EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Testing an Industrial Code Generator With Model-Based Testing

Hoeijmakers, Tijs P.H.

Award date:
2022

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Technische Universiteit Eindhoven
University of Technology
Department of Mathematics and Computer Science

Testing an Industrial Code Generator With Model-Based Testing

Tijs Hoeijmakers

Supervisors:
Dr. Wilbert Alberts (ASML)
Ir. Niels Brouwers (Capgemini)
Dr. Ivan Kurtev (TU/e)

Graduation Committee:
Dr. Wilbert Alberts (ASML)
Ir. Niels Brouwers (Capgemini)
Dr. Ivan Kurtev (TU/e)
Dr. Tim Willemse (TU/e)

Eindhoven, September 2022

Abstract

Capgemini has co-developed with ASML a family of Domain Specific Languages called ASOME. These languages allow ASML to specify on a high abstraction level the Data, Control, and Algorithms (DCA) aspects of a system in separation. A code generator has been developed that generates C++ code that should conform to the specification. Code generators are developed by humans and therefore typically not bug-free. Since the users need to rely on these code generators to produce code that matches their input specification, it is clear that there is a strong need for these code generators to work correctly. A solution from Model-Driven Engineering for the testing process of a system is Model-Based Testing (MBT). This technique allows generating test cases automatically, which enables testing a system extensively. We will research how to apply MBT to the code generator for Domain Data Models. In particular, we will apply MBT on the code generator of ASOME using the PyModel tool. We will cover all steps involved in the application of MBT. That includes the selection of an MBT tool, making Domain Data Models compatible with the MBT tool, the development of an adaptor, the creation and analysis of different testing strategies, dividing an infinite state space up into a finite number of equivalence classes, and the actual application of MBT on generated code. We will evaluate on the MBT procedure and see that apart from some drawbacks, MBT can in fact be used in an industrial context to recover bugs in a mature code generator.

Preface

The creation of this Master Thesis spreads a period of six months. During this period I acquainted myself with the topic of Model-Based Testing. This topic was unknown to me beforehand. I did however already have a background in the formal aspect of the verification of models through courses at the Technical University of Eindhoven (TU/e). This knowledge helped me in getting acquainted with the topic at a fast pace.

This master thesis is the endpoint of my master's in Computer Science and Engineering on the TU/e. I am grateful that I could finish my studies at this university, and for the opportunities the university gave me. I would like to express my sincere gratitude towards Wilbert Alberts, Niels Brouwers, and Ivan Kurtev. They have offered me guidance and support throughout this journey. The meetings always contained fine suggestions and a sense of humor, making them joyful and insightful. Their sharpness taught me to think more carefully about the design and semantics of specification languages. I would also like to thank Capgemini Engineering for giving me the opportunity to carry out the Master Thesis in their company and for providing me support. Furthermore, I am grateful that ASML allowed me to work on a project that is developed on their behalf.

I thank my parents and sister for their support throughout this trajectory and for keep supporting me through the ups and downs. Finally, I thank my friends for their encouragement and support. They made my study time an enjoyable period of my life.

Tijs Hoeijmakers
Eindhoven, The Netherlands
September 2022

Acronyms

API Application Programming Interface.

ASOME ASML Software Modeling Environment.

AST Abstract Syntax Tree.

ASM Abstract State Machine.

CRUD+A Create, Read, Update, Delete and Add.

DAG Directed Acyclic Graph.

DCA Data Control and Algorithms.

DSL Domain-Specific Language.

DMDSL Domain-Interface Modeling Domain Specific Language.

EFSM Extended Finite State Machine.

EMF Eclipse Modeling Framework.

FSM Finite State Machine.

GPL General Purpose Language.

IUT Implementation Under Test.

LTS Labelled Transition System.

MBT Model-Based Testing.

MC/DC Modified Condition Decision Coverage.

MDE Model-Driven Engineering.

OCL Object Constraint Language.

OOP Object-Oriented Programming.

STS Symbolic Transition System.

SUT System Under Test. (In the context of this thesis, this term is typically avoided. We use the term IUT instead.)

V&V Verification and Validation.

Contents

Contents v

1 Introduction 1
1.1 Model-Driven Engineering . . . 1
1.2 Introduction to the ASOME Languages Family . . . 2
1.3 Problem Statement . . . 2
1.4 Research Questions . . . 3
1.5 Approach . . . 3
1.6 Outline . . . 3

2 Testing a Code Generator 4
2.1 Techniques to Test a Code Generator . . . 4
2.1.1 Unit Testing . . . 4
2.1.2 Back-To-Back Testing . . . 5
2.1.3 Model-Based Testing . . . 5
2.1.4 Formally Prove Correctness . . . 6
2.2 Model-Based Testing . . . 6
2.2.1 General Context . . . 6
2.2.2 Code Generator Context . . . 7

3 The ASOME Software Modeling Environment 9
3.1 Introduction to DMDSL . . . 9
3.2 Basic Specification in DMDSL . . . 10
3.2.1 Example . . . 10
3.3 Formal Semantics . . . 12
3.3.1 Static Semantics . . . 13
3.3.2 Dynamic Semantics . . . 14

4 Model-Based Testing in the Context of DMDSL 21
4.1 DMDSL Testing Models . . . 21
4.2 Making DMDSL Models MBT Compatible . . . 22
4.3 MBT Tool Evaluation . . . 23
4.4 Adaptor . . . 23
4.5 Test Report . . . 23

5 Selecting a Model-Based Testing Tool 25
5.1 Approach . . . 25
5.1.1 A Systematic Review of Tools . . . 25
5.1.2 Procedure Used in DMDSL context . . . 26
5.2 Formalisms in Model-Based Testing . . . 27
5.2.1 Popular Formalisms . . . 27
5.2.2 Selecting a Formalism . . . 29

5.3 General Formal Concepts . . . 29
5.3.1 Formal Conformance . . . 29
5.4 Model-Based Testing with Labelled Transition Systems . . . 31
5.4.1 IOCO on Labelled Transition Systems . . . 31
5.4.2 IOCO for Symbolic Specifications . . . 34
5.4.3 Tools . . . 38
5.5 Model-Based Testing with Model Programs . . . 46
5.5.1 Conformance on Model Programs . . . 46
5.5.2 Tools . . . 49
5.6 Result of MBT selection . . . 51

6 Encoding Models in an MBT Tool **52**
6.1 Scope of Modeling . . . 52
6.2 Generation of PyModel Model . . . 53
6.3 Abstract Syntax Elements . . . 54
6.3.1 Multiplicity . . . 54
6.3.2 Entity . . . 55
6.3.3 Instance . . . 55
6.3.4 Association . . . 56
6.3.5 Link . . . 56
6.4 Model-Dependent Constructs . . . 57
6.4.1 Entities . . . 57
6.4.2 Associations . . . 57
6.5 Dynamic Language Constructs . . . 58
6.5.1 State Variables . . . 58
6.5.2 PyModel Actions . . . 58
6.6 Exploration of PyModel Models . . . 62

7 MBT Tool Adaptor **64**
7.1 Architecture of Adaptor . . . 64
7.1.1 Stepper . . . 65
7.1.2 Main . . . 65
7.1.3 CallTranslator . . . 66
7.2 Concretization of Actions . . . 66
7.2.1 Create . . . 66
7.2.2 Read . . . 67
7.2.3 Update . . . 68
7.2.4 Delete . . . 68
7.2.5 Add . . . 68
7.3 Abstraction of Output . . . 68
7.3.1 Create . . . 68
7.3.2 Read . . . 70
7.3.3 Update . . . 70
7.3.4 Delete . . . 71
7.3.5 Add . . . 71
7.4 Generating an Adaptor . . . 71

8 Application of MBT **73**
8.1 Domains of Actions . . . 73
8.1.1 Create . . . 73
8.1.2 Read . . . 77
8.1.3 Update . . . 78
8.1.4 Delete . . . 78
8.1.5 Add . . . 78

8.2 Testing Strategies for Model Exploration . . . 78
8.2.1 Random Strategy . . . 79
8.2.2 Exhaustive Create Add Strategy . . . 80
8.2.3 Default State Coverage . . . 81
8.2.4 State Coverage 1 . . . 81
8.2.5 State Coverage 2 . . . 81
8.2.6 State Coverage 3 . . . 90
8.2.7 State Coverage 4 . . . 91
8.2.8 Coverage for output messages . . . 91
8.3 Strategy Analysis . . . 91
8.3.1 Bench . . . 92
8.4 Testing a Model . . . 94
8.4.1 Finding Issues . . . 95
8.4.2 Applying PyModel on Capgemini Test Models . . . 98
8.5 Discussion . . . 102

9 Conclusions and Future Work **104**
9.1 Answers to Research Questions . . . 104
9.2 Future Directions and Conclusion . . . 106

Bibliography **107**

Appendix **109**

A TorXakis **110**
A.1 Model Translation . . . 110
A.2 Source cascade deletion . . . 120

B Test Models **123**

C Error Output **129**
C.1 Read Output Messages . . . 129
C.2 Update Output Messages . . . 129
C.3 Delete Output Messages . . . 130
C.4 Add Output Messages . . . 131

Chapter 1

Introduction

1.1 Model-Driven Engineering

Since the digital revolution started, a lot of technological advancements are made that significantly contribute to society. With these advancements also comes a whole new set of challenges, among which is the development of correct software. There is no denying that society nowadays heavily relies on software. In almost every electronic device, there is some software controlling it. Coffee machines, cars, vacuum cleaners, and with the rise of IoT even lights have some software running on them. Since the devices and systems we develop are getting increasingly more complex, the complexity of writing correct software also grows. To keep up with this, new methods are created that assist in writing software (e.g. intelligent IDEs, testing frameworks, programming with AI assistance, etc.). However, due to the increasing complexity, developing correct software remains a big challenge. As a consequence, developers nowadays spend a significant part of their development time merely on debugging [5]. For some systems, such as a computer game, a subtle rare bug can be acceptable. On the other hand, it is of major importance that the software controlling a nuclear reactor is correct. The consequences of incorrectness can be astronomical. Even on more common devices, such as a pacemaker, it is of vital importance that the device behaves as expected. It is clear that there is a need for new techniques that help to develop correct software.

A software development methodology that is gaining more and more interest is the field of Model-Driven Engineering (MDE). This is a development methodology that focuses on the creation and usage of models as primary software artifacts. In this methodology, creation of models becomes an important part of the development cycle. Domain-Specific Languages (DSL) are created that allow for the development of models on a high abstraction level, exploiting domain-specific knowledge. These models can often be transformed into a general-purpose language (GPL) and sometimes even be verified using tools. By working on a high abstraction level, the expressiveness of the written lines of code increases. Furthermore, since the specification language is designed for a specific domain, the language can be designed in such a way that the expressiveness of a single line of code further increases by exploiting domain-related properties in the language design. Fred Brooks describes in his famous book '*The Mythical Man-Month: Essays on Software Engineering*' that on average a professional programmer contributes 10 lines of code per day, no matter the programming language chosen [6]. Even today, when dividing the lines of code in a project by the number of man-hours spent, it turns out that the average is still close to his estimation. To reduce development time, it is clear that these 10 lines should be rather expressive. One can imagine that much more meaningful code can be written in a DSL that gets transformed into hundreds of lines of C++ code, as can be done in 10 lines of Assembly Language. This high expressiveness of code in DSLs, therefore, allows for faster development cycles. Furthermore, by working on a higher abstraction level it should be easier to specify the system. Validation techniques on the model can help verify that the system satisfies the requirements. MDE could thus be an important tool to write correct software and reduce development time.

1.2 Introduction to the ASOME Languages Family

The ASML Software Modeling Environment (ASOME) is a set of DSLs for usage in ASML. This set of languages employs the ASML DCA Pattern. This is a pattern in which the Data, Control, and Algorithms of a system are separated from each other. Using these languages, aspects of ASML's scanners architecture can be defined and verified in isolation. This should facilitate in the creation of correct software for these scanners.

One of the languages in the ASOME language family is the "Domain Interface Modeling DSL" (DMDSL). This language handles the Data aspect of the DCA pattern. The DMDSL language allows the user to define domain interfaces. In such a domain interface, constructs such as *Entities*, *ValueObjects*, and their corresponding relations can be described. The operations the interfaces provide access to are the Create, Read, Update, Delete and Add operations (CRUD+A). In particular, *Entities* can be instantiated using the Create operation. Such an instance can be added to a corresponding Entity repository using the Add operation. Instances from the repository can be read from the repository using the Read operation. The instances can also be removed from the repository using the Delete operation. Finally, the instances can also be changed using the Update operation. On the basis of such interfaces, a realization satisfying the interfaces can be generated. In this work, we will focus on this DMDSL language, and test whether the transformation from the high-level DMDSL specification to C++ happens correctly using a technique called Model-Based Testing (MBT).

1.3 Problem Statement

An important reason for using a DSL to develop a system is to decrease the chances for the system to show undesired behavior. However, can we be confident that a code generator for the DSL will produce code conforms to the specification? Suppose a specification in a DSL satisfies all desirable properties, but still the generated code is faulty. Then still the developer of the model is left with incorrect code, and it is not this developer who is to blame. That would be very undesirable and arguably worse. Therefore, a high level of confidence in this translation process is needed. A need to get very thorough testing of the code generator emerges. The testers of the code generator of the DMDSL language point out the difficulties in the testing process of the code generator of this language. It is time-consuming to come up with a testcase for some given model. Having lots of testcases also puts a burden on maintainability. It is even mentioned by testers of Capgemini that compilation time becomes an issue when there are too many test models. For complex models, it is often not easy to see what the expected behavior of the generated code should be which contributes to the difficulties of writing testcases for the code generator.

Model-Based Testing is a testing approach in which testcases are derived from a model on which the implementation is based, and in the context of code generators even generated from. In the model, on a high abstraction level, the intended behavior of the implementation is specified. Hence, information on how the implementation of the system should behave is encoded in such a model. In order to test the actual implementation, it is then possible to derive tests, including the test verdicts, from a model automatically. With MBT one is then able to test whether an implementation seems to conform to the model by producing testcases automatically. This allows for a testing procedure that is at least quantitatively more extensive than feasible with manually written testcases.

To apply MBT in the context of a code generator, we need to research how the ideas translate to such a setting. We want to be able to generate a high-quality test suite to get high confidence in the correctness of the tested code generator. Therefore, we need a way of determining what an appropriate MBT solution is. Furthermore, we need to figure out how we can apply such an MBT solution, and observe if it is feasible to discover bugs in a code generator that is currently in use.

1.4 Research Questions

The main research question in our research will be: *How to apply Model-Based Testing to a code generator?*

Answering this question gives rise to the following sub-questions

1. How is Model-Based Testing applied to improve the testing of code generators?
2. How to assess which Model-Based Testing tool is suitable for a code generator?
   2.1 Which Model-Based Testing tool is suitable for the problem setting?
3. How to apply a *given* Model-Based Testing tool to a code generator?
   3.1 How to make test models compatible with a Model-Based Testing tool?
   3.2 How to create an adapter that can apply abstract test cases to generated code?
4. What are the benefits and drawbacks of applying Model-Based Testing on a code generator in an industrial context?

To answer these questions we will do a case study on the DMDSL code generator.

1.5 Approach

To answer question 1, we will investigate the techniques that are currently used to test code generators. We will research the literature to see how MBT has been applied to code generators. Finally, we will describe on an architectural level how MBT can be applied to test the DMDSL Code generator.

To answer question 2, we will investigate what assessment criteria for Model-Based Testing are. We will check which assessment criteria are most important for models in the DMDSL language. Finally, we experiment with a few concrete MBT tools so that we can select a tool that works in the context of the code generator of the DMDSL language.

To answer question 3, we will apply a selected MBT tool on the DMDSL code generator. We will research how DMDSL specifications can be made compatible with the Model-Based Testing Tool. Then we investigate how the MBT tool can apply the generated test cases on the generated code. Finally, we will investigate how the MBT tool can be used to generate high-quality tests.

To answer question 4, we will describe the benefits and drawbacks encountered during the MBT process on the DMDSL code generator.

1.6 Outline

In chapter 2, we will investigate the literature for different techniques that can help in the verification of a code generator. Furthermore, we give an introduction to MBT, and how it can be used to verify a code generator. In chapter 3, we will introduce the semantics of the DMDSL language. In chapter 4, we will put MBT in the context of the DMDSL code generator, and discuss the general architecture of the process. In chapter 5, we will discuss the procedure of selecting an MBT tool. This involves a discussion of popular mathematical modeling formalisms, a discussion of a popular testing theory, and some experiments with different MBT tools. In chapter 6, we will discuss the process of translating DMDSL models into a specification language supported by an MBT tool. In chapter 7, we will discuss the development of an adaptor. In chapter 8, we will create different testing strategies, do some analysis on these strategies, and apply MBT to generated code to put the code generator to the test. Finally, in chapter 9 we will make a general conclusion, and consider future work.

Chapter 2

Testing a Code Generator

In this chapter, we will research existing testing techniques applied to code generators. We will also dive deeper into the technique of Model-Based Testing.

2.1 Techniques to Test a Code Generator

There are several techniques to test a code generator. We will consider popular techniques to test a code generator.

2.1.1 Unit Testing

A common approach to the *testing* of software is the application of *unit testing*[2]. In unit testing, components of the implementation under test (IUT) are called by executing testcases from the unit testing framework. In the testcase, the output of the component is compared against the expected output. As Stahl et al. point out, applying this to a code generator naively has significant disadvantages [27]. In a code generator, the specification expressed in a DSL is translated to a GPL. Hence, naively one would compare the generated code against the expected code. In the unit test, one could do a string comparison to see if the output of the code generator matches the expected output. The disadvantages they point out are as follows:

- Many false positives: even a small change of formatting would make the test code fail, even though the generated code is semantically equivalent and correct.
- Reliance on low-level implementation: the creator of the test has to specify the expected output for the translation process. This makes concrete assumptions about the translation process. There are almost always uncountable many ways how a specification could be translated into code. When some change is made to the code generator, which results in producing different code that still conforms to the specification, the unit test would already fail even though the output is correct.

As a result, Stahl et al. suggest executing the generated code and observing if it has the desirable effect. Since we have specified a model of the generated IUT, it should follow from the semantics of the modeling language what the desired behavior is. For example, in the context of ASOME, the developers manually created test models to validate the features of interest. Then in manually written tests, it is checked that for manually specified inputs, the output produced by the executed code is equivalent to the output one would expect on the basis of the model. There is, however, currently not a way of executing the model itself. Hence, the test designers need to manually conclude from the informally described dynamic semantics what they expect the output should be. This tactic will provide a certain degree of confidence in the implementation. However, since the creation of these testcases is *very labor-intensive*, there will (at least quantitatively) not be such an extensive testing procedure achievable. Furthermore, when changes to the DSL happen

due to requirement changes, new testcases need to be written, and old testcases might need to be rewritten which is very time-consuming.

2.1.2 Back-To-Back Testing

In [17] Jörges and Steffen also recognized the problem of testing a code generator with unit tests. They followed the suggestion of Stahl et al. to test the generated code's effect instead of concrete syntax. This lead them to applying *back-to-back* testing on a code generator.

In [35] the Back-To-Back testing technique is introduced. Vouk describes back-to-back testing as a strategy that involves the production of two or more functionally equivalent programs. These programs are then tested with the same input, and outputs are compared. In the situation of testing a code generator, one of the 'programs' would be an executable version of the manually crafted test model, and the other program would be the GPL of the test model produced by the code generator. Now the test designer would specify some input. The input will be run on both the model and the generated code. All outputs produced by the running model and by running the generated code on the input are compared. To apply this technique, it is needed that the Model can be executed in some way. Zamani points out that in MDE one often wants to explore the models by executing them, in order to apply *dynamic* software Verification and Validation (V&V) techniques [12]. He points out that for most dynamic V&V tools a prerequisite is to *trace* the execution of executable models. To support the execution of models, an executable modeling language must provide *execution semantics*. For models that can be transformed to a GPL using a code generator, there is not always a strong need to develop a method to execute a model and explore its traces on the level of the high-level specification. In particular, the code generator can transform the high-level specification into code, and the execution of the code will reflect the dynamic semantics of the model. In such a situation, some additional work needs to be done to apply *back-to-back* testing. It is obvious that using the generated GPL as both inputs for *back-to-back* testing will be a self-comparison and no meaningful results can be concluded. Zamani describes that in such a situation there are three approaches possible to provide execution semantics:

- *Denotational* semantics approach: Describe the semantics of the language in mathematical terms.
- *Translational* approach: The specification of the model is translated into another executable language for execution.
- *Operational* approach: The execution behavior of the model is defined by an interpreter. This can be considered a virtual machine that can execute the model through a series of transitions, moving between the states.

From the point of view of the author of this thesis, the *denotational* approach should be considered to be the first step in the *translational* and *operational* approach.

Both the translational and operational approaches are used in practice. Papadopoulus researched the testing of a code generator for a DSL called Maverick, which is used and created by the Dutch bank ING [13]. He produced a concise definition of the semantics of Maverick by developing a *definitional interpreter*[24], following the *operational* approach. This gave a concise definition of the model's execution semantics. In the work of Frenken a code generator for the OIL DSL, which is developed by Océ, is tested [10]. In this work, they express the execution semantics of the OIL language in terms of the mCRL2 language. Here the *translational* approach is chosen.

2.1.3 Model-Based Testing

The technique of MBT builds on the technique of *back-to-back* testing, with the extension that the testcases will be automatically generated. Utting and Legeard define in their book "Practical Model-Based Testing: A Tools Approach" the term Model-Based Testing as follows: "Model-based

testing is the automation of black-box test design. A Model-Based Testing tool uses various test generation algorithms and strategies to generate tests from a behavioral model of the IUT" [20]. The MBT strategy applies back-to-back testing but uses an algorithm to generate the inputs that are used for the IUT and the executable model. By observing output from the model, the tool can derive test cases. Typically, when MBT is applied, it is used for assisting in the development of a system. A model is created on the basis of the requirements of a system that will be or has been developed. After creating a satisfactory model, we test by applying MBT whether the IUT satisfies the desirable behavior as specified by the model. The book of Utting and Legeard points out that typically it is pointless to do Model-Based Testing if the IUT is automatically generated via the code generator when also this same model is used input model of the MBT solution. In that situation, both the tests and implementation are derived from the same source. Hence, errors in the model will propagate to the IUT. The test and implementation suffer in this situation from *lack-of-independence*. Therefore, they state that applying MBT using the model from which the IUT was derived, and the generated code is useless, except if one wants to test the correctness of the test generation and code generation tools themselves. Indeed, the latter is what we're trying to do in this work! Hence, creating test models, and observing whether these are translated correctly into code using MBT is a way of testing the code generator.

Researching the internet for techniques to test code generators confirms that MBT is a suggested technique in testing code generators. For example, in the work of Stürmer and Conrad, an overview of a practice-oriented testing approach for code generation tools is provided [29]. This paper suggests testing a code generator using the principles of MBT. The fact that it can generate tests automatically, allows for the creation of a test suite with lots of test cases. This would not be feasible with manually written test cases. Furthermore, when the design of the test model changes, one only has to make the change to the test model, and all test cases will be updated. Hence, extensive testing of the code generator can be done in a maintainable way. The technique is however not exhaustive. That is, if all test cases are passed, we still can not be sure whether the IUT will be behaviorally equivalent to the input model.

2.1.4 Formally Prove Correctness

Another approach is to formally verify each transformation rule of the code generator. For example, in the work of Blech and Glesner, an example is shown of how a proof could be done to show that transformed code preserves the semantics of the transformed programs in single-static-assignment form [4]. This relates however mostly to compiler optimizations which is not the purpose here. Nevertheless, also in our code generator, it should from a theoretical point of view be possible to prove that each transformation step preserves the semantics. Stürmer et al. point out that due to the high rate of technological innovation, language changes appear in relatively short cycles, making formal verification in practice infeasible [30]. Furthermore, there is no commonly agreed way to define the semantics of the input DSLs. The target language also often does not have a fully defined semantics. Even the generator itself can be written in a DSL without fully fledged semantics, which makes writing a proof in practice rather difficult.

2.2 Model-Based Testing

2.2.1 General Context

In this work we make use of the following definition of Model-Based Testing:

Definition 1 (Model-Based Testing)**.** Model-Based Testing is a testing technique where software is tested by using a model for both generation of testcases, and in providing a verdict whether the software passes the testcases.

The technique can be particularly helpful in finding bugs that are state dependent. It could be the case that some features of the software are not available from the initial state, but only

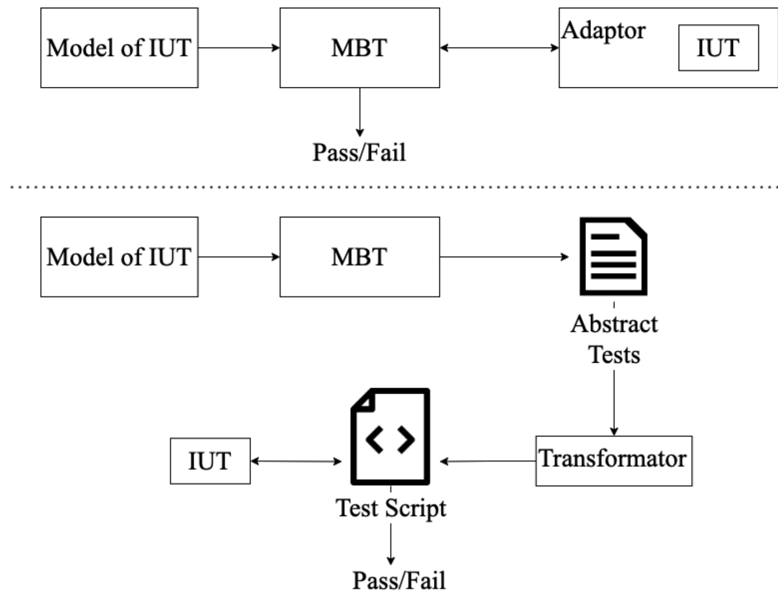


Figure 2.1: Visualization of MBT using respective Adaptor and Transformator approach.

after some inputs have been provided. Finding these kinds of bugs with unit testing is very time-consuming. One would need to manually come up with a set of inputs to get the software in the desired state. Only then it can be checked if the desired behaviour is shown in the state. Determining for the given state what the appropriate behaviour is, is not always easy. For each test case there will be time spent on coming up with a test sequence of interest, and time spent in determining what the expected behaviour of the implementation would be. The latter is referred to as the *oracle problem* [20]. The MBT tool typically solves this problem by generating test cases on the basis of the provided specification, by for example exploring the model and generating an execution trace. When the model is expressed as transition system this can for example be done by applying graph exploration algorithms on the transition system. During the exploration, the expected outputs can be observed. The observed outputs can serve as *oracles* for the generated test cases. Using a model, typically uncountably many test sequences can be derived.

The model of a system is on a higher abstraction level than the system itself. The test cases derived using MBT are therefore typically abstract test cases. The test engineer should then provide a way for these abstract test cases to become runnable on the IUT. The book of Utting and Legeard [20] points out that bridging this step is often done by a transformation tool or adaptor as seen in figure 2.1:

- Transformation tool: a program that can map an abstract test into an executable test script. For example, it maps the abstract tests to unit tests for the IUT.
- Adaptor: Some code that will serve as a wrapper around the IUT. The wrapper can receive the abstract tests as input and will be able to execute these on the IUT.

The book distinguishes between two types of deploying MBT. In *online/on-the-fly testing*, the MBT tool will be in a direct connection with the IUT and the tests will be executed during production. It is clear that in such a situation the Adaptor approach would be more appropriate. In *offline testing*, a collection of concrete testing scripts will be generated. In *offline testing*, the transformation tool approach would be more appropriate.

2.2.2 Code Generator Context

We would like to apply Model-Based Testing to test a code generator. We will consider the application of MBT to test whether a given test model is translated correctly to a GPL program

to be the *first dimension* of applying MBT on a code generator. Here MBT is used implicitly to test the code generator. We can also distinguish a *second dimension* of applying MBT to a code generator. This will be a more direct application of MBT. In particular, here the IUT is the code generator itself, which is a system that takes as input a model and produces generated code as output. The task of the MBT tool is now to come up with interesting test models, and check whether generated code conforms to the specification.

- In the work of Stürmer and Conrad, the idea of Model-Generation for testing a code generator is shown [28]. The specific part of the transformation of the code generator is considered where guarded transitions of labelled transition systems are translated into if-then-else code. To test all transformations, one would like to have complete coverage. They describe how from the Classification-Method a model can be generated that exploits all transformations. Stürmer and Conrad point out that the number of possible test models that could be derived from a classification tree is very high [29].
- In [23] the CoGenTe tool is presented. The authors have developed a tool called CoGenTe for Code Generator Testing. The tool takes as input a syntactic and semantic meta-model of the modeling language, and a test specification specifying a coverage criterion of the meta-model. The tool then produces a test suite that can be used to test code generators for this language. The CoGenTe applies the idea of the *second dimension* and can generate test models itself. To test whether a code generator produces code that conforms to the generated test model, CoGenTe applies the technique of the *first dimension*.

Successful execution of the test cases provides confidence that the translation process worked correctly. The goal is that the testers of the code generator use MBT as a testing technique to reduce the number of bugs. It is not the intention that end-users apply MBT for their created models. We desire that the end-users can rely on the code generator to translate their specifications correctly.

Chapter 3

The ASOME Software Modeling Environment

In our work, we investigate the application of Model-Based Testing on the code generator of the DMDSL language. In this chapter, we will introduce the DMDSL language and introduce the formal semantics of DMDSL models.

3.1 Introduction to DMDSL

The DMDSL language copies ideas from the Repository Pattern [19]. In this pattern, the data aspect is shared between software components by means of a shared repository that is accessible to the software components. In the DMDSL the user can make a specification that contains constructs such as *Entities* and *ValueObjects*. *Entities* are similar to classes in Object-Oriented Programming (OOP). In the dynamic semantics of the model, entities can be instantiated, giving corresponding *(Entity) Instances* which would be similar to objects in OOP. Each instance will have a unique *identifier*. For each Entity, a corresponding repository is created. An instance of an *Entity* can be stored in the repository of the corresponding *Entity*. In particular, we can do Create, Read, Update, Delete and Add (CRUD+A) operations to manipulate the content of the repositories. The *ValueObjects* can be seen as tuples of attributes and are owned by an Entity or another ValueObject. The *Composition* relation is used to denote ownership of such ValueObject. The ValueObjects can not be stored by themselves in repositories and do not have a unique identifier. Entities can also have relations with each other. We distinguish between the *Association*, and *Specialization* relation.

> Note: To be more precise, the DMDSL language allows specifying a Repository Service Specification and a Repository Service Realization. By connecting a Repository Service Specification to Domain Interfaces, the user can indicate the interfaces the Repository Service Specification should offer. It is however often the case that many realizations of such specification are possible. There can even be a single realization satisfying multiple domain interfaces, allowing for different accessibility levels dependent on the interface used. Therefore, in the Repository Service Realization, on a lower abstraction level, it can be specified how the actual implementation should implement the Repository Service Specification. One can also decide to generate a default Repository Service Realization from the Service Specification. *In our work, we will restrict ourselves to the models created within the context of a single Domain Interface. In particular, we assume the Repository Specification provides only one Domain Interface, and the default generated Repository Service Realization is used.* While it is good to know that these concepts exist, there is no strict need for the reader of to be familiar with these concepts.

3.2 Basic Specification in DMDSL

In this section, we will explain the most important concepts of the DMDSL language. We will provide a running example to assist in understanding the semantics and syntax.

3.2.1 Example

In figure 3.1 we have a simple example model containing a Dog entity and a Person entity.

Entity Multiplicity The Dog entity has a *minimum multiplicity* of 0 and a *maximum multiplicity* of 1. The Person entity has a *minimum multiplicity* of 0 and a *maximum multiplicity* of 3. This means that during execution at any time there should be at least 0 Dog instances in the repository, and at most 1 Dog instance can exist. Analogously, at least 0 Person instances should be in the repository, and at most 3 Person instances can exist.

> Note: One might wonder why the minimum multiplicity restricts the number of instances in the repository, and the maximum multiplicity bounds the existence of instances. These constraints intend to put boundaries on the number of instances in the repository. One should however be aware that already at creation time one could hit memory boundaries of the system running the software. Hence, it is desirable that the maximum constraint will already be enforced at creation time. Since the maximum constraint enforces the number of instances that can exist, it will indirectly also limit the number of instances that can exist in the repository of the corresponding entity.

