 4 and (unassigned) variable B not equals (<>) 7, returning 1 when true and 0 when false.

The high-level elements in this Finan model are either single value variables or variables containing an array of values. The Finan model file only specifies model elements, for example, the variables and expressions, while the imported Finan context files (line 1 and 2 of Listing 7.1) define elements like functions, enumerations and constants. The variables in Listing 7.1 only have a limited number of properties (formula, data type and frequency), although there may be many more dependent on the requirements of the Finan model. The FFL variables are transformed to the generic elements GenericVariable or GenericArrayVariable depending on the value of the *frequency* property. A *document* frequency results in a GenericVariable element while a *column* frequency results in an GenericArrayVariable element. The column frequency is used when the variable can have multiple values depending on an index and can be used, for example, to define a variable that represents values in multiple years.

The FFLTransformer Xtend class transforms the Finan models to generic common elements models, while preserving all the important information used by the framework. This class was partly generated using the mapping DSL discusses in Section 4.2 and extended manually, so all domain-specific elements could be properly transformed. In Listing 7.2, 7.3, 7.4 and 7.5 snippets of the FFLTransformer class are shown, to illustrate the transformation from FFL elements to generic elements.

```
def GenericAbstractElement transform(String prefix, AssignmentStatement assignment){
    var valueItem = assignment.value.valueItems.head;
    if(valueItem.value instanceof Expression){
        var Expression expression = valueItem.value as Expression;
        switch(variableType.get(prefix)){
            case "column" : variables.put(prefix + "." + assignment.element.name, new GenericArrayVariable(prefix + "." + assignment.element.name, transform(expression), ARRAYLENGTH))
            case "document" : variables.put(prefix + "." + assignment.element.name, new GenericVariable(prefix + "." + assignment.element.name, transform(expression)))
        }
    return variables.get(prefix + "." + assignment.element.name);
}
```

Listing 7.2: FFL AssignmentStatement transformation

The code in Listing 7.2 transforms an AssignmentStatement element (used in the Finan model to define the assignment of a value to a variable) to a GenericVariable or GenericArrayVariable element, depending on the value of the frequency property. This property value was already extracted before transforming the domain elements by traversing all the AssignmentStatement elements. A prefix parameter of type String is used to define the context in which the assignment is done, thereby creating elements like *A.formula* where *A* defines the context that, in this case, refers to variable *A*. This prefix together with the name of the assignment element make up the identifier for the variable. The variable is saved in the variables list and returned to the generated Xtend file as explained in Section 4.2, so it can use the transformed elements to initiate the test generation by calling the methods of the GenericOptions class.

```
    def GenericAbstractElement transform(Variable variable, CellSpecifier cell){
2       var GenericAbstractVariable referenceVariable = transform(variable) as
    GenericAbstractVariable;
        var GenericAbstractElement referenceIndex = transform(cell.column);
4       return new GenericAbstractVariableIndexReference(referenceVariable,
    referenceIndex);
    }
```

Listing 7.3: FFL variable index reference transformation

The code in Listing 7.3 transforms an FFL Variable element with a CellSpecifier element, used as an index specifier, to a GenericAbstractVariableReference element. This is done by transforming the two elements and creating a new GenericAbstractVariableReference with the results. A GenericAbstractVariable element is used to keep the implementation independent of the variable type, thereby improving the modularity of the transformer class.

```
1   def GenericAbstractElement transform(FunctionDeclaration function,
    ArgumentList arguments){
        //Transform the arguments
3       var resultArguments = new ArrayList<GenericAbstractElement>();
        for(argument : arguments.arguments){
5           if(argument != null && argument.expression != null){
                resultArguments.add(transform(argument.expression));
7           }
            else{
9               throw new Exception("Null argument encountered in FFL transformer
    : " + function + " and arguments: " + arguments);
            }
11      }
        //Check for predefined supported function
13      if(functions.get(function.name) != null){
            return new GenericFunctionReference(functions.get(function.name),
    resultArguments);
15      }
        else{
17          switch(function.name){
```

```
                //Check for If function, by checking name
19              case "If" : return transform(function, resultArguments)
                default : throw new Exception("Unsupported parameterized Function
 Type: " + function)
21          }
        }
23  }
```

Listing 7.4: FFL function transformation

The code in Listing 7.4 transforms a FunctionDeclaration and ArgumentList element into a new GenericFunctionReferenceElement or GenericIfConditional. In the FFL language, *if-then(-else)* constructs are defined as Function elements, resulting in an extra check to correctly transform both elements. First the arguments of the ArgumentList element are transformed and inserted in a new ArrayList as shown in line 3 till 11 of Listing 7.4. The list of supported functions should be defined and filled with function elements before running the transformation process, using the function name as identification. If the list of functions contains the name of the FunctionDeclaration, a new reference to this function is returned with the transformed arguments (line 12 till 15 of Listing 7.4). If the function list does not contain the name of the FunctionDeclaration, the transformation tests if the function is an *if-then(-else)* construct by comparing the name of the function to 'If' as shown in line 17 and 18 of Listing 7.4. If this is true, the original FFL elements are forwarded to the GenericIfConditional transformation method. If both cases are false, an exception is thrown stating that an unsupported function is encountered.

```
1   def GenericIfConditional transformIf(FunctionDeclaration function,
    ArgumentList arguments){
        var resultArguments = new ArrayList<GenericAbstractElement>();
3       for(argument : arguments.arguments){
            if(argument != null && argument.expression != null){
5               resultArguments.add(transform(argument.expression));
            }
7           else{
                throw new Exception("Null argument encountered in FFL transformer
    : " + function + " and arguments: " + arguments);
9           }
        }
11      switch(resultArguments.size){
            case 2 : return new GenericIfConditional(function.name,
    resultArguments.get(0), resultArguments.get(1))
13          case 3 : return new GenericIfConditional(function.name,
    resultArguments.get(0), resultArguments.get(1), resultArguments.get(2))
            default : throw new Exception("Unsupported number of parameters in
    IfConditional " + function)
15      }
    }
```

Listing 7.5: FFL conditional transformation

The code in Listing 7.5 transforms a FunctionDeclaration and ArgumentList element to a GenericIfConditional. We chose to not use the transformed arguments, since this would make the method dependent on the code in Listing 7.4. Another method was implemented with a list of transformed arguments as parameter, which is not shown here. So again, the arguments are transformed and inserted in a new ArrayList. After that, the size of this list is checked. If the size equals 2, a GenericIfConditional is returned with a GenericNull for the else-clause, and if the size equals 3 a GenericIfConditional is returned with all assigned elements. If another value is found for the size, a Java exception is thrown stating that the number of arguments is not supported.

By applying the FFLTransformer to the elements described in Listing 7.1, the following generic elements were generated, where the numeric values are GenericNumber elements with a double object representing the value, but depicted here as numbers to improve readability:

- Variable A.formula with default value GenericNull
- Variable B.formula with default value GenericNull
- ArrayVariable V1.formula with default value 10, V1[3] = 30, V1[4] = 40 and V1[5] = 50.
- ArrayVariable V2.formula with default value VariableReference referencing ArrayVariable V1.formula with index BinaryExpression(Variable A.formula - 1).
- ArrayVariable V3.formula with default value 15, V3[7] = 40, V3[8] = 20 and V3[9] = 15.
- ArrayVariable V4.formula with default value VariableReference referencing ArrayVariable V3.formula with index Variable B.formula
- Variable V5.formula with default value IfConditional with condition = (BinaryExpression (BinaryExpression 4 == ArrayVariable A.formula) && (BinaryExpression ArrayVariable B.formula != 7)) returning 1 when true, and 0 when false.

7.3 Generation

Now that the domain-specific elements are transformed to generic elements, the generation phase of the transformation chain begins. In this phase the values for the unassigned variables are generated.

Two elements in the example of Section 7.2 do not have a value assigned to them (value equals GenericNull), which are variable A.formula and B.formula (hereafter called A and B). For these two elements, values have to be generated. As explained in Section 5.2, values are generated when unassigned variables are used in binary expressions. These binary expressions are found in variable V5.formula, namely 4 == A and B != 7, resulting in the generation of four numeric values. For A, the values 4 and 5 (4 incremented with 1) are generated, so both the true clause (4=4) and false clause (5=4) are evaluated. For B, the values 8 and 7 are generated so again the true clause (8!=7) and false clause (7!=7) are evaluated. These values are saved in the context of variable V5.formula, so that other elements can also use A or B without interfering with the values A and B of variable V5.formula.

Using the generated values, test cases can be generated to reach branch/condition coverage. Both variables are assigned two values, so a total of four generic test cases are generated by the case generator described in Section 6.1.

1. **Case 1:** A = 4, B = 8
2. **Case 2:** A = 4, B = 7
3. **Case 3:** A = 5, B = 8
4. **Case 4:** A = 5, B = 7

7.4 Specification

During the specification phase of the transformation chain, the generated generic test cases are transformed to executable tests for the Java artifacts generated using the Finan model and the artifact generator developed by Topicus. Executable JBehave stories and Selenium tests are generated for the four cases described in Section 7.3. For both types of tests (JBehave stories and Selenium tests) a generator was developed that transforms the generic test cases to an executable test.

7.4.1 JBehave

Since users define their own JBehave steps that convert the story code to executable code, the FFL JBehave story generator is similar to the generic JBehave story generator. However, the stories result in FES methods being executing by using the steps made available by Topicus.

Listing 7.6 shows the FFL JBehave generator code used to transform generic test case preconditions to JBehave *When* clauses.

```
public String variable2Precondition(GenericAbstractVariable variable){
    String[] variableNameElements = variable.getName().split("\\.");
    String variableName = variableNameElements[0];
    String propertyName = variableNameElements[1];

    Object value = options.evaluateElement(variable.getValue());
    if(propertyName.equals("formula")){
        return "When variable " + variableName + " gets value " + value;
    }
    else{
        System.err.println("Encountered unsupported property name in
    FFLVariable2Precondition");
        return null;
    }
}
```

Listing 7.6: FFL JBehave precondition transformation

Since the FES uses model variable names as identification (so A and B), the model variable names have to be extracted from the saved variable names (A.formula and B.formula). To achieve this, the variables names are parsed using the Java *split(String)* function with "." as the delimiter. The value of the variable is again determined using the JavaScript engine. As a proof of concept only the value of the *formula* property can be set, yet more property support can be added by Topicus in the future due to the modular setup. Combining the name and the value, a JBehave *When* case is added to the result.

For the conversion of postconditions, the code in Listing 7.7 is used.

```
public String variable2Postcondition(GenericAbstractVariable variable){
    String[] variableNameElements = variable.getName().split("\\.");
    String variableName = variableNameElements[0];
    String propertyName = variableNameElements[1];
    //Base value is N/A
    Object value = "N/A";

    //GenericNull should be N/A
    if(variable.getValue() instanceof GenericNull){
        value = "N/A";
    }
```

```
12
        //Check if the value is a variable index reference
14      else if(variable.getValue() instanceof GenericAbstractVariableIndexReference)
        {
            GenericAbstractVariableIndexReference reference = (
        GenericAbstractVariableIndexReference) variable.getValue();
16          //If the index is a variable
            if(reference.getIndex() instanceof GenericAbstractVariable){
18              GenericAbstractVariable indexVar = (GenericAbstractVariable)
        reference.getIndex();
                //If the index variable is null
20              if(indexVar.getValue() instanceof GenericNull){
                    return "Then variable " + variableName + " should return value N/
        A";
22              }
            }
24          //Calculate the index
            double index = (Double) options.evaluateElement(reference.getIndex());
26          //Calculate the value using the index
            value = options.evaluateElement(reference.getVariable().getValue((int)
        index));
28          //Check if the return value is null
            if(value == null){
30              return "Then variable " + variableName + " should return value N/A";
            }
32          //Return the result
            else{
34              return "Then column variable " + variableName + " should return value
         " + value + " for column " + (int)index;
            }
36      }
        //Check if the value is a variable reference
38      else if(variable.getValue() instanceof GenericAbstractVariableReference){
            GenericAbstractVariableReference reference = (
        GenericAbstractVariableReference) variable.getValue();
40          //If the reference variable is null
            if(reference.getVariable().getValue() instanceof GenericNull){
42              value = "N/A";
            }
44          //If the reference is a generic variable, get the value
            else if(reference.getVariable() instanceof GenericVariable){
46              value = options.evaluateElement(reference.getVariable().getValue());
            }
48          //Else return N/A, since no index is defined
            else{
50              value = "N/A";
            }
52      }
        //Else calculate the value by evaluating the variable
54      else{
            //If the reference is an array variable and no index is defined
56          if(variable instanceof GenericArrayVariable){
                value = "N/A";
58          }
            else{
```

```
60              value = options.evaluateElement(variable.getValue());
            }
62      }

64      //Base result String on property
        if(propertyName.equals("formula")){
66          return "Then variable " + variableName + " should return value " + value;
        }
68      else if(propertyName.equals("title")){
            return "Then variable " + variableName + " has title " + value;
70      }
        else if(propertyName.equals("inputRequired")){
72          if("false".equals(value.toString())){
                return "Then variable " + variableName + " is not required";
74          }
            else return "Then variable " + variableName + " is required";
76      }
        else if(propertyName.equals("locked")){
78          if("false".equals(value.toString())){
                return "Then variable " + variableName + " is unlocked";
80          }
            else return "Then variable " + variableName + " is locked";
82      }
        else if(propertyName.equals("visible")){
84          if("false".equals(value.toString())){
                return "Then variable " + variableName + " is invisible";
86          }
            else return "Then variable " + variableName + " is visible";
88      }
        else{
90          System.err.println("Encountered unsupported property name in
        FFLVariable2Postcondition");
            return null;
92      }
    }
```

Listing 7.7: FFL JBehave postcondition transformation

The same method is used to obtain the test data. Since the variables in FFL can have different properties, different cases are implemented to switch on the property resulting in different JBehave *Then* cases as shown in line 64 till 88 of Listing 7.7. When a variable does not have a value, the FES expects the string "N/A".

The transformation of Case 1, described earlier, results in the JBehave case shown in Listing 7.8.

```
1 Scenario: case 1
  When variable A gets value 4.0
3 When variable B gets value 8.0

5 Then variable A should return value 4.0
  Then variable B should return value 8.0
7 Then variable V1 should return value 10.0
```

```
Then column variable V2 should return value 30.0 for column 3
Then variable V3 should return value 15.0
Then column variable V4 should return value 20.0 for column 8
Then variable V5 should return value 1.0
```

LISTING 7.8: FFL JBehave case 1 result

7.4.2 Selenium

As Selenium tests depend on the structure and elements of the website that is tested, the FFL Selenium generator contains two hard-coded functions to achieve an executable test. These functions, used to setup and tear down the Topicus website, are given in Appendix F. The FFL transformation code is however similar to the generic Selenium test generator code.

Listing 7.9 shows the FFL Selenium generator code used to transform generic test case preconditions to Selenium code for setting a variable value.

```
public String Variable2Precondition(GenericAbstractVariable variable){
    String[] variableNameElements = variable.getName().split("\\.");
    String variableName = variableNameElements[0];
    String propertyName = variableNameElements[1];

    Object value = options.evaluateElement(variable.getValue());
    if(propertyName.equals("formula")){
    return "\tdriver.findElement(By.xpath(\"//*[@selenium-id='" + variableName +
    "']//*[@selenium-id='container']/input[@selenium-id='input']\")).sendKeys(\""
     + value + "\");" + "\n"
            + "\tdriver.findElement(By.xpath(\"//*[@selenium-id='" + variableName
     + "']//*[@selenium-id='container']/input[@selenium-id='input']\")).sendKeys(
    Keys.ENTER);" + "\n"
            + "\tThread.sleep(3000);\n"
            + "\tdriver.findElement(By.xpath(\"//*[@selenium-id='Q_Map05']\")).
    click();\n"
            + "\tThread.sleep(3000);\n";
    }
    else{
        System.err.println("Encountered unsupported property name in
    FFLVariable2Precondition");
        return null;
    }
}
```

LISTING 7.9: FFL Selenium precondition transformation

The required data is obtained in the same way as in the JBehave generator (line 1 till 6 of Listing 7.9), yet the generated String representation of the precondition differs to comply to the Topicus website and Selenium requirements. Topicus has developed a Selenium plugin that lets the front-end developers index web elements using the name of

the model variable. Using this plugin, the web element representing the model variable can be located with an XPath expression that checks whether the selenium-id of the web element is equal to the name of the model variable. Having located the correct web element, a value can be assigned to it by traversing the tree nodes of this element until an input element is reached. Using this input web element, the method *sendKeys(String)* can be used to assign a value to it, thereby assigning a value to the model variable web element. This is followed by sending a *Keys.ENTER* action to the same element to confirm the value entry. The code of assigning a value to a variable input element is shown in line 8 and 9 of Listing 7.9.

Since the structure of the web elements is website specific, part of the String representation of setting the value of a variable web element is hard-coded. The variable names and generated values are variable as can be seen in Listing 7.9. After the value is entered, the model should recalculate its expressions and formulas using the newly entered values. This is done by clicking a Topicus website navigation web element (called 'Q_Map05'), thereby initiating a page refresh, as shown in line 11 of Listing 7.9. Before and after this action a sleep method is added to the test to avoid delay errors.