Association relation Furthermore, there is an *association* relation named “owner” from the Dog entity towards the Person entity. The source multiplicity of this association relation has a *minimum source-multiplicity* of 0, and a *maximum source-multiplicity* of 1. This denotes that an arbitrary Person instance has to be the owner of at least 0 dogs, and can be the owner of at most 1 dog. The *minimum target-multiplicity* is 3, and the *maximum target-multiplicity* is 5. This means that an arbitrary Dog instance has at least 3 owners, and at most 5 owners. For concrete instances, the association relations will get *materialized.* E.g., when we create a Dog instance and specify the owners, we will materialize the ‘owner’ association relation. We will refer to the materialization of association relations as links. A Dog instance will thus have links to Person instances that are the ‘owners’ of the dog.

Cascade Deletion Note that at the source of the association relation in figure 3.1 there is a garbage bin. This symbol denotes that the *source cascade deletion* property is enabled. This means that for an arbitrary Dog Instance d that has a link towards some Person Instance p, that if p gets deleted, d will get deleted as well. Analogously there could instead be a garbage bin at the arrow of the association relation. Then, the *target cascade delete* property would be enabled. For this property, we must have that when d gets deleted all the persons that d points to with the *owner* relation (among which the p instance) get deleted as well. It is clear that for a large Domain Data Model, with many cascade delete options enabled, a single deletion could result in the deletion of many instances due to the propagation of the deletion caused by cascade deletion.

Constructability Entities can be either *Constructable* or *Unconstructable.* The construction hat of Dog in 3.1 indicates the Dog entities are *Constructable* in the Domain Interface, meaning the Entity can be instantiated. A crossed construction hat means that the Dog is *Unconstructable* in the Domain Interface. Note that an entity *Unconstructable* from one interface could actually be *Constructable* via some other interface.

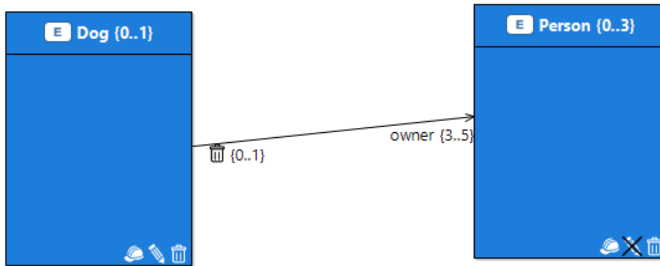


Figure 3.1: A simple data model containing a Dog and Person entity.

Mutability Entities can be either *Editable*, *Uneditable,* or *Immutable*. The pen of Dog in figure 3.1 indicates the Dog entities are *Editable* in the Domain Interface. This means that instances can be updated after addition to the repository via the interface. A pen with a single strike through denotes the entity is *Uneditable*. This means the instances of this entity can not be modified via this interface after it has been added to the repository. The entity may however be editable from another interface. A crossed pen as seen for the Person entity denotes this entity is *Immutable*. This is a stronger property than the *Uneditable* property. When an entity is *Immutable* it means it can not be edited from the interface, with the extra guarantee that there is no other interface from which these entities can be edited.

Deletability Entities can be either *Deletable*, *Undeletable,* or *Undestructable*. The garbage bin of Dog in figure 3.1 indicates the Dog entities are *Deletable* in the Domain Interface. This means that instances can be deleted from the repository via the interface. A garbage bin with a single strike through it denotes the entity is *Undeletable*. This means instances of this entity can not be deleted from the repository via this interface. It may be possible that such an instance is deleted via a different interface. A crossed garbage bin means that the Dog is *Undestructable* in the Domain Interface. This means that instances cannot be deleted by anyone once stored in the repository.

> Note: The entities may contain attributes, and own certain ValueObjects. In this way data can be stored in entity instances. These aspects are not covered in the semantics of Derasari. We will also not consider this aspect of the DMDSL language in this work. From a behavioral perspective, the features of interest are a result of association relations.

When considering the semantics in the context of a single domain interface (as we will do in this thesis) *Undeletable* and *Unconstructable* are considered to be equivalent terms in the formal semantics of the work of Derasari. Similarly, an Entity that is *Uneditable* or *Immutable* he considers to result in the same execution semantics and considers them to be equivalent terms in the context of a single domain interface.

In figure 3.2, some execution behavior allowed by the model in figure 3.1 is visualized. Instances of entities are represented as circles in the figure. Links between instances have been visualized as arrows between the circles.

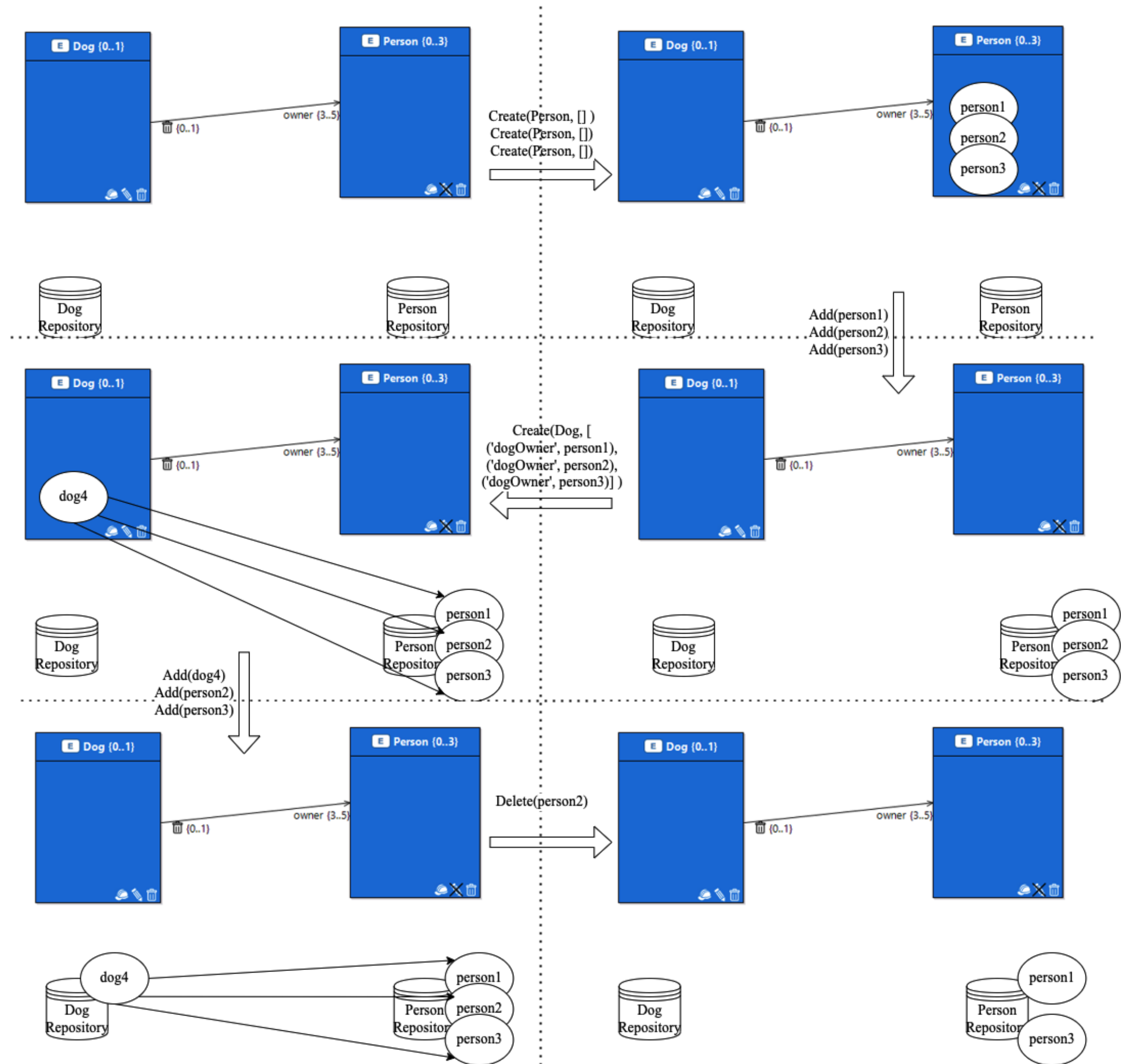


Figure 3.2: Example trace allowed by the dynamic semantics of the example model in figure 3.1.

3.3 Formal Semantics

In the thesis of Derasari, research is done on the DMDSL language [8]. There was no formal semantics of this DSL. The static and dynamic semantics of this language are given in text and diagrams. Derasari pointed out that this can lead to misconceptions between language engineers, which can lead to implementation gaps. The result of this is that static constraints in the model could be missed, or that operations in the dynamic semantics might need to be reconsidered. It is undesirable that it is possible to define models that are *inconsistent*.

Note: We will refer to the work of Derasari for the definition of an *inconsistent model* i.e a model that allows for *repository inconsistency*. An example violation of model consistency would be if during execution it would be possible for the number of instances of an entity type to be greater than the maximum multiplicity of the corresponding entity type. Derasari addressed this issue by formalizing both the static and dynamic semantics of a subset of the DMDSL language and checked with Alloy [14] that the semantics were sound w.r.t. model consistency.

In the thesis of Derasari a more detailed description of the concepts of DMDSL is provided. In section 2.2 of the thesis, the static semantics of the modeling language are described in natural language. In section 2.3 of the thesis, the dynamic semantics of DMDSL models are described in natural language on the basis of execution in terms of function calls of the C++ code. In 2.4 of the thesis, the static constraints are provided in natural language. These static constraints are defined to exclude models that can cause inconsistencies at runtime, or prevent models that allow undefined behavior. In section 3 of the thesis, these concepts are formalized. In particular, in section 3.2 the static semantics are formalized, and in section 3.6 of the thesis, the dynamic semantics are formalized. Finally, in section 4 of the thesis, the semantics are expressed in the Alloy specification language. Alloy is then used to check that the CRUD+A operations are sound with respect to repository consistency. Alloy is not sufficient as verification proof. It can only show the existence of bugs, but not the absence of bugs. The tool has been built with the *small-scope hypothesis* in mind [1]. This hypothesis states that a significant part of bugs can be found by testing a program with input in a very small scope. Assuming this holds true for ASOME models, the verification in Alloy does provide us with high confidence that the semantics are indeed sound with respect to model consistency.

3.3.1 Static Semantics

A Domain Data Model is typically specified in the DMDSL language using the diagram editor. Besides drag-and-dropping elements in the graphical interface, the architect can also set certain properties, to get the desired Domain Data Model. By drag-and-dropping elements, and setting properties, the architect can create a valid Domain Data Model, without knowing the syntax of the underlying specification language. While creating the model via the diagram editor, the underlying textual specification is adjusted. The textual specification of the model is in the .ASOME file. There are however static constraints that need to be respected by the specified models. An example of a property of an entity would be its minimum and maximum multiplicity. It is clear that a minimum multiplicity of an entity should be smaller or equal to the maximum multiplicity. To prevent the architect from creating models that violate these static constraints, the Object Constraint Language (OCL) is used. We will now briefly provide the formal semantics of the DMDSL language as specified in the work of Derasari.

In section 3 of the work of Derasari, a subset of the DMDSL language is formalized into mathematical concepts. The static semantics of an ASOME model M can be considered as a tuple $\langle Entity, Association\rangle$, where $Entity$ is the set of entity types, and $Association$ is the set of associations in the model. An Entity has certain properties. Let $e \in Entity$, then $e = \langle C, M, D, N\rangle$ for $C, M, D \in \mathbb{B}$, and $N \in \mathbb{N}\times\mathbb{N}^1$. $N = \langle minimum, maximum\rangle$ is a tuple denoting the multiplicity of the Entity e

- $C =$ **True** means the Entity is Constructable.
- $C =$ **False** mens the Entity is Unconstructable.
- $M =$ **True** means the Entity is Editable.
- $M =$ **False** means the Entity is Uneditable or Immutable.
- $D =$ **True** means the Entity is Deletable.
- $D =$ **False** means the Entity is Undeletable or Undestructable.

Again, note that by means of a boolean value, no distinguishment between Uneditable and Immutable, and Undeletable and Undestructable has been made. For models containing multiple domain interface, this does make a difference, and the mathematical representation should then be adapted. Furthermore, the semantics ignore the notion of ValueObjects, Attributes, and Inheritance. It does, however, consider the association relation between entities.

Let $a \in Assocation$, then $a = \langle source, target, SProperty, TProperty\rangle$.

- $source : Entity$: Element of $Entity$. Is the source entity of the association.
- $target : Entity$: Element of $Entity$. Is the target entity of the association.
- $SProperty, TProperty : \mathbb{B} \times (\mathbb{N} \times \mathbb{N})$: Association-end property (respectively source association property, or target association property), which is a tuple $\langle cascade, multiplicity \rangle$.
- $cascade : \mathbb{B}$: Boolean denoting the cascading deletion property of the association end.
- $multiplicity : \mathbb{N} \times \mathbb{N}$[1]: Tuple containing the multiplicity $\langle minimum, maximum \rangle$ corresponding to the association end.

We will now provide an example of a DMDSL model.

Example 3.3.1. Let $M = \langle Entity, Association \rangle$, where

$$Entity = \{Dog, Person\}$$
$$Association = \{\langle Dog, Person, \langle \mathbf{True}, (0,1) \rangle, \langle \mathbf{False}, (3,5) \rangle \rangle\}$$

$$Dog = \langle \mathbf{True}, \mathbf{False}, \mathbf{False}, (0,1) \rangle$$
$$Person = \langle \mathbf{True}, \mathbf{False}, \mathbf{False}, (0,3) \rangle$$

This example has been visualized in Figure 3.1.

In section 3.3 in the work of Derasari some well-formedness constraints are specified for these semantic models. These are not of relevance to us, since we are not the creators of the test models. We assume, that these constraints are enforced by the creator of the models. (In fact, they should be enforced by OCL in de ASOME Modeling environment).

3.3.2 Dynamic Semantics

Instances In the dynamic semantics, the entities can be instantiated. These instances can be represented as a tuple $\langle id \rangle$ where id is an element from the universe of identifiers. We leave implicit what this universe is, but this universe could for example be the set of natural numbers. The Entity (type) of an instance can be recovered via the *type* function (see state paragraph).

Links In the dynamic semantics, the associations will get materialized in the form of links. These links can be represented as a tuple $\langle instance1, association, instance2 \rangle$ where $instance1, instance2$ are elements in the universe of instances, and $association \in Association$. Furthermore, we require that $type(instance1) = association.source$, and $type(instance2) = association.target$.

States In the work of Derasari, the dynamic semantics have been formalized as a transition system. States in this transition system are currently defined by instances that exist, the contents of the repositories, and links between instances. When this description is formalized the a state of a given ASOME model M is a tuple $S = \langle I, REPO, L, type, output \rangle$ where:

- I denotes the set of instances that are created (but do not need to be stored in a repository).
- $REPO$ contains the set of instances that are stored in repositories (it follows that $REPO \subseteq I$ in any state).

[1]Strictly speaking the domain is $\mathbb{N} \times (\mathbb{N} \cup \{\infty\})$, but we will restrict ourselves to finite multiplicities due to possible memory limitations. The ASOME tool also warns when the number of entities are not bounded in their multiplicities. For association relations, the source and target multiplicities do not have such a warning. By the *small scope hypothesis*, one would expect bugs to occur already for 'small' models.

- L is a bag (i.e. multiset) of links[2]. E.g., in fig 3.1 we have the association relation 'owner'. A concrete Dog Instance can then have a link to some Person target instance. The link is the materialization of this 'owner' association relation. The mathematical notation for links is $\langle source, association, target\rangle$ with $source \in I, target \in I, association \in Association$

- $type : I \rightarrow Entity$ is the function that maps instances in I to their corresponding entity type. Between different states, instances can be created and deleted, and thus the *type* function is subject to change.

- $output \in OutputMessages$ presents the success or error output as a result of executing the action leading to the current state (initially output will contain `Success`). We have that $OutputMessages$ = {`Success`, `Entity_MultiplicityMaximum`, `Entity_Unconstructable`, `Entity_Immutable`, `Entity_Undestructable`, `Entity_Immutable`, `Entity_Undestructable`, `Entity_UnexpectedAssociation`, `Entity_MissingAssociation`, `Association_SourceMaximum`, `Association_TargetMaximum`, `Association_TargetMinimum`, `Link_TargetNotInRepository`, `Instance_NotInRepository`, `Instance_AlreadyInRepository`}

It follows that in the semantics a state is thus not only defined by the instances, instances in repositories and existing links, but also by the *type* mapping and the *output* state variable. We can, however, consider states for which $I, REPO, L, type$ are the same to be *equivalent states*. This has been explicitly encoded in Alloy to verify the semantics.

The set of transitions are $\{Create, Read, Update, Delete, Add\}$. The result of a transition is that state variables will be modified accordingly.

We will now have a look how the transitions affect the state variables. We sometimes deviate from the notation in the work of Derasari to make them easier to read, or in order to be a bit more precise.

3.3.2.1 Create

The $Create(e, links)$ action takes as argument, an Entity e, and a bag of *links*. The action tries to create an instance $new \notin s.I$, where *new* is the source of each link in *links*. If the *Create* action, can successfully be done from a given state s, and arguments e and *links*, the resulting state s' will look as follows:

$$
\begin{aligned}
s'.I &= s.I \cup \{new\} \\
s'.L &= s.L \uplus links \qquad \text{(The } \uplus \text{ is used as union operator for bags)} \\
s'.REPO &= s.REPO \\
s'.output &= \texttt{Success} \\
s'.type &= s.type[new \rightarrow e]
\end{aligned}
$$

When the action can not be successfully done, then all state variables remain unchanged, except the *output* state variable of the resulting state s' will reflect why this is the case. For the *Create* action we distinguish between the following error outputs.

- `Entity_Unconstructable`
  - Rationale: It should not be allowed to create an instance of an entity, that is not constructable.
  - Formalization: $e.C$ = **False**.
- `Entity_MultiplicityMaximum`

[2] Note that an instance can have multiple links of some association relation to the same target instance. Therefore, a bag is used, instead of a set.

  - Rationale: It should not be allowed to create an instance of an entity, of which the maximum multiplicity is already reached.
  - Formalization: $|\{i \in s.I \mid type(i) = e\}| = e.N.maximum$.

- `Entity_UnexpectedAssociation`
  - Rationale: It should not be allowed to create an instance of an entity, where one of the links materializes an association that is not an outgoing association of entity.
  - Formalization: $\{a \in Association \mid \exists_{link}[link \in links : a = link.association]\} \not\subseteq e.OutgoingAssociations$. In which $e.OutgoingAssociations := \{a \in Association \mid a.source = e\}$

- `Entity_MissingAssociation`
  - Rationale: For each 'required association' a link should be provided. We consider a 'required association' to be an outgoing association of the entity with minimum target multiplicity ≥ 1. (Note, in the work of Derasari this output message is not properly defined. It states that all outgoing associations of entity e should be a subset of the associations used in the provided link set *links*. This does not need to hold, since the target minimum multiplicity of an association relation can be 0, and then no association needs to be provided.)[3]
  - Formalization: $\{a \in Association \mid a.source = e \wedge a.TProperty.minimum \geq 1\} \not\subseteq \{link.association \in Assocation \mid link \in links\}$

- `Association_TargetMaximum`
  - Rationale: It should not be allowed to create an instance of an entity that has more association targets than allowed by the corresponding association relation.
  - Formalization: $\exists_a[a \in e.OutgoingAssociations : |\{link \in links \mid a = link.association\}| > a.TProperty.multiplicity.maximum]$.

- `Association_TargetMinimum`
  - Rationale: It should not be allowed to create an instance of an entity that has less association targets than allowed by the corresponding association relation.
  - Formalization: $\exists_a[a \in e.OutgoingAssociations : |\{link \in links \mid a = link.association\}| < a.TProperty.multiplicity.minimum]$.

- `Association_SourceMaximum`
  - Rationale: It should not be allowed to create an instance of an entity, when the result would be that one of the target instances would now have too many incoming links for one of its incoming association relations.
  - Formalization: $\exists_{a,i}[a \in e.OutgoingAssociations \wedge i \in s.Instances \wedge i.Entity = a.target.Entity : |i.incomingLinks(a) \uplus links(a,i)|] > a.SProperty.multiplicity.maximum$,

    where

$$\begin{aligned} i.incomingLinks(a) &:= \{link \in s.L \mid link.association = a \wedge link.target = i\} \\ links(a,i) &:= \{link \in links \mid link.association = a \wedge link.target = i\} \end{aligned}$$

[3]The careful reader may already spot this, but `Association_TargetMinimum` actually implies `Entity_MissingAssociation`. During the application of MBT, this will also become clear.

3.3.2.2 Read

The *Read* action has not concretely been formalized in the work of Derasari. Fortunately, it is an easy operation and the formalization is rather simple. The $Read(identifier)$ action takes as argument an id $identifier$. If the *Read* action, can successfully be done, it means that an instance in the repository exists, of which the corresponding id is $identifier$. From a given state s, and argument $identifier$ the variables of s' will look as follows:

$$\begin{aligned} s'.I &= s.I \\ s'.L &= s.L \\ s'.REPO &= s.REPO \\ s'.output &= \texttt{Success} \\ s'.type &= s.type \end{aligned}$$

When the action can not be successfully done, then all state variables remain unchanged, except the *output* state variable will reflect why this is the case. For the *Read* action, we distinguish between the following error outputs.

- `Instance_NotInRepository`
  - Rationale: It should not be possible to read from the repository using an identifier of which there is no instance in the repository that has such an identifier.
  - Formalization: $\neg\exists_x[x \in s.REPO : x.id = identifier]$.

3.3.2.3 Update($instance, links$)

The $Update(instance, links)$ action takes as arguments, an instance, and a bag of $links$. If the *Update* action, can successfully be done, the outgoing links of the *instance* will be replaced by $links$. From a given state s, and arguments $identifier$ and $links$, s' will look as follows:

$$\begin{aligned} s'.I &= s.I \\ s'.L &= (s.L \setminus \{link \in s.L \mid link.source = instance\}) \uplus links \\ s'.REPO &= s.REPO \\ s'.output &= \texttt{Success} \\ s'.type &= s.type \end{aligned}$$

Note: We slightly deviate from the formalization done in the work of Derasari. In his work, it was assumed that *links* only contained links corresponding to a *single* outgoing association, and only replaces those links. Those semantics would match better with the actual IUT. In our *Update* function, we assume that *links* replaces *all* links in $s.L$ in which *instance* is the source. This makes the function more general and creates more overlap with the Create operation, in which *links* also contains *all* outgoing links of *instance*.

When the action can not be successfully done, then all state variables except for the *output* state variable remain unchanged. The *output* variable will reflect why the *Update* action could not be done successfully. For the *Update* action, we distinguish between the following error outputs.

- `Entity_UpdateImmutableType`
  - Rationale: It should not be possible to modify an entity that is not mutable.
  - Formalization: $instance.Entity.M = \textbf{False}$.

- `Link_TargetNotInRepository`
  - Rationale: It should not be possible to make an instance (in the repository) point to a target instance that is not in the repository.
  - Formalization: $\exists_l[l \in links : l.source = instance \wedge l.target \notin s.REPO]$
- `Association_TargetMinimum`
  - Rationale: It should not be allowed for an instance of an entity to exist, that has less association targets than allowed by the corresponding association relation.
  - Formalization: $\exists_a[a \in instance.Entity.OutgoingAssociations : |\{link \in links \mid a = link.association\}| < a.TProperty.multiplicity.minimum]$.
- `Association_TargetMaximum`
  - Rationale: It should not be allowed for an instance of an entity to exist, that has more association targets than allowed by the corresponding association relation.
  - Formalization: $\exists_a[a \in instance.Entity.OutgoingAssociations : |\{link \in Links \mid a = link.association\}| > a.TProperty.multiplicity.maximum]$.
- `Association_SourceMaximum`
  - Rationale: It should not be allowed to update an instance of an entity, when the result would be that one of the target instances would now have too many incoming links of one of the association relations.
  - Formalization: $\exists_{a,i}[a \in e.OutgoingAssociations \wedge i \in s.REPO \wedge i.Entity = a.target.Entity : |i.incomingLinks(a) \uplus links(a,i)|] > a.SProperty.multiplicity.maximum$.

    where

$$\begin{aligned} i.incomingLinks(a) &:= \{link \in s.L \mid link.association = a \wedge link.target = i\} \\ links(a,i) &:= \{link \in links \mid link.association = a \wedge link.target = i\} \end{aligned}$$

3.3.2.4 Delete

The $Delete(identifier)$ action takes as an argument, an identifier $identifier$. If the $Delete$ action, can successfully be done, the instances with id $identifier$ will be removed from its repository. Furthermore, instances that should also be deleted (due to cascade deletion) should also be removed from the repository. We will denote the set of instances that will be removed with $deletion_set$. From a given state s, using argument $identifier$, the variables of s' will look as follows:

$$\begin{aligned} s'.I &= s.I \setminus deletion_set \\ s'.L &= s.L \setminus \{link \in s.L \mid \exists_{so}[so \in deletion_set \mid s.L.source = so \vee s.L.target = so]\} \\ s'.REPO &= s.REPO \setminus deletion_set \\ s'.output &= \texttt{Success} \\ s'.type &= \{i \mapsto s.type(i) \mid i \in s.I\} \end{aligned}$$

In which $deletetion_set$ will be defined as follows:

$$\begin{aligned} R^0 &:= \{instance\} \text{ (where } instance \in s.I \wedge instance.id = identifier) \\ R^{n+1} &:= \{i \in s.I \mid \exists_l[l \in s.L : l.source = i \wedge l.target \in R^n \wedge l.SProperty.cascade = \textbf{True}] \\ &\vee \exists_l[l \in s.L : l.target = i \wedge l.source \in R^n \wedge l.TProperty.cascade = \textbf{True}]\} \quad \text{(for } n \in \mathbb{N}) \end{aligned}$$

$$deletion_set := \bigcup_{i=0}^{\infty} R^i$$

The $deletion_set$ denotes the instances that get deleted as a result. Note that applying a deletion using $identifier$ can have many deletions as a result due to the cascade deletion properties. The $deletion_set$ is in fact a sort of transitive closure on the link set. In the work of Derasari, this closure has not been formalized. To be explicit, we have provided the exact definition of $deletion_set$.

When the action can not be successfully done, then all state variables remain unchanged, except the *output* state variable will reflect why this is the case. For the *delete* action, we distinguish between the following error outputs.

- `Instance_NotInRepository`
  - Rationale: It should not be allowed to perform delete on an identifier if there is no corresponding instance for it in the repository.
  - Formalization: $\neg\exists_{instance}[instance \in s.REPO : instance.id = identifier]$
- `Entity_DeleteOnUndestructable`
  - Rationale: It should not be allowed to perform delete on an identifier if the corresponding entity is not Deletable.
  - Formalization: $\exists_y[y \in deletion_set : y.D = \textbf{False}]$].
- `Entity_MultiplicityMinimum`
  - Rationale: It should not be allowed to perform delete on an identifier if the result would be that after deletion, the entity minimum multiplicities would be harmed by removing the $deletion_set$.
  - Formalization: $|\{i \in s.REPO \mid i.Entity = instance.Entity\} \setminus \{i \in deletion_set \mid i.Entity = instance.Entity\}| < instance.Entity.multiplicity.minimum$. where $instance \in s.REPO$ for which $instance.id = identifier$
  - One should not create models where the minimum multiplicity of an entity is > 0, where the entity also has an outgoing association relation with source cascade deletion enabled. This restriction is enforced by OCL.

3.3.2.5 Add

The $Add(instance)$ action takes as an argument, instance $instance$. If the Add action, can successfully be done, $instance$ will be added to its repository. From a given state s, and argument $instance$ the variables of s' will look as follows:

$$\begin{aligned} s'.I &= s.I \\ s'.L &= s.L \\ s'.REPO &= s.REPO \cup \{instance\} \\ s'.output &= \texttt{Success} \\ s'.type &= s.type \end{aligned}$$

When the action can not be successfully done, then all state variables remain unchanged, except the *output* state variable will reflect why this is the case. For the *Add* action we distinguish between the following error outputs.

- `Instance_AlreadyInRepository`
  - Rationale: It should not be possible to add an instance to the repository if it is already in there.

  - Formalization: $instance \in s.REPO$

- `Link_TargetNotInRepository`

  - Rationale: It should not be possible to add an instance to the repository if it has a link to an instance that is not yet in the repository.
  - Formalization: $\exists_{link}[link \in s.L : link.source = instance \wedge link.target \notin s.REPO]$

Chapter 4

Model-Based Testing in the Context of DMDSL

In this chapter, we will identify the aspects that are involved in applying Model-Based Testing to the DMDSL code generator. In Figure 4.1, the architecture of applying MBT on a code generator is visualized. The concepts involved will be explained in the sections of this chapter.

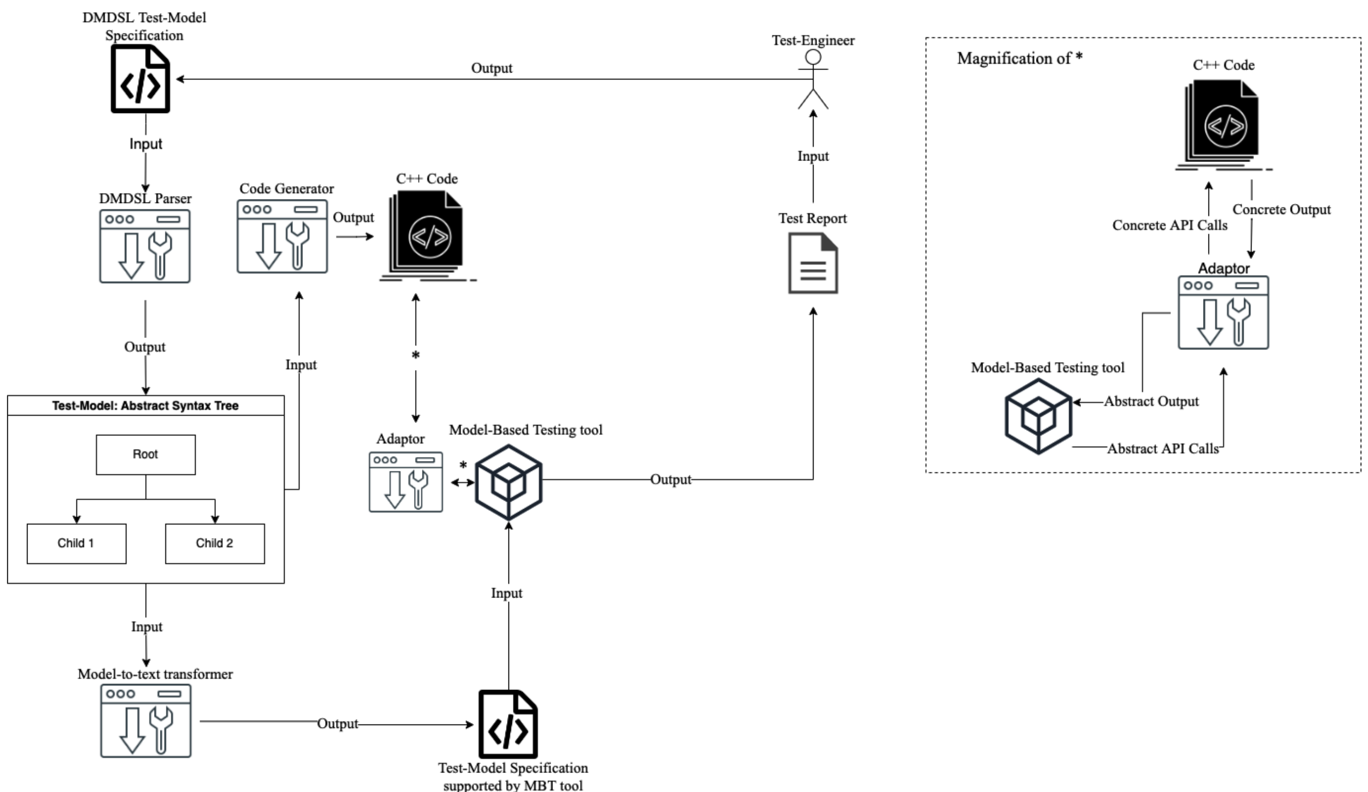


Figure 4.1: Architecture of applying Model-Based Testing on the DMDSL code-generator.

4.1 DMDSL Testing Models

To apply MBT on the code generator of the DMDSL language, we will apply the *first dimension* of MBT as introduced in section 2.2.2. There is also literature available that chooses the *second*

dimension of applying MBT on code generator. In [7] research is done on the generation of input test models where also the DMDSL code generator is used as a case study. They surveyed a number of academic tools for this and concluded that the underlying fundamentals are solid and useful but still additional work is needed to bring them to the required quality for industrial application. In this work, we will consider test models as given. The research required to generate test models automatically will be out of the scope of this work.