For the conversion of postconditions, the code snippet in Listing 7.10 is used.

```
public String variable2Postcondition(GenericAbstractVariable variable){
2       String[] variableNameElements = variable.getName().split("\\.");
        String variableName = variableNameElements[0];
4       String propertyName = variableNameElements[1];
        //Base value is null
6       Object value = null;

8       //GenericNull should be null
        if(variable.getValue() instanceof GenericNull){
10          value = null;
        }
12
        //Check if the value is a variable index reference
14      else if(variable.getValue() instanceof GenericAbstractVariableIndexReference)
        {
            GenericAbstractVariableIndexReference reference = (
        GenericAbstractVariableIndexReference) variable.getValue();
16          //If the index is a variable
            if(reference.getIndex() instanceof GenericAbstractVariable){
18              GenericAbstractVariable indexVar = (GenericAbstractVariable)
        reference.getIndex();
                //If the index variable is null
20              if(indexVar.getValue() instanceof GenericNull){
                    return "Encountered index with value Null";
22              }
            }
24          //Calculate the index
            double index = (Double) options.evaluateElement(reference.getIndex());
26          //Calculate the value using the index
```

```
            value = options.evaluateElement(reference.getVariable().getValue((int)
        index));
28          //Check if the return value is null
            if(value == null){
30              return null;
            }
32      }
        //Check if value is a variable reference
34      else if(variable.getValue() instanceof GenericAbstractVariableReference){
            GenericAbstractVariableReference reference = (
        GenericAbstractVariableReference) variable.getValue();
36          //If the reference variable is null
            if(reference.getVariable().getValue() instanceof GenericNull){
38              value = null;
            }
40          //If the reference is a generic variable get the value
            else if(reference.getVariable() instanceof GenericVariable){
42              value = options.evaluateElement(reference.getVariable().getValue());
            }
44          //Else return null, since no index is defined
            else{
46              value = null;
            }
48      }
        //Else calculate the value by evaluating the variable
50      else{
            //If the reference is an array variable and no index is defined
52          if(variable instanceof GenericArrayVariable){
                value = null;
54          }
            else{
56              value = options.evaluateElement(variable.getValue());
            }
58      }
        //Clean output file for now, Proof of Concept check
60      if(value == null || !variableName.equals("OnroerendGoed_tpVerschil")){
            return "";
62      }

64      //Base result String on property
        if(propertyName.equals("formula")){
66          double doubleValue = (Double)value;
            return "\tAssert.assertEquals(\"" + (int)doubleValue + "\", driver.
        findElement(By.xpath(\"//*[@selenium-id='" + variableName + "']//*[@selenium-
        id='OtsyFirst']//*[@selenium-id='container']//*[@selenium-id='input']\")).
        getText());" + "\n";
68      }
        else{
70          System.err.println("Encountered unsupported property name in
        FFLVariable2Postcondition");
            return null;
72      }
    }
```

LISTING 7.10: FFL Selenium postcondition transformation

Again the same method as in the JBehave generator is used to obtain the variable name and value. For this generator only the *formula* property is transformed, since this element should be checked on the Topicus website. The checking of the value is done using the *Assert.assertEquals(String, String)* method of Selenium as shown in line 67 of Listing 7.10.

The transformation of Case 1, described earlier, results in the Selenium test case shown in Listing 7.11.

```
@Test
public void testCase1() throws Exception {
    driver.findElement(By.xpath("//*[@selenium-id='A']//*[@selenium-id='container
    ']/input[@selenium-id='input']")).sendKeys("4.0");
    driver.findElement(By.xpath("//*[@selenium-id='A']//*[@selenium-id='container
    ']/input[@selenium-id='input']")).sendKeys(Keys.ENTER);
    Thread.sleep(3000);
    driver.findElement(By.xpath("//*[@selenium-id='Q_Map05']")).click();
    Thread.sleep(3000);
    driver.findElement(By.xpath("//*[@selenium-id='B']//*[@selenium-id='container
    ']/input[@selenium-id='input']")).sendKeys("8.0");
    driver.findElement(By.xpath("//*[@selenium-id='B']//*[@selenium-id='container
    ']/input[@selenium-id='input']")).sendKeys(Keys.ENTER);
    Thread.sleep(3000);
    driver.findElement(By.xpath("//*[@selenium-id='Q_Map05']")).click();
    Thread.sleep(3000);

    Assert.assertEquals("4", driver.findElement(By.xpath("//*[@selenium-id='A
    ']//*[@selenium-id='OtsyFirst']//*[@selenium-id='container']//*[@selenium-id
    ='input']")).getText());
    Assert.assertEquals("8", driver.findElement(By.xpath("//*[@selenium-id='B
    ']//*[@selenium-id='OtsyFirst']//*[@selenium-id='container']//*[@selenium-id
    ='input']")).getText());
    Assert.assertEquals("10", driver.findElement(By.xpath("//*[@selenium-id='V1
    ']//*[@selenium-id='OtsyFirst']//*[@selenium-id='container']//*[@selenium-id
    ='input']")).getText());
    Assert.assertEquals("30", driver.findElement(By.xpath("//*[@selenium-id='V2
    ']//*[@selenium-id='OtsyFirst']//*[@selenium-id='container']//*[@selenium-id
    ='input']")).getText());
    Assert.assertEquals("15", driver.findElement(By.xpath("//*[@selenium-id='V3
    ']//*[@selenium-id='OtsyFirst']//*[@selenium-id='container']//*[@selenium-id
    ='input']")).getText());
    Assert.assertEquals("20", driver.findElement(By.xpath("//*[@selenium-id='V4
    ']//*[@selenium-id='OtsyFirst']//*[@selenium-id='container']//*[@selenium-id
    ='input']")).getText());
    Assert.assertEquals("1", driver.findElement(By.xpath("//*[@selenium-id='V5
    ']//*[@selenium-id='OtsyFirst']//*[@selenium-id='container']//*[@selenium-id
    ='input']")).getText());
}
```

Listing 7.11: FFL Selenium case 1 result

7.5 Conclusion

This case demonstrates that the framework is applicable to a DSL already in use by an organization. Framework users have to design and implement a transformer that maps the metamodel to the common elements metamodel and a number of test generators, dependent on the number of types of tests to be supported. We developed the Finan Financial Language transformers in collaboration with company stakeholders and (future) framework users, and these can be modified and extended by Topicus when necessary. Using the developed transformers, tests could be generated for a multitude of models without modifying the framework elements.

Chapter 8

Final remarks

8.1 Conclusions

In this thesis, we described the design and implementation of a domain-specific testing framework for the generation of tests using domain-specific models. These tests can be used to verify correctness of the artifacts generated using the models, or systems that use these artifacts. As a result, artifact generation errors and bugs in the systems that use these artifacts could be detected. By generating the test instead of manually developing them, development time is reduce while usability is improved.

We investigated and explained several testing techniques and their application to domain-specific languages. We chose to implement branch/condition coverage, since the source/-model code is available and it is a strong white-box testing technique. We also applied model-based testing techniques by using a metamodel to improve maintainability and flexibility.

To generate tests for a multitude of domain-specific languages, a generic metamodel called the *common elements metamodel* was developed. This metamodel was set-up so that it supports a large number of common programming language features. By making use of this generic metamodel, the generation of tests consists of three phases: generalization, generation and specification. The test case generation algorithm is based on the elements of the generic metamodel, resulting in a generation approach that is independent of the domain-specific language.

Using this approach does however require the user to define and implement a transformation from the domain metamodel to the generic metamodel and from the generated generic test cases to domain-specific tests, the generalization and specification phase, respectively. For the transformation of common elements, a mapping DSL was developed, thereby reducing the workload of the framework user.

We analyzed which common elements should be supported, by using the Common Language Specification. A selected subset of this specification was determined for the development of the generic metamodel. The generic metamodel was developed in Ecore and implemented using Java. The other framework components were also implemented using Java. Their functions can however be called from domain-specific classes, for example, the Xtend classes generated by the Xtext framework.

In this thesis, we used the Topicus Finance case study, to show that the framework can be applied to actively used domain-specific languages and models. We also showed that the framework supports a multitude of languages and resulting test types.

In future work, reusability can be further tested by applying the framework to other domain-specific languages. The correctness of the transformation and generation could be proved using formal verification. To make the framework applicable to other (more diverse) languages, it has several areas that should be extended as discussed in Section 8.3. Taken into account the information gained during the literature study and framework development, we advise that the framework should be kept modular, extensible and small. Since domain-specific languages are diverse, the framework cannot pose a large number of restrictions on the domain-specific languages without restricting the amount of supported languages. The generic framework code is published at `https://github.com/ratenbuuren/DSLTestingFramework`.

8.2 Research answers

The research questions defined for this thesis have been answered:

RQ1. Which testing techniques are available and what is their coverage?

In Chapter 2 we analyzed several testing techniques and described as well as compared their coverage. There are several testing techniques, for example, black-box testing, white-box testing, model-based testing and automated web testing, each with their own coverage, benefits and drawbacks.

RQ2. How can these testing techniques be applied to domain-specific languages?

After identifying the available testing techniques, research was performed on how these techniques could be applied to DSLs. The focus of this analysis was on generating tests for the artifacts developed using the domain-specific models. Since the generation was based on the source code, white-box testing techniques could be applied. Model-based testing techniques are also used to create a generic metamodel and generation algorithm thereby improving maintainability and flexibility.