Since we apply the *first dimension* of applying MBT to test code generators, we will create testcases that test whether the generated code conforms to the model, and in this way implicitly test the code generator. In the literature, the implementation is often referred to as the System Under Test (SUT) or Implementation Under Test (IUT). We will consistently use the notation IUT to refer to the generated C++ code. The system that is implicitly tested using MBT is actually the code generator itself. The term SUT, therefore seems more appropriate for the code generator itself. To avoid confusion we will avoid the term SUT.

4.2 Making DMDSL Models MBT Compatible

Unfortunately, DMDSL is not a specification language supported by an MBT tool. In order to apply an existing MBT tool, we need to transform the test model written in DMDSL to a specification language supported by an MBT tool (translational approach). To do this process automatically, we need to create a *model-to-text* transformation. The DMDSL language is built with the Eclipse Modeling Framework (EMF). A metamodel of the DMDSL language has been built using EMF. A DMDSL Model is then simply an instantiation of the metamodel of the DMDSL language constructs.

Xtext is a DSL to describe the concrete syntax of a language, and how it is mapped to an in-memory representation. The Xtext language is used to define the grammar of the DMDSL language. After describing the grammar in Xtext, a parser of the DMDSL language is automatically provided. After parsing a textual specification in the DMDSL language, the parser can give a corresponding Abstract Syntax Tree (AST), which is in fact the instantiation of the DMDSL metamodel. This AST is referred to as the (ecore) model. It is the model as described in the textual specification in the DMDSL language. To transform a DMDSL specification to a textual specification, we then simply need to read the elements of interest from the model, and transform it to the textual specification of the target language. Since we only cover part of the DMDSL language, it is expected that we will use a target driven transformation here. The resulting textual specification should encode the behavioral semantics of the input DMDSL model in the target specification language. The tasks we obtained are (1) *express the semantics of the DMDSL into the target specification language.* (2) *Develop a tool to do these transformations automatically.*

Domain Data Models are not typical models to use for MBT. A more typical example would be a coffee machine. As input, this machine accepts coins and notices button presses. As an output, this machine can produce different kinds of beverages. In such a system we can identify clear input and output actions, and the execution of these actions would result in state changes. It is clear that we are dealing with a reactive system here. Such a reactive system can be semantically represented by a Labelled Transition System or some other type of state-dependent formalism. In our setting of Domain Data Models, we are focussed on the processing of data using the repository pattern [19]. In this pattern we are dealing with Create, Read, Update, Delete and Add actions (CRUD+A). The corresponding semantic concepts for the application of MBT are not obvious. From the work of Derasari, it has been made clear that these Domain Data Models do in fact also have clear inputs and outputs (see section 3.3.2). The inputs are the CRUD+A actions, and the outputs are success/corresponding error message after the operation. One could wonder whether this is sufficient for MBT. When an Add action is taken, and `Success` is reported as expected, can we actually be sure that the implementation added the instance to the repository? We can not immediately be sure. However, when later in the test sequence a Read action is done using the identifier of the added instance we would expect a `Success` to occur. When a Read action is done with an identifier from an instance that is not in the repository, we would expect a corresponding

error message. The effect of executing a CRUD+A action affects the output of future actions. When the output actions during a test sequence are consistently as expected, one builds more and more confidence that these operations are indeed performed as reported by the output messages. We can thus in fact consider the IUT of a Domain Data Model to be a reactive system, and use popular MBT techniques that apply to reactive systems.

4.3 MBT Tool Evaluation

That test sequences that the MBT tool creates contain actions with corresponding input arguments. In our context that would be for example the type of operation (CRUD+A), but also the arguments for the operation such as the instance identifier needed in a deletion. The MBT tool should in some way generate these action arguments. We should, however, be careful that these will be sensible. For example, when an instance of entity type E is created with some identifier i, it is probably desirable to also test deleting an instance with identifier i later in the test sequence. We should thus be careful in the selection of the MBT tool so that it supports the generation of sensible test cases that are sufficiently sophisticated. This goes hand in hand with the supported constructs of the specification language of the MBT tool. Furthermore, the tool should be sufficiently scalable to handle the transformed DMDSL models. The following task emerges: *(3) Make an assessment of which MBT tools are appropriate in the context of DMDSL*.

4.4 Adaptor

Since the MBT tool will create test cases on the basis of a high-level DMDSL specification, the MBT tool will only be able to generate test cases on a higher abstraction level. In the context of DMDSL, this will be a sequence of CRUD+A actions with corresponding arguments. These abstract actions should be turned into concrete method calls on the implementation. Furthermore, the concrete outputs the IUT produces, such as exceptions, should be transformed back into the abstract DMDSL output messages as defined in section 3.3.2. We should therefore create an adaptor. The following task emerges: *(4) Develop an adaptor to translate abstract tests to executable tests for the generated C++ code.*

4.5 Test Report

When a test model is given, an MBT tool is selected, a tool is developed to transform the test-model in a specification language supported by the MBT tool, and an adaptor is developed. The MBT tool can then be used to test the generated code of the test model (IUT). It is not unlikely some problems will pop up, as a result of the exhaustive testing. These problems need to be manually investigated. A problem does not necessarily mean that there is a bug in the code generator:

- It could be the case that there is some error in transforming the model to the specification language supported by the MBT tool. Then, the model-to-text transformer needs to be changed.

- It could also be the case that the Adaptor, did not handle some input properly. Then, the adaptor needs to be changed.

> Note: The fact that in both the adaptor and translated model bugs can occur seems problematic. However, since the code generator is developed in isolation from the adaptor and generator, it is not typical that the exact same bug occurs in the generated implementation and in the generated specification and adaptor for the MBT tool (given the dynamic semantics are properly defined). Therefore, problems of the adaptor or translated model will typically be exposed in the test report, allowing the tester to fix them.

- Another problem that could occur is that there was some misconception about the semantics of the modeling language. For example, there could be misconceptions about what a property of the test model actually means. This could result in execution semantics of the IUT that are different from the dynamic semantics used in the MBT for the test model. Nevertheless, it is good that MBT can point out these misconceptions, so that discussion between the developers and testers occurs and forces them to refine the semantics of the language. Perhaps even some constraints need to be added, and the test model is a type of model that we actually would like to reject.
- Finally, it could be the case that there is some actual problem with the implementation. The developers should then be made aware of this so that they can inspect the test report, and find the cause of the bug. The test report could for example show a trace that would lead to incorrect behavior of the IUT. The developer should inspect such a trace, find the cause of the bug, and fix the bug. After fixing the bug the MBT testing process can be continued. If it is not feasible to fix the bug in time, and it is desirable to continue the MBT process, one could try to restrict the exploration of the model by for example avoiding traces where this specific bug occurs, to try and find different types of bugs.

Chapter 5

Selecting a Model-Based Testing Tool

In chapter 4, we explained an approach for applying MBT on a code generator. An essential piece in this puzzle is the MBT tool itself. There are many different MBT tools available. It is important to select an MBT tool appropriate for our setting. In this chapter, we will investigate how to select a proper MBT tool. We will first consider popular formalisms behind MBT tools. Then, for formalisms that seem most applicable, we will investigate the underlying theory and consider popular tools that rely on these. Finally, we will experiment to see if it is possible to encode the dynamic DMDSL semantics in the underlying specification language and assess whether the tools are sufficiently scalable to capture more complex DMDSL models.

5.1 Approach

In this section, we will research how we can select an MBT tool for the DMDSL context. We will start off by considering existing literature on this topic, and continue with a proposal of an approach used in the DMDSL context.

5.1.1 A Systematic Review of Tools

The selection of an MBT tool is important. One MBT tool could be better for a use case than some other MBT tool. Unfortunately, in literature where MBT is applied, the selection procedure of the MBT tool is usually left out. In the paper of Shafique and Labiche an attempt at a systematic review of nine prominent MBT tools is done on tools that support state-based models[1] [25]. They point out the many difficulties in this process. (1) There are widely varying modeling notations, and the tool capabilities are often very dependent on the input language. Therefore, they restrict themselves to state-based models. (2) Even when restricting oneself to state-based models there are still many different constructs: Finite State Machine, Extended Finite State Machines, UML state machines, Labeled Transition Systems, and more, making tools hard to compare. (3) Furthermore, they state that academic and open source tools are often outdated and contain incomplete documentation. On the contrary commercial tools often have up-to-date manuals, but vendors often do not provide enough technical information. Since the vendors have a profit motive, it is not unsurprising the vendors do not publish research. (4) They also conclude that there is a gap between MBT tool support and research on MBT since supposedly a large part of research in this field does not translate into tools.

The authors created four groups of criteria in their evaluation:

[1] They consider both transition-based notation (e.g. Labelled Transitions Systems, Finite State Machines), but also pre/post notation (e.g. Abstract State Machines) to be state-based models.

1. Model-flow criteria: Adequacy criteria on how test cases are built. Examples of these are state coverage and transition coverage.

2. Script-flow criteria: If the modeling notation allows for mechanisms that go beyond state machines, such as pre- and post-conditions, we may use traditional test criteria such as 'modified condition/decision coverage' (MC/DC) to specify coverage of these code segments.

3. Data criteria: Criteria referring to the selection of input values for arguments in abstract tests cases. They consider criteria such as boundary-value testing.

4. Requirement criteria: Criteria that rely on traceability between model elements and requirements.

It is good to be aware of these criteria. In the context of the DMDSL code generator, the test-models are given, so we are not so much interested in the requirements criteria. The data criteria are of particular interest since the actions are very data-driven. It should be noted, however, that the domains over which we operate are from a theoretical point of view often of infinite size. Furthermore, from the semantics described in section 3.3.2, it does not become clear how criteria such as boundary-value testing can be applied to such models. In particular, the transitions of the transitions systems do not have guards where boundary value testing is usually applied on. It depends on how a DMDSL model is translated to a different specification language if the defined data criteria are really applicable. The script-flow criteria are more of relevance. The CRUD+A actions have quite a complexity hidden in the calculation of resulting state variables. Unfortunately, only one of the nine tools, has some serious support for control-flow criteria. The remaining, either do not support the criteria, or the modeling notation does not involve any scripts. Unfortunately of the nine tools covered, five are no longer available. Some of the commercial tools covered are still available, but as the paper is from 2010, it is uncertain to what extent the testing tools of these commercial companies still follow the original approach. Finally, the Model-flow criteria is of interest to get a notion of a good coverage. Also, here the selection based on this criteria is quite tough. It is not yet clear how the DMDSL model will translate to a model supported by the MBT tool. It is thus not clear what kind of coverage criteria will apply to the resulting model. What if the resulting model has an infinite state space or an infinite number of transitions? Is a notion of state coverage or transition coverage in such a situation even useful? Shafique and Labiche put the result of their tool evaluations in several tables. In the different tables one can observe that a significant part of the cells is filled either with N (does not support the criterion), N/A (not of relevance), ? (simply unknown whether it is supported). It shows how inherently hard it is to come up with good general criteria to compare MBT solutions with each other. Furthermore, it is hard to assess to what extent the criteria apply and to what extent they are relevant if it is not clear how the models will be expressed in the underlying specification language.

5.1.2 Procedure Used in DMDSL context

Shafique and Labiche have taken a systematic approach in the review of MBT tools. We have seen that it is quite hard to apply their result in practice in the selection of an MBT tool. We will take a more practical approach in the selection of an MBT tool (in the context of the DMDSL language). The steps we take are as follows:

1. We have a look at popular formalisms used in Model-Based Testing for the encoding of the models. Then we select a few of formalisms that seem most promising for expressing DMDSL models. This relies on the concepts the formalisms provide, e.g. does it allow for an infinite number of states, does it allow for symbolic treatment of data.

2. We have a look at the general concepts and definitions used in formal theories of Model-Based Testing.

3. For the selected formalism, we look at the underlying theory an the tools used in the application of MBT. This should give more confidence whether the formalism is appropriate for applying Model-Based Testing. For example, to what extent is non-determinism or partial specification supported by the underlying testing theory? It is also good in the assessment to understand how tools can derive test sequences and verdicts to assess whether the formalism is appropriate. We do the following:

   (a) We try to manually describe how DMDSL models can be translated *to the selected formalism.*

   (b) We try out popular MBT tools that rely on the selected formalism and testing theory. In particular, we try to translate constructs of test models in the *underlying specification language.* It may be the case that the underlying formalism from a theoretical point of view is more expressive than the specification language used by the MBT tool is.

   (c) Assess how the Model-Based Testing tool could explore the model and assess whether this can be done for models on the intended complexity level.

The selection of the appropriate formalism and which specification language is used will mostly depend on answering 'How to express feature X in semantics Y using specification language Z?'. We will now start applying this procedure to find an MBT tool that is applicable in DMDSL context.

5.2 Formalisms in Model-Based Testing

In this section, we will have a look at the different formalisms to specify models. Then, we will consider the procedure of selecting formalisms that are good for capturing the dynamic semantics of DMDSL test models in an MBT context.

5.2.1 Popular Formalisms

In Chapter 3.3.2 the dynamic semantics of the DMDSL have been formalized into some transition system. The textual specifications that are supported by Model-Based Testing tools are often influenced by a mathematical formalism. These typically will be different from the formalism as specified in the work of Derasari. Fortunately, there are formalisms that are rather close. To assess which MBT tools are appropriate for our context, it is sensible to list some of the popular formalisms. Based on different literature, we constructed a (non-exhaustive) list of popular formalisms used in MBT and a short description.

Finite State Machine (FSM) (with Input and Output) (Mealy Machines) In testing theories, one of the simpler concepts is the Finite State Machines with input and output, which are also known as *mealy machines* [20]. In the classical setting, we have that input and output strictly alternate. That is, we have that output is produced on the basis of input and previous state. Hence, on the basis of some input, a transition is fired in which output is produced, and consecutive state is reached. Furthermore, in Finite State Machines, the transitions must be input-enabled and the number of states and transitions should be finite. The fact that it is input-enabled does not harm expressiveness. Typically, unexpected inputs would be modeled as self-loops. It is usually required that the FSM is *deterministic.* That is, every outgoing transition for a state has a unique input label. Many of the test generation algorithms on FSMs require non-determinism of the FSM. It may however be desirable to have a notion of non-determinism in the modeling, since it allows for specification on a higher abstraction level. In such case, the FSM would probably not be a desirable formalism to use. An example of an MBT tool that relies on FSMs is the TestOptimal tool (`https://testoptimal.com/`).

Symbolic Input/Output Finite State Machine In this type of state machines, the transitions can have some input argument from a possibly infinite domain [22]. By specifying guards, a transition can be enabled or disabled, on the basis of the provided input. Furthermore, the output produced may depend on the input. Note, however, that the states do not have state variables (in particular we have a finite state space), and the input only influences the output and next state.

Extended Finite State Machines The Extended Finite State Machine (EFSM) builds upon the notion of these Symbolic Input Finite State Machine with Output. In these state machines, the state space can be greatly reduced by assigning state variables to the states of the system. On the basis of the test objective, one can decide to partition the state space into interesting areas and represent these as states in the diagram. E.g. $state_1$ when state variable $x < 1$ and $state_2$ when state variable $x \geq 1$. In the EFSM the transitions can have guards so that the desired behavior can be modeled. In the example, we could have transitions $state_1 \xrightarrow{increaseX/\{x:=x+1\}/[x<1]} state_1$ and $state_1 \xrightarrow{increaseX/\{x:=x+1\}/[x\geq1]} state_2$. In such representation, $state_1$ and $state_2$ are representatives for a much larger state space. An example of an MBT tool that relies on this formalism is the ModelJUnit Library [20].

Labelled Transition System (LTS) with Input and Output These are systems containing a set of states, and labelled (input/output) transitions between the states. Hence, it may be possible that, from a single state, multiple outputs are possible, allowing observable output non-determinism. It is also possible that from a state, there are transitions leading to different states, but share the same input/output label, which is also a form of non-determinism. Furthermore, states do not need to accept all transitions, allowing for *partiality.* Unfortunately, test generation for FSMs does not apply to LTSs, but work on test generation is done [33]. An example of an MBT tool that relies on this formalism is the TorX tool [31].

Symbolic Transition System (STS) with Input and Output The LTS with input and output has been extended to a symbolic formalism. In this formalism, the labelled transitions may have input arguments. Furthermore, the transitions may have guards, and the states can have state variables. By this extension, the STSs have some similarities with the EFSM. Note, however, that the STSs allow for *partiality*, and *non-determinism* in both output and input actions. Furthermore, input actions do not directly produce output. Examples of MBT tools that rely on this formalism are the TorXakis tool [32] and the Axini tool (`www.axini.com`).

Extended Finite Automaton / UML State machine A popular formalism in MBT are UML State machines. These are an extension of a finite automaton expressed in UML notation. Typically these transition systems have an initial state and a final state. Furthermore, there are transition arrows between the states. The transitions have a trigger, constraint, and transitional behavior. It follows that there is a large similarity with EFSMs. Something exceptional about UML State Machines is that they allow for the clustering of states in a *superstate.* An example of an MBT tool that relies on this formalism is the MoMuT testing tool [18].

Abstract State Machine The basic Abstract State Machine (ASM), which was formerly known as *evolving algebras*, is a finite set of *transition rules* of the form **if** *Condition* **then** *Updates* which transforms abstract states. The introduction and first use of evolving algebras in a paper were in 1988 by Yuri Gurevich [11]. It was introduced as ‘a computation model that is more powerful and more universal than standard computation models’. The ASM provides a formal method to describe an algorithm by providing a mathematical view of a program state. Some of the current Model-Based Testing tools make use of Model Programs that are inspired by the ideas of ASMs. Examples of MBT tools that make use of Model Programs inspired by ASMs are Spec Explorer [34], NModel [15], and PyModel [16].

5.2.2 Selecting a Formalism

The book 'Practical Model-Based Testing' [20] lists *pre/post (or state-based) notation* and *transition-based notation* as the most popular paradigms. In the former, the system is modeled as a collection of variables, representing the internal state of the system, plus operations modifying those variables. A corresponding mathematical formalism would be the *Abstract State Machine*. In the latter, the focus is on describing the *transitions* between different states (modeled as locations) of the systems. The remaining formalisms introduced in 5.2.1 (FSMs, LTSs, etc.) would be more close to this category. As a basic guideline, the book points out that pre/post notations are best for *data-oriented* systems. The transition-based notation is considered best for control-oriented systems. It is not always clear if the system is more data-oriented than control-oriented. Furthermore, there is not always such a clear distinction between the two paradigms. In particular, the state-based notation can often also be used to model transition systems. One could use some state variable that indicates the node considered. The enabledness of outgoing transitions could then simply be made dependent on this node state variable. In this way, it would be possible to express a model that originally uses transition-based notation using state-based notation. On the other hand, the transition-based notation may also support more complex data structures. We have seen this for example in EFSM, and in STSs, allowing these structures also to operate over arbitrary data structures.

Since the DMDSL language covers the data aspect of systems, it is clear that the type of models we are considering are more *data-oriented*. It, therefore, seems best to focus on the formalisms found in the *pre/post notation* and the *transition-based* structures that have sufficient support for complex data structures. When looking at the dynamic semantics as defined by Derasari, we consider a transition system, with state variables. Each state has the outgoing CRUD+A transitions, and output is represented by changing the output state variable. Since it is clearly defined how the actions manipulate the state variables, we would not expect it to be hard to express such a system as Model Program (ASM), we will conceptually describe how this could be done. Furthermore, we expect it will also be possible to encode a DMDSL model in a Transition-Based Notation that supports complex data structures such as a Symbolic Transition System or Extended Finite State Machine. We will try to encode the model as an STS with input and output. This formalism has in comparison to EFSMs a richer underlying formal theory. In practice, EFSMs are actually more close to Model Programs. If the research in STSs shows that having locations is beneficial, we may decide to research the translation to EFSMs as well. We use the semantics defined by Derasari, as a middle-ground, and transfer these concepts into the specification language that underlies corresponding formalisms.

5.3 General Formal Concepts

It depends on the semantics of the input specifications supported by the MBT tool how an MBT tool can derive verdicts for test cases. Constructs such as non-determinism and symbolic actions can make this process non-trivial. Formal theories have been developed that influence how to do analysis, derive verdicts from a specification, and how one could derive test cases. These theories differ in the underlying mathematical formalism, which accordingly influences the specification supported by the MBT. Since it is important for us to assess what kind of MBT tool will be most appropriate for our context setting, we will investigate some of the formal theories behind MBT that are relevant. We will start of with the general concepts seen in MBT theories.

5.3.1 Formal Conformance

We wish to know whether the IUT implements the provided specification. The IUT is, however, not a formal object. As a matter of fact, in practice, the implementation could be a physical device to which only input can be provided, and output can be observed. Hence, the IUT will serve as a *black box*.

We assume there is some universe $\mathcal{SPECS}$ of specifications, a universe $\mathcal{IMPS}$ of implementations. A conformance relation *conforms-to* $\subseteq \mathcal{IMPS} \times \mathcal{SPECS}$ then expresses whether an IUT conforms to a specification, i.e. IUT is a correct implementation of the specification.

As mentioned, the IUT is not a formal object and is considered to be a black box. Still, we would like to be able to formally reason about the implementation. Hence, in the literature, it is assumed that for an implementation IUT there exists a formal model of the implementation $i_{IUT} \in \mathcal{MODS}$ in which $\mathcal{MODS}$ is the universe of formal models. This is referred to as the *test hypothesis*. This assumption allows us to formalize the *conforms-to* relation. The formalization is now **imp** $\subseteq \mathcal{MODS} \times \mathcal{SPECS}$ where IUT *conforms-to* S if and only if i_{IUT} **imp** S

An MBT tool should use S to generate testcases. Based on the observations of the IUT the MBT tool should be able to assess whether the implementation passes a test case T or fails the testcase. We say that a test suite T_S (set of testcases) is passed if the implementation passes all testcases in it. We desire to generate a test suite T_S such that

$$IUT \textit{ conforms-to } S \text{ if and only if } i_{iut} \textbf{ passes } T_S$$

Such a test suite T_S would be a *complete* test suite. It will be able to reject all implementations that do not satisfy the specification. Furthermore, all implementations that implement the specification should pass the testcase. In practice, such test suite T_S would unfortunately be of infinite size. We will provide an example that even for very simple systems, such a test suite would be of infinite size.

Example 5.3.1 (Complete test suites are often of infinite size)**.** Suppose a specification S of a light switch is provided in CCS [21] with as input actions $\{toggle?\}$ and output actions $\{lightOff!, lightOn!\}$. The specification S looks as follows:

$$LightSwitch := toggle?.lightOff!.toggle?.lightOn!.LightSwitch$$

The corresponding Labelled Transition System is visualized in figure 5.1. To assess whether the IUT shows the desired behavior, we would like to generate testcases. It might be the case that for first input $toggle?$ we indeed observe from the implementation output $lightOff!$, and for second input $toggle?$ output $lightOn!$. However, perhaps the implementation will fail to provide the correct output after executing this input sequence 100 times (or more). The expected output after 100 toggles would be $lightOff!$, but since the IUT is considered to be a black box we can not be sure if the output would be correct without testing this sequence. It is clear that the specification has only a few states, but it is unknown to us how many possible states i_{IUT} has. To reject all faulty implementations of a $LightSwitch$, we would need a test suite of infinite size.

We can split the notion of *completeness* up into two separate conditions:

Definition 2 (Definition of completeness, soundness, and exhaustiveness)**.** A test-suite T_S is complete *if and only if* T_S is both a sound and exhaustive test-suite. Soundness and completeness are defined as follows:
Soundness:

$$\text{if } IUT \textit{ conforms-to } S \text{ then } i_{iut} \textbf{ passes } T_S$$

Exhaustiveness:

$$i_{iut} \textbf{ passes } T_S \text{ then } IUT \textit{ conforms-to } S$$

Since completeness is in practice not achievable when considering the IUT as black box, but we do desire that correct implementations pass the testcases, we desire the generated test suites to be *sound*.

Now that we have seen the general concepts in testing theories, we can start exploring some testing theories seen in MBT, and see if these are applicable in DMDSL context.

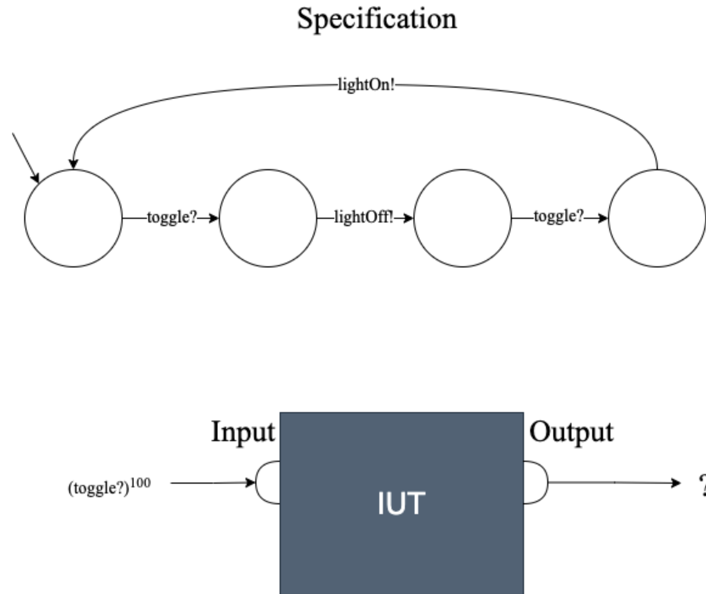


Figure 5.1: Example of specification, and IUT represented as black-box.

5.4 Model-Based Testing with Labelled Transition Systems

It would be interesting to see if we are able to express the DMDSL models as STS. MBT applied on STSs builds on the testing theory used in LTSs. The STSs can in fact be semantically represented as an LTS. This allows the conformance relation **ioco** that is used on LTSs to be used as conformance relation for MBT using STSs. We will therefore first have a look at LTSs and the **ioco** conformance relation, then we will see how STSs are defined and have LTS semantics. Afterwards, we will from a conceptual point of view try to express a DMDSL model as an STS, and try to encode such a model in a specification language. In particular, the commercial Axini tool, and the academic TorXakis tool and their specification languages will be considered. It will turn out that we will get a negative result for both tools in terms of practical application to DMDSL models. As a result, this subsection will not be of particular interest for the remainder of the thesis. This subsection may still be of interest to the reader to see how an extensive testing theory is applied to the case study.

5.4.1 IOCO on Labelled Transition Systems

We will provide a brief repetition of the **ioco** theory on Labelled Transitions as seen in the work of Tretmans [33]. This formal theory forms the basis of several different MBT tools. To assess whether the theory applies to our setting some knowledge of this theory is required.

The **ioco** relation is a conformance relation defined on Labelled Transition Systems with Input and Output. LTSs allow for capturing non-determinism and partiality in the specifications. Sometimes it is desirable to observe that the IUT does not produce any output at all. The LTSs considered therefore also support the absence of output (quiescence). These concepts are all respected by the **ioco** relation.

We first provide the definition of a Labelled Transition System.

Definition 3. A Labelled Transition System (LTS) *is a quaternary tuple* $\mathcal{L} = \langle Q, L, T, q_0 \rangle$

- Q is a possibly infinite set of states
- $q_0 \in Q$ is the initial state

- L is a possibly infinite set of action labels. The τ action denotes the unobservable action. All other actions are in L and are observable. We have that the set of action labels is $L \cup \{\tau\}$. We can use L_τ to denote this set.
- $T \subseteq Q \times L_\tau \times Q$ is a transition relation. For $(s_0, a, s_1) \in T$ we often write $s_0 \xrightarrow{a} s_1$.

Instead of referring to $\mathcal{L}$, the initial state s_0 is often used to refer to the corresponding transition system.

Definition 4. Let A be a set. Then A^* is the set of all finite sequences over A including the empty sequence ϵ. If $\sigma_1, \sigma_2 \in A^*$ are finite sequences, then $\sigma_1 \cdot \sigma_2$ is the concatenation of σ_1 and σ_2.

Definition 5. Let $p = \langle Q, L, T, q_0 \rangle$ be a labelled transition system with $s, s' \in Q$, and $a \in L$.

$$s \xRightarrow{\epsilon} s' \overset{val}{=} (s = s' \vee s \xrightarrow{\tau \cdot \ldots \cdot \tau} s')$$
$$s \xRightarrow{a} s' \overset{val}{=} \exists_{q_1, q_2}[q_1, q_2 \in Q : (s \xRightarrow{\epsilon} q_1 \xrightarrow{a} q_2 \xRightarrow{\epsilon} s')]$$

Definition 6. Let p be (a state of) a labelled transition system.

- p **after** $\sigma \overset{\text{def}}{=} \{p' \in Q \mid p \xRightarrow{\sigma} p'\}$.
- p is *strongly converging* if there is no state of p that can perform an infinite sequence of internal transitions.
- p is *image finite* if for all $\sigma \in L^*$, p **after** σ is *finite*.
- $\mathcal{LTS}(L)$ is the class of *all image finite and strongly converging* labelled transition systems with labels in L.

Tretmans points out that the form of LTSs as defined in Definition 3 is usually sufficient for analyzing and reasoning about applications. However, in the context of testing, it is not desirable to abstract from the initiative and direction of an action. The environment communicates with the system via an *input* action, and the system communicates with the environment via an *output* action. Therefore, the concept of *labelled transition systems with inputs and outputs* is introduced.

Definition 7. A labelled transition system with inputs and outputs is a Quinary tuple $\langle Q, L_I, L_U, T, q_0 \rangle$ where

- $\langle Q, L_I \cup L_U, T, q_0 \rangle$ is a labelled transition system in $\mathcal{LTS}(L_I \cup L_U)$.
- L_I is a *finite* set of input labels and L_U is a *finite* set of output labels.

The input labels will be decorated with a ? and output labels with !. The class of *labelled transition systems with inputs and outputs* is denoted with $\mathcal{LTS}(L_I, L_U)$

Except for the addition of some syntactic sugar, the labelled transition systems with inputs and outputs do not really differ from regular labelled transition system. The **ioco** relation restricts itself for the implementation to a special type of transition system referred to as *input-output transition systems*.

Definition 8. An *input-output transition system* $\langle Q, L_I, L_U, T, q_0 \rangle$ is a labelled transition system with inputs and output for which

$$\forall_q [q \in Q \wedge \ q \text{ is reachable from } q_0 : \forall_\sigma [\sigma \in L_I : q \xRightarrow{\sigma}]]$$

(where $q \xRightarrow{\sigma} \overset{val}{=} \exists_{q'}[q' \in Q : q \xRightarrow{\sigma} q']$). I.e. all input actions are enabled in any state reachable from the initial state.

We denote the class of *input-output* transition systems with $\mathcal{IOTS}(L_I, L_U)$ where $\mathcal{IOTS}(L_I, L_U) \subseteq \mathcal{LTS}(L_I, L_U)$

For many systems, the view as an *input-output transition system* is sensible. For example, for a vending machine, the user can always press any of the buttons. However, when no money is inserted into the machine, the vending machine typically will not respond to this input action. In such a situation, a self-loop could be modeled in the transition system. Also, in the context of a DMDSL model, this requirement is sensible. At any point in time, the user can do a CRUD+A operation. Even in a state where some instance with identifier i does not exist, it should be possible to do the Delete(i) transition. As expressed by Derasari, this would be represented as a self-loop and the output variable would be changed.

For an input-output transition system, the environment can at any point in time decide to perform an input action. The IUT can autonomously decide when to perform an output action. It is therefore, of relevance for the environment to be aware if the system is in a state in which no output action is possible anymore, and thus the system cannot autonomously proceed. We call such a state *quiescent* and denote a quiescent state d with $\delta(d)$. It turns out that it is convenient if the system is able to express to 'see nothing'. Hence, the possibility of δ-transitions is added to the language.

Definition 9. Let $p = \langle Q, L_I, L_U, T, q_0 \rangle \in \mathcal{LTS}(L_I, L_U)$

- A state q of p is quiescent, denoted with $\delta(q)$ when $\forall_\mu[\mu \in L_U \cup \{\tau\} : q \overset{\mu}{\not\rightarrow}]$
- $L_\delta \overset{\text{def}}{=} L \cup \{\delta\}$
- $p_\delta \overset{\text{def}}{=} \langle Q, L_I, L_U \cup \{\delta\}, T \cup T_\delta, q_0 \rangle$ where $T_\delta \overset{\text{def}}{=} \{q \overset{\delta}{\rightarrow} q \mid q \in Q \wedge \delta(q)\}$
- The suspension traces of p are $Straces(p) \overset{\text{def}}{=} \{\sigma \in L_\delta^* \mid p_\delta \overset{\sigma}{\Longrightarrow}\}$

Definition 10. Let $q \in Q$ of some transition system $p = \langle Q, L_I, L_U, T, q_0 \rangle$. Then:

- $out(q) \overset{\text{def}}{=} \{o \in L_U \mid q \overset{o}{\rightarrow}\} \cup \{\delta \mid \delta(q)\}$
- $out(Q) \overset{\text{def}}{=} \bigcup\{out(q) \mid q \in Q\}$

Finally, we will provide the **ioco** relation presented in the work of Tretmans:

Definition 11 (**ioco** relation)**.** Given a set set of input labels L_I and a set of output labels L_U, the relation $\mathbf{ioco} \subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{LTS}(L_I, L_U)$ is defined as follows:

$$i \ \mathbf{ioco} \ s \overset{val}{=} \forall_\sigma[\sigma \in Straces(s) : out(i \ \mathbf{after} \ \sigma) \subseteq out(s \ \mathbf{after} \ \sigma)]$$

Note that the definition of **ioco** considers suspension traces from s, and does not say anything about suspension traces of i that are not suspension traces for s. This allows for partial specifications (*partiality*).

In section 5.2 in the work of Tretmans, an algorithm for complete test generation is provided [33]. In section 5.3 it is pointed out that this algorithm can detect all and only all non-**ioco** correct implementations. We call the test suite T this algorithm can compute *complete*. In practice, however, such a test suite would be of infinite size, and the algorithm would not terminate. One would thus typically create only *sound* test suites. As Tretmans points out, exhaustiveness is more a theoretical result and usually not practically applicable. We will now provide formal definitions of these terms in the context of the **ioco** theory.

Definition 12. Let s be a specification and T a test suite; then for **ioco**

$$T \ \textit{is complete} \overset{\text{def}}{=} \forall_i[i \in \mathcal{IOTS}(L_I, L_U) : i \ \mathbf{ioco} \ s \iff i \ \mathbf{passes} \ T]$$

$$T \ \textit{is sound} \overset{\text{def}}{=} \forall_i[i \in \mathcal{IOTS}(L_I, L_U) : i \ \mathbf{ioco} \ s \implies i \ \mathbf{passes} \ T]$$

$$T \ \textit{is exhaustive} \overset{\text{def}}{=} \forall_i[i \in \mathcal{IOTS}(L_I, L_U) : i \ \mathbf{ioco} \ s \impliedby i \ \mathbf{passes} \ T]$$

In practice when **ioco** is applied, the relation will be restricted to a subset of suspension traces:

Definition 13. Let $\mathcal{F} \subseteq (L_I \cup L_U \cup \{\delta\})^*$ be a set of suspension traces (typically generated from specification s) and let $i \in \mathcal{IOTS}(L_I, L_U)$ and $s \in \mathcal{LTS}(L_I, L_U)$

$$i \ \mathbf{ioco}_{\mathcal{F}} \ s \stackrel{\text{def}}{=} \forall_{\sigma}[\sigma \in \mathcal{F} : out(i \ \mathbf{after} \ \sigma) \subseteq out(s \ \mathbf{after} \ \sigma)]$$

In LTSs, the labels on the transition system indicate the interaction that a system may have with its environment. This is also in line with the transition system induced by a DMDSL model as described in the work of Derasari [8]. The output actions of the CRUD+A operations have however been made part of the content of the state, whereas for an LTS you would expect these to be an output action label. Furthermore, when the state variables of two states in the transition system are different, we would expect in the LTS different states/locations to exist for reflecting this. Since state variables are not supported for an LTS one would need to find LTS semantics for the transition system defined in the work of Derasari.

5.4.2 IOCO for Symbolic Specifications

The IOCO Theory forms a rich formal theory for MBT using LTSs. If we attempt to express the model of Derasari in LTS semantics, we will encounter problems. The CRUD+A actions, have been formalized using input arguments. When these input arguments would be 'hard-coded' in the transitions, then the number of transitions for a single state would be very high, and possibly infinite. For example, for the 'Deletion' of an instance, an identifier needs to be passed. One would expect the domain of identifiers to be of infinite size. However, having an infinite number of outgoing transitions is typically not allowed by the tools. One would need to finitize such a domain. Depending on how these are finitized, this potentially could still give rise to a large number of transitions. Furthermore, the states do not allow for state variables. Due to the high number of combinations of possible for the state state variables, we would expect the state space to be very large, and depending on how the system is modeled, of infinite size. For tools relying on LTSs, this is typically problematic. In the work of Frantzen, Tretmans and Willemse [9] these problems are pointed out. To allow for the modeling of such systems, they augment the LTS as *Symbolic Transition Systems* (STSs) and lift the **ioco** test theory to these structures. Finally, they present an on-the-fly algorithm for generating and executing test cases for STSs.

Analogous to the paper of Frantzen, Tretmans and Willemse, we will first remember the reader to some of the first-order logic concepts and then introduce the STSs.

5.4.2.1 First Order Logic

Suppose a first order structure is given

- A logical signature $\mathfrak{S} = (F, P)$
  - F is a set of function symbols, where each $f \in F$ has an arity $n \in \mathbb{N}$ denoting the number of arguments. When $n = 0$ for some function f we will say that f is a constant.
  - P is a set of predicate symbols. Each $p \in P$ has an arity of $n \in \mathbb{N}^+$.
- A model $\mathfrak{M} = (\mathfrak{U}, (f_{\mathfrak{M}})_{f \in F}, (p_{\mathfrak{M}})_{p \in P})$
  - $\mathcal{U}$ denotes the universe, where $\mathcal{U} \neq \emptyset$.
  - For $f \in F$ with $\mathbf{arity}(f) = n$ we have that $f_{\mathfrak{M}} : \mathcal{U}^n \to \mathcal{U}$.
  - For $p \in P$ with $\mathbf{arity}(p) = n$ we have $p_{\mathfrak{M}} \subseteq \mathcal{U}^n$.

Definition 14 ((ground)terms). Let $\mathfrak{X}$ be a set of variables.

- Suppose $X \subseteq \mathfrak{X}$. Now $t \in \mathfrak{T}(X)$ is an element that can be constructed from function symbols in F and variables in X. We denote t as *term over* X. Furthermore, we use $\mathbf{var}(t)$ to denote the set of variables used in term t. In particular, we have that $\mathbf{var}(t) \subseteq X$.

- Let $t \in \mathfrak{T}(\emptyset)$, then t is constructed using only function symbols in F, and thus $\mathbf{var}(t) = \emptyset$. We denote t as *ground term*.

Definition 15 (Term-mapping)**.** A term-mapping is a function $\sigma : \mathfrak{X} \rightarrow \mathfrak{T}(\mathfrak{X})$ that assigns to each variable in $\mathfrak{X}$ a term over $\mathfrak{X}$. Furthermore, we denote with $\mathfrak{T}(Y)^X$ the set of term mappings containing term-mappings with type $X \rightarrow \mathfrak{T}(Y)$. Furthermore, the substitution of terms $\sigma(x)$ for $x \in \mathbf{var}(t)$ is denoted with $t[\sigma]$.

Example 5.4.1. Let $\mathfrak{S}(F, P)$ with $F = \{false, not, and\}$ with respective arities $0, 1, 2$, and let $P = \{equiv\}$ with arity 2. A possible model would be the boolean algebra with the signature $\texttt{false}, \neg, \wedge$ and the $\overset{val}{=}$ predicate on propositions. Example terms would be $not(x), not(and(x, y))$. Example of ground terms would be: $and(not(false), not(false)), not(and(not(false), not(false)))$. Now let $X = \{x, y\}, Y = \{x\}$. Now we have that $\sigma = \{x \mapsto x, y \mapsto false\}$ is a term mapping in the set $\mathfrak{T}(Y)^X$. Now consider $t := not(and(x, y))$ and we thus have $t \in \mathfrak{T}(X)$. Furthermore, note that $t[\sigma] = not(and(x, false))$

Definition 16 (Valuation)**.**

- A valuation is defined as a function $\vartheta : \mathfrak{X} \rightarrow \mathfrak{U}$.
- We write $\vartheta \in \mathfrak{U}^X$ when $\vartheta : X \rightarrow \mathfrak{U}$ for $X \subseteq \mathfrak{X}$. When $y \in \mathfrak{X} \setminus X$ then $\vartheta(y) = *$ for an arbitrary element $*$ of the set $\mathfrak{U}$.
- Suppose $\vartheta \in \mathfrak{U}^X, \varsigma \in \mathfrak{U}^Y$ where $X \cap Y = \emptyset$. Then, the union is defined as:

$$(\vartheta \cup \varsigma)(x) \overset{\text{def}}{=} \begin{cases} \vartheta(x) & \text{if } x \in X \\ \varsigma(x) & \text{if } x \in Y \\ * & \text{otherwise} \end{cases}$$

Definition 17 (Satisfaction)**.** Let φ be a predicate. The satisfaction of φ w.r.t to a valuation ϑ is denoted with $\vartheta \models \varphi$.

Definition 18 (Term-evaluation)**.** The extension of a valuation ϑ to evaluate whole terms is called a *term-evaluation*. This is denoted with $\vartheta_{\text{eval}} : \mathfrak{T}(\mathfrak{X}) \rightarrow \mathfrak{U}$.

Example 5.4.2. Consider the model of boolean algebra on the signature of example 5.4.1. Let $\varsigma = \{x \rightarrow \texttt{false}, y \rightarrow \texttt{false}\} \in \mathfrak{U}^{\{x,y\}}$. We have that $\varsigma_{\text{eval}}(and(x, y)) = \texttt{false}$ and $\varsigma_{\text{eval}}(x) = \texttt{false}$. It is clear that for $\varphi = equiv(and(x, y), x)$ we have that $\varsigma \models \phi$.

5.4.2.2 IOCO on STSs

Definition 19 (Symbolic Transition System)**.** A Symbolic Transition is a septuple $\langle L, l_0, \mathcal{V}, \iota, \mathcal{I}, \Lambda, \rightarrow \rangle$

- L is a *countable* set of locations.
- $l_0 \in L$ is the initial location.
- $\mathcal{V}$ is a *countable* set of location variables.
- ι is an *initialization* of the location variables. (i.e. ι is a mapping $\mathcal{V} \rightarrow \mathfrak{T}(\emptyset)$ where $\mathfrak{T}(\emptyset)$ is the set of *ground terms*. Ground terms, are build from function symbols, and do not contain variables.)
- $\mathcal{I}$ is a set of *interaction variables* with $\mathcal{I} \cap \mathcal{V} = \emptyset$.
- Λ is a finite set of *gates*. The *unobservable* gate is denoted with τ, and $\tau \notin \Lambda$. We use Λ_τ to denote $\lambda \cup \{\tau\}$. We say that each $\lambda \in \Lambda_\tau$ has an arity(λ) denoting the number of distinct interaction variables. We have that arity(τ) = 0 is fixed since the unobservable gate does not allow for interaction, and therefore will not contain interaction variables.

- $\rightarrow \subseteq L \times \Lambda_\tau \times \mathfrak{T}(V \cup \mathcal{I}) \times \mathfrak{T}(\mathcal{V} \cup I)^{\mathcal{V}} \times L$, which denotes the switch relation. Instead, we can write $l \xrightarrow{\lambda,\phi,\rho} l'$ to denote $(l, \lambda, \phi, \rho, l') \in \rightarrow$. Hence, we have that for such a transition, λ would serve as a gate, ϕ would serve as a guard, and ρ would serve as an update mapping.

We will use Λ_I to denote the set of input gates, and Λ_U for the set of output gates. Furthermore, in a visualization we will denote this respectively with ? and !.

Definition 20 (**ioco** on STSs)**.** Let $S = \langle L, l_0, \mathcal{V}, \iota, \mathcal{I}, \Lambda, \rightarrow \rangle$ be an STS. The LTS semantics of S are given by $[\![S]\!] = \langle Q, \Sigma, T, q_0 \rangle$

- $Q = L \times \mathfrak{U}^{\mathcal{V}}$ as set of states.
- $\Sigma = \bigcup_{\lambda \in \Lambda_\tau} (\{\lambda\} \times \mathfrak{U}^{\text{arity}(\lambda)})$ is the set of actions, where $\Sigma_U = \bigcup_{\lambda \in \Lambda_U} (\{\lambda\} \times \mathfrak{U}^{\text{arity}(\lambda)})$ and $\Sigma_I = \bigcup_{\lambda \in \Lambda_I} (\{\lambda\} \times \mathfrak{U}^{\text{arity}(\lambda)})$.
- $q_0 = (l_0, \text{eval} \circ \iota) \in S$ is the initial state.
- $T \subseteq Q \times \Sigma \times Q$ where T is the transition relation defined by

$$\frac{l \xrightarrow{\lambda,\varphi,\rho} l' \quad \text{type}(\lambda) = \langle \nu_1, \ldots, \nu_n \rangle \quad \varsigma \in \mathfrak{U}^{\text{type}(\lambda)} \quad \vartheta \cup \varsigma \models \varphi \quad \vartheta' = (\vartheta \cup \varsigma)_{\text{eval}} \circ \rho}{(l, \vartheta) \xrightarrow{(\lambda, \langle \varsigma(\nu_1), \ldots, \varsigma(\nu_n) \rangle)} (l', \vartheta')}$$

Since this provides LTS semantics to STSs, this allows using the **ioco** test relation as seen in section 5.4.1 to be applied on these structures.

5.4.2.3 Expressing the Semantics of a DMDSL Model as STS

We will transfer the semantics as defined by Derasari as an STS. We will refer to this STS as *Model idea 1*, which has been visualized in figure 5.2. Let

- $L := \{inputState, outputState\}$
- $l_0 := inputState$
- $\mathcal{V} := \{\textsc{Repo}, \textsc{Instances}, \textsc{Links}, \textsc{Output}\}$
- $\iota := \{\textsc{Repo} \rightarrow \emptyset, \textsc{Instances} \rightarrow \emptyset, \textsc{Links} \rightarrow \emptyset, \textsc{Output} \rightarrow \texttt{Success}\}$
- $\mathcal{I} := \{entity, targets, identifier, instance, output\}$
- $\Lambda := \{Create(entity, targets), Read(identifier), Update(instance, targets),$ $Delete(instance), Add(instance), OutputMessage(output)\}$
- $$\begin{aligned}\rightarrow = \{ &\big(inputState, Create(entity, targets), [\textbf{True}], \{(\textsc{Instances}, \textsc{Repo}, \textsc{Links}, \textsc{Output}) \\ &:= UpdateFor((\textsc{Instances}, \textsc{Repo}, \textsc{Links}, \textsc{Output}), Create(entity, targets))\}, outputState\big), \\ &\big(inputState, Read(identifier), [\textbf{True}], \{(\textsc{Instances}, \textsc{Repo}, \textsc{Links}, \textsc{Output}) \\ &:= UpdateFor((\textsc{Instances}, \textsc{Repo}, \textsc{Links}, \textsc{Output}), Read(identifier))\}, outputState\big), \\ &\big(inputState, Update(identifier, targets), [\textbf{True}], \{(\textsc{Instances}, \textsc{Repo}, \textsc{Links}, \textsc{Output}) \\ &:= UpdateFor((\textsc{Instances}, \textsc{Repo}, \textsc{Links}, \textsc{Output}), Update(identifier, targets))\}, outputState\big), \\ &\big(inputState, Delete(identifier), [\textbf{True}], \{(\textsc{Instances}, \textsc{Repo}, \textsc{Links}, \textsc{Output}) \\ &:= UpdateFor((\textsc{Instances}, \textsc{Repo}, \textsc{Links}, \textsc{Output}), Add(instance))\}, outputState\big), \\ &\big(inputState, Delete(identifier), [\textbf{True}], \{(\textsc{Instances}, \textsc{Repo}, \textsc{Links}, \textsc{Output}) \\ &:= UpdateFor((\textsc{Instances}, \textsc{Repo}, \textsc{Links}, \textsc{Output}), Delete(identifier))\}, outputState\big), \\ &\big(outputState, OutputMessage(output), [output == \textsc{Output}], \{\}, inputState\big) \\ &\}\end{aligned}$$

As state variables, we will use INSTANCES to denote the set of instances that were created and not yet deleted. We use REPO to denote the set of instances that are in their repository. We use LINKS as the set of links between instances. Initially, these are initialized with the empty set. We have a specialized function $UpdateFor$ that can update the state variables on the basis of the state variables of the current state, and the action executed for some given arguments. How $Updatefor$ should be defined follows from the dynamic semantics in chapter 3.3.2, where is specified how the contents of the state variables should look, after executing an action given the current state variables. From the *inputState* the $UpdateFor$ determines that successful execution of the action is possible, the state variables INSTANCES, REPO, LINKS will be updated accordingly and OUTPUT will contain the output message `Success`. If the $UpdateFor$ execution determines that successful execution is not possible, the state variables INSTANCES, REPO, LINKS remain unchanged, and the *Output* variable will be set to the output message denoting the reason that execution was not possible (as specified in chapter 3.3.2). Note that all complexity is actually hidden in the $UpdateFor$ function. Since the CRUD+A actions are always enabled, we can set the guards for each of these actions to **True**. In the *outputState* state there is only one transition possible, in which the content of the *Output* variable is communicated with a corresponding output gate.

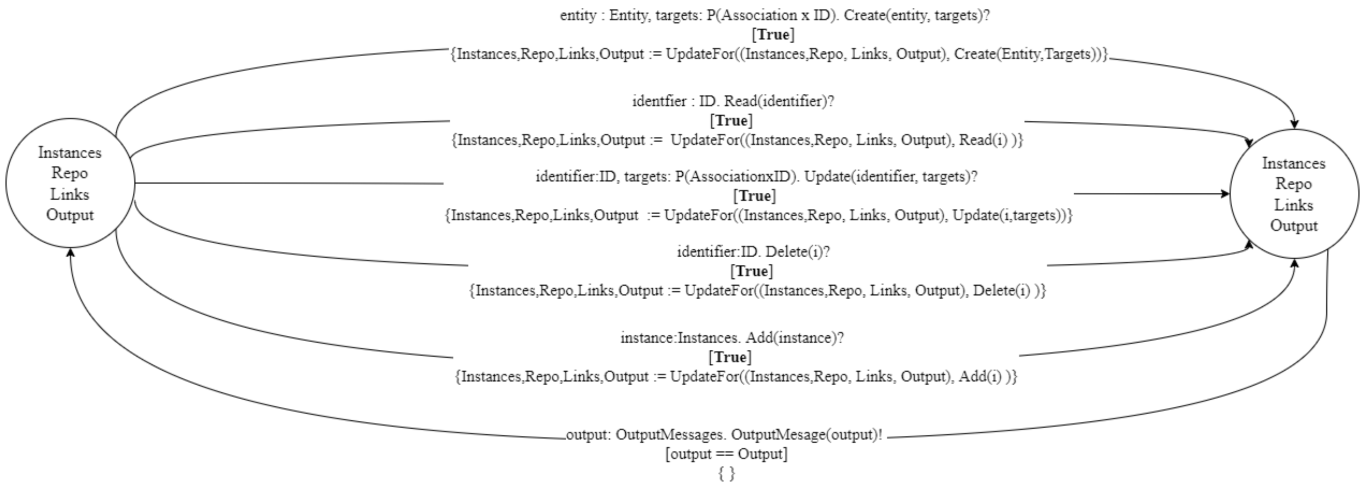


Figure 5.2: *Model idea 1* - Dynamic Semantics Captured as STS.

The STS thus has two states: an input state, for receiving actions, and an output state that can only output the result of the action. Although such modeling would be sound, it does not truly use the power of these transition systems. The idea behind such transition-based notation would be to create separate locations for which the corresponding states have *significantly different behavior*. This fact is also pointed out in [20], where they point out that for such formalisms you could even create a single location, and transfer all logic, and possible output values in self-loops for this single state. Of course, implicitly there are many states, but these will all be mapped to a single location.

We could try to improve on the initial model, and get rid of the OUTPUT state variable, and instead model for each type of error message a separate state that only allows for outputting the specific error message. By adding guards we can ensure that the transition is only enabled, when input arguments are passed for which such an error would be the result. We will provide part of such an STS in figure 5.3 where only the *Delete* action has been modeled. For the Delete action, we would have an outgoing arrow for the occurrence of `Instance_NotInRepository`, `Entity_DeleteOnUndestructable`, `Entity_MultiplicityMinimum` errors, and a transition if neither of these errors would occur, and a `Success` message is expected. In the figure we observe in the guard of the first outgoing transition a $isInstanceNotInRepositoryError((\text{INSTANCES}, \text{REPO}, \text{LINKS}), id)$, which takes as argument the state variables, and the id of the instance on which the delete action is performed. Such function can decide on the basis of the state variables if indeed the `Instance_-`

`NotInRepository` error could occur for the input arguments. As specified in the work of Derasari, this could occur if $x \notin Repo$ (where x is the instance that has as identifier the corresponding id).

> Note that multiple errors could occur simultaneously. For example, doing a Create operation with insufficient targets, to an entity for which the maximum entity multiplicity would be exceeded. Then the implementation could output both `Entity_MultiplicityMaximum` but also `Association_TargetMinimum`. We make use of the non-deterministic property of STSs, in which multiple transitions can be enabled, for the same action (and arguments). The result is that the MBT tool does not know which of the states of the specification is supposed to match with the state the implementation is in. This requires the underlying MBT tool to be more sophisticated in order to handle this notion.

The introduction of states for each output message will increase the states of the model from two, to the 'number of distinguished output messages'+1(for the initial/input state). Note that if the MBT tool would try to achieve state coverage, it would indirectly cover all possible output messages. We could slightly adapt it and describe it as an STS in which the output states are unique for each of the CRUD+A actions. For example, the `Instance_NotInRepository` error could occur when trying to read an instance that is not in the repository, but also when trying to delete an instance that is not in the repository. If we decide to make a unique state for outputting `Instance_NotInRepository` after executing a Read action, and a different state reachable after executing a Delete action, we would slightly increase the state space. Then, when the MBT tool would try to achieve state coverage, then indirectly we would get coverage of the output messages on a per-action basis. We will call this *model idea 2*. This would still be reflected in figure 5.3, since only the Delete action has been visualized. Note that for, for example, the `Success` output message, we now have a separate location for each action.

We have provided two ways to model the dynamic semantics of DMDSL as STS. Even after increasing the state space so that for each output action a unique state exists, we do not truly make use of different behavior on a different location basis. The global idea remains that one the CRUD+A actions is chosen, to which a corresponding output will occur on the basis of the state variables, and a modification of the state variables will happen. The CRUD+A actions, will always be enabled, but only a different output message will happen on the basis of the arguments and the current state variables. It is not "wrong notation" to use STS as formalism, but it gives the intuition that we do not truly make use of the mechanisms it offers. The complexity is hidden in the functions to update the state variables. It depends on the tool support to what extent such model would be good in practice. If there are sophisticated techniques, relying on the state variables to create interesting test cases then such a model could still make sense. We would however expect that for such a notation the tool would be mostly focused on the locations of the STS in its possibility of generating test cases.

5.4.3 Tools

Examples of tools that make use of Labelled Transition Systems and the **ioco** test relation are TorX [31] and JTorX [3]. JTorX is a re-implementation of TorX, but has some additional features. Unfortunately, these tools are no longer developed, nor available. A different tool that is based on TorX and JTorX is TorXakis [32]. The TorXakis tool keeps data symbolically, and unlike JTorX and TorX it does not unfold the data in all possible concrete data values. The TorXakis tool manages to work over infinite structures by making use of a constraint solver. It currently uses Z3 and CVC4 as SMT solvers. Using such a solver, the tool can come up with arguments of a transition, satisfying all input constraints. The tool is still actively developed and implements the test generation for symbolic transition systems as seen in [9] closely. The creators note that TorXakis is an experimental tool, used in research education and some case studies in industries. Furthermore, it is pointed out that the tool is not always powerful enough to be sufficiently scalable for a complex system. The usability is sometimes insufficient. Furthermore, it is pointed out that test selection is currently still mainly random.

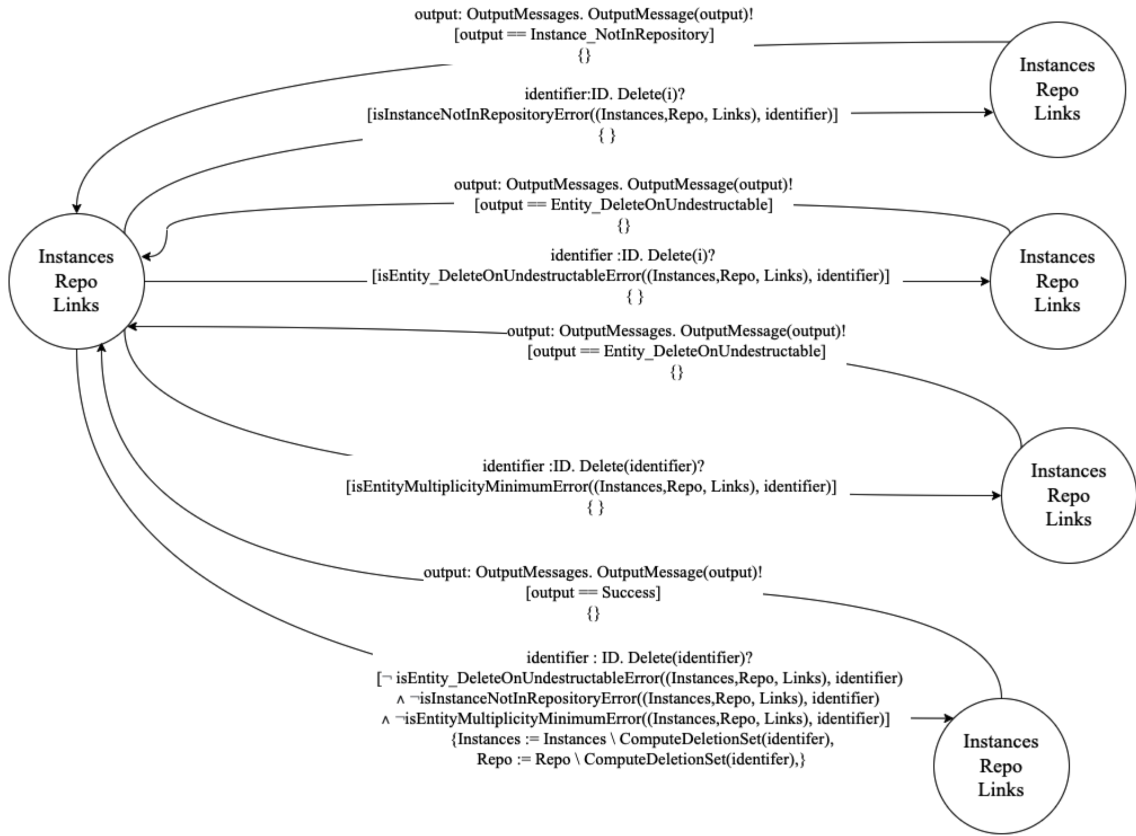


Figure 5.3: *Model idea 2* - The dynamic semantics of the delete action of DMDSL represented as STS.

We will also have a look at the Axini MBT tool. Axini is a company that developed an MBT tool in a Software As A Service (SAAS) format. The tool is commercial and not publicly available. The tool implements the **ioco** theory, and allows for symbolic treatment of data. One would expect due to its commercial nature that such a tool is better applicable for industrial use cases, and overall has better usability. Axini states 'Our platform cleverly chooses and generates test cases to maximize the test coverage'. What these exact strategies are, are not public information, but these words seem more promising in comparison to the mostly random testing strategy of TorXakis. This tool also runs a solver in the background, allowing it to operate over infinite domains. We will try out both tools, to check out the potential of applying these as MBT solutions in the context of the code generator.

5.4.3.1 Axini

In Axini an LTS or STS is encoded using the Axini Modeling Language (AML). It provides a pleasant notation to encode a model. By using mechanisms such as looping, one is able to express a large LTS or STS with only a small specification. When experimenting with the Axini tool, we encounter a few problems. The tool has as data types integer, string, decimal, boolean, date, time, list, hash, struct and enum. One of the data types that are lacking is a set. In particular, for the state variables, the order of the elements in $Instances, Repo, Links$ do not matter. Hence, it would be desirable to use set notation to express this. Since these are not supported we could use lists instead. Result of that is that states can be behaviorally equal, but have different state variables due to the different ordering in the lists. It thus lacks a form of *data abstraction* to express this. Furthermore, containment checking for a set can typically be done in $O(1)$ time,

whereas for lists this typically takes $O(n)$ (although we do not know how this is implemented under the hood.)

If we try to encode a model as shown in *model idea two*, we already encounter limitations of the tool. In particular, the constraints that Axini allows you to specify can not be complex. It only supports basic mathematical operations such as $<, >, \neq, =$, and some boolean checks on lists (e.g. list containment). While the tool allows you to create more advanced functions using the Ruby language, these functions can not be used for solving. This means that an input value (that needs to be generated by the constraint solver) can not be used as argument for a function in the constraint. The auxiliary function can only be used for some fixed value in the constraint, or for a given argument in the update block. Some of the easier checks, such as 'isInstanceNotInRepositoryError', can be expressed as constraint. It is simply checked if the instance argument is a member of the *Repo* list. However, if we get to slightly more complex constraints, such as $isEntityMultiplicityMinimumError$ we already run into problems. Now we desire the input argument to be an instance of an entity type, of which removing the instances causes a minimum multiplicity violation. One would typically iterate over the $Instances$ list, and check if the input argument would cause such a violation. This is however not expressible as a constraint in the AML language. For even more sophisticated constraints, this becomes even more clear. For example, when an entity is deleted, this can cause a lot of deletions due to cascade deletion. Each of the instances that get deleted as a result of the operation can induce error output messages. To determine which elements would be deleted, one would typically need to compute which instances are reachable via the link set from the instance that will be deleted, for which the corresponding association relation has source cascade deletion enabled. One could compute such closure using a fixed-point computation. This is unfortunately not expressible with the limited mathematical operations. It would be expressible in the Ruby language, but the tool can not manage to create input using such a Ruby function, since these are not allowed to be used in constraints. The simpler model such as in figure 5.2 would (although awkward) be probably expressible in the AML language. Note that here the constraints are actually empty, and hence there is not really any solving involved. The $Instances, Repo, Links$, and *Output* variables would simply be updated on the basis of the randomly generated input. This input would be fully random and there is no incline for the solver to generate any 'interesting' test cases at all. The tool would quickly terminate since coverage of the two locations and transitions is almost immediately reached.

5.4.3.2 TorXakis

In TorXakis an STS is encoded using the TorXakis language. The TorXakis language is a functional programming language. The language has only as built-in types the Bool, Int, String, and Regex type. It has the option for users to define their own (recursive) Algebraic Data Type. This allows the user to create for example a list data type. It however does not seem to be possible to express a set as such a data type. Hence, also the TorXakis tool lacks this form of data abstraction. It is unfortunate that there is not a broader list of predefined data types, but it is good that we can define datatypes. This allows us to create a datatype for the concepts such as Multiplicity, Entity, Instance, Association, Link, etc. as we have seen in chapter 3. Furthermore, the language allows us to make user-defined functions for which solving is supported. Hence, we can make guards containing these functions. This would make TorXakis much more expressive than Axini is.