RQ3. How to deal with different language constructs and syntax?

In Chapter 3 we established a selection of common language elements to support. The common elements were implemented into a generic metamodel described in Section 4.1. Using the metamodel for the generation algorithms, the generation of test cases is independent of the language, its constructs and syntax. By also keeping the generated test cases generic, different types of tests can be generated using the same model.

RQ4. How to assess the reusability and verify the quality of the testing framework?

We assessed the reusability of the framework by generating two types of tests, JBehave stories and Selenium tests, for two different languages, a newly designed expression language called Precedence and the Finan Financial Language. This shows that the test framework is usable for different languages and testing at multiple testing levels.

We verified the quality informally by introducing several errors in the artifact generator and systems that use these artifacts, and all were detected by the generated tests. The usability and effectiveness were verified by questioning the stakeholders in the case study company. The stakeholders stated that the framework would be helpful in their organization, since it significantly reduced testing efforts. However, since only a limited number of elements is supported, the framework should be extended for future purposes.

8.3 Future work

During the development of our framework, we identified several research opportunities for future work:

- **Extend common elements metamodel**

  The generic metamodel used during the development of the framework is a subset of the Common Language Specification and could be extended to support additional (common) elements. The common elements metamodel also contains mock functions. A library containing frequently used functions, for example, *min()*, *max()* and *abs()*, could be developed so that framework users can reuse these functions.

- **Extend mapping DSL**

  The DSL used for the generation of the domain model to generic model transformer currently support a subset of the common elements. This language can be extended to support more elements, thereby reducing the effort and time it takes to develop the transformer. Support for frequent domain-specific elements could also be added.

- **Extend value generation functionality**

  The developed value generator supports generation of GenericBooleans, GenericStrings, GenericNumbers and GenericUnaryExpressions. If the subexpression of the binary expression is of an unsupported type, the value is calculated by evaluating the element using the JavaScript engine. This could cause exceptions, for example, when the expression contains an unassigned value. The value generator support should therefore be extended to support more types.

  One possible way is to rewrite the generation algorithm so that it works recursively, thereby being independent of the depth of the unassigned variables. This could however significantly increase computation time, which could be solved by letting the user specify a maximum depth.

- **Extend test generators**

  The generic test cases have currently only been transformed to JBehave stories and Selenium tests. Additional test types could be supported by the development of new transformers. These might however be dependent on the domain.

- **Extend test functionality**

  The test functionality could be extended to support range/code coverage. Next to that, the test feedback could be converted back to feedback on the model. This would enable the domain experts to evaluate the test feedback and modify the model without the need of a programmer or tester.

- **Versioning**

  When a model is modified and saved, the current algorithm starts the generation process from scratch. Versioning could be added so that only the modified elements are reevaluated, thereby improving computation time, efficiency and giving the user a clear overview of the modification effects.

Appendix A

Expression mapping grammar

```
grammar xx.ExpressionMapping with org.eclipse.xtext.common.Terminals

generate expressionMapping "http://www.ExpressionMapping.xx"

/**
 * A grammar that describes the mapping from a domain-specific model to the
    generic model and generates a Xtend transformer class that performs the
    mapping
 * Consists of:
 * Package statements (the package of the generated Transformer)
 * Import statements (imports used by the generated Transformer)
 * Grammar (the name of the grammar, can be self defined)
 * MainType (Generalizes type that is first parsed, like Expression interface)
 * Types (Supported types by the grammar that should be converted)
 * Mappings (Defines the mappings from Types to Generic Types)
 */

Model:
    package=PackageStatement
    imports+=ImportStatement*
    grammardef=Grammar
    maintype=MainType
    types+=Type*
    mappings+=Mapping*;

//Defines the grammar name

Grammar:
    'Grammar' name=ID
;

//Defines the package

PackageStatement:
    'Package' name=Fqn ';'
;

//Defines an import statement
```

```
37
   ImportStatement:
39     'import' importedNamespace=FqnWithWildCard
   ;
41
   //Defines the general expression type
43
   MainType:
45     'MainType' name=Fqn
   ;
47
   //Defines the subclasses of the MainType
49
   Type:
51     'Type' name=Fqn
   ;
53
   //Mapping can map binary, unary, parenthesized expressions and literals
55
   Mapping:
57     'MAP Variable Declaration' varDeclMapping=VarDeclMapping ';'
       | 'MAP Binary Expression' binaryMapping=BinaryMapping ';'
59     | 'MAP Unary Expression' unaryMapping=UnaryMapping ';'
       | 'MAP Parenthesized Expression' ParenthesizedMapping ';'
61     | 'MAP Literal' literalMapping=LiteralMapping ';'
       ;
63
   //Maps a variable declaration, expects a name, and method to get the right hand
       assignment
65
   VarDeclMapping:
67     'of' 'type' varDeclType=[Type] 'and' 'nameMethod' nameMethod=Fqn 'and' '
       valueMethod' valueMethod=Fqn
   ;
69
   //Maps a binary expression, expects a Type, left child, right child, operator
       Method and operator representation.
71 //Converts it to the supported BinaryTargets
   BinaryMapping:
73     parameterType=[Type] 'with' binaryLeft=BinaryLeft binaryOperator=OperatorDef
       binaryRight=BinaryRight 'to' 'type' binaryTarget=BinaryTarget
   ;
75
   //Maps a unary expression, expects a Type, left child, right child, operator
       Method and operator representation.
77 //Converts it to the supported UnaryTargets
   UnaryMapping:
79     parameterType=[Type] 'with' unaryExp=UnaryExpression unaryOperator=
       OperatorDef 'to' 'type' unaryTarget=UnaryTarget
   ;
81
   //Maps a parenthesized expression, expects a inner expression method
83 //Converts it to the supported ParenthesizedTargets

85 ParenthesizedMapping:
```

```
        parameterType=[Type] 'with' innerExp=InnerExpression 'to' 'type'
        parenthesizedTarget=ParenthesizedTarget
87  ;

89  //Maps a literal expression, expects a Type and operator Name. Converts it to the
         supported LiteralTargets
    LiteralMapping:
91      parameterType=[Type] 'to' 'type' literalTarget=LiteralTarget 'using' '
        operation' operationName=Operator
    ;
93
    //Defines the left child of a binary expression
95
    BinaryLeft:
97      'leftChildMethod' '=' value=ID
    ;
99
    //Defines the right child of a binary expression
101
    BinaryRight:
103     'rightChildMethod' '=' value=ID
    ;
105
    //Defines a binary or unary operator
107
    OperatorDef:
109     'operatorMethod' operatorMethod=Fqn 'returns' operator=Operator
    ;
111
    //Defines a expression child of a unary expression
113
    UnaryExpression:
115     'unaryExpressionMethod' '=' value=ID
    ;
117
    //Defines a expression child of a parenthesized expression
119
    InnerExpression:
121     'innerExpressionMethod' '=' value=ID
    ;
123
    // Supported Binary targets
125
    enum BinaryTarget returns BinaryTarget:
127     MULTIPLICATION = 'Multiplication'
        | DIVISION = 'Division'
129     | ADDITION = 'Addition'
        | SUBTRACTION = 'Subtraction'
131     | EQUALS = 'Equals'
        | NOTEQUALS = 'Not Equals'
133     | GT = 'Greater Than'
        | GTE = 'Greater Than Equals'
135     | LT = 'Less Than'
        | LTE = 'Less Than Equals'
137     | MOD = 'Modulo'
```

```
    | OR = 'Logic OR'
139     | AND = 'Logic AND'
;
141
// Supported Unary targets
143
enum UnaryTarget returns UnaryTarget:
145     NEGATE = 'Negate'
    | POSITIVE = 'Positive'
147     | NEGATIVE = 'Negative'
;
149
// Supported Parenthesized targets
151
enum ParenthesizedTarget returns ParenthesizedTarget:
153     GenericParenthesizedExpression = 'Parenthesized'
;
155
// Supported Literal targets
157
enum LiteralTarget:
159     GenericNumber='NUMBER'
    | GenericString='STRING'
161     | GenericBoolean = 'BOOLEAN'
;
163
FqnWithWildCard:
165     Fqn ('.*')?
;
167
Fqn:
169     ID ('.' ID)*
;
171
Operator:
173     Fqn
    | STRING
175 ;
```