We will make an attempt to see if we can express the DMDSL semantics in the TorXakis tool. Furthermore, we translate a simple model into the TorXakis language. To prevent overwhelming ourselves, we will consider a model without associations. Furthermore, since we do not have associations we will also ignore the update Action (since there are no links to update anyways). The visualization of the DMDSL model and the full TorXakis code are provided in appendix A. We will now elaborate on the translation to TorXakis

Datatypes We will first create datatypes for the static constructs that are used in the DMDSL language. We will show for sake of example some of the constructs translated as Algebraic Data

Types. For the full list of Algebraic Data Types, see the listing in the appendix A. In the listing we define the `Multiplicity` data type. This has a tuple constructor requiring an integer for the minimum, and an integer for the maximum. The `Instance` datatype requires in the constructor an entityName (of type ValidEntityName, where ValidEntityName will be the datatype reflecting the entity names of the model that will be translated) and an id, which is modeled as an integer here. Finally, we will also create a datatype `InstanceList`. We need such a list to model the *Instances* state variable. Note that the `InstanceList` is defined recursively as one would expect for these list data types. It is unfortunate that the TorXakis language does not support generic data types. As a result, for each type that we want to use in lists, we need to create a separate list data type.

```
TYPEDEF
  Multiplicity ::= Tuple {min :: Int; max :: Int}
ENDDEF

TYPEDEF
  Instance ::= Instance { entityName :: ValidEntityName; id :: Int}
ENDDEF

TYPEDEF
  InstanceList ::= Nil | Cons { hd :: Instance; tl :: InstanceList}
ENDDEF
```

Now we express the model in the form of a TorXakis State Automaton called *DmdslModel*, which uses `CreateInput`, `DeleteInput`, `AddInput`, `ReadInput` as gates (channels) for corresponding operations, and the `Output` as gate for output of the IUT. Then we declare states for each error and success message as seen in *model idea 2*. Then we declare variables `instances`, `repo`, and `links` as state variables. Besides these state variables, there is also a `highestId` state variable, that is used to assign ids to instances in the creation process so that the uniqueness of the id can be guaranteed (i.e. `highestId` as id when creating an instance, and after creation, the variables gets incremented). These state variables are then initialized to the emptylist (`Nil`) for `instances`, `repo` and `links`, and the `highestId` is initialized to 0.

```
STAUTDEF DmdslModel [CreateInput :: CreateOperation;
                      DeleteInput :: DeleteOperation;
                      AddInput :: AddOperation;
                      ReadInput :: ReadOperation;
                      Output :: OutputMessage]()

::= {- DECLARE STATES-}
    STATE
        initialState,
        outputState,
        Entity_Unconstructable_State,
        Entity_MultiplicityMaximum_State,
        Instance_NotInRepository_State_Delete,
        Instance_NotInRepository_State_Read,
        Entity_Undestructable_State,
        Instance_AlreadyInRepository_State,

        successfulCreationState, successfulDeletionState,
        successfulAdditionState, successfulReadState

{- DECLARE STATE VARIABLES-}
```

```
    VAR
        instances :: InstanceList;
        repo :: InstanceList;
        links :: LinkList;
        highestId :: Int

{- INITIALIZE STATE VARIABLES -}

    INIT  initialState{
    instances := Nil;
    repo := Nil;
    links := Nil;
    highestId := 0
    }
```

We can now start to describe the transitions. We will only cover the Delete transition here for sake of example. Since for now, we do not consider models with associations, only the instance of the provided id will be deleted. The errors that could result from this operation would be `Entity_DeleteOnUndestructable`, `Instance_NotInRepository`, and `Entity_MultiplicityMinimum`.

- `Entity_DeleteOnUndestructable`: To check if a deletion happens on an instance that is undeletable/undestructable, we at least require that the provided id, is an id of an instance. Therefore, in the guard of the transition (indicated with [[...]]) leading to the `Entity_Undestructable_State`, we check the id is the id of an instance in the instance list. Then using an `isEntityDeleteOnUndestructableError` function it is checked whether the corresponding entity is undeletable/undestructable.
- `Instance_NotInRepository`: Analogously, we have a transition leading to `Instance_NotInRepository_State_Delete` that is only enabled when the passed id, is not the id of an instance in the repository list.
- `Entity_MultiplicityMinimum`: There is also a transition leading to a `Entity_MultiplicityMinimum_State` which is only enabled, if deleting the instance with corresponding id would cause the number of instances of the corresponding entity to dive below the minimum multiplicity.
- `Success`: Finally, when none of the error conditions hold (which is specified in the guard), it must mean that we can successfully delete the instance, and a transition to `successfulDeletionState` would be enabled. This transition also removes the instance from the `instances` and `repo` state variables (modification of state variables is denoted with {...}), since the instance that will be deleted, should be removed from both lists according the semantics.

```
    TRANS


    {-BEGIN: DELETION-}
      {-ERROR DELETION -}

      initialState ->
      DeleteInput ? deleteInput
      [[ isEntityDeleteOnUndestructableError(
                id(deleteInput), instances)
        /\ isIdInInstanceList(id(deleteInput), instances)]]
      {  }
```

```
      -> Entity_Undestructable_State


    initialState ->
    DeleteInput ? deleteInput
    [[ isInstanceNotInRepositoryError(id(deleteInput), repo) ]]
    {  }
      -> Instance_NotInRepository_State_Delete


    initialState ->
    DeleteInput ? deleteInput
    [[isEntityMultiplicityMinimumError(id(deleteInput), repo)
    /\ isIdInInstanceList(id(deleteInput), repo)]]
    {  }
      -> Entity_MultiplicityMinimum_State



    {-SUCCESFUL DELETION -}
    initialState ->
    DeleteInput ? deleteInput
    [[not( isInstanceNotInRepositoryError(id(deleteInput), repo)
       )
    /\
    not( isEntityDeleteOnUndestructableError(id(deleteInput),
       instances) /\ isIdInInstanceList(id(deleteInput),
       instances) )
    /\
    not( isEntityMultiplicityMinimumError(id(deleteInput), repo)
    /\ isIdInInstanceList(id(deleteInput), repo) )
    ]]
    { repo := removeInstanceFromInstanceList(id(deleteInput),
       repo);
      instances := removeInstanceFromInstanceList(id(deleteInput)
         , instances) }
    -> successfulDeletionState
  {-END: DELETION-}
```

The functions used in the guards of the delete transition are as follows:

- The `isEntityDeleteOnUndestructableError` get as argument `id` and `instances` list. The function traverses the `instances` list until an instance is found that has `id` as its identifier. Then for corresponding instance it is checked whether it is destructable.
- The `isInstanceNotInRepositoryError` gets as argument an `id` and `repo` list. The function traverses the `repo` list, until an instance is found, for which the id matches the identifier of the corresponding instance, and true is returned. If such instance can not be found, false will be returned.
- The `isEntityMultiplicityMinimumError` has as argument `id` and `instances` list. First, the corresponding instance with identifier `id` is found. Then it is checked what the entity of corresponding instance is. Finally, the `instances` list is traversed to check the number of instances of this entity type, allowing to conclude if deletion would violate the minimum number of instances.

For the exact implementation of these functions, see appendix A.1

We also present the transitions for the error states that can only output the corresponding error, and the success states that can only output a `Success` output message.

```
    Entity_Unconstructable_State -> Output ! Entity_Unconstructable
        -> initialState
    Entity_MultiplicityMaximum_State -> Output !
       Entity_MultiplicityMaximum -> initialState
    Instance_NotInRepository_State_Delete -> Output !
       Instance_NotInRepository -> initialState
    Entity_Undestructable_State -> Output ! Entity_Undestructable
       -> initialState
    Instance_AlreadyInRepository_State -> Output !
       Instance_AlreadyInRepository -> initialState
    Instance_NotInRepository_State_Read -> Output !
       Instance_NotInRepository -> initialState


    {- SUCCESS STATES -}
    successfulCreationState -> Output ! Success -> initialState
    successfulDeletionState -> Output ! Success -> initialState
    successfulAdditionState -> Output ! Success -> initialState
    successfulReadState -> Output ! Success -> initialState


ENDDEF


MODELDEF Model ::=
    CHAN IN   CreateInput,  DeleteInput, AddInput, ReadInput
    CHAN OUT  Output
    BEHAVIOUR DmdslModel [CreateInput, DeleteInput, AddInput,
       ReadInput, Output]()
ENDDEF
```

In appendix A.1, one can see a full (but simple) DMDSL model expressed in the TorXakis language. When we use the stepper to make the tool step through the model, we can already observe that for a very simple model, the tool gets slower when more steps are taken. In particular on an iMac 2012 3,2GHz i5 we observe that after 100 transitions, taking a next transition already takes longer than 1 second. After 200 transitions this takes even longer than 4 seconds. On the one hand, this is not very surprising. When more instances are created, it takes more time to check if an instance with an id is in the list. The list is an Algebraic Data Type that is inductively defined. To check for containment, in the worst case the entire list needs to be checked ($O(n)$) time. Also, to check for an EntityMultiplicityMinimumError, it needs to be counted for a given Entity, how many instances in the repository exist with the corresponding Entity (Type). This, of course, is also an $O(n)$ traversal. Note that since these functions are used in the guards of transitions, they will actually impose restrictions on the input value the *solver* tries to generate. When there are more restrictions, it is harder for the solver to find an input value that satisfies all constraints. Apparently, this already puts quite a burden on the Z3 solver running in the background.

It is possible to now extend to models with association relations. An important aspect of this, is to determine which elements should get deleted as a result of the Delete action due to cascade deletion. In algorithm 1, a typical procedure is described to calculate the *Deletion_set* (when only source cascade deletion is considered).

Algorithm 1 COMPUTEDELETIONSET(*instances*)

1: $predecessors = \emptyset$
2: **if** $|instances|$=0 **then**
3: **return** $\emptyset$
4: **end if**
5: **for** $instance \in instances$ **do**
6: $predecessors := predecessors \cup$ SCDPREDECESSORS($instance$)
7: **end for**
8: $Deletion_Set := instances \cup predecessors$
9: $Deletion_Set := Deletion_Set \cup$ (COMPUTEDELETIONSET($predecessors$))
10: **return** $Deletion_Set$

Algorithm 2 SCDPREDECESSORS(*instance*)

1: $SCDPredecessors \leftarrow \emptyset$
2: **for** each $link \in s.Links$ **do**
3: **if** $link.target = instance \wedge link.association.SProperty.cascade$ **then**
4: $SCDPredecessors := SCDPredecessors \cup \{link.source\}$
5: **end if**
6: **end for**
7: **return** $SCDPredecessors$

The elements that are removed as a result of deleting an instance can be found by exploring the *links* set of the current state state s in a Breadth-First Search style algorithm, where only links are considered in which source cascade deletion is enabled. This has been implemented in Algorithm 1, and computes for an input set *instances*, what the elements are that would be removed as a result of deleting all elements in *instances*. The algorithm works as follows: When there are no elements in *instances*, the result of deleting *instances*, is that no elements are deleted. Hence, $\emptyset$ is returned. When there are elements in *instances*, we compute for each instance x in the *instances* set what its SCDPredecessors are (the instances that have a link to x where in the corresponding association relation source cascade deletion is enabled). The SCDPredecessors and *instances* are added to $Deletion_Set$ since the SCDPredecessors are then also instances that need to be deleted. For these SCDPredecessors, we also compute the corresponding deletion set, by making a recursive call to the algorithm, and add corresponding instances to $Deletion_Set$. The runtime of the algorithm to compute $Deletion_Set$ would be $O(l \cdot n)$ where n is the size of the $s.Instances$ and l the size of $s.Links$ (for state s).

While the imperative algorithm could be improved in runtime if the incoming links of a node could be found efficiently (using for example an adjacency list), these structures are lacking in TorXakis. We could now translate these algorithms into functional style in TorXakis. The iteration over lists can be done functionally using the head (`hd`) and tail (`tl`) functionalism recursively. E.g., Algorithm 2 is implemented in TorXakis as follows:

```
FUNCDEF SCDPredecessorOfInstance(instance :: Instance; links ::
    LinkList;
returnList :: InstanceList) :: InstanceList
::=
  IF links == Nil THEN returnList
  ELSE
    IF target(hd(links)) == instance /\ ( cascade(sproperty(
        getAssociationByAssociationName(associationName(hd(links))))
        )
== True )
```

```
      THEN SCDPredecessorOfInstance(instance, tl(links), Cons(
          source(hd(links)),
returnList))
    ELSE SCDPredecessorOfInstance(instance, tl(links), returnList)
    FI
  FI
ENDDEF
```

Note: In the current definition of the link data type we put the association name in the link and not the actual association object. The reason for this is that otherwise the output screen would be cluttered with information. Hence, in the current implementation, this function would actually take $O(la)$ time with a the number of associations. The association list should be traversed to find the corresponding association.

The full algorithm can be found in appendix A.2. One can observe that we make use of an append operator instead of the union operator. While the implementation of the union operator is possible (see A.3), it would be highly inefficient. When applied to two lists, for each element that would be added, the other lists need to be traversed to see if it is contained in the list. Hence, this would give a complexity of $O(nm)$ just for determining a single union, whereas in an imperative language this would typically be done in $O(n)$ time. We instead use the append operator and accept that duplicates will occur. However, appending two lists also requires appending each element of one list to the other, giving a runtime of $O(n)$. In an imperative language appending two lists can often be done in $O(1)$ (e.g. using a Linked List). It is not hard to see that the functional variant of Algorithm 1 has a high complexity. The result would be that for even a few links, there would be very heavy burden on the constraint solver, making exploration in practice infeasible, even for small models.

To conclude: The TorXakis tool can be used to encode the semantics of a DMDSL Model, however, due to its programming language, for part of the operations there does not appear to be a way to efficiently implement them. The functional language is currently not optimized to implement the desired operations. The creator of the tool states in the user documentation "TorXakis currently misses good usability, scalability does not always match the requirements of complex systems". Unfortunately, we have to agree that this indeed holds true in our context, making the tool practically not applicable in the context of DMDSL Models where some complex operations play a role. One could decide to further limit the scope of testing in a TorXakis model, and try to narrow down the models to test aspects in separation. This has not been investigated further. We also discovered in the attempts to translate a DMDSL model to STS semantics, that locations did not appear to be useful in the context of DMDSL model. The general idea remains that all CRUD+A actions are always enabled, and input is followed up with the output. Tool support for creating interesting test cases on the basis of the state variables was not available.

5.5 Model-Based Testing with Model Programs

Given the negative results for using *state/location-based* semantics for applying MBT, we will now have a look at *data-oriented* semantics. We now investigate if we can make use of Model Programs to encode the semantics of a DMDSL model. We start by considering some of the underlying formal theory, consider existing tools and finally describe conceptually how it can be done.

5.5.1 Conformance on Model Programs

One of the other formalisms we have briefly covered is the Abstract State Machine. In these structures, *states* are first-order structures over arbitrary data. The states are modified by *transition*

rules in the form of **if** *Condition* **then** *Updates*. In practice, we see the ideas of the Abstract State Machines in MBT applied in the form of *Model Programs* (e.g. in SpecExplorer [34], NModel [15] and PyModel [16]).

While there is not a unified formalism for these Model Programs, we will provide one, and an appropriate conformance relation. The formalism and conformance relation are based on [34] and [15]. A Model Program $P = (\mathcal{M}, \mathcal{V}, \mathcal{U})$ provides a finite set $\mathcal{M}$ of action methods, and a set of state variables $\mathcal{V}$, a set $\mathcal{U}$ denoting the universe of data values. States are defined in terms of the contents of their state variables. That is, two states are equal *if and only if* the content of their state variables are equal. Each action $m \in \mathcal{M}$ has a fixed arity (number of input arguments). For each action method m, one can define a precondition $Pre_m : \mathcal{U}^n \rightarrow \mathbb{B}$, where n is the arity of m. We then have for input vector $\vec{x} \in \mathcal{U}^n$, that $Pre_m(\vec{x})$ evaluates to a boolean value indicating whether the action method is enabled for $\vec{x}$ from a given state s. If $Pre(\vec{x})$ evaluates to true for s, then from s the m transition can be taken with the input parameters $\vec{x}$ which will produce a new state t. In this state t the state variables of s have been modified by m. These preconditions are sometimes referred to as guards. Besides modifying the state variables, the action methods may also produce output. We use $m(\vec{x})/o$ to denote that action method m produces output o for input vector $\vec{x}$ when taken from the current state. In these Model Programs a distinction between two types of actions is made:

- *Controllable actions*: Actions that can be executed on demand by the test tool.
- *Observable actions*: Actions the test tool can only execute when observed from the IUT.

Note that these 2 types of actions correspond to the input actions and output actions respectively of the LTSs with Input and Output of the **ioco** theory. In the **ioco** theory, the Transition System allowed for a notion of non-determinism in a way that two outgoing transitions of a state can have the same label, but lead to a different state. This is a notion of non-determinism that is not possible in Model Programs. In a Model Program, the execution of an action method will produce a new state on the basis of input arguments, and the state variables of the current state. Hence, there will be only one possible successor assuming the actions do not contain randomness (which should not be allowed). The state variables of a state thus uniquely identify the state the model is in. The Model Programs do support the following notion of *non-determinism*: A state can have multiple *observable actions* enabled. On the basis of observing the IUT, it is determined what the successor state is. We will see an example of this in example 5.5.2. This notion of *non-determinism* is also available for output actions in the LTSs of the **ioco** theory. When the Model Program lacks observable actions, the program is deterministic and is referred to as a *closed system*. A system that includes observable actions is referred to as an *event-driven system*.

Let MP be a model program. It is now important to wonder when *IUT conforms to MP*. In section 5.3.1 we have introduced this notion, and now provide the inductive definition in the context of model programs:

Definition 21 (State Conformance)**.** Let S_{MP} be a state in the model program. Let S_{IUT} be an arbitrary state of the i_{IUT} (which is the assumed model program of the implementation as specified by the *test assumption*).

- If there is a controllable action a allowed from S_{MP}, producing S'_{MP}, then a must be allowed from S_{IUT} and producing state S'_{IUT} such that S'_{MP} and S'_{IUT} conform.
- If there is an observable action a allowed from S_{IUT}, producing S'_{IUT}, then a must be allowed from S_{MP} and producing state S'_{MP} such that S'_{MP} and S'_{IUT} conform.

Definition 22 (Model Program Conformance Relation)**.** MP and IUT conform if and only if the initial states of MP and i_{IUT} conform.

Note that it is not clearly defined how output produced by the actions may affect the conformance relation. Hence, when actions produce output, one could adapt the definition slightly.

We will provide an example of a model program to get some more familiarity with the notions. We also specify how output influences the conformance relation.

Example 5.5.1 (Stack)**.** Suppose we wish to make a model program of a stack that operates on the integers. We require from a stack that it works with a last-in, first-out mechanism. We will provide some code of a model program in a hypothetical programming language for model programs.

```
State Variables:
        stack_list = []

Controllable Actions:
        push_enabled(x):
                return True
        push(x):
                stack_list.add_back(x)

        pop_enabled():
                list.size() > 0
        pop_back():
                return stack_list.pop_back()
```

The state variable of the model program is a `stack_list`. The action methods are the `push` and `pop` actions. These respectively add a provided element to the list, and remove the last element from the list. We specify in the `pop_enabled()` guard that pop is only allowed when there are elements in the list. We consider it undefined behavior what the implementation does in such cases, and therefore do not consider traces where we pop an empty stack. Furthermore, we require from the conformance relation that the produced output of the controllable actions should be equal.

Example 5.5.2 (Linear Data structure specification)**.** Suppose we have a specification on a higher abstraction level, and only require from the implementation that it is a linear data structure that respects insertion, and removal in arbitrary order. For such a specification, both the implementation of a Stack and a Queue would be accepted.

```
State Variables:
        list = []


Controllable Actions:
        insert_Enabled(x):
                return True
        insert(x):
                list.add(x)
        remove_send_enabled():
                list.size() > 0

        remove_send():
                //No implementation needed


Observable Actions:
        remove_receive_enabled(x):
                return contained(x, list)
        remove_receive(x):
                list.remove_first(x, list)
```

In the above example, we have as controllable action the `insert` action. For the remove action, we can not know which element the implementation will remove. As long as it is an element in the list, it conforms to the specification. We will make use of observable actions, to model this non-deterministic behavior. In this example, we have as a controllable remove action `remove_send`, that will inform the IUT of the removal action. We then observe the output of the remove action via the observable `remove_receive` action. When this action is 'observed', it suffices to check if the item x that the implementation removed is in the list state variable, which is done in its corresponding precondition `remove_received_enabled`. If this action is not enabled in the Model Program, a problem regarding conformance would be detected (see definition 21). Note that this Model Program would consider both a Queue and a Stack as valid implementations, whereas in the previous example only a Stack would be considered a valid implementation. Non-determinism (via observable actions) is used to accept any element from the list as valid output.

5.5.2 Tools

Popular tools that rely on these types of model programs are Spec Explorer [34], NModel [15] and PyModel [16]. The Spec Explorer is an MBT tool developed by Microsoft Research. Jonathan Jacky, Margus Veanes, Colin Campbell and Wolfram Schulte wrote the book 'Model-Based software Testing and Analysis with C#' which also has been created at Microsoft Research. In this book, they dive into the technique of Model-Based Testing using Model Programs. To support the book, they developed the NModel Library. They point out that many of the ideas shown in the book are applicable to other tools, among which Spec Explorer. Jonathan Jacky, one of the authors of the book, then developed PyModel, of which the foundation can be found in NModel.

The latest release of Spec Explorer became public in 2010, and Microsoft unfortunately no longer supports this tool. On the SpecExplorer page of Microsoft, they also refer to NModel as an alternative. Unfortunately, the corresponding page also seems to be no longer available. It seems this framework is also no longer supported nor available. The PyModel tool its framework and website are still available. Jonathan Jacky provided its last functional update in 2013. The tool was written in Python 2, which harms usability in a time where Python 2 has been declared end-of-life, and Python 3 has been standardized. Fortunately, in early 2022 a GitHub contributor zlorb created a new fork, where the PyModel tool has been updated to Python 3. Jonathan Jacky, has now placed a url on the GitHub page to refer to this updated version, where it supposedly will be actively developed. We will therefore have a closer look at the PyModel tool since it currently seems most relevant.

5.5.2.1 Expressing the Semantics of ASOME as Model Program

We will use the PyModel tool to create a Model Program. One of the advantages of being able to encode a Model using an existing programming language is that you can make use of all features the often extensive programming language has to offer. When the model is encoded with a specification language that is specific for the MBT tool, the number of built-in data types and algorithms could be very limited, as we have seen for TorXakis and Axini. The research that went into the popular known programming languages is typically much more extensive than one can expect for a specification language specific to a MBT tool. As a result, operations can be encoded much more efficiently in such a language. Furthermore, as is pointed out in [15], being able to encode models in a familiar programming language, makes the tool usable by most people involved in the technical aspect of the software production process. We will now consider roughly how to translate the main concepts to Model Programs of PyModel:

State Variables Following the state variables defined in chapter 3.3.2. We can use a Python set `instances` for I, a Python set `repo` for $REPO$, and a Python list `links` to represent the bag semantics of L.

Actions The actions are the *Create*, *Read*, *Update*, *Delete*, and *Add* actions. The actions should modify the state variables as described in chapter 3.3.2. In PyModel, the actions can simply implemented as Python functions. Implementing these actions, should not induce many problems in the Python programming language. The CRUD+A actions can all be *controllable actions* since such actions can be executed on demand by the test tool. The output that will be produced, will depend on the state (and thus state variables) from which the action is fired, and the provided input arguments.

Remember that sometimes multiple messages are allowed as output messages according the specification. For example, when trying to delete an instance of an entity that is undestructable, but deleting it would also violate the minimum multiplicity of the corresponding entity type the `Entity_Undestructable`, or the `Entity_MultiplicityMinimum` is both considered to be valid output. The IUT conforms to the specification when either of these messages is outputted. Such non-deterministic behavior could in fact be modeled as *observable action* in a Model Program. In the guard of the *observable* action we can then check whether the output that is provided, is one of the allowed outputs (e.g., after executing an action, we maintain a list of valid outputs, and in the guard of the observable action, we check for containment in the valid output list). Having non-deterministic behavior adds some complexity between the bridge of the PyModel mode, and the IUT, since in principle at any time such observable action can occur.

In the context of DMDSL models the output action that is produced, does not actually influence the next state. Either the Model Program only finds a `Success` acceptable, and the state variables would be modified accordingly, or the Model Program finds elements in a set of error messages acceptable and the state variables remain unchanged. Which of the error messages the IUT outputs, does not influence the next state of the Model Program. Furthermore, it is deterministic whether a `Success` message occurs or whether an error message will be output. This property allows us to actually model it as a *closed system.* We could, for each action, produce as action-output the set of outputs that are allowed to occur, and modify the state accordingly. After executing the action on the IUT, and the output is passed back, it suffices to *check for containment of the IUT output in the set that is output by the action of the Model Program*. Since in such a setting we do not have observable actions and make use of action output, we can adapt definition 21 to our context.

Definition 23 (State Conformance for DMDSL models)**.** Let S_{MP} be a state in the model program. Let S_{IUT} be an arbitrary state of the i_{IUT} (which is the assumed model program of the implementation as specified by the *test assumption*).

- If there is a controllable action a allowed from S_{MP} producing S'_{MP} and output O, then a must be allowed from S_{IUT} and producing state S'_{IUT} *and output* o' such that S'_{MP} and S'_{IUT} conform *and* $o' \in O$.

Example 5.5.3. Suppose that from the state S_{MP} a *Create* action using *Entity*1 and an empty list of links would produce output {`Entity_MultiplicityMaximum, Entity_Unconstructable`} i.e. we have `Create(Entity1, []) /` {`Entity_MultiplicityMaximum, Entity_Unconstructable`}. Then, we desire from S_{IUT} that the `Create(Entity1,[])` action can be performed, and that the output produced is either `Entity_MultiplicityMaximum` or `Entity_Unconstructable`. If for example S_{IUT} would produce a `Success`, then a problem regarding conformance is detected (since definition 23 requires `Success` ∈ {`Entity_MultiplicityMaximum, Entity_Unconstructable`} which clearly does not hold).

Guards Since the CRUD+A actions are always enabled, we do not need guards. We may however decide to use guards if we want to restrict to specific kinds of traces in the testing process.

Domains In **ioco** theory on STSs, there is a reliance on constraint solvers to construct input arguments. In Model Programs, finite domains need to be provided for the actions. Sometimes the argument of an action has an infinite domain. Then, the tester should try to finitize the domain in

a sensible manner. One way is to provide fixed finite domains for the action arguments. One can, however, also provide a lambda expression that creates a finite domain for the action arguments dependent on the current state. This is a very powerful feature in our context. For example, the `Add` operation requires an instance. It depends on the state what instances exist. On a state basis, you can provide a finite domain for this. For the `identifier` parameter, one can construct for a given state a finite domain of identifiers of instances that currently exist. Of course, one can additionally add some identifiers that do not have a corresponding instance. Making use of the *uniformity assumption*, it suffices to only add a limited amount of such identifiers. There is probably no value in having an infinite number of identifiers of which no corresponding instance exists (even though these identifiers exist from a theoretical point of view). We will further discuss the finitization of these domains in the testing strategy in chapter 8.1.

Testing Strategy To guide the tool in Model-Based Testing, the PyModel tool allows you to write a *custom* strategy. That is for a provided state, using the finite domains, a finite set of enabled transitions is constructed by the PyModel. One can manually write an algorithm in Python to select one of the enabled actions. A few example strategies *ActionNameCoverage* and *StateCoverage* are provided by the tool.

Since the tool allows one to write complex operations in the actions, makes use of finite domains that can be re-evaluated on a state basis, and offers functionality to write custom strategies, this tool seems very promising for our context. Translating a simple DMDSL model as PyModel showed, that it was indeed capable of generating test cases in an efficient manner, even when associations are considered. We will not provide the test model and translation here, as we will continue to use PyModel in our case study.

5.6 Result of MBT selection

We have considered two different formalisms, and three tools in total. We summarize the information obtained during our selection procedure in table 5.1

Tool	Formalism	Conformance Relation	Expressiveness	Scalable	Treatment Symbolic Data
TorXakis	STS	**ioco**	Sufficient	Insufficient	Solver on action argument with support for functions in constraints.
Axini	STS/LTS	**ioco**	Insufficient	N/A	Solver on action argument without support for functions in constraints.
PyModel	Model Program/ASM	Definition 21	Sufficient	Sufficient[2]	(State-dependent) finitized domains.

Table 5.1: Summary of results in tool selection procedure.

The analysis showed that of the three tools considered, PyModel has the most potential to apply MBT in the context of the DMDSL code generator. The underlying specification language is sufficiently expressive, and seems to be sufficiently scalable. Furthermore, the fact that domains of actions can be finitized on the basis of the current state, make it very useful in the DMDSL model context.

[2]Given the finitized domains are not too large.

Chapter 6

Encoding Models in an MBT Tool

We use PyModel as a Model-Based Testing tool for testing the code generator of the DMDSL language. In this chapter, we will investigate how to translate the dynamic semantics of a DMDSL model to the language used in PyModel (i.e., mostly Python concepts). First, we will discuss the scope of test models covered. Then we will briefly discuss how generation is done, and what files are produced. Then, we will go more in-depth on the files that are produced. In particular we discuss how to transform the elements found in the Abstract Syntax of DMDSL models into PyModel constructs. Then, we will transform the dynamic constructs of the DMDSL language to PyModel constructs. Being able to make this translation step allows us to make a *model-to-text* transformation so that arbitrary DMDSL models can be automatically transformed into PyModel models. Finally, we will briefly discuss how PyModel provides a way to debug a translated model.

6.1 Scope of Modeling

We will concern ourselves with DMDSL models in the testing process, that satisfy the following assumptions:

- The code is generated for intraprocess communication.
  - Code can also be generated for interprocess communication. In that case, domain interfaces are deployed in such a way that multiple processes can connect to it via a shared memory segment. To make the scope of testing feasible in this work, will not consider this behavior, and restrict ourselves to code generated for intraprocess communication.
- All entities and association relations have finite multiplicities.
  - The ASOME tool gives a warning if the entities do not have finite multiplicities. It is however not forbidden, but it goes against the design guidelines of ASML. A lack of finite multiplicities could result in overflowing memory when too many entities are created. The association multiplicities do not need to be bounded. We however restrict ourselves to models where these are bounded. Since one could create test models in which these multiplicities are very large, and additionally, it is expected that bugs already occur for 'small' test models (*small scope hypothesis*), it is not expected that the testing procedure will suffer greatly from this assumption.
- Of the association relations, we only consider association relations in which source cascade deletion is enabled.
  - While, we will support association relations in which target cascade deletion is enabled, making such models in ASOME is not (yet) supported. The semantics made by Derasari

are on this perspective actually ahead of the implementation. We also do not consider association relations in which cascade deletion is disabled since the semantics of Derasari do not cover this behavior properly[1]. Furthermore, we do not consider the inheritance relation, since it is not covered in the semantics.

6.2 Generation of PyModel Model

To translate a DMDSL model to a PyModel model we need to do a model-to-text transformation. The DMDSL language is built using the Eclipse Modeling Framework (EMF). A metamodel of the DMDSL language has been built using EMF and the syntax has been defined using Xtext. The latter automatically provides a parser for the DMDSL language. We thus only need to be able to transform a given model to a textual PyModel specification i.e., to make a *model-to-text* transformation. We select Acceleo (`https://www.eclipse.org/acceleo`) as a template-based code generator framework to implement this model-to-text transformation. The primary reason for this is that the same framework is used to develop the code generator for the DMDSL language, making embedding the model-to-text transformation in the ASOME tool easier.

The model-to-text transformation takes as input a DMDSL model, and produces the files listed below and visualized in figure 6.1:

- `LanguageConstructs.py`: Here are the Abstract Syntax elements *independent* of the provided model.
- `GeneratedFromModel.py`: Here are the elements that *depend on the input model* (the entities and associations).
- `CRUDA` folder: Here is the PyModel translation for handling output for the CRUD+A actions. A folder is generated containing for each of the CRUD+A actions a file. In such a file, for each output message that might occur for the CRUD+A action, a method is provided that checks whether the output message occurs for the action (e.g. a method to check whether `Association_SourceMaximum` occurs). These are all *independent* of the input model.
- `Model_Translation.py`: Here will be the specification of the Model Program (declaration of state variables, action methods, guards and domains of actions). The action method makes calls to the methods in the CRUDA folder, to check what output occurs for the action executed. Furthermore, the action methods modifies the state variables accordingly. The file is practically *independent* of the model, except for the fixed domain of entities which is dependent on the test model.
- `stepper.py`: A file that serves as a bridge between the IUT and the PyModel model. This will be part of the adaptor and is covered in chapter 7. The stepper is practically *independent* of the input model, except a reference to the generated code will be made, which is dependent on the name of the model.

The model-to-text could thus actually only output `Model_Translation.py` since the remaining files are practically independent of the test-model under consideration. We decide to output all files to get a complete package for testing.

After integration of the *model-to-text* transformation in the ASOME environment, it is possible in the ASOME tool to generate a PyModel specification by right-clicking on the project as seen in figure 6.2. With orange, we indicate that the file does not rely on the input, with yellow, we indicate that only a small part of the file relies on the input model, and with green, we indicate that the files largely depend on the model.

[1]Some issues regarding the respecting of multiplicity constraints when associations do not have source cascade deletion enabled were found by Wilbert Alberts during the creation of this thesis. Investigation into why these did not give problems with respect to Repository Consistency in Alloy is required.

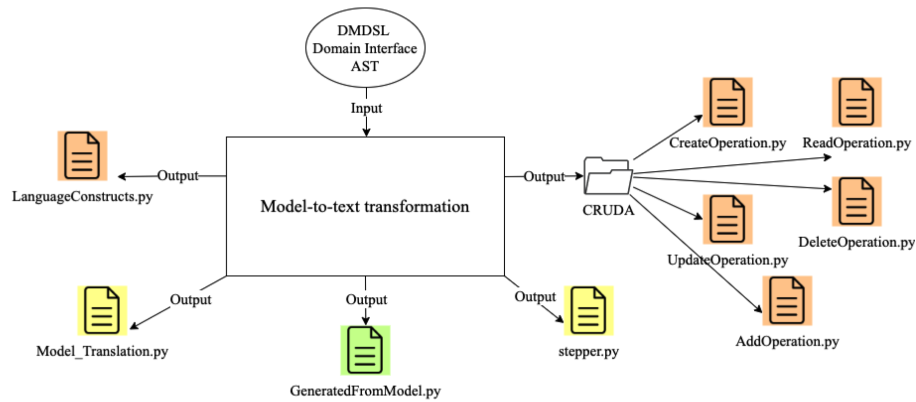


Figure 6.1: Files produced by the *model-to-text* transformation.

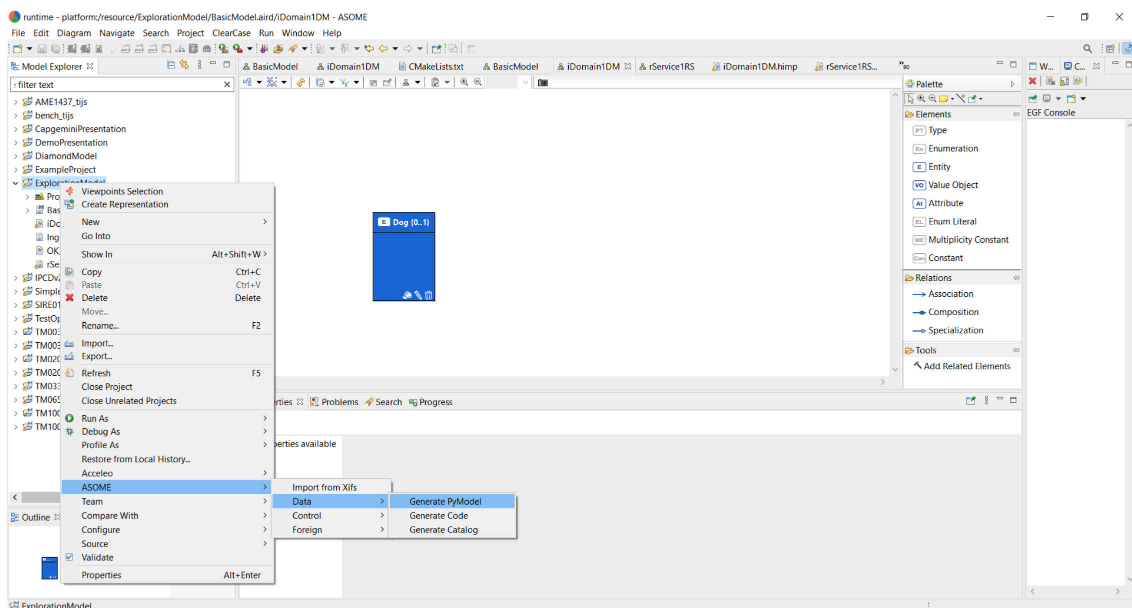


Figure 6.2: Generating a PyModel specification from a DMDSL Model.

6.3 Abstract Syntax Elements

There are certain language constructs that we need to be able to specify in the specification language of the MBT tool. The elements of the Abstract Syntax are independent of the input model and will be generated in `LanguageConstructs.py`.

6.3.1 Multiplicity

The notion of a multiplicity is a tuple $(minimum, maximum)$. We will map this notion of multiplicity to python as follows:

```
class Multiplicity:
    def __init__(self, min, max):
        self.min = min
        self.max = max
```

6.3.2 Entity

An element $e \in$*Entity* has been formalized as a tuple $\langle C, M, D, N \rangle$ where C denotes constructability, M mutability, D deletability, and N denotes the entity multiplicity. We will translate this into a Python concept as follows:

```
class Entity:
    def __init__(self, is_constructable, is_mutable, is_deletable, multiplicity):
        self.is_constructable = is_constructable
        self.is_mutable = is_mutable
        self.is_deletable = is_deletable
        self.multiplicity = multiplicity

        self.out_associations = None
        self.in_associations = None
```

Additionally, we will define a few member functions:

- get_outgoing_associations(): At the first call, it will use the global `associations` list to determine the outgoing associations of the corresponding entity. These will be added to the `out_associations` list member variable. (This allows returning them in $O(1)$ time for all consecutive calls.)
- get_incoming_associations(): At the first call, it will use the global `associations` list to determine the incoming associations of the corresponding entity. These will be added to the `in_associations` list member variable. (This allows returning them in $O(1)$ time for all consecutive calls.)
- get_number_of_instances(): The function iterates over state variable `instances` (see section 6.5.1) to count the number of instances with corresponding entity type.

These turn out to be practical in the dynamic semantics.

6.3.3 Instance

Entities can be instantiated. In the semantics specified in the work of Derasari, we have that every instance corresponds to some entity, and each instance can be uniquely identified by its id. We model this notion of instance as a separate concept, instead of relying on Python instantiations of an Entity (see the remark in section 6.4.1).

The translation is as follows:

```
class Instance:
    def __init__(self, Entity, instance_name, id):
        self.Entity = Entity
        self.id = id
        self.instance_name = instance_name

    def get_id(self):
        return self.id

    def __eq__(self, other):
        if other is None:
            return False
        return self.id == (other.id) and self.Entity == (other.Entity)

    def __repr__(self):
        return fsm_string(self.instance_name)
```

Hence an instance is defined by the id, the corresponding entity (type) and the instance name. The instance name is a new notion that turns out to be practical for the adaptor between model and code.

6.3.4 Association

To define *Associations*, we first need the notion of an *AssociationEndProperty*. The notion of an *AssociationEndProperty* has been defined as a tuple $\langle Cascade, Multiplicity \rangle$. We will translate this into a Python construct as follows:

```
class AssociationEndProperty:
    def __init__(self, is_cascade, multiplicity):
        self.is_cascade = is_cascade
        self.multiplicity = multiplicity
```

An *Association* is defined as a tuple $\langle source, target, SProperty, TProperty \rangle$. We can now translate this into a Python construct as follows:

```
class Association:
    def __init__(self, association_name, source_entity, target_entity,
    source_association_property, target_association_property):
        self.association_name = association_name
        self.source_entity = source_entity
        self.target_entity = target_entity
        self.source_association_property = source_association_property
        self.target_association_property = target_association_property

    def __eq__(self, other):
        return self.association_name == other.association_name and \
        self.source_entity == other.source_entity and \
        self.target_entity == other.target_entity
```

Even though it is not added in the formal model, there should also be an association name. An association between two entities is named in the model, and in practice, it is practical to keep track of this information. There could be two association objects that are exactly equal in terms of $\langle source, target, SProperty, TProperty \rangle$ but differ in association name. Since sometimes implicit copies of the association object happen, we also define an equality operator. As expected, two associations are equal when both associations have the same source entity, target entity, and association name.

6.3.5 Link

A link is defined as tuple $\langle source, association, target \rangle$. We can model this in Python as follows:

```
class Link:
    def __init__(self, source_instance, association, target_instance):
        self.source_instance = source_instance
        self.association = association
        self.target_instance = target_instance

        self.id = uuid.uuid4()

    def __eq__(self, other):
        return self.id == other.id

```

Since implicit copies of links happen during execution, it is not sufficient to use the default equality operator, which checks if the compared elements are really the same object in memory. This does not hold true when a copy is compared to the original. Comparison on the basis of equality of `source_instance`, `target_instance`, and `association` is also not sufficient. For different links, this equality could hold (remember the bag semantics of links in states), even though they are not the same representative of the link. Therefore, we add a hidden member variable that assigns a unique id to the links. The equality operator is now implemented by the equality of this hidden id.

6.4 Model-Dependent Constructs

The remaining constructs will be created on the basis of the model considered, and need information from the model under consideration. We will consider these elements here. They will be generated in `GeneratedFromModel.py`

6.4.1 Entities

For each Entity in the model, a separate class will be generated that inherits from the *Entity* class, with the `is_constructable`, `is_mutable`, `is_deletable`, and `multiplicity` set accordingly. The model-to-text transformation thus needs to generate these according the provided model. E.g., for the model in figure 3.1 the model-to-text transformation generates:

```
class Dog(Entity):

    def __init__(self):
        super().__init__(True, True, True, Multiplicity(0, 1))


    def __repr__(self):
        return fsm_string("Dog")

class Person(Entity):

    def __init__(self):
        super().__init__(True, False, True, Multiplicity(0, 3))


    def __repr__(self):
        return fsm_string("Person")

DogSingleton = Dog()
PersonSingleton = Person()
```

Furthermore, the default `__new__(cls)` function (which is called to create a Python Object) is overwritten with an alternative implementation that ensures that for each of these entities, only one instance can exist. We apply the *Python Singleton pattern* to the generated Entity sub-classes. We desire that for each entity (type), only one Python object can exist. Global variables `DogSingleton` and `PersonSingleton` are instances created to use as *representatives* for the corresponding entity (type). We will model instances of these entities separately.

> Note: Python itself allows for the instantiation of classes. Although we could try to relate the Python instances of an Entity class to the notion of instance as used in the semantics of DMDSL, we decided to keep these concepts separate from each other to avoid confusion. This allows us to define precisely what an instance of an entity means according to the semantics of the DMDSL language, without relying on analogous concepts that are inherited from Python instances.

6.4.2 Associations

In the formal model *Association* is a set of existing associations. For each association in the input model an Association class will be instantiated accordingly, and added to the global variable `associations` that stores all associations. The model-to-text transformation thus needs to generate these according to the provided model. E.g., for the model in figure 3.1 we generate:

```
associations = [
    Association("owner", DogSingleton, PersonSingleton,
                                AssociationEndProperty(True, Multiplicity(0,1)),
                                AssociationEndProperty(False, Multiplicity(3,5)))
```

```
5 ]
```

6.5 Dynamic Language Constructs

In the representation of the dynamic semantics in PyModel, we first determine the state variables. As this notion of the PyModel model is similar to the notion of the transition system as introduced in the work of Derasari, the translation of concepts will be similar. Deviations will be there, but these are often subtle.

6.5.1 State Variables

We identify a state using the variables:

- `instances`: The set of instances that were created (but were never deleted). These instances may or may not have been added to their corresponding repositories.
- `repo`: The set of instances that are currently in the corresponding repository.
- `links`: A list of links between instances that currently exist.
- `all_identifiers`: A list of all identifiers of the instances that have ever been created.
- `highest_id`: an id that is strictly greater than the identifiers of instances that were ever created.

Note that `instances` corresponds to I, `repo` corresponds to $REPO$ and `links` corresponds to L in the work of Derasari. We don't need the evolving *type* function, as in our translation we can simply check for a given instance what the entity type is (it is a member of the Instance class). Furthermore, we also do not use the *output* state variable. Output will be handled via transitions, and will be considered in 6.5. Additionally the `all_identifiers` list is added since for the testing process it is good to keep track of all the identifiers that were ever created (which will be elaborated on in chapter 8). The `highest_id` is used, so that for each instance that is created, a unique identifier can be used as the corresponding identifier. We simply use `highest_id` as identifier, and increment it.

6.5.2 PyModel Actions

6.5.2.1 Create

In the semantics, a $create(e, links)$ transition is defined. As an input argument, this transition takes an entity e and a bag of *links*. The translation to PyModel is as controllable action `Create(entity_links, instance_name, instance_id)` where the arguments are as follows:

- `entity_links`: A tuple ⟨`entity`,`new_links`⟩ containing the entity `entity` of the created instance and a bag of links `new_links` in which the created instance will be the source.
- `instance_name`: This will be the representative for the newly created instance. (I.e., the actual instance that is generated by executing the implementation will be matched to this abstract instance name.)
- `instance_id`: This will be the abstract id for the instance that is created. (In the IUT the actual instance that will be created also gets assigned an id, which is unknown in the context of the PyModel model. The abstract id will be used as a representative for this id.)

> Note: The Read and Delete actions are executed by passing as argument the identifier of the instance. The actual instance first needs to be added to the repository before the repository 'knows' what instance corresponds to the identifier. The Add action, therefore, uses the actual instance as argument in the call. We use `instance_name` as representative for using the actual instance as argument, and `instance_id` for using the identifier as argument.

When none of the error conditions hold, the instance of the entity can be successfully created. Derasari defined the relation between the current state s and next state s' as follows:

$$\begin{aligned} s'.I &= s.I \cup \{newInstance\} \\ s'.L &= s.L \cup links \\ s'.Repo &= s.Repo \\ s'.output &= \texttt{Success} \\ s'.type &= s.type[newInstance \rightarrow e] \end{aligned}$$

In the PyModel transition system the following happens:

$$\begin{aligned} s'.I &= s.I \cup \{newInstance\} \\ s'.L &= s.L \cup links \\ s'.Repo &= s.Repo \\ s'.highestIdentifier &= s.highestIdentifier + 1 \\ s'.allIdentifiers &= s.allIdentifiers \cup \{newInstance.id\} \end{aligned}$$

It may also be the case that some error conditions hold, and then corresponding error messages are output. For sake of example we will provide the concrete action method for the Create method:

```
1  def Create(entity_links, instance_name, instance_id):
2      global instances, repo, links, all_identifiers, highest_id
3  
4      entity = entity_links[0]
5      links_needed = entity_links[1]
6  
7      check_log = CRUDA.CreateOperation.creation_check(entity, links_needed)
8      no_errors = (len(check_log) == 0)
9      output = {}
10     if no_errors:
11         output = {"Success"}
12         entity_instance = Instance(entity, instance_id)
13         # Add created instance
14         instances.add(entity_instance)
15         # Add new identifier
16         all_identifiers.append(entity_instance.id)
17 
18         # Add the new links and set the source of these links to the newly created
       instance
19         for link in links_needed:
20             link.source_instance = entity_instance
21         links = links + list(links_needed)
22         highest_id = highest_id + 1
23     else:
24         # Set the output to the errors
25         output = check_log
26     return list_output(output)
```

Error Messages The creation of an instance of an entity may fail. Therefore a set of failure cases, and corresponding output messages has been defined. We will translate these to the PyModel

model using the helper functions below. In some of the helper functions, we use arguments corresponding to the following:

```
    entity = entity_links[0]
    new_links = entity_links[1]
    new_link_associations = {link.association for link in new_links}
    outgoing_associations = entity.get_outgoing_assocations()
    required_associations = {association for association in outgoing_associations
    if  association.target_association_property.multiplicity.min >= 1}
```

1. $e.C = \textbf{False} \implies \mathit{output} = \texttt{Entity_Unconstructable}$

```
def is_entity_unconstructable_error(entity):
    return not entity.is_constructable):
```

2. $|\{i \in s.I \mid \mathit{type}(i) = e\}| = e.N.\mathit{maximum} \implies \mathit{output} = \texttt{Entity_MultiplicityMaximum}$

```
def is_entity_multiplicity_maximum_error(entity):
    return entity.get_number_of_instances() >= entity.multiplicity.max:
```

3. $\{a \in \mathit{Association} \mid \exists_{link}[\mathit{link} \in \mathit{links} : a = \mathit{link.association}]\} \not\subseteq e.\mathit{OutgoingAssociations} \implies \mathit{output} = \texttt{Entity_UnexpectedAssociation}$

```
def is_unexpected_association_error(outgoing_associations,
    new_link_associations):
    return not new_link_associations.issubset(outgoing_associations):
```

4. $\{a \in \mathit{Association} \mid a.\mathit{source} = e \land a.\mathit{TProperty.minimum} \geq 1\} \not\subseteq \{\mathit{link.association} \in \mathit{Assocation} \mid \mathit{link} \in \mathit{links}\} \implies \mathit{output} = \texttt{Entity_MissingAssociation}$

```
def is_entity_missing_association_error(required_associations,
    new_link_associations):
    return not required_associations.issubset(new_link_associations):
```

5. $\exists_a[a \in e.\mathit{OutgoingAssociations} : |\{\mathit{link} \in \mathit{link} \mid a = \mathit{link.association}\}| > a.\mathit{TProperty.multiplicity.maximum}] \implies \mathit{output} = \texttt{Association_TargetMaximum}$

```
 def __number_of_links_with_association(association, links):
    count = 0
    for link in links:
        if link.association == association:
            count += 1
    return count


def is_association_target_maximum_error(outgoing_associations, new_links):
    for association in outgoing_associations:
        if __number_of_links_with_association(association, new_links) >
    association.target_association_property.multiplicity.max:
            return True
    return False

```

6. $\exists_a[a \in e.\mathit{OutgoingAssociations} : |\{\mathit{link} \in \mathit{links} \mid a = \mathit{link.association}\}| < a.\mathit{TProperty.multiplicity.minimum}] \implies \mathit{output} = \texttt{Association_TargetMinimum}$

```
def is_association_target_minimum_error(outgoing_associations, new_links):
    for association in outgoing_associations:
        if __number_of_links_with_association(association, new_links) <
    association.target_association_property.multiplicity.min:
            return True
    return False
```

7. $\exists_{a,i}[a \in e.OutgoingAssociations \wedge i \in s.Instances \wedge i.Entity = a.target.Entity : |i.incomingLinks(a) \uplus links(a,i)|] > a.SProperty.multiplicity.maximum \implies output =$ `Association_SourceMaximum`

```
def __number_of_incoming_links_to_instance(instance, association, links):
    count = 0
    for link in links:
        if link.target_instance == instance and link.association == (
    association):
            count += 1
    return count

def is_source_maximum_multiplicity_violation(outgoing_associations, new_links)
    :
    from Model_Translation import repo, links
    for association in outgoing_associations:
        for instance in repo:
            number_of_existing_links_pointing_to_instance =
    __number_of_incoming_links_to_instance(instance, links)
            number_of_new_links_pointing_to_instance =
    __number_of_incoming_links_to_instance(instance, new_links)
            if number_of_existing_links_pointing_to_instance +
    number_of_new_links_pointing_to_instance \
                    > association.source_association_property.multiplicity.max
    :
                return True
        return False
```

6.5.2.2 Read

In the semantics of Derasari $read(identifier)$ transition has not formally been defined. It is fortunately very trivial. The translation to PyModel is as follows: `Read(identifier)`

- `identifier`: The id of the instance we wish to read from the corresponding repository.

When no errors occur, all state variables remain unchanged and `Success` is output. For the translation of the error output see appendix C.2.

6.5.2.3 Update

In the semantics an $update(instance, links)$ has formally been defined. As input argument this transition takes an instance *instance* and a bag of links *links*. The translation to PyModel is as follows: `Update(instance_links)`.

- `instance_links`: a tuple $\langle$`instance`, `links`$\rangle$ containing the instance to update and a bag of links that replaces the original set of links that instance is the source of.

When none of the error conditions hold, the instance can successfully be updated and the links set will be modified accordingly and `Success` is output. All other state variables remain the same. For the translation of the error output see appendix C.2.

6.5.2.4 Delete

In the formal semantics the $delete(identifier)$ has been defined. As input argument this transition takes an instance identifier $identifier$. The translation to PyModel is as follows: `Delete(identifier)`.

- `identifier`: an identifier of an instance.

Deletion executed on an instance, will due to cascade deletion induce a `deletion_set` of instances that should be deleted. Using a fixed point computation algorithm we can determine the `deletion_set` as follows:

```
def __get_deletion_set(instance):
    global instances, repo, links, all_identifiers
    deletion_set = {instance}

    is_changed = True
    deletion_set_size = len(deletion_set)
    last_added = {instance}
    while is_changed:
        new_elements = set([])
        for instance in last_added:
            for link in links:
                if link.target_instance == instance and link.association.
    source_association_property.is_cascade:
                    new_elements.add(link.source_instance)
                if link.source_instance == instance and link.association.
    target_association_property.is_cascade:
                    new_elements.add(link.target_instance)
        deletion_set = deletion_set.union(new_elements)
        last_added = new_elements
        if len(deletion_set) == deletion_set_size:
            is_changed = False
        else:
            deletion_set_size = len(deletion_set)
    return deletion_set
```

Since the number of instances in a given state is finite, termination is guaranteed.

Each of the instances in the `deletion_set` should be removed from the `repo` and `instances`. When none of the error conditions hold, the `deletion_set` can successfully be removed. For the translation of the error output see appendix C.3.

6.5.2.5 Add

In the semantics an *add*(*instance*) has formally been defined. As input argument this transition takes an instance *instance* The translation to PyModel is as follows: `Add(instance)`.

- `instance`: an instance object.

When none of the error conditions hold, the instance is added to the repository state variable, and all other state variables remain the same. For the translation of the error output see appendix C.4.

6.6 Exploration of PyModel Models

We could wonder about the correctness of the PyModel specification. How can we know that correctly translated the semantics? That is a fair point, but we should realize that the PyModel specification is encoded at a much higher abstraction level than the IUT. This makes the implementation of the PyModel specification easier and makes it more feasible to keep an overview and less likely for bugs to occur. Furthermore, one should not forget that the generation of the PyModel specification is implemented in isolation from the generation of the IUT. The chances are small that the exact same bug that occurs in the C++ code, is also in the model program. When something is incorrectly encoded in the PyModel specification program, but correct in the IUT, we would expect that during MBT some unexpected behavior happens that should make us aware of issues. PyModel also allows you to create an FSM of the model program, by exploring it to a provided transition depth. In this FSM, we can explore the transitions that are enabled and inspect the resulting state. The FSM that PyModel can generate is written in the DOT graph description language. These .dot files can be converted to a .svg to visualize the graph in an internet browser. By hovering the mouse over the states, one is able to inspect the state variables

of a state, and check if these variables are as one would expect them to be. This also helps in detecting problems in the model program at an early stage.

We explore the DMDSL model of figure 6.2 as an FSM with transition depth 13 in figure 6.3. If we explore the .svg of the FSM in the browser and hover over state 1, then we indeed observe that the `instances` list now contains `instance0`. From state 1, we have that a `read(0)` transition indeed induces an `Instance_NotInRepository`, error as we can also see from the image. In state 2, `instance0` got added, and we can observe (although the output is a bit cluttered) a `Success` output when reading `instance0`. You might expect some more transitions to be visualized in this image. Due to the finitization of the domains, we only have the provided transitions for this simple model explored until transition depth 13. In chapter 8, we will elaborate on this finitization process.

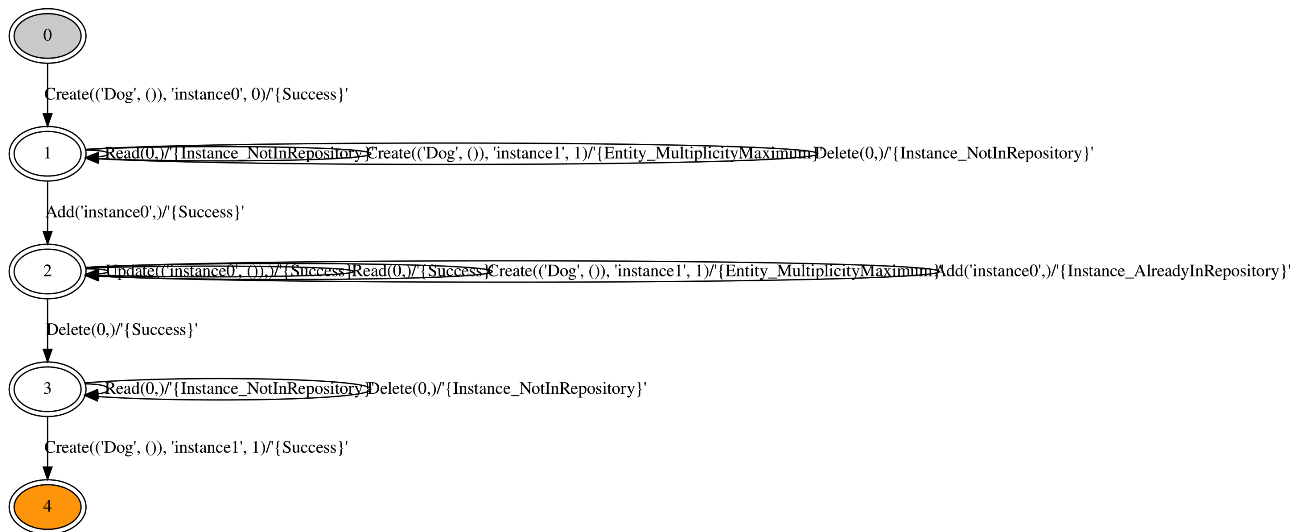


Figure 6.3: FSM of an example model.

Chapter 7

MBT Tool Adaptor

Now that we are able to encode DMDSL test models in PyModel, we want to start connecting PyModel to the IUT. We will need an adaptor for this. In this chapter, we will first discuss the high-level architecture of the adaptor. Then, we will explain how PyModel actions (i.e. abstract actions) are translated into concrete API calls (i.e. concrete actions). Afterwards, we will discuss how PyModel action outputs can be related to concrete outputs (e.g. exceptions). Finally, we will discuss how to make the adapter compatible with the generated code of some DMDSL model. In particular, we discuss how to overcome the challenge that the adapter relies on interfaces of the IUT that are dependent on the DMDSL model considered.

7.1 Architecture of Adaptor

The PyModel tool is able to generate abstract test cases on-the-fly. The MBT tool should be able to communicate these test sequences with the implementation in order to check if the behavior of the implementation conforms to the specification. The model is at a high abstraction level and we should be able to accommodate the low-level differences of interfaces and output messages so that checks on this higher abstraction level can be done. We, therefore, need to develop an adaptor.

As explained, the API of the model, which is often rather abstract, does not exactly match the API of the IUT. The task of the adaptor is to bridge this gap. In [20] the adaptor is assigned the following responsibilities:

- *Setup:* Setup the IUT to make it ready for testing.
- *Concretization:* Translate model-level abstract operation calls and abstract input values to concrete IUT calls.
- *Abstraction:* Obtain the IUT results from the concrete calls and translate them back into abstract values and pass these back to the model for comparison.
- *Teardown:* Shut down the IUT at the end of each test sequence.

In our case, the adaptor consists out of multiple components, on which we will elaborate in the coming subsections.

- `stepper.py`: Handles communication with the C++ process being tested. In our case, this is the wrapped version of the IUT.
- `main.cpp`: Handles incoming and outcoming communication of the C++ process, and communicating this to the CallTranslator.
- CallTranslator: Wrapper around the IUT that makes actual API calls to the IUT, and translates concrete output messages s.a. exceptions back to abstract output messages.

The different components of the adaptor are visualized in figure 7.1. in the testing context. The different elements of the adaptor are marked in green.

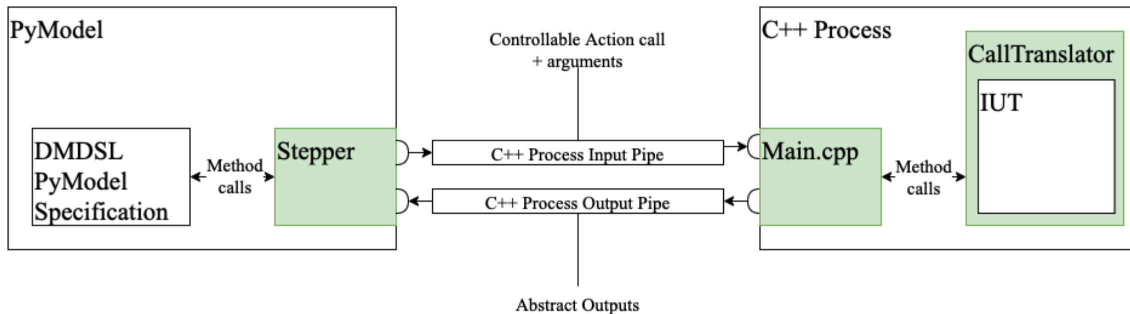


Figure 7.1: Visualization of the adaptor placed into the testing context.

7.1.1 Stepper

The MBT tool needs to communicate with the IUT in some way. The PyModel tool allows for defining a so-called stepper. In `stepper.py` one should implement how the action calls should be sent to the process that is tested, and how messages observed from the process should be translated for the MBT tool. In our case, the tested process is the wrapped version of the generated DMDSL C++ code. The PyModel framework requires the tester to implement a `test_action` method with parameters `aname`, `args` and `modelResult`. Respectively, these correspond to the action name, arguments, and output of the transition taken. We should thus ensure in `stepper.py` that `aname` and `args` are communicated with the C++ process and that the output of the process that is tested conforms with `modelResult`. In the stepper, we spawn the process in which the IUT lies and connect to the input pipe and output pipe of the process so that the stepper can send the abstract action calls and their arguments to the IUT and receive the corresponding output of the IUT via the output pipe. `stepper.py` checks if the received output is an element of `modelResult` to check if conformance is respected.

7.1.2 Main

In the `main.cpp` file, the *setup* phase of the adaptor takes place. In particular, a `serviceBundle` for the domain interface of the IUT is instantiated. The `serviceBundle` serves as a gateway to the generated DMDSL code of the model. We use the pointer to the `serviceBundle` in the creation of the CallTranslator. This class will be explained in more detail in the next subsection. The `main.cpp` also serves as the communication interface of the C++ code. In the `main.cpp` file, we will handle the input on the stdin pipe. The first input is the action (Create, Read, Delete, Add, Update). Corresponding to the action that is provided as input via the stdin pipe, a corresponding subroutine will be called, which handles the parameters provided via stdin. For example, for the Create action these are as follows:

- entityType: The type of entity of which an instance should be created.
- associations: The association list (in json format) to specify the targets in the creation process.
- instanceName: The abstract instanceName for the soon-to-be-created instance.
- instanceId: The abstract instance id that is used for the soon-to-be-created instance.

Now the corresponding action in the CallTranslator will be called with the input arguments passed via strings. The CallTranslator will return a string as output as a result of the call. `main.cpp` will provide this on the stdout output pipe.

7.1.3 CallTranslator

The generated IUT of a DMDSL model is a set of C++ files. Typically, the programmer would create .cpp files and write code to perform CRUD+A actions as desired in his project. To apply MBT, we desire that these actions are dynamically callable so that *on-the-fly* testing is possible. The adaptor should thus be able to translate abstract action calls that PyModel is generating, to actual method calls at execution time. Therefore, we will develop in C++ a CallTranslator that serves as a *wrapper* around the generated C++ code. The tasks of the CallTranslator are the *concretization* and *abstraction* tasks of an adaptor. The CallTranslator functions as a bridge between the abstract calls and the actual calls on the generated C++ code. In other words, using the abstract arguments and abstract action names, it will make concrete calls to the generated DMDSL code via appropriate `serviceBundle` calls (*concretization task*). Furthermore, it will translate the concrete output messages (e.g., error exceptions) to the abstracted output messages (*abstraction task*). Since there is quite a deviation between the respective calls and output messages, this is not trivial.

7.2 Concretization of Actions

In this section we will elaborate how the CallTranslator can concretize the abstract actions. In figure 7.2, the relevant concrete interfaces of the generated DMDSL code are visualized.

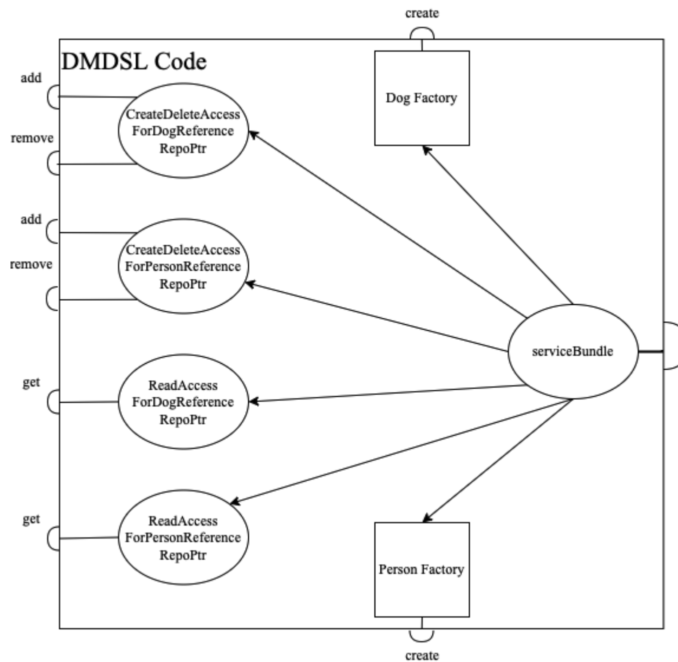


Figure 7.2: Visualization of the objects the `serviceBundle` points, and corresponding relevant interfaces for concretization of the model in figure 3.1.

7.2.1 Create

The abstract PyModel call is `Create(entity_links, instance_name, instance_id)`, where `entity_links`= ⟨`entity`, `links`⟩ is a tuple containing the entity and links for the newly created instance. `instance_name` will be the abstract name and `instance_id` will be the abstract id.

In the generated code, the `serviceBundle` provides access to a `CreateDeleteAccessReferenceRepo` for each entity. This repository then provides a factory for the corresponding entity.