Listing A.1: Expression mapping DSL grammar

Appendix B

Precedence grammar

```
1  grammar org.xtext.operator.Precedence with org.eclipse.xtext.common.Terminals
     hidden(WS, ML_COMMENT, SL_COMMENT)
3
   generate precedence "http://www.xtext.org/operator/Precedence"
5
   import "http://www.eclipse.org/emf/2002/Ecore" as ecore
7
   ExpressionsModel:
9      elements += AbstractElement*
       ;
11
   AbstractElement:
13     Variable | ListVariable | Expression | Function | MockDef | MockEntry |
       Container
       ;
15
   Variable:
17     'variable' type=ReturnType name=ID ('=' expression=AbstractElement)?
       ;
19
   ListVariable:
21     'listvariable' type=ReturnType name = ID ('=' expression=AbstractElement)?
   ;
23
   Function:
25     'function' type=ReturnType name=ID functionType=FunctionType '(' (parameters
       +=AbstractElement (',' parameters+=AbstractElement)*)? ')'
   ;
27
   Container:
29     'container' name=ID '{' ( elements+=AbstractElement (',' elements+=
       AbstractElement)*)? '}'
   ;
31
   MockDef:
33     'mocking' 'function' type=ReturnType name=ID
   ;
35
```

```
    MockEntry:
37      'mock' type=[MockDef] 'for' 'input' '(' (parameters+=AbstractElement (','
        parameters+=AbstractElement)*)? ')' 'returns' result=AbstractElement
    ;
39
    enum ReturnType:
41      DOUBLE = "double"
        | BOOL = "bool"
43      | STRING = "String"
    ;
45
    enum FunctionType:
47      ABS='abs'
        | ROUND='round'
49      | CEIL='ceil'
        | FLOOR='floor'
51      | MAX='max'
        | MIN='min'
53      | POW="pow"
        | SQRT="sqrt"
55      | IF='if'
    ;
57
    Expression:
59      Or
        ;
61
    enum OrOperator returns BinaryOperator:
63      OR='||'
    ;
65
    Or returns Expression:
67      And ({BinaryExpression.left=current} op=OrOperator right=And)*
        ;
69
    enum AndOperator returns BinaryOperator:
71      AND='&&'
    ;
73
    And returns Expression:
75      Equality({BinaryExpression.left=current} op=AndOperator right=Equality)*
        ;
77
    enum EqualityOperator returns BinaryOperator:
79      EQUAL='==' | NOTEQUAL='<>'
    ;
81
    Equality returns Expression:
83      Comparison (
            {BinaryExpression.left=current} op=EqualityOperator
85          right=Comparison
            )*
87      ;

89  enum ComparisonOperator returns BinaryOperator:
```

```
    GREATER='>' | SMALLER='<' | GREATEREQUAL='>=' | SMALLEREQUAL='<='
;

Comparison returns Expression:
    MulOrDiv (
        {BinaryExpression.left=current} op=ComparisonOperator
        right=MulOrDiv
    )*
;

enum MulOrDivOperator returns BinaryOperator:
    MULTIPLY='*' | DIVIDE='/'
;

MulOrDiv returns Expression:
    PlusOrMinus (
        {BinaryExpression.left=current} op=MulOrDivOperator
        right=PlusOrMinus
    )*
;

enum PlusOrMinusOperator returns BinaryOperator:
    PLUS='+' | MINUS='-'
;

PlusOrMinus returns Expression:
    UnaryExpression (
        {BinaryExpression.left=current} op=PlusOrMinusOperator
        right=UnaryExpression
    )*
;

enum UnaryOperator:
    NEGATE='!' | POSITIVE='@' | NEGATIVE='#'
;

UnaryExpression returns Expression:
    ({UnaryExpression}
        op=UnaryOperator expression=PrimaryExpression
    )
    | PrimaryExpression
;

PrimaryExpression returns Expression:
    ParenthesizedExpression
    | Atomic
;

ParenthesizedExpression returns Expression:
    {ParenthesizedExpression}
    '(' innerExpression=AbstractElement ')'
;

Atomic returns Expression:
    {DoubleConstant} value=Double
```

```
145     | {BoolConstant} value=Boolean
        | {StringConstant} value=STRING
147     | {MockRef} 'func' mockFunction=[MockDef] 'using' '(' parameters+=
        AbstractElement (',' parameters+=AbstractElement)* ')'
        | {VariableRef} 'var' variable=[Variable]
149     | {ListVariableRef} 'listvar' listvariable=[ListVariable] ('['cell=
        AbstractElement']')?
        | {ContainerRef} 'cont' container=[Container] ('['cell=AbstractElement']')?
151 ;

153 Double returns ecore::EDouble:
        INT('.'INT)?
155 ;

157 Boolean returns ecore::EBoolean:
        'true'
159     | 'false'
    ;
```