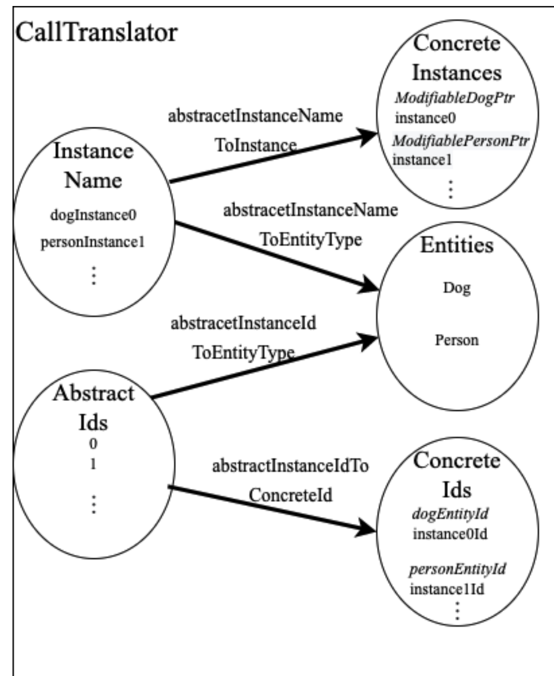


Figure 7.3: Mappings preserved in the CallTranslator.

The factory has a create method where for each association the corresponding target instances need to be provided as arguments. If in the model it is specified that for an association, precisely one target instance needs to be provided, this will be a C++ instance of the corresponding target entity. If it requires $\{0, ..., 1\}$ target instances for an association, this will be a boost::optional of a C++ instance of the target entity. In all other cases, a vector of instances of the target entity needs to be provided. The CallTranslator will call the factory method for `entity`. Furthermore, it splits the provided `links` based on the association name and puts them into the appropriate format (a concrete instance, boost::optional, or vector). Then, a call to the `createModifiable` function of the appropriate factory is made in which CallTranslator passes the target instances as arguments in the order desired by the interface. Finally, it will turn out that the `CallTranslator` needs to maintain some mappings to translate the remaining CRUD+A actions. The `abstractInstanceNameToInstance` maps the `instance_name` to the created instance, and the `abstractInstanceIdToConcreteId` maps the abstract `instance_id` to the actual id of the newly created instance. It is less obvious (but will soon be clear) that we also need to preserve the entity types corresponding to the abstract instances and abstract ids. The `abstractInstanceNameToEntity` maps `instance_name` to `entity` and `abstractInstanceIdToEntity` maps `instance_id` to `entity`. These mappings are needed to properly translate the calls of the remaining CRUD+A actions.

7.2.2 Read

The abstract PyModel call is `Read(identifier)`. The Read action is performed by calling the ReadAccessReferenceRepoPtr of the appropriate entity type via the `serviceBundle`. The `CallTranslator` uses the mapping `abstractInstanceIdToEntity` to get the corresponding entity type. It can use this information to get the ReadAccessReferenceRepoPtr of that entity. This repository pointer has a `get` method corresponding to our defined Read action. As an argument, it expects the identifier of the instance one likes to read from the repository. The `CallTranslator` uses `abstractInstanceIdToConcreteId` to transform an `identifier` into the actual concrete identifier, which then can be passed to the `get` operation to perform the Read action on the

generated C++ code.

7.2.3 Update

The abstract PyModel call is `update(instance_links)`, where `instance_links`= ⟨`instance`, `links`⟩ is a tuple containing the instance that will get its outgoing links replaced by `links`. Via the `abstractInstanceNameToInstance`, the adapter can find the corresponding instance. A mutable instance has for each association a dedicated method to replace the targets. The adaptor will group the links based on the association name and call for each group the corresponding replacement method of the instance, in order to replace the links.

7.2.4 Delete

The abstract PyModel call is `delete(identifier)`. The `serviceBundle` also has a `CreateDeleteAccessReferenceRepoPtr` for each entity. Using the `abstractInstanceIdToEntity` the `CallTranslator` can get the entity of the instance belonging to the `identifier`. It can then use this information to get the repository belonging to the entity. This repository then provides a `remove` method corresponding to our defined `delete` action. As an argument, it expects the identifier of the instance one likes to delete from the repository. The `CallTranslator` uses `abstractInstanceIdToConcreteId` to transform `identifier` to the actual concrete identifier, which then can be passed to the `remove` operation to perform the Delete action on the generated C++ code.

7.2.5 Add

The abstract PyModel call is `add(instance)`. The `serviceBundle` has a `CreateDeleteAccessReferenceRepoPtr` for each entity. Using the `abstractInstanceNameToEntity` the `CallTranslator` can get the entity of the instance belonging to the `instance`. It can then use this information to get the repository belonging to the entity. This repository then provides an `add` method corresponding to our defined `add` action. As an argument, it expects the instance one likes to add to its repository. The `CallTranslator` uses `abstractInstanceNameToInstance` to transform `instance` to the actual concrete instance, which then can be passed to the `add` operation to perform the Add action on the generated C++ code.

7.3 Abstraction of Output

Typically, before the development of the code generator, the semantics of the input models and observable input and output messages are clearly defined. For the DMDSL language, this has not been formally worked out. Therefore, applying MBT after the fact raises some problems. Normally, one should consider the IUT as a black box, and the code of the IUT should not be inspected. Since it is unknown what the exact error messages are, we should try to relate these concrete error/output messages to the abstract error messages specified in the dynamic semantics. We have created this mapping by communicating with the developers, inspecting the code, and manually executing traces on the IUT.

7.3.1 Create

Entity_Unconstructable When a create call is made using an entity type that has been specified in the domain interface as unconstructable, the dynamic semantics expects to have an `Entity_Unconstructable` output message. However, in the implementation, the Factory Pattern is used for the creation of instances of each entity type. For an entity that is unconstructable, there will be no create method in the corresponding factory. In this way, *it is statically enforced* that no instance can be created of an entity that is unconstructable. Theoretically, it could be the case that the code generator is not functioning correctly, and for some entities that are constructable, no create function is made, and for some that entities are unconstructable, a create function does

exist in the corresponding factory. A way to overcome this would be for the adaptor to perform static analysis and inspect if the corresponding methods are there. There will, however, be no power of MBT itself involved here, since it is not state-dependent. It would make more sense to write unit tests that perform such static analysis. From an economical perspective, a manual inspection would probably be sufficient. We will leave this check out of scope.

This gives rise to the question of how to overcome such difference in interface. An option would be to restrict our PyModel model and add a guard to the Create action so that we can not take the Create action for entities that are unconstructable. It is, however, probably undesirable to adapt the high-level specification to the implementation. A future new implementation could not have this static enforcement but output a corresponding error message, which would also be valid behavior according to the dynamic semantics (and strictly speaking, would match it even closer). We will not choose this route. The task of the adaptor is to form a bridge between the MBT tool. We can bridge this difference as follows. In the generation of the adaptor, the create function for entity types that are constructable will call the corresponding function of the IUT. For entity types that are unconstructable, the adaptor itself will return the `Entity_Unconstructable` output message. In this way, it is statically checked that entities that are constructable have a corresponding create function (otherwise the adaptor would fail to compile since it refers to methods that do not exist). It is not excluded that there is no create function for entities that are not constructable. As mentioned, this is not where the power of MBT lies, and this is not really the type of bug we desire to discover here.

Entity_MultiplicityMaximum It turns out that *there is no error message* in the IUT for exceeding the `Entity_MultiplicityMaximum`. If the code is built for interprocess communication, the error `ERxEXC::Exception`("The number of allowed entity instances is exhausted...") is thrown when too many instances are added to the repository. The formal semantics, however, require this error already to be thrown when too many instances are created. When creating a generator for interprocess communication, we could catch this exception and handle it by outputting `Entity_MultiplicityMaximum`. We will develop an adaptor for intraprocess-communication only. The fact that this error message is missing already gives the suggestion of a flaw in the code generator. Is it indeed the case that the maximum can be exceeded?

> Note: The fact that an error message can not be matched does not by definition need to be problematic. Although perhaps not typical, it could be the case that models that could induce an error that is not matched will be rejected by the model constraints. Hence, such a model is not supposed to be used for code generation. Furthermore, as we observed in section 5.4.2.3, there may be multiple error messages possible as output. It is up to the implementation to decide which exact error message it will output. Theoretically, it could be possible that the error message will only be thrown in conjunction with some other error message, of which the other error message does have a translation in the implementation and is actually enforced. Then the implementation will still conform to the specification. We expect in our context that an error message that can not be matched will be problematic. It is expected that MBT will point this problem out to us.

Entity_MultiplicityMinimum When the minimum multiplicity of entities is underrun, an `std::out_of_range("...Repository has to contain at least ...")` is thrown. Note that when the minimum multiplicity of an entity is non-zero, in the initial state there should already be entities in the repository. Therefore, constructor delegates are created when the minimum multiplicity is non-zero, which will be invoked when the `serviceBundle` is called. The developer needs to provide corresponding create calls so that the minimum multiplicity is respected. When the constructor delegates are not filled, the out-of-range exception should already occur. Therefore, we try to catch such an exception at the creation of the `serviceBundle` in `main.cpp`.

Entity_UnexpectedAssociation During creation, the target instances to which the newly created instance will have a link should be provided. Similar to entity constructability, *it will be statically enforced* that only instances of an entity to which an association exists, can be used as target instances in the create method. We will resolve this by making the adaptor output `Entity_UnexpectedAssociation` when an instance as a target is provided to which the entity of the 'to be created instance' has no association.

Association_TargetMaximum For a target maximum of 1, the target maximum *will be statically enforced* (since during the creation of the entity, a target instance needs to be provided). For target maximum of {0..1} a boost::optional needs to be provided, and thus again the maximum *will be statically enforced.* We will reflect the static enforcement by letting the Adapter output `Association_TargetMaximum` in such cases. However, for $\{x..y\}$ where $y > 1 \land y \geq x$ a vector of target instances is expected. One would expect the vector to be checked for the appropriate size. However, there does *not appear to be an error check on the size of the vector.* This should induce problems in the testing.

Association_TargetMinimum Analogously to `Association_TargetMaximum`, the association target multiplicity of 1 and {0..1} are *statically enforced.* We will reflect the static enforcement by letting the Adapter output `Association_TargetMinimum` in such cases. *No output message* can be identified for the case that $\{x..y\}$ where $y > 1 \land y \geq x$. The comments in the generated code confirm that these constraints are not actually enforced, showing that the developers of the code generator were aware of this.

Assocation_SourceMaximum In the models the associations also have a source multiplicity that indicates, for a target instance, the maximum number of source instances that can have a link to it. The creation process, however, *does not appear to have an error output* for this.

There are quite a lot of possible error output messages for the Create action. It turned out that only the `Entity_MultiplicityMinimum` could be properly matched, which seems rather disappointing in applying Model-Based Testing.

7.3.2 Read

Instance_notInRepository We observe that providing an id to the repository of an instance that is not in the repository triggers the `std::out_of_range(..., repository does not contain it.)` exception. The adaptor catches such exceptions and returns `Instance_notInRepository` accordingly.

7.3.3 Update

Entity_UpdateImmutabletype In the interface of the generated code, there will only be methods to change the target instances of an instance when the corresponding entity type is mutable. Hence, this is statically enforced in the generated code. We will handle this in the adaptor by outputting `Entity_UpdateImmutableType` when trying to update an entity that is not mutable.

Link_TargetNotInRepository There does not seem to be any check in the code for this.

Association_TargetMaximum The error checking is analogous to the error checking for the `Association_TargetMaximum` in the creation of an entity.

Association_TargetMinimum The error checking is analogous to the error checking for the `Association_TargetMinimum` in the creation of an entity.

Association_SourceMaximum The error checking is analogous to the error checking for the `Association_SourceMaximum` in the creation of an entity.

7.3.4 Delete

Instance_NotInRepository When providing an identifier of an instance that is not actually in the repository, no exception is thrown. However, from the comments of the generated code, it turns out that the delete function returns a `size_t` object of the number of items deleted. They note that the instances deleted via cascade deletion are not taken into account. I.e., it can only be 0 or 1, depending on whether the instance was deleted or not. Hence, when 0 is returned, we expect the instance to not be in the repository and the adaptor would output `Instance_NotInRepository`.

Entity_DeleteOnUndestructable The repository interface of an entity that is not destructable will not have a remove function. Hence, this error is statically enforced. We will resolve this by adding in the generator the `Entity_DeleteOnUndestructable` output for calls to entities that are not destructable.

Entity_MultiplicityMinimum We already considered this error message for the creation of instances in the initial state. However, the minimum multiplicity could also be underrun by an arbitrary state where deletion of an instance would result in violating `Entity_MultiplicityMinimum`. The exception that is thrown is again the `std::out_of_range("...Repository has to contain at least ...")` exception. We can simply try to catch it in the adaptor and output the `Entity_MultiplicityMinimum` output message.

7.3.5 Add

Instance_AlreadyInRepository When trying to add an instance to the repository, the exception `std::invalid_argument("cannot add ..., repository contains it already.")` is thrown. We can simply try to catch it in the adaptor and output the `Instance_AlreadyInRepository` output message.

Link_TargetNotInRepository When trying to add an instance to its repository, for which links to target instances have been established, it is necessary for these target instances to be in the repository. It turns out that when trying to add such an instance, no error message will be thrown.

7.4 Generating an Adaptor

The interfaces the adaptor needs to access are dependent on the DMDSL model. For example, where in the abstract model we expect a `Create(Entity, Links)` interface, in the actual implementation the factory of `Entity` needs to be accessed. This factory provides a `Create` method, where the target instances need to be provided as separate arguments grouped on association name. Another example is that entities that are unconstructable do not have a Create method. Here the adaptor needs to return `Entity_Unconstructable` since there is no method for it to call. It is clear that we can not make an adaptor independent from the input model. Hence, we should also generate the Adaptor, so that the adaptor can call the appropriate methods. We again use Acceleo to perform the transformation automatically. Now when the PyModel Model is generated, it will also include the adaptor. In figure 7.4, it is visualized what files are generated. The files generated for PyModel are made transparent to emphasize the files generated for the adaptor. We again use orange to indicate that the file does not rely on the input, with yellow we indicate that only a small part of the file relies on the input model, and with green we indicate that the file largely depends on the model.

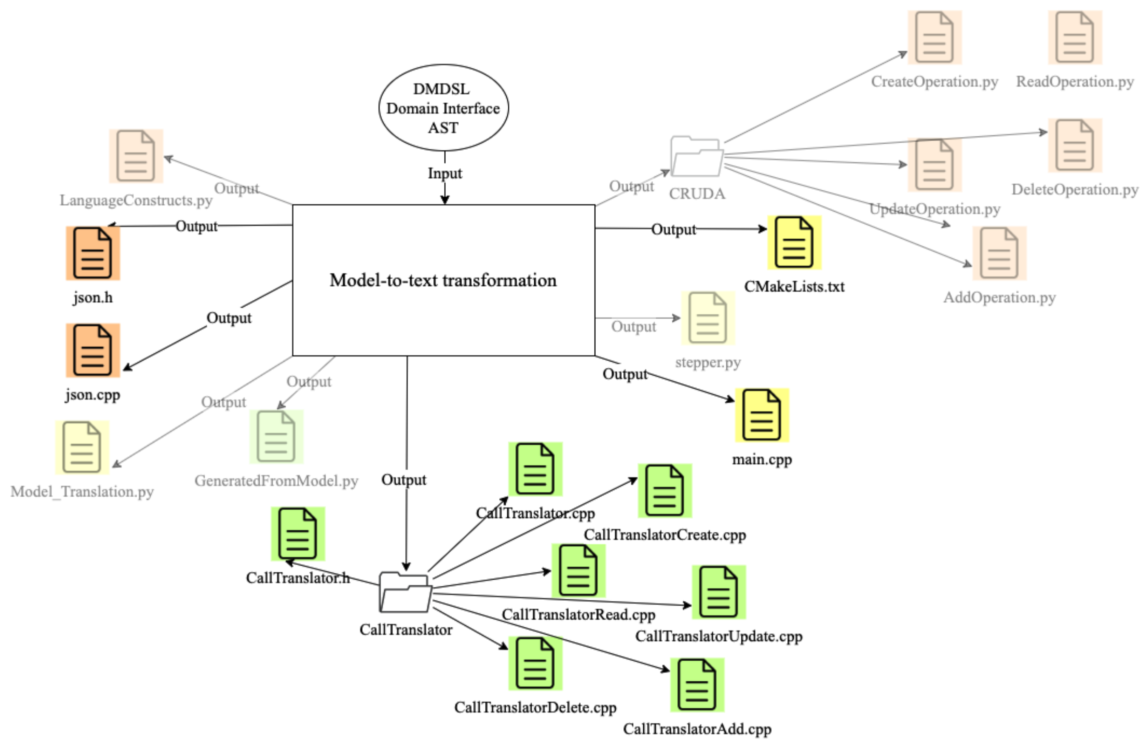


Figure 7.4: Generation of files visualized.

Chapter 8

Application of MBT

In this chapter, we will investigate how to use PyModel in practice to test the code generation of DMDSL models. We will start by considering how to finitize the domains of the actions in the DMDSL model. Then, we will discuss several strategies that one can use to test DMDSL Models. In one of these strategies, we will introduce a notion of state equivalence, that allows us to divide the state space up into a finite number of equivalence classes. We will also do some analysis on the different strategies. After that, we will actually apply Model-Based Testing on a given model and see if we manage to find some problems with the code. Finally, we will run the MBT tool on a set of test models using a strategy that simulates behavior when most of the preconditions of actions are satisfied. This allows us to draw some conclusions with regard to the correctness of the code generator when the programmers respect the preconditions of all actions.

8.1 Domains of Actions

Each of the CRUD+A actions relies on input arguments out of a certain domain. For some of the input arguments, it is clear that a finite domain exists. For some other arguments, this is less clear, and a possibly infinite domain could exist. Since PyModel iterates over all enabled actions in conjunction with the possible arguments to determine which transition to take, it is not possible to work with infinite domains, and practically infeasible to work with domains that are too large. We need a form of *data abstraction* to deal with this. A typical solution is to *finitize* these domains. The key challenge here is to finitize them in such a way that the corresponding arguments are still most likely to expose failure. Since we have knowledge about the system, we have some ideas about what inputs are likely to produce the same behavior. The underlying assumption is called the *uniformity assumption* [20]. We will consider each of the CRUD+A actions and their corresponding arguments, and apply our domain knowledge to come up with arguments that are most likely to expose failure.

PyModel requires you to specify for each of the actions, and corresponding arguments what the finite domain is. It is, however, allowed to make these finite domains dependent on the state. Especially in our context, this is a powerful feature of PyModel. It depends on the state what instances exist and thus what instances could be added to the repository. There is no way to specify such a domain initially since in the initial state there won't be any instances at all. Domains that are state-dependent are defined using a lambda term. The lambda term will be re-evaluated from the current state, to determine the finite domains of each action for the current state. In this way, it is determined what the outgoing transitions are.

8.1.1 Create

In the formal semantics of the Create action, the arguments are the entity (`entity`) of the instance that will be created and the links (`new_links`) in which the new instance will participate as the

source. For the creation of an `entity` instance, for each outgoing association, some subset of the elements in $s.I$ (instances of state s) *will be involved* as target instances. Suppose that there are 20 elements in $s.I$. Then, the number of subsets will be 2^{20}. Note, however, that a single instance can be used multiple times as a target (even for a single association relation). Therefore, the number of options for `new_links` is actually of *infinite size.* While one could decide to limit the behavior of the model so that we only consider subsets of I, calculating a domain of size 2^{20} will not be feasible in practice, even though it is finite. We can observe that a significant part of the options in the infinite domain will lead to an error due to the multiplicities of the source and target of an association relation. E.g., when an association relation between some EntityA and an EntityB has a target multiplicity of 4, then trying to create an instance of EntityA with 100 links to an Entity B instance (could even be 100 times the same instance), would clearly lead to an error due to exceeding the maximum number of target instances.

We should consider a finitized domain that invokes the most interesting behavior of this action but is also computationally feasible to calculate in each state. The idea of *boundary value* testing is to choose test input at the boundaries of the domain [20]. In this way, we hope to satisfy the *uniformity assumption.* In particular, we consider the outgoing association relations of the entity. Then for each outgoing association, we consider the following groups of target instances:

- Group 1: No target instances.
  - Interesting to observe if the implementation handles no target instances as expected. Can it deal properly with a data type for passing no target instances? (e.g. boost::optional::none, empty collection, some other way this is implemented?)
- Group 2: One target instance.
  - Interesting to observe if the implementation handles a single target instance as expected. Can it deal properly with the data type used for passing a single instance? (e.g. a concrete instance is passed, a collection containing a single instance, some other way this is implemented?)
- Group 3: One element below the minimum number of target instances.
  - Interesting to observe if the implementation handles the check on the minimum number of target instances properly, and fails as expected.
- Group 4: The minimum number of target instances.
  - Interesting to observe if the implementation handles the check on the minimum number of target instances properly, and succeeds as expected.
- Group 5: The maximum number of target instances.
  - Interesting to observe if the implementation handles the check on the maximum number of target instances properly, and succeeds as expected.
- Group 6: Exceeding the maximum number of target instances by one.
  - Interesting to observe if the implementation handles the check on the maximum number of target instances properly, and fails as expected.
- Group 7: The midrange ($\frac{\text{minimum}+\text{maximum}}{2}$) of target instances.
  - Interesting to observe if the implementation handles the check on a non-boundary case of target instances properly, and succeeds as expected.

At first, one might think Group 3 and Group 2 are not really needed since if it is possible to go below the minimum number of target instances, this would happen as well in Group 1 (or possibly also for Group 2). We consider these to be special cases since it is typical that implementations use a different data type for passing nothing or a single instance.

There are many options possible for these groups. Suppose we have an association relation from EntityA to EntityB with a target multiplicity {3..5}, and there are five instances of EntityB in existence. Then we have 1 option for group 1, 5 options for group 2, 5^2 options for group 4, 5^5 options for group 5, 5^6 options for group 6, and 5^4 options for group 4. This would already lead to a 'finite' domain of $1 + 5^1 + 5^2 + 5^5 + 5^6 + 5^4$ options just for this single association relation. This is of course still rather large. We will not create all the options for each of the groups, but instead, we will simply do a single 'sample with replacement' to fill the groups. That is, for group 2, an arbitrary EntityB instance will be picked. For group 3, an arbitrary 'minimum − 1' target instances will be picked, etc. Hence, for the creation of each type of entity, there will be 7 options to use as target instances for each of the corresponding outgoing associations. Hence, in the worst case, there would be $\sum_{e \in Entities} \prod_{assoc \in e.outgoingAssociations} 7$ Create transitions possible from a given state. This can still be a large number. Typically, the number of outgoing associations is fortunately limited. It should be noted that due to the 'sample with replacement', it typically will happen when the same state is visited again, different outgoing transitions will be available for the Create action due to randomness. If one finds this undesirable, the 'Pick with replacement' can be done using a random seed that is, for example, based on a hash value unique for each state.

In practice, it turns out that having a factor of 7 in the product still creates too many options in the finite domain, making domain calculation rather slow. Suppose one is considering a model of 3 entities, each with 3 outgoing associations. Then, one would expect in worst case $(7 \cdot 7 \cdot 7) + (7 \cdot 7 \cdot 7) + (7 \cdot 7 \cdot 7) = 1029$ possible Create transitions per association for a state. Therefore, in practice, we will limit ourselves to $Group1, Group6$, and $Group7$. In this way, we will still test going below the minimum number of target instances (if this is possible), exceeding the number of target instances, and providing an appropriate number of target instances. In worst case one would now expect $(3 \cdot 3 \cdot 3) + (3 \cdot 3 \cdot 3) + (3 \cdot 3 \cdot 3) = 81$ Create transitions per association, which is significantly less.

> Note: The groups are formed based on the type of Entity of which an instance is going to be created. Hence, the `new_links` set of target instances is dependent on which `entity` is going to be created. The domain of `entity` and `new_links` should thus *not be finitized in isolation*. Otherwise, PyModel could attempt to create an instance of some EntityB using a `new_link` set intended for the creation of the instance of some other EntityA. To fix this, we combine the two parameters into a single parameter `entity_links`. This will be a tuple ⟨`entity`, `new_links`⟩ where the first element is the entity of the newly created instance, and the second element is its outgoing link set. By merging `entity` and `new_links`, we can ensure the tool will pick only links that are 'intended' for the creation of an instance with entity type `entity`.

Example 8.1.1. Consider the model in figure 8.1. Suppose that $s.Instances = \{Entity2Instance0, Entity1Instance1\}, s.Repo = \emptyset, s.Links = \{(Entity1Instance1, \text{'entity2'}, Entity2Instance0)\}$. Now during the calculation of the enabled transitions, one of the options will be the creation of an instance of Entity3 called $Entity3instance2$. Now for each outgoing association relation of Entity3, the groups will be determined:

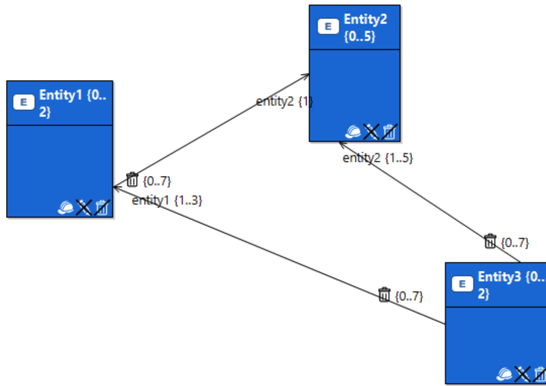


Figure 8.1: Example Model.

- 'entity2' association:

$$Group1 = \emptyset$$

$$\begin{aligned} Group6 = \{&(Entity3instance2, \text{'entity2'}, Entity2Instance0),\\ &(Entity3instance2, \text{'entity2'}, Entity2Instance0),\\ &(Entity3instance2, \text{'entity2'}, Entity2Instance0),\\ &(Entity3instance2, \text{'entity2'}, Entity2Instance0),\\ &(Entity3instance2, ,\text{'entity2'}, Entity2Instance0),\\ &(Entity3instance2, \text{'entity2'}, Entity2Instance0)\}\end{aligned}$$

$$\begin{aligned} Group7 = \{&(Entity3instance2, \text{'entity2'}, Entity2Instance0),\\ &(Entity3instance2, \text{'entity2'}, Entity2Instance0),\\ &(Entity3instance2, \text{'entity2'}, Entity2Instance0, \}\end{aligned}$$

- 'entity1' association:

$$Group1 = \emptyset$$

$$\begin{aligned} Group6 = \{&(Entity3instance2, \text{'entity1'}, Entity1Instance1),\\ &(Entity3instance2, \text{'entity1'}, Entity1Instance1),\\ &(Entity3instance2, \text{'entity1'}, Entity1Instance1),\\ &(Entity3instance2, \text{'entity1'}, Entity1Instance1), \}\end{aligned}$$

$$\begin{aligned} Group7 = \{&(Entity3instance2, \text{'entity1'}, Entity1Instance1),\\ &(Entity3instance2, \text{'entity1'}, Entity1Instance1)\}\end{aligned}$$

Now, we consider the possible arguments for `new_links` (given that `entity`= $Entity3$). The options are ('entity2': $Group1$, 'entity1':$Group1$), ('entity2': $Group1$, 'entity1':$Group2$), ('entity2': $Group1$, 'entity1':$Group3$), ('entity2': $Group2$, 'entity1':$Group1$), etc. The `new_links` argument of for example ('entity2': $Group1$, 'entity1':$Group3$) would then be the concatenation of $Group1$ and $Group3$. To be more precise, given `entity` $=$ $Entity3$ the options for `new_links` are the concatenation of the elements in the tuples from the cartesian product between the set of groups

of association relation ‘entity2’ and the set of groups of association relation ‘entity1’. Using $Group1 + Group3$ as `new_links` argument for the creation of $Entity2$ is not sensible (it does not even have any outgoing association relations). We have a dependency of the `new_links` parameter on the `entity` parameter. Therefore, these are determined in conjunction via a tuple as explained in the note.

Also, an identifier (`instance_id`) and the name of the instance (`instance_name`) are passed as an argument. We distinguish between an identifier and a name since some of the CRUD+A actions are performed using the identifier, such as the Read action for which we will use `instance_id`, whereas the Add action requires the actual instance as argument in which we will use `instance_name`. The identifier of an instance is determined on the basis of the `highest_id` state variable to ensure uniqueness of the id. The `instance_name` is simply the concatenation of the word ‘instance’ and the identifier.

To summarize:

- `entity_links`: We use a lambda expression that returns a set of tuples $\langle$`entity`, `new_links`$\rangle$, where `entity` comes from the fixed domain of entities, and based on the `entity`, the `new_links` list is constructed using the grouping method.
- `instance_id`: We use a lambda expression that takes the highest id using the `highest_id` state variable.
- `instance_name`: We use a lambda expression that concatenates the word ‘instance’ and `highest_id`.

8.1.2 Read

The Read action takes as an argument an identifier called `identifier`, and tries to read a corresponding instance from the repository. The identifier is modeled as an integer and thus has an infinite domain.

> Note: Modeling the identifiers as a finite set is not desirable. One might think that when all entities are bounded, one could create a finite set of identifiers containing $\sum_{e \in Entities} e.max$ unique identifiers. Due to the creation and deletion of instances, that would mean that an identifier i that was used previously for some instance that got deleted, would need to be reused for the creation of an instance at a later point of time if sufficient Create and Delete actions are done. This is undesirable if one wants to maintain a one-to-one correspondence between abstract identifiers and concrete identifiers in the implementation. Otherwise, housekeeping in the Adaptor with regard to reuse should be done. Furthermore, it is probably desirable that doing a Read operation using an identifier of an instance that was deleted results in a corresponding `Instance_NotInRepository` error, instead of succeeding because it is now bound to some different instance.

Option 1: We use a lambda expression that returns `all_identifiers`. This list contains all the identifiers that of instances that were ever created. Hence, we would test the operation using instances that exist that can be in or out of the repository, and using the identifiers of instances that no longer exist

Option 2: We take the identifiers of all instances that currently exist (i.e., instances both in and not in the repository). Furthermore, #`instances` arbitrary identifiers from the `all_identifiers` list are picked (this set also contains identifiers of instances that were deleted). We use a lambda expression that concatenates both lists.

Option 2 is introduced since in Option 1, for longer test sequences, a significant part of the identifiers in `all_identifiers` will correspond to instances that no longer exist. Furthermore, the `all_identifiers` domain would grow unboundedly, making the exploration increasingly slow. Initially, Option 1 will be used, but during analysis, the refinement to Option 2 will be proposed.

8.1.3 Update

The update Action takes as argument a tuple $\langle$`instance, links`$\rangle$. The action tries to replace the outgoing links of the `instance` with the provided `links`. Analogous to the Create action we use a lambda expression that returns a set of tuples $\langle$`instance, links`$\rangle$, where `instance` comes from the domain of state variable `instances` and based on the corresponding entity `new_links` is constructed using the grouping method.

8.1.4 Delete

The Delete Action takes as an argument an identifier and tries to Delete the corresponding instance from the repository. We will use the same finite domain as used in the Read action.

8.1.5 Add

The Add action takes as an argument an instance and tries to Add the instance to the repository. We use a lambda expression that returns the state variable `instances`. This will contain both instances that are in the repository and instances that are not in the repository so that both success and error output messages are tested.

8.2 Testing Strategies for Model Exploration

The MBT tool creates traces by exploring the PyModel Model. Typically in many states, multiple actions are possible. Furthermore, the set of traces that are possible is almost always of infinite size. It follows that exploration of all possible traces will not be feasible. An MBT tool typically has some predefined strategies to find 'interesting' traces. It could be of interest to make the testcase generation focus on putting the PyModel Model in different states, aiming for state coverage. It could also be of interest to discover all possible actions and aim for transition coverage. In this section, we will investigate how to create strategies so that the MBT tool explores DMDSL models in a clever way, resulting in hopefully interesting testcases being generated. In particular, an interesting strategy is explored in which an equivalence relation on the set of states is introduced. This allows us to divide an infinite state space up into a finite number of equivalence classes. In addition, we will point out some of the deficiencies of the strategies.

Definition 24 (Enabled transition)**.** We will refer to an action with corresponding arguments, (for which corresponding guard evaluates to true in the current state), as an *enabled transition.*

Pymodel Strategy PyModel allows users to define their own testing strategy. A Python file describing the testing strategy contains a method `test_action(enabled)` where `enabled` is a list of tuples $(aname, args, result, next, properties)$ representing the enabled transitions from the current state, which the PyModel tool will provide. *As a tester, one should fill in this method body, and use the information of the `enabled` to determine which transition should be taken.* The information provided of each enabled transition is:

- The enabled transition has *aname* as name of an action.
- The enabled transition uses *args* as input arguments.
- Execution of the transition produces the output *result*.
- Execution of the transition would result in a state with the state variables set as described by *next*.
- (*properties* are not of relevance in our context.)

In a strategy it is only possible to influence for a given state which of the enabled transitions it takes, based on the tuples in the *enabled* list. It is not possible to check for the resulting state *next* what its outgoing transitions are. This limits the creation of more sophisticated strategies that look further then one transition ahead.

We will now have a look at some testing strategies. The Random Strategy and Default State Coverage are provided by the tool. The remainder of the described strategies have been implemented within the scope of this thesis.

8.2.1 Random Strategy

By default, the model will be explored using the random strategy. That is, a randomly enabled transition will be taken. We will see in section 8.4.1 that the strategy is quite good in finding issues where model constraints are not enforced and semantics are not properly defined. It turns out that in practice it is not so good in exploring the model very well. Especially in a model with many entities, and lots of association relations it can be hard to properly explore the model. We refer to this strategy as Random Strategy.

8.2.1.1 Chaining Problem

In figure 8.2 we have a chain of entities. It is hard to properly instantiate Entity1. There need to be at least two target instances of Entity2 for the successful creation of Entity1, and these need to be provided as target in the creation call. For the creation of Entity2, there needs to exist at least one Entity3 instance, and it must be provided as a target. Finally, for the creation of an instance of Entity3 at least one Entity4 instance needs to exist and it needs to be provided as target. The creation process should thus traverse this chain from back to front to properly instantiate Entity1.

It is clear that especially for smaller test sequences the probability that Entity1 would get successfully instantiated is small using the Random Strategy. We will run PyModel on the model of figure 8.2 with a test sequence of size 100, three times and report the average data obtained. In figure 8.3 we observe the spread of output messages. We observe that the `Success` message do not contain a large part of the Create, and Add actions. In the Update and Delete actions, no `Success` output occurs, which is not surprising since all entities are undestructable and immutable. It is probably desirable that the Create and Add actions contain more `Success` messages since these actions will manipulate the contents of the state variables. Furthermore, a close look at these actions shows that on average $4\frac{1}{3}$ Entity4 instances, 3 Entity3 instances and $1\frac{1}{3}$ Entity2 instances are created. In none of the runs it was able to create an Entity1 instance. In the test sequence, it added on average $4\frac{1}{3}$ Entity4 instances, $1\frac{2}{3}$ Entity3 instances, $\frac{1}{3}$ Entity2 instances to the repository. Also, in none of the runs it managed to add an Entity1 instance to the repository.

What becomes clear from inspecting the trace is that adding instances to the repository suffers from the same issue. All link targets of an instance need to be in the repository in order to successfully add it (otherwise `Link_TargetNotInRepository` occurs). Not very surprisingly, for the Add operation, it is thus even more difficult to be executed successfully for instances in which the corresponding Entity is at the beginning of the chain. For an instance of Entity1 to be added, first the creation process should have properly traversed the chain. Then, in the adding process, the chain needs to be properly traversed again. If one would increase the size of the test sequence, it is expected that eventually creation and adding of an Entity1 instance will happen. E.g., if we increase it to 300, we observe that often it will manage to create an instance of Entity1. To also add it, a larger test sequence is needed. Note that in the model, the instances are undestructable. If they would in fact be destructable, it would be even harder to instantiate Entity1, since during the random exploration instances in the front of the chain can be deleted. We will refer to this problem for creating and adding instance as the *chaining problem.*

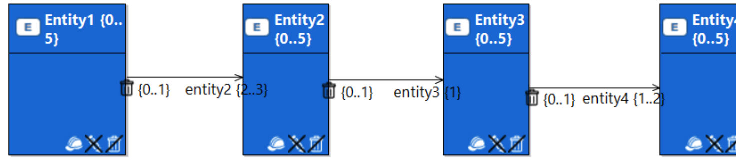


Figure 8.2: Example model with chain of entities.

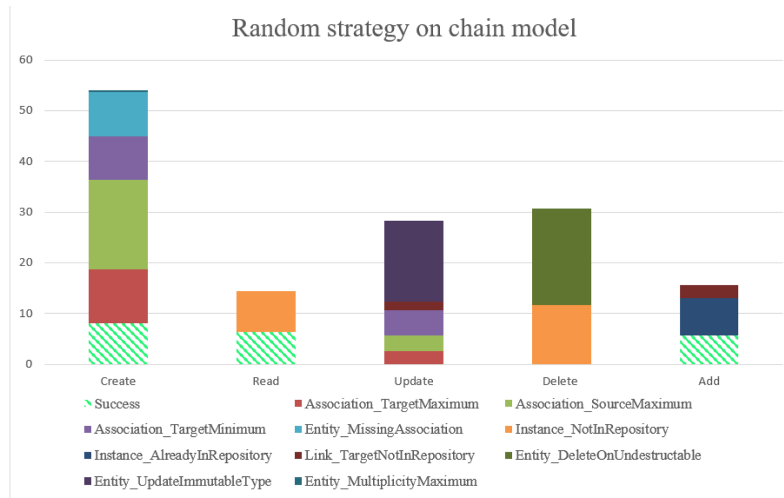


Figure 8.3: Results of random strategy on chain model.

8.2.2 Exhaustive Create Add Strategy

A way to overcome the *chaining problem* is to have a creation phase before the random exploration. In the creation phase, the strategy will only pick enabled Create transitions that will have `Success` as output. Due to the boundaries of the entities, it is guaranteed there will only be a finite number of consecutive create transitions possible (the creation phase will be smaller than $\sum_{e \in Entities} e.max$ steps since the multiplicities are bounded). In this way, automatically we will have that instances of Entity4 will be instantiated before the creation of Entity3. Analogously we will have that instances of Entity3 will be instantiated before the creation of Entity2. Finally, we then have that instances of Entity2 will exist before the creation of Entity1. If this is not the case, then the entities can not successfully be instantiated. As the Add action suffers from the same issue, it is desirable to have an exhaustive Add phase after the Create phase. Finally, after the Create and Add phase, the state variables `instances`, `repo`, and `links` will be very populated. The strategy can then switch back to random exploration. It is expected that lots of interesting behavior would be exposed during the random exploration after the Create and Add phase since the state variables are very filled. We refer to this strategy as Exhaustive Create Add.

By applying the Exhaustive Create Add to the model of figure 8.2. we have that in the average of three test sequences applying this strategy, we have that 5 Entity4 instances are created and all are added to the repository. $4\frac{1}{3}$ Entity3 instances are created of which 4 instances are added to the repository. $2\frac{1}{3}$ Entity2 instances are created and all are added to the repository. 1 Entity1 instance is created and it is added to the repository. Indeed, it seems like the technique managed to solve the problem of the creation and adding instances for entities that are in the front of such a chain.

Note: In the test sequence of EXHAUSTIVE CREATE ADD more Entity3 instances were created on average, then added to the repository. In the random phase after exhaustively creating and adding instances, it might be possible that new options for the creation of entities appear. The finite domains are reconsidered, which potentially could give new options. When the entities are mutable, it could also be the case that links are changed, allowing for the creation of more entities. E.g., suppose that we have 4 Entity2 instances for the model in figure 8.2. Theoretically, two Entity1 instances can be created. However, if the first created Entity1 instance has 3 Entity2 instances as a target, due to the source maximum of the association relation it is not possible to create another Entity1 instance. If the number of target instances of Entity1 is decreased during the test sequence, new options for the creation of an Entity1 instance could emerge.

8.2.3 Default State Coverage

The default state coverage strategy is implemented by maintaining a list of states, and the number of times that state has been visited. In the selection of a transition, an arbitrary transition is picked that would lead to a new state. If no such transition exists (i.e. all transitions would lead to a state that has already been visited), it will pick the transition leading to a state that has been visited least often. This is a very appealing strategy since the strategy tries to get the model in as many different states as possible by avoiding repetition as much as possible. Since the model is built on a higher abstraction level, we expect IUT to be in a different kind of state as well, increasing the likelihood that a state can be found in which the IUT does not show the behavior that we desire. The PyModel tool provides this strategy for us. This strategy is referred to as DEFAULT STATE COVERAGE strategy.

8.2.4 State Coverage 1

Remember that in the model program a state has been defined using the state variables `instances`, `repo`, `links`, `highest_identifier`, `all_identifiers`

Intuitively, when we consider an input Model with an entity called *EntityA*, and the test sequence executes the creation of an *EntityA* instance, adds the instance to the repository, and finally deletes the newly created instance, we would end up in the initial state again. Since we make use of `highest_identifier` and `all_identifiers`, this is not the case. In the new state, the `highest_identifier` will have been increased to 1, and the `all_identifiers` set now contains identifier 0, since an instance with such identifier existed before. We can thus improve on the default state coverage strategy and consider arbitrary states s, s' equal if and only if the $s.\texttt{instances} = s'.\texttt{instances} \wedge s.\texttt{repo} = s'.\texttt{repo} \wedge s.\texttt{links} = s'.\texttt{links}$. We will use STATE COVERAGE 1 (SC1) to refer to this strategy.

8.2.5 State Coverage 2

We will now introduce a new state coverage strategy. The strategy makes use of a new notion of state equivalence. We will first define this concept. Then, we will see that Graph Isomorphism reduces to the newly defined notion of state equivalence, which shows that determining equivalent states is a complex problem. Then, we will see how this notion of state equivalence can still be used in practice. Finally, we will compare the strategy against the STATE COVERAGE 1 strategy. Furthermore, we will see how it performs on the chaining problem.

8.2.5.1 Introducing State Equivalency

One might think that it would be sufficient to leave out `highest_identifier` and `all_identifiers` out of the state variables in comparing states of state coverage. A large part of states,

however, will allow for traces that are rather similar. Note that the input models puts boundaries on the number of entities that can be created. Hence, when one would abstract from the identifiers one could create a model program (although typically much too large for exploration) with a finite state space. Hence, it makes sense to adapt the notion of state coverage so that we can abstract from the actual identifiers. It may be good to formally define a notion of equivalent states, so that we can try to prevent visiting states of which an equivalent state has been seen before.

We will provide an example:

Example 8.2.1. Suppose *state*1 is:

$$\begin{aligned} Instances &= \{DogInstance1, DogInstance2, OwnerInstance3, OwnerInstance4\} \\ Repo &= \emptyset \\ Links &= \{(DogInstance1, owner, OwnerInstance3), (DogInstance1, owner, OwnerInstance4)\} \end{aligned}$$

and suppose *state*2 is:

$$\begin{aligned} Instances &= \{DogInstance5, DogInstance4, OwnerInstance9, OwnerInstance4\} \\ Repo &= \emptyset \\ Links &= \{(DogInstance4, owner, OwnerInstance9), (DogInstance4, owner, OwnerInstance4)\} \end{aligned}$$

Note that when abstracting from identifiers, *state*1 and *state*2 would behave very similarly. In particular, actions executed on *OwnerInstance*9 in *state*2 would correspond to actions performed on *OwnerInstance*3 in *state*1 and vice versa. Analogously actions *DogInstance*4 in *state*2 would correspond to actions on *DogInstance*1 in *state*1, *OwnerInstance*4 would correspond to *OwnerInstance*4 in *state*1, and finally *DogInstance*5 in *state*2 would correspond to *DogInstance*4 in *state*1 and vice versa. Considering the induced transition system of *state*1, would be equal to the induced transition system of *state*2 when the arguments are renamed under their respective correspondence (i.e. *OwnerInstance*9, would get renamed to *OwnerInstance*3, *DogInstance*4 would get renamed to *Doginstance*1 etc..)

We observe some properties desirable for this notion of equivalency:

- Injectivity in Instance Matching: No two instances in *state*1 should correspond to the same instance in *state*2: Suppose it would be allowed that two instances in *state*1 would correspond to a single instance in *state*2, then deleting one of the instances in *state*1 (leading to state $state1'$) would correspond to deleting the instance of *state*2 (leading to $state2'$). Now, $state1'$ still allows for deleting the other instance, whereas the matched instance of *state*2 no longer exist in $state2'$, and therefore corresponding behavior would not induce the same output.

- Surjectivity in Instance Matching: Each instance in *state*2 should correspond to at least an instance in *state*1: Suppose there would be an instance in *state*2 that would not have a corresponding instance in *state*1, then *state*2 would for example allow for deleting the corresponding instance, whereas *state*1 would not have a corresponding behavior.

- Repository containment in instance matching: The allowed traces are also affected by repository containment of the instances. Suppose an instance i in *state*1 that is in the repository is matched to an instance i' in *state*2 that is not in the repository. Then performing a delete action on i in *state*1 would result in a `Success` output, whereas performing a delete action on i' in *state*2 would result in an `Instance_NotInRepository` output. Hence, in the matching, the repository containment should also correspond, so that traces under renaming are equal. Therefore, we desire that repository containment is also preserved in the instance matching.

- Link matching: Since the links have behavioral consequences due to cascade deletion, we desire that the links also have a one-to-one correspondence in this notion of equivalent states. In particular, it would be desirable that $(instance_x, \text{association_name}, instance_y) \in state1.Links \iff (instance_x', \text{association_name}, instance_y') \in state2.Links$ where $instance_x'$ would be the instance in *state*2 that is matched to $instance_x$ and $instance_y'$ would be the instance in *state*2 that is matched to $instance_y$.

Let's now formally define a notion when states are considered to be equivalent:

Definition 25 (State equivalency)**.** We say that $state1$ is equivalent to $state2$, denoted $state1 \sim state2$, if and only if

- There exists a bijection $\phi : state1.Instances \rightarrow state2.Instances$ with $x \in state1.Repo \iff \phi(x) \in state2.Repo$ and $x.Entity = \phi(x).Entity$
- We have that for all instances $instance_x, instance_y \in state1.Instances$

$$(instance_x, \text{association_name}, instance_y) \in state1.Links$$
$$\iff$$
$$(\phi(instance_x), \text{association_name}, \phi(instance_y)) \in state2.Links$$

Having this notion of equivalency could be of interest in the state coverage. We are now able to divide the state space up into equivalence classes in which the states behave equivalently up to renaming of identifiers.

> Note: It is left to future research to further formalize this notion. It would be of interest to give the Model Programs LTS semantics, and try to prove that two 'equivalent states' are bisimilar under LTS semantics for some relabelling function. This is out of scope of this work.

Under the assumption that the multiplicities of the entities and associations are bounded, we have a finite number of equivalence classes. It should however be noted that for non-trivial models, the number of equivalence classes will be rather large, and reaching state coverage modulo the defined notion of equivalency does not seem feasible in practice. Nevertheless, if we manage to encode this notion of equivalency in the state coverage strategy, then state coverage would try to avoid the states that are equivalent and produce hopefully more interesting test sequences.

8.2.5.2 Graph Isomorphism Reduces to State Equivalency

The provided definition might remind the reader of the definition of Graph Isomorphism. In this section we will show that the problem of determining whether two graphs are isomorphic can be reduced to determining whether two states are equivalent. Since the Graph Isomorphism problem is not known to be tractable (solvable in polynomial time) nor to be NP-Complete, the reduction will show that it is unlikely that we would be able to come up with an efficient procedure for determining state equivalency. For the theoretically oriented reader the following section might be of interest. From a practical point of view, the conclusion is most important.

Definition 26 (Graph Isomorphism)**.** Two directed/undirected graphs $G = (V, E)$ and $H = (W, F)$ are isomorphic if there is a bijective function $f : V \rightarrow W$ such that for all $v, w \in V$:

$$(v, w) \in E \iff (f(v), f(w)) \in F$$

We will now show that determining whether undirected graphs are isomorphic can be reduced to determining state equivalency.

Theorem 8.2.2. *Graph isomorphism on undirected graphs reduces to state equivalency.*

Proof. It has been proven that the Graph Isomorphism problem reduces to Directed Acyclic Graph (DAG) isomorphisms as follows [36]: let $G = (V, E)$ be an undirected graph. Now let $G' = (V', E')$ where

- $V' = V \cup E$. I.e. each vertex of G will be a vertex in G' and each edge of G will be a vertex in G'. To the former, we refer to as *'regularNode'*, and the latter we refer to as *'edgeNode'*.

- $E' = \{(x, y) \in V \times E \mid x \text{ is an endpoint of } y\}$. I.e. for each edge $e \in E$, there is a directed edge in E' from the endpoints of e to the corresponding 'edgeNode' of e in V'.

For an example transformation from Graph Isomorphism on an undirected graph to Graph Isomorphism on a directed graph, consider the first transformation step in example 8.2.3. We can conclude, if we would be able to solve the isomorphism problem for the DAGs, we would be able to solve the Isomorphism Problem for undirected graphs. Interesting to observe is that *the type of DAGs the undirected graphs reduce to, can be reduced to determining state equivalency*. Consider the DMDSL model in figure 8.4.

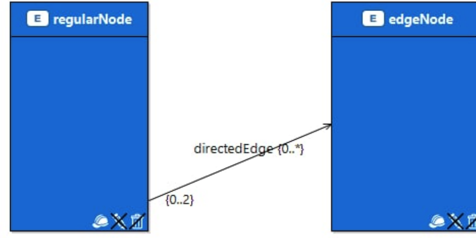


Figure 8.4: ASOME Model for Graph Isomorphism Reduction

> Note: One could wonder why we reduce the isomorphism problem on undirected graph to state equivalency, instead of reducing the isomorphism problem on DAGs state equivalency. It turns out that the type of DAGs undirected graphs reduce to, are expressible as state graphs for a general ASOME model (see figure 8.4). Arbitrary DAGs are not expressible as state graphs for a general ASOME model. That would make a proof very cumbersome.

Now consider 2 arbitrary undirected graphs $G_1 = (V_1, E_2), G_2 = (V_2, E_2)$ on which we'd like to determine whether these are isomorphic. We reduce the problem to the DAG Isomorphism problem, as described by the reduction transformation. Thus we consider the graphs $G'_1 = (V'_1, E'_1), G'_2 = (V'_2, E'_2)$ with $V'_1 = V_1 \cup E_1$ and $V'_2 = V_2 \cup E_2$ and $E'_1 = \{(x, y) \in V_1 \times E_1 \mid x \text{ is an endpoint of } y\}$ and $E'_2 = \{(x, y) \in V_2 \times E_2 \mid x \text{ is an endpoint of } y\}$. Now we will reduce the DAG isomorphism problem of G'_1 and G'_2 to state equivalency. Let $state1$ and $state2$ be defined as follows:

- $state1.Instances := V'_1$ (where $v.Entity := regularNode$ for $v \in V_1$ and $v.Entity := edgeNode$ for $v \in E_1$)
- $state_1.Repo := \emptyset$
- $state_1.Links := \{(x, directedEdge, y) \mid (x, y) \in E'_1\}$

And

- $state2.Instances := V'_2$ (where $v.Entity = regularNode$ for $v \in V_2$ and $v.Entity := edgeNode$ for $v \in E_2$)
- $state2.Repo := \emptyset$
- $state2.Links := \{(x, directedEdge, y) \mid (x, y) \in E'_2\}$

This is clearly a polynomial time reduction. We will now show that showing $state1 \sim state2$ is equivalent to showing G'_1 and G'_2 are isomorphic:

- $\Rightarrow$): Suppose that $state1 \sim state2$. Then, we can pick a bijection with $\phi : state1.Instances \rightarrow state2.Instances$ with $x \in state1.Repo \iff \phi(x) \in state2.Repo$ and $x.Entity = \phi(x).Entity$ and

$$\begin{aligned}&(x, directedEdge, y) \in state1.Links\\ &\iff\\ &(\phi(x), directedEdge, \phi(y)) \in state2.Links\end{aligned}$$

  We will now show that ϕ satisfies all requirements as a witness for DAG isomorphism of G'_1 and G'_2

  - Since $state1.Instances = V'_1$ and $state2.Instances = V'_2$ it follows that ϕ is a bijection between V'_1 and V'_2.
  - We will now show that $(x, y) \in E'_1 \iff (\phi(x), \phi(y)) \in E'_2$ is preserved for all $x, y \in V'_1$. Let $x, y \in V'_1$ be picked arbitrarily. Assume that $(x, y) \in E'_1$. Then by definition of $state1.Links$ we have that $(x, directedEdge, y) \in state1.Links$. Then by definition of $\sim$ it follows that $(\phi(x), directedEdge, \phi(y)) \in state2.Links$. By set definition of $state2.Links$ then follows that $(\phi(x), \phi(y)) \in E'_2$. Now assume that $(\phi(x), \phi(y)) \in E'_2$. By definition of $state2.links$ we then have that $(\phi(x), directedEdge, \phi(y)) \in state2.Links$. Since $x, y \in V'_1$ we have that $x, y \in state1.Instances$ and thus by definition of $\sim$ it then follows that $(x, directedEdge, y) \in state1.Links$. From set definition of $state1.Links$ then follows that $(x, y) \in E'_1$.

  Hence ϕ is a bijection that maps V'_1 to V'_2 with $(v, w) \in E'_1 \iff (\phi(v), \phi(w)) \in E'_2$ for all $v, w \in V'_1$ allowing us to conclude that G'_1 and G'_2 are isomorphic.

- $\Leftarrow$) : Suppose that G'_1 and G'_2 are isomorphic. Then we can pick a bijective function $\phi : V'_1 \rightarrow V'_2$ such that for all $v, w \in V'_1 : (v, w) \in E'_1 \iff (\phi(v), \phi(w)) \in E'_2$. We will show that ϕ is also a witness for DAG isomorphism between $state1$ and $state2$.

  - Since $state1.Instances = V'_1$ and $state2.Instances = V'_2$, it follow that ϕ is a bijection between $state1.Instances$ and $state2.Instances$.
  - Furthermore, since $state1.Repo = \emptyset = state2.Repo$ it follows that for ϕ the requirement $x \in state1.Repo \iff \phi(x) \in state2.Repo$ is trivially satisfied.
  - Since $(v, w) \in E'_1 \iff (\phi(v), \phi(w)) \in E_2$, from set definition of $state1.Links$ and $state2.Links$ it follows that

$$\begin{aligned}&(x, directedEdge, y) \in state1.Links\\ &\iff\\ &(\phi(x), directedEdge, \phi(y)) \in state2.Links\end{aligned}$$

    is preserved.
  - Since 'edgeNodes' have 2 incoming arrows, and 'regularNodes' do not have incoming arrows, it follows from $(v, w) \in E'_1 \iff (\phi(v), \phi(w)) \in E_2$ that 'regularNodes' will be mapped via ϕ to 'regularNodes' and 'edgeNodes' will be mapped via ϕ to 'edgeNodes'. Hence, $x.Entity = \phi(x).Entity$ will be preserved.

  Hence ϕ is a bijection that maps $state1.Instances$ to $state2.Instances$ with $x \in state1.Repo \iff \phi(x) \in state2.Repo$ and $x.Entity = \phi(x).Entity$ and all instances $instance_x, instance_y \in state1.Instances$

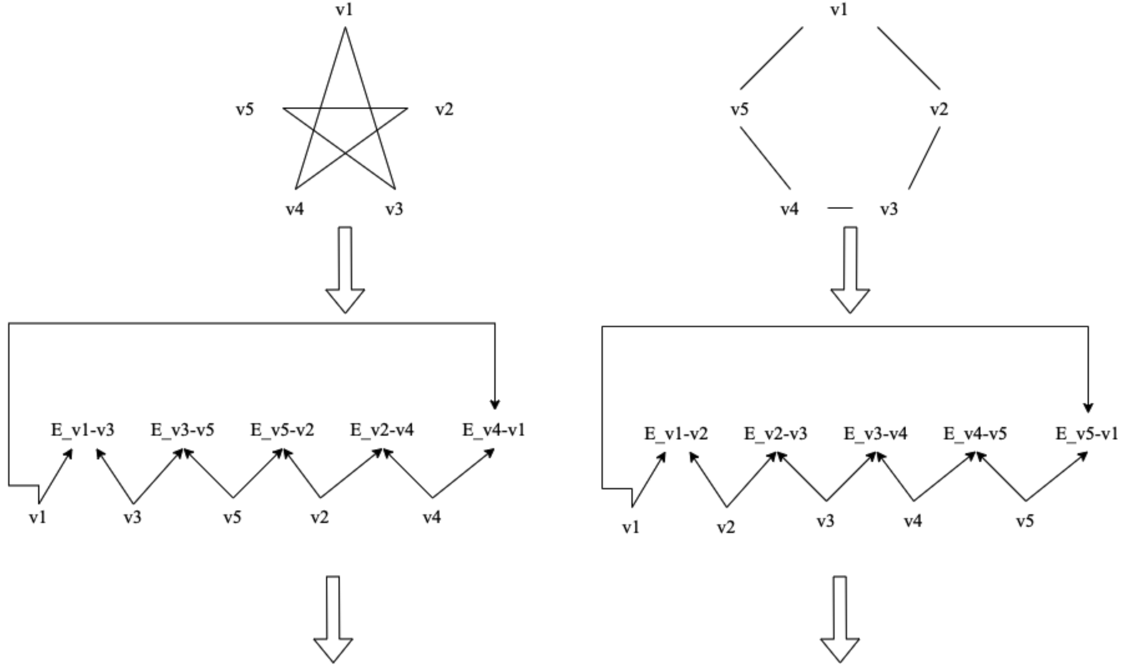
$$\begin{aligned}&(instance_x, \text{association_name}, instance_y) \in state1.Links\\ &\iff\\ &(\phi(instance_x), \text{association_name}, \phi(instance_y)) \in state2.Links\end{aligned}$$

  We conclude $state1 \sim state2$.

Since we have proven that the isomorphism problem on undirected graphs reduces to an isomorphism problem on a type of DAGs that can be reduced to determining state equivalency for a DMDSL model, we can conclude that Graph Isomorphism reduces to state equivalency.

□

Example 8.2.3. Suppose we wish to see if the following two undirected graphs G_1 (star shape) and G_2 (circle shape) are isomorphic. Then we would apply the following 2 transformation steps to reduce the problem to a state equivalency problem.



$$\begin{aligned}
state1.Instances &= \{v1, v2, v3, v4, v5, E_v1\text{-}v3, E_v3\text{-}v5, E_v5\text{-}v2, E_v2\text{-}v4, E_v4\text{-}v1\} \\
state1.Repo &= \emptyset \\
state1.Links &= \{(v1, \text{`directedEdge'}, E_v1\text{-}v3), (v3, \text{`directedEdge'}, E_v1\text{-}v3), (v3, \text{`directedEdge'}, E_v3\text{-}v5), \\
&\quad (v5, \text{`directedEdge'}, E_v3\text{-}v5), (v5, \text{`directedEdge'}, E_v5\text{-}v2), (v2, \text{`directedEdge'}, E_v5\text{-}v2), \\
&\quad (v2, \text{`directedEdge'}, E_v2\text{-}v4), (v4, \text{`directedEdge'}, E_v2\text{-}v4), (v4, \text{`directedEdge'}, E_v4\text{-}v1), \\
&\quad (v1, \text{`directedEdge'}, E_v4\text{-}v1)\}
\end{aligned}$$

$$\begin{aligned}
state2.Instances &= \{v1, v2, v3, v4, v5, E_v1\text{-}v2, E_v2\text{-}v3, E_v3\text{-}v4, E_v4\text{-}v5, E_v5\text{-}v1\} \\
state2.Repo &= \emptyset \\
state2.Links &= \{(v1, \text{`directedEdge'}, E_v1\text{-}v2), (v2, \text{`directedEdge'}, E_v1\text{-}v2), (v2, \text{`directedEdge'}, E_v2\text{-}v3), \\
&\quad (v3, \text{`directedEdge'}, E_v2\text{-}v3), (v3, \text{`directedEdge'}, E_v3\text{-}v4), (v4, \text{`directedEdge'}, E_v3\text{-}v4), \\
&\quad (v4, \text{`directedEdge'}, E_v4\text{-}v5), (v5, \text{`directedEdge'}, E_v4\text{-}v5), (v5, \text{`directedEdge'}, E_v5\text{-}v1), \\
&\quad (v1, \text{`directedEdge'}, E_5\text{-}v1)\}
\end{aligned}$$

It turns out that it is possible to reduce the graph isomorphism problem to the defined notion of state equivalency. That means, if we would manage to write an efficient algorithm to determine

whether two states are equivalent, then we could use this algorithm to solve the graph isomorphism problem for two arbitrary undirected graphs. This problem is not known to be solvable in polynomial time nor to be NP-complete. Therefore, it is unlikely that we would manage to create an efficient algorithm for checking state equivalency.

8.2.5.3 Application of State Equivalency

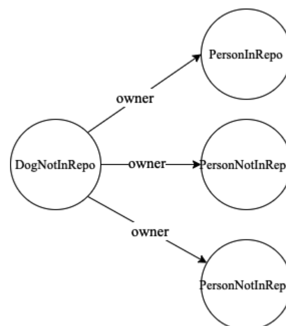
The result of the previous subsection seems a bit disappointing. It raises the question whether the notion of state equivalency is only of interest from a theoretical standpoint. In this subsection, we will provide a different reduction. We will show that state equivalency also reduces to Graph Isomorphism on labelled DAGs. This reduction allows us to use the mechanisms that have been discovered to tackle this problem in practice. In particular, we will make use of a hashing procedure that has the property that if two graphs have a different hash value, the graphs are not isomorphic. Using a reduction from state equivalency to graph isomorphism this hashing procedure can be used to exclude that two states are equivalent states.

Now, suppose we wish to make use of the introduced notion of state equivalency in our strategy. We desire that the enabled transition is picked of which the equivalence class of the resulting state has been visited least often. We will refer to a state of which no equivalent state has been seen in the test sequence as a *novel state*. Note that in a state often many transitions are enabled. Even with our finitized domains, it is typical to have more than 50 transitions enabled. Each of these transitions will have a resulting state. Suppose PyModel, discovered in the running sequence so far only 50 states. This suggests that in the worst case we have to do 50×50 comparisons, to determine which of the possible next states are *novel states*/corresponding equivalence class has been visited least often. Fortunately, this does not mean we need to solve for step 50×50 problems that are at least as hard as 50×50 Graph Isomorphism problems.

We have shown that the Graph Isomorphism problem reduces to state equivalency, showing that finding an efficient state equivalence procedure is not likely. There exists also an easy reduction from state equivalency to DAG isomorphism on labelled graphs.

Definition 27 (State graph)**.** Let *state* be a given state of a DMDSL Model Program. Let $G = (V, E)$ be a DAG, where $V = \{x \mid x \in state.I\}, E = \{(x, y) \mid (x, associationName, y) \in state.L\}$, and labelling function $\lambda(x) = x.EntityName\ [(x \in REPO)?\texttt{InRepo} : \texttt{NotInRepo}]$. I.e. the labels of the nodes encode the entity name of the corresponding instance, and whether the corresponding instance is in the repository. Then G is the *state graph* of *state*

Example 8.2.4. Consider the model in 3.1. Clearly the dynamic semantics of this model allow for a state *state* with $state.I = \{dogInstance0, personInstance1, personInstance2, personInstance3\}$, $state.L = \{(dogInstance0, \text{owner}, personInstance1), (dogInstance0, \text{owner}, personInstance2), (dogInstance0, \text{owner}, personInstance3)\}, state.REPO = \{personInstance2\}$. Then the corresponding state graph looks as follows:



Proof that DAG isomorphism for state graphs is equivalent to proving state equivalency is left to the reader. The proof is in fact quite trivial and can be done in a similar style as done in subsection 8.2.5.2. The result of this reduction is that we can use all existing techniques and implementations for graph isomorphism on labelled DAGs in determining state equivalency. In [26] the Weisfeiler-Lehman Graph Isomorphism test is introduced. This procedure can be used to assign hashes to graphs, such that when $graph1$ and $graph2$ are isomorphic then $hash(graph1) = hash(graph2)$, and when $graph1$ and $graph2$ are not isomorphic there are strong guarantees that $hash(graph1) \neq hash(graph2)$. Hence, such a method can be used to exclude that graphs are isomorphic, and when applied on state graphs exclude corresponding states would not be equivalent. Using such a hash function, we can determine for a potential next state that if its hash value has not been encountered earlier in the trace, we are *guaranteed that this potential next state is a novel state.* Given that equal hashes typically won't occur for graphs that are not isomorphic, this seems to be a good strategy to get a better state coverage modulo state equivalency. If the hash function would often give equal hashes on state graphs that are not isomorphic, the result would be that the strategy would avoid lots of states that are in fact novel states, which still is not that problematic. Fortunately, this should not happen often in practice. The hashing procedure has been implemented in the NetworkX python package. The package allows to apply this procedure also in Directed Acyclic Graphs