Listing B.1: Precedence language grammar

Appendix C

Precedence transformer

```
/**
 * Class that transforms Precedence elements to generic framework elements
 */

public class PrecedenceTransformer {

    Map<String, GenericAbstractVariable> variables = new HashMap<String,
    GenericAbstractVariable>();
    Map<String, GenericContainer> containers = new HashMap<String,
    GenericContainer>();
    Map<String, GenericFunction> functions = new HashMap<String, GenericFunction
    >();

    /**
     * Transforms the different types of elements by using polymorphism
     * @param element the element to be transformed
     * @return the transformed element
     */

    def GenericAbstractElement transform(AbstractElement element){
        //System.out.println("Transforming: " + element)
        if(element instanceof VariableRef){
            transform(element as VariableRef)
        }
        else if(element instanceof BinaryExpression){
            transform(element as BinaryExpression)
        }
        else if(element instanceof UnaryExpression){
            transform(element as UnaryExpression)
        }
        else if(element instanceof ParenthesizedExpression){
            transform(element as ParenthesizedExpression)
        }
        else if(element instanceof DoubleConstant){
            transform(element as DoubleConstant)
        }
        else if(element instanceof BoolConstant){
            transform(element as BoolConstant)
```

```
        }
        else if(element instanceof StringConstant){
            transform(element as StringConstant)
        }
        else if(element instanceof Container){
            transform(element as Container)
        }
        else if(element instanceof ContainerRef){
            transform(element as ContainerRef)
        }
        else if(element instanceof Function){
            transform(element as Function);
        }
        else if(element instanceof Variable){
            transform(element as Variable);
        }
        else if(element instanceof MockDef){
            transform(element as MockDef);
        }
        else if(element instanceof MockEntry){
            transform(element as MockEntry);
        }
        else if(element instanceof MockRef){
            transform(element as MockRef);
        }
        else if(element instanceof ListVariable){
            transform(element as ListVariable);
        }
        else if(element instanceof ListVariableRef){
            transform(element as ListVariableRef);
        }
        else {
            throw new Exception("Transformer encountered unsupported
    AbstractElement type: " + element);
        }
    }

    /**
     * Transforms MockDef elements
     * @param mockDef the MockDef to be transformed
     * @return the transformed MockDef
     */

    def GenericFunction transform(MockDef mockDef){
        if(!functions.containsKey(mockDef.name)){
            functions.put(mockDef.name, new GenericFunction(mockDef.name));
        }
        return functions.get(mockDef.name);
    }

    /**
     * Transforms MockEntry elements
     * @param entry the MockEntry to be transformed
     * @return the transformed MockEntry
     */
```

```
90
    def GenericFunction transform(MockEntry entry){
92      var List<GenericAbstractElement> parameters = new ArrayList<
    GenericAbstractElement>();
        for(AbstractElement element : entry.parameters){
94          parameters.add(transform(element));
        }
96      functions.get(entry.type.name).insertEntry(parameters, transform(entry.
    result));
        return functions.get(entry.type.name);
98  }

100 /**
     * Transforms MockRef elements
102  * @param reference the MockRef to be transformed
     * @return the transformed MockRef
104  */

106 def GenericFunctionReference transform(MockRef reference){
        var List<GenericAbstractElement> parameters = new ArrayList<
    GenericAbstractElement>();
108     for(AbstractElement element : reference.parameters){
            parameters.add(transform(element));
110     }
        return new GenericFunctionReference(functions.get(reference.mockFunction.
    name), parameters);
112 }

114 /**
     * Transforms Variable elements
116  * @param variable the Variable to be transformed
     * @return the transformed Variable
118  */

120 def GenericAbstractVariable transform(Variable variable){
        //New variable
122     if(variable.expression == null){
            variables.put(variable.name, new GenericVariable(variable.name));
124         return variables.get(variable.name);
        }
126     //Existing variable
        else{
128         variables.put(variable.name, new GenericVariable(variable.name,
    transform(variable.expression)));
            return variables.get(variable.name);
130     }
    }
132
    /**
134  * Transforms VariableRef elements
     * @param reference the VariableRef to be transformed
136  * @return the referenced Variable
     */
138
    def GenericAbstractVariable transform(VariableRef reference){
```

```
140         return variables.get(reference.variable.name);
        }
142
        /**
144      * Transforms ListVariable elements
         * @param listVariable the ListVariable to be transformed
146      * @return the transformed ListVariable
         */
148
        def GenericAbstractVariable transform(ListVariable listVariable){
150         //New variable
            if(listVariable.expression == null){
152             variables.put(listVariable.name, new GenericListVariable(listVariable
        .name));
                return variables.get(listVariable.name);
154         }
            //Existing variable
156         else{
                variables.put(listVariable.name, new GenericListVariable(listVariable
        .name, transform(listVariable.expression), 10));
158             return variables.get(listVariable.name);
            }
160     }

162     /**
         * Transforms ListVariableRef elements
164      * @param reference the ListVariableRef to be transformed
         * @return the referenced ListVariable
166      */

168     def GenericAbstractVariable transform(ListVariableRef reference){
            return variables.get(reference.listvariable.name);
170     }

172     /**
         * Transforms Container elements and their contents
174      * @param container the Container to be transformed
         * @return the transformed Container
176      */

178     def GenericContainer transform(Container container){
            var genericContainer = new GenericContainer();
180         for(element: container.elements){
                genericContainer.addElement(transform(element));
182         }
            containers.put(container.name, genericContainer);
184         return genericContainer;
        }
186
        /**
188      * Transforms ContainerRef elements
         * @param reference the ContainerRef to be transformed
190      * @return the referenced Container
         */
192
```

```
    def GenericAbstractElement transform(ContainerRef reference){
        //Reference has a index
        if(reference.cell != null){
            var cell = transform(reference.cell) as GenericNumber;
            var container = containers.get(reference.container.name);
            return container.get(cell.value.intValue);
        }
        //No index present
        else return containers.get(reference.container.name);
    }

    /**
     * Transforms Function elements and their parameters
     * @param function the Function to be transformed
     * @return the transformed Function
     */


    def GenericAbstractElement transform(Function function){
        var parameters =  new ArrayList<GenericAbstractElement>();
        for(parameter : function.parameters){
            parameters.add(transform(parameter));
        }
        if(function.functionType == FunctionType.^IF){
            return transformIf(function);
        }
        else if(functions.get(function.name) != null){
            return new GenericFunctionReference(functions.get(function.name),
parameters);
        }
        else{
            throw new Exception("Unsupported Function: " + function)
        }
    }

    /**
     * Transforms IF Function elements and their parameters
     * @param function the IF Function to be transformed
     * @return the transformed IF Function
     */

    def GenericIfConditional transformIf(Function function){
        var parameters =  new ArrayList<GenericAbstractElement>();
        for(parameter : function.parameters){
            parameters.add(transform(parameter));
        }
        switch(parameters.size){
            case 2 : return new GenericIfConditional(function.name, parameters.
get(0), parameters.get(1))
            case 3 : return new GenericIfConditional(function.name, parameters.
get(0), parameters.get(1), parameters.get(2))
            default: throw new Exception("Unsupported number of parameters in
IfConditional " + function)
        }
    }
```

```
244
    /**
246  * Transforms BinaryExpression elements
     * @param expression the BinaryExpression to be transformed
248  * @return the transformed BinaryExpression
     */
250
    def GenericBinaryExpression transform(BinaryExpression expression){
252     if(expression.op.equals(BinaryOperator.GREATER)){
            return new GenericBinaryExpression(transform(expression.left),
    transform(expression.right), GenericBinaryOperator.GT)
254     }
        if(expression.op.equals(BinaryOperator.SMALLER)){
256         return new GenericBinaryExpression(transform(expression.left),
    transform(expression.right), GenericBinaryOperator.LT)
        }
258     if(expression.op.equals(BinaryOperator.GREATEREQUAL)){
            return new GenericBinaryExpression(transform(expression.left),
    transform(expression.right), GenericBinaryOperator.GTE)
260     }
        if(expression.op.equals(BinaryOperator.SMALLEREQUAL)){
262         return new GenericBinaryExpression(transform(expression.left),
    transform(expression.right), GenericBinaryOperator.LTE)
        }
264     if(expression.op.equals(BinaryOperator.EQUAL)){
            return new GenericBinaryExpression(transform(expression.left),
    transform(expression.right), GenericBinaryOperator.EQUALS)
266     }
        if(expression.op.equals(BinaryOperator.NOTEQUAL)){
268         return new GenericBinaryExpression(transform(expression.left),
    transform(expression.right), GenericBinaryOperator.NOTEQUALS)
        }
270     if(expression.op.equals(BinaryOperator.OR)){
            return new GenericBinaryExpression(transform(expression.left),
    transform(expression.right), GenericBinaryOperator.OR)
272     }
        if(expression.op.equals(BinaryOperator.AND)){
274         return new GenericBinaryExpression(transform(expression.left),
    transform(expression.right), GenericBinaryOperator.AND)
        }
276     if(expression.op.equals(BinaryOperator.MULTIPLY)){
            return new GenericBinaryExpression(transform(expression.left),
    transform(expression.right), GenericBinaryOperator.MULTIPLICATION)
278     }
        if(expression.op.equals(BinaryOperator.DIVIDE)){
280         return new GenericBinaryExpression(transform(expression.left),
    transform(expression.right), GenericBinaryOperator.DIVISION)
        }
282     if(expression.op.equals(BinaryOperator.PLUS)){
            return new GenericBinaryExpression(transform(expression.left),
    transform(expression.right), GenericBinaryOperator.ADDITION)
284     }
        if(expression.op.equals(BinaryOperator.MINUS)){
286         return new GenericBinaryExpression(transform(expression.left),
    transform(expression.right), GenericBinaryOperator.SUBTRACTION)
```

```
        }
288     else {
            throw new Exception("Unsupported BinaryOperator Type: " + expression.
    op)
290     }
    }
292
    /**
294  * Transforms UnaryExpression elements
     * @param expression the UnaryExpression to be transformed
296  * @return the transformed UnaryExpression
     */
298
    def GenericUnaryExpression transform(UnaryExpression expression){
300 if(expression.op.equals(UnaryOperator.NEGATE)){
        return new GenericUnaryExpression(transform(expression.expression),
    GenericUnaryOperator.NEGATE)
302     }
    if(expression.op.equals(UnaryOperator.POSITIVE)){
304     return new GenericUnaryExpression(transform(expression.expression),
    GenericUnaryOperator.POSITIVE)
        }
306 if(expression.op.equals(UnaryOperator.NEGATIVE)){
        return new GenericUnaryExpression(transform(expression.expression),
    GenericUnaryOperator.NEGATIVE)
308     }
    }
310
    /**
312  * Transforms ParenthesizedExpression elements
     * @param expression the ParenthesizedExpression to be transformed
314  * @return the transformed ParenthesizedExpression
     */
316
    def GenericParenthesizedElement transform(ParenthesizedExpression expression)
    {
318     return new GenericParenthesizedElement(transform(expression.
    innerExpression));
    }
320
    /**
322  * Transforms DoubleConstant elements
     * @param expression the DoubleConstant to be transformed
324  * @return the transformed DoubleConstant
     */
326
    def GenericNumber transform(DoubleConstant expression){
328     return new GenericNumber(expression.value);
    }
330
    /**
332  * Transforms BoolConstant elements
     * @param expression the BoolConstant to be transformed
334  * @return the transformed BoolConstant
     */
```

```
336
    def GenericBoolean transform(BoolConstant expression){
338         return new GenericBoolean(expression.value);
    }
340
    /**
342      * Transforms StringConstant elements
     * @param expression the StringConstant to be transformed
344      * @return the transformed StringConstant
     */
346
    def GenericString transform(StringConstant expression){
348         return new GenericString(expression.value);
    }
350
    /**
352      * Method to define predefined and mock functions
     */
354
    def void instantiateDefaultFunctions(){
356         functions.put("Abs", new AbsoluteValueFunction());
        functions.put("Round", new RoundFunction());
358         functions.put("Sum", new SumFunction());
    }
360
}
```

Listing C.1: PrecedenceTransformer class used to transform Precedence elements to generic elements

Appendix D

Generated JBehave story

```
Scenario: case 1
When variable A gets value 4
When variable C gets value 'test'
When variable B gets value false

Then variable A should return value 4.0
Then variable B should return value false
Then variable C should return value test
Then function Foobar should return value true clause

Scenario: case 2
When variable A gets value 4
When variable C gets value 'test'
When variable B gets value true

Then variable A should return value 4.0
Then variable B should return value true
Then variable C should return value test
Then function Foobar should return value false clause

Scenario: case 3
When variable A gets value 4
When variable C gets value 'Not test'
When variable B gets value false

Then variable A should return value 4.0
Then variable B should return value false
Then variable C should return value Not test
Then function Foobar should return value false clause

Scenario: case 4
When variable A gets value 4
When variable C gets value 'Not test'
When variable B gets value true

Then variable A should return value 4.0
Then variable B should return value true
Then variable C should return value Not test
```

```
39 Then function Foobar should return value false clause

41 Scenario: case 5
   When variable A gets value 3
43 When variable C gets value 'test'
   When variable B gets value false
45
   Then variable A should return value 3.0
47 Then variable B should return value false
   Then variable C should return value test
49 Then function Foobar should return value false clause

51 Scenario: case 6
   When variable A gets value 3
53 When variable C gets value 'test'
   When variable B gets value true
55
   Then variable A should return value 3.0
57 Then variable B should return value true
   Then variable C should return value test
59 Then function Foobar should return value false clause

61 Scenario: case 7
   When variable A gets value 3
63 When variable C gets value 'Not test'
   When variable B gets value false
65
   Then variable A should return value 3.0
67 Then variable B should return value false
   Then variable C should return value Not test
69 Then function Foobar should return value false clause

71 Scenario: case 8
   When variable A gets value 3
73 When variable C gets value 'Not test'
   When variable B gets value true
75
   Then variable A should return value 3.0
77 Then variable B should return value true
   Then variable C should return value Not test
79 Then function Foobar should return value false clause
```

Listing D.1: JBehave story generated from the example model in Chapter 6

Appendix E

Generated Selenium test

```java
package test;

import org.junit.*;
import org.openqa.selenium.*;
import org.openqa.selenium.firefox.FirefoxDriver;

public class SeleniumTest {

private WebDriver driver;

@Before
public void setUp() throws Exception {
    driver = new FirefoxDriver();
    driver.get("http://example.com/");
    driver.findElement(By.id("inputUsername")).sendKeys("username");
    driver.findElement(By.id("inputPassword")).sendKeys("password");
    driver.findElement(By.id("loginButton")).click();
    }

@Test
public void testCase1() throws Exception {
    driver.findElement(By.xpath("//*[@example-id[='A']/input[@example-id='value']
    ")).sendKeys("4.0");
    driver.findElement(By.xpath("//*[@example-id[='C']/input[@example-id='value']
    ")).sendKeys("test");
    driver.findElement(By.xpath("//*[@example-id[='B']/input[@example-id='value']
    ")).sendKeys("false");

    Assert.assertEquals("4.0", driver.findElement(By.xpath("//*[@example-id[='A
    ']//*[@example-id='value']")));
    Assert.assertEquals("false", driver.findElement(By.xpath("//*[@example-id[='B
    ']//*[@example-id='value']")));
    Assert.assertEquals("test", driver.findElement(By.xpath("//*[@example-id[='C
    ']//*[@example-id='value']")));
    Assert.assertEquals("true clause", driver.findElement(By.xpath("//*[@example-
    id[='Foobar']//*[@example-id='value']")));
}

```

```
    @Test
33  public void testCase2() throws Exception {
        driver.findElement(By.xpath("//*[@example-id[='A']/input[@example-id='value']
        ")).sendKeys("4.0");
35      driver.findElement(By.xpath("//*[@example-id[='C']/input[@example-id='value']
        ")).sendKeys("test");
        driver.findElement(By.xpath("//*[@example-id[='B']/input[@example-id='value']
        ")).sendKeys("true");
37
        Assert.assertEquals("4.0", driver.findElement(By.xpath("//*[@example-id[='A
        ']//*[@example-id='value']")));
39      Assert.assertEquals("true", driver.findElement(By.xpath("//*[@example-id[='B
        ']//*[@example-id='value']")));
        Assert.assertEquals("test", driver.findElement(By.xpath("//*[@example-id[='C
        ']//*[@example-id='value']")));
41      Assert.assertEquals("false clause", driver.findElement(By.xpath("//*[@example
        -id[='Foobar']//*[@example-id='value']")));
    }
43
    @Test
45  public void testCase3() throws Exception {
        driver.findElement(By.xpath("//*[@example-id[='A']/input[@example-id='value']
        ")).sendKeys("4.0");
47      driver.findElement(By.xpath("//*[@example-id[='C']/input[@example-id='value']
        ")).sendKeys("Not test");
        driver.findElement(By.xpath("//*[@example-id[='B']/input[@example-id='value']
        ")).sendKeys("false");
49
        Assert.assertEquals("4.0", driver.findElement(By.xpath("//*[@example-id[='A
        ']//*[@example-id='value']")));
51      Assert.assertEquals("false", driver.findElement(By.xpath("//*[@example-id[='B
        ']//*[@example-id='value']")));
        Assert.assertEquals("Not test", driver.findElement(By.xpath("//*[@example-id
        [='C']//*[@example-id='value']")));
53      Assert.assertEquals("false clause", driver.findElement(By.xpath("//*[@example
        -id[='Foobar']//*[@example-id='value']")));
    }
55
    @Test
57  public void testCase4() throws Exception {
        driver.findElement(By.xpath("//*[@example-id[='A']/input[@example-id='value']
        ")).sendKeys("4.0");
59      driver.findElement(By.xpath("//*[@example-id[='C']/input[@example-id='value']
        ")).sendKeys("Not test");
        driver.findElement(By.xpath("//*[@example-id[='B']/input[@example-id='value']
        ")).sendKeys("true");
61
        Assert.assertEquals("4.0", driver.findElement(By.xpath("//*[@example-id[='A
        ']//*[@example-id='value']")));
63      Assert.assertEquals("true", driver.findElement(By.xpath("//*[@example-id[='B
        ']//*[@example-id='value']")));
        Assert.assertEquals("Not test", driver.findElement(By.xpath("//*[@example-id
        [='C']//*[@example-id='value']")));
65      Assert.assertEquals("false clause", driver.findElement(By.xpath("//*[@example
        -id[='Foobar']//*[@example-id='value']")));
```

```
    }
67
    @Test
69  public void testCase5() throws Exception {
        driver.findElement(By.xpath("//*[@example-id[='A']/input[@example-id='value']
        ")).sendKeys("3.0");
71      driver.findElement(By.xpath("//*[@example-id[='C']/input[@example-id='value']
        ")).sendKeys("test");
        driver.findElement(By.xpath("//*[@example-id[='B']/input[@example-id='value']
        ")).sendKeys("false");
73
        Assert.assertEquals("3.0", driver.findElement(By.xpath("//*[@example-id[='A
        ']//*[@example-id='value']")));
75      Assert.assertEquals("false", driver.findElement(By.xpath("//*[@example-id[='B
        ']//*[@example-id='value']")));
        Assert.assertEquals("test", driver.findElement(By.xpath("//*[@example-id[='C
        ']//*[@example-id='value']")));
77      Assert.assertEquals("false clause", driver.findElement(By.xpath("//*[@example
        -id[='Foobar']//*[@example-id='value']")));
    }
79
    @Test
81  public void testCase6() throws Exception {
        driver.findElement(By.xpath("//*[@example-id[='A']/input[@example-id='value']
        ")).sendKeys("3.0");
83      driver.findElement(By.xpath("//*[@example-id[='C']/input[@example-id='value']
        ")).sendKeys("test");
        driver.findElement(By.xpath("//*[@example-id[='B']/input[@example-id='value']
        ")).sendKeys("true");
85
        Assert.assertEquals("3.0", driver.findElement(By.xpath("//*[@example-id[='A
        ']//*[@example-id='value']")));
87      Assert.assertEquals("true", driver.findElement(By.xpath("//*[@example-id[='B
        ']//*[@example-id='value']")));
        Assert.assertEquals("test", driver.findElement(By.xpath("//*[@example-id[='C
        ']//*[@example-id='value']")));
89      Assert.assertEquals("false clause", driver.findElement(By.xpath("//*[@example
        -id[='Foobar']//*[@example-id='value']")));
    }
91
    @Test
93  public void testCase7() throws Exception {
        driver.findElement(By.xpath("//*[@example-id[='A']/input[@example-id='value']
        ")).sendKeys("3.0");
95      driver.findElement(By.xpath("//*[@example-id[='C']/input[@example-id='value']
        ")).sendKeys("Not test");
        driver.findElement(By.xpath("//*[@example-id[='B']/input[@example-id='value']
        ")).sendKeys("false");
97
        Assert.assertEquals("3.0", driver.findElement(By.xpath("//*[@example-id[='A
        ']//*[@example-id='value']")));
99      Assert.assertEquals("false", driver.findElement(By.xpath("//*[@example-id[='B
        ']//*[@example-id='value']")));
        Assert.assertEquals("Not test", driver.findElement(By.xpath("//*[@example-id
        [='C']//*[@example-id='value']")));
```

```
101        Assert.assertEquals("false clause", driver.findElement(By.xpath("//*[@example
           -id[='Foobar']//*[@example-id='value']")));
    }
103
    @Test
105 public void testCase8() throws Exception {
        driver.findElement(By.xpath("//*[@example-id[='A']/input[@example-id='value']
           ")).sendKeys("3.0");
107     driver.findElement(By.xpath("//*[@example-id[='C']/input[@example-id='value']
           ")).sendKeys("Not test");
        driver.findElement(By.xpath("//*[@example-id[='B']/input[@example-id='value']
           ")).sendKeys("true");
109
        Assert.assertEquals("3.0", driver.findElement(By.xpath("//*[@example-id[='A
           ']//*[@example-id='value']")));
111     Assert.assertEquals("true", driver.findElement(By.xpath("//*[@example-id[='B
           ']//*[@example-id='value']")));
        Assert.assertEquals("Not test", driver.findElement(By.xpath("//*[@example-id
           [='C']//*[@example-id='value']")));
113     Assert.assertEquals("false clause", driver.findElement(By.xpath("//*[@example
           -id[='Foobar']//*[@example-id='value']")));
    }
115
    }
```

LISTING E.1: Selenium test generated from the example model in Chapter 6

Appendix F

Selenium functions

```
2  /**
    * Returns the prefix for a valid Selenium class
4   * @param modelName the name of the model under test
    * @return the prefix for a valid Selenium class
6   */

8  public String generateSCORECARDTESTMODELPrefix(String modelName){
       return "package test;" +
10             "\n" +
               "\n" +
12             "import java.util.concurrent.TimeUnit;" + "\n" +
               "\n" +
14             "import org.junit.*;" + "\n" +
               "import org.openqa.selenium.*;" + "\n" +
16             "import org.openqa.selenium.firefox.FirefoxDriver;" + "\n" +
               "\n" +
18             "public class " + modelName + "Selenium {" + "\n" +
               "\n" +
20             "private static WebDriver driver;" + "\n" +
               "\n" +
22             "@BeforeClass" + "\n" +
               "public static void setUp() throws Exception {" + "\n" +
24             "\tdriver = new FirefoxDriver();" + "\n" +
               "\tdriver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);"
       + "\n" +
26             "\tdriver.get(\"http://squirtle:8080/f4c-web/\");" + "\n" +
               "\tdriver.findElement(By.id(\"inputUsername\")).sendKeys(\"username
       \");" + "\n" +
28             "\tdriver.findElement(By.id(\"inputPassword\")).sendKeys(\"password
       \");" + "\n" +
               "\tdriver.findElement(By.id(\"loginButton\")).click();" + "\n" +
30             "\tdriver.findElement(By.xpath(\"//*[@selenium-id=\'customerName\'
       and @title=\'Test\']\")).click();" + "\n" +
               "\tdriver.findElement(By.xpath(\"//*[@selenium-id=\'
       SCORECARDTESTMODEL\']\")).click();" + "\n" +
32             "\tdriver.findElement(By.xpath(\"//*[@selenium-id=\'description\']\")
       ).click();" + "\n" +
```

```
            "\tdriver.findElement(By.xpath(\"//*[@selenium-id=\'Q_Map05\']\")).
    click();" + "\n" +
            "}\n" +
            "\n" +
            "@Before" + "\n" +
            "public void setupCase() throws Exception{" + "\n" +
            "\tdriver.findElement(By.xpath(\"//*[@selenium-id=\'26\']\")).click()
    ;" + "\n" +
            "\tdriver.findElement(By.xpath(\"//*[@selenium-id='OnroerendGoed' and
     @lvl='4']//*[@selenium-id='junctionLink']\")).click();" + "\n" +
            "}" + "\n" +
            "\n";
}

/**
 * Returns the postfix for a valid Selenium class
 * @return the postfix for a valid Selenium class
 */

public String generateSCORECARDTESTMODELPostfix(){
    return "@After" + "\n" +
            "public void tearDownCase() throws Exception {" + "\n" +
            "\tdriver.findElement(By.xpath(\"//*[@selenium-id='OnroerendGoed' and
     @lvl='4']//*[@selenium-id='submit']\")).click();" + "\n" +
            "\tdriver.findElement(By.xpath(\"//*[@class='button icon ok']\")).
    click();" + "\n" +
            "}" + "\n" +
            "\n" +
            "@AfterClass" + "\n"+
            "public static void tearDown() throws Exception {" + "\n" +
            "\tdriver.findElement(By.id(\"user\")).click();" + "\n" +
            "\tdriver.findElement(By.xpath(\"//*[@selenium-id='logout']\")).click
    ();" + "\n" +
            "\tThread.sleep(2000);" + "\n" +
            "\tdriver.quit();" + "\n" +
            "}" + "\n" +
            "\n" +
            "}";
}
```

Listing F.1: Java functions used to generate a executable Selenium test

Bibliography

[1] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000. ISSN 0362-1340. doi: 10.1145/352029.352035. URL http://doi.acm.org/10.1145/352029.352035.

[2] Arie van Deursen. Domain-specific languages versus object-oriented frameworks: A financial engineering case study. *Smalltalk and Java in Industry and Academia, STJA'97*, pages 35–39, 1997.

[3] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. Model-based dsl frameworks. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 602–616, New York, NY, USA, 2006. ACM. ISBN 1-59593-491-X. doi: 10.1145/1176617.1176632. URL http://doi.acm.org/10.1145/1176617.1176632.

[4] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. ISBN 978-1-4812-1858-0. URL http://www.dslbook.org.

[5] Stuart Kent. Model driven engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43703-7. URL http://dl.acm.org/citation.cfm?id=647983.743552.

[6] Object Management Group. Mda guide rev. 2.0. page 15, 2014. URL http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.

[7] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011. ISBN 1118031962, 9781118031964.

[8] Emin Gün Sirer and Brian N. Bershad. Using production grammars in software testing. In *Proceedings of the 2Nd Conference on Conference on Domain-Specific Languages - Volume 2*, DSL'99, pages 1–1, Berkeley, CA, USA, 1999. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267936.1267937.

[9] William E. Lewis and W. H. C. Bassetti. *Software Testing and Continuous Quality Improvement, Second Edition*. Auerbach Publications, Boston, MA, USA, 2004. ISBN 0849325242.

[10] Zhen Ru Dai. Model-Driven Testing with UML 2.0. Technical report, Computing Laboratory, University of Kent, 2004.

[11] Olli-Pekka Puolitaival and Teemu Kanstrén. Towards flexible and efficient model-based testing, utilizing domain-specific modelling. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*, DSM '10, pages 8:1–8:6, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0549-5. doi: 10.1145/2060329.2060349. URL http://doi.acm.org/10.1145/2060329.2060349.

[12] D. Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, Aug 1996. ISSN 0018-9219. doi: 10.1109/5.533956.

[13] Carlos Solis and Xiaofeng Wang. A study of the characteristics of behaviour driven development. In *Proceedings of the 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, SEAA '11, pages 383–387, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4488-5. doi: 10.1109/SEAA.2011.76. URL http://dx.doi.org/10.1109/SEAA.2011.76.

[14] Dionny Santiago, Adam Cando, Cody Mack, Gabriel Nunez, Troy Thomas, and Tariq M King. Towards domain-specific testing languages for software-as-a-service. In *MDHPCL@ MoDELS*, pages 43–52, 2013.

[15] Tariq M. King, Gabriel Nunez, Dionny Santiago, Adam Cando, and Cody Mack. Legend: An agile dsl toolset for web acceptance testing. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 409–412, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2645-2. doi: 10.1145/2610384.2628048. URL http://doi.acm.org/10.1145/2610384.2628048.

[16] World Internet Stats. World Internet Users and 2014 Population Stats. http://www.internetworldstats.com/stats.htm, 2014. [Online; accessed 12-May-2015].

[17] Chien-Hung Liu, David C. Kung, Pei Hsia, and Chih-Tung Hsu. Structural testing of web applications. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, ISSRE '00, pages 84–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0807-3. URL http://dl.acm.org/citation.cfm?id=851024.856240.

[18] Xinchun Wang and Peijie Xu. Build an auto testing framework based on selenium and fitnesse. In *Proceedings of the 2009 International Conference on Information*

Technology and Computer Science - Volume 02, ITCS '09, pages 436–439, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3688-0. doi: 10.1109/ITCS.2009.228. URL http://dx.doi.org/10.1109/ITCS.2009.228.

[19] Andreas Bruns, Andreas Kornstadt, and Dennis Wichmann. Web application tests with selenium. *IEEE Software*, 26(5):88–91, 2009. ISSN 0740-7459. doi: http://doi.ieeecomputersociety.org/10.1109/MS.2009.144.

[20] Selenium Project. Brief History of The Selenium Project. http://www.seleniumhq.org/docs/01_introducing_selenium.jsp#brief-history-of-the-selenium-project, 2015. [Online; accessed 12-May-2015].

[21] A. Holmes and M. Kellogg. Automating functional tests using selenium. In *Agile Conference, 2006*, pages 6 pp.–275, July 2006. doi: 10.1109/AGILE.2006.19.

[22] Pierre Bourque and Richard E. Fairley. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Computer Society, Version 3.0 edition, 2014. ISBN 0-7695-5166-1.

[23] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996. ISBN 0262011530.

[24] ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI)*. Geneva, Switzerland, 6 edition, June 2012. URL http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[25] Bruno Richard Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. John Wiley & Sons, 2000. URL http://www.brpreiss.com/books/opus5. 635 pp. ISBN 0-471-34613-6.

[26] Niek Hulsman. Technische documentatie - Finan ModeltaalOntwikkelomgeving. June 2013.

[27] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005. ISSN 0360-0300. doi: 10.1145/1118890.1118892. URL http://doi.acm.org/10.1145/1118890.1118892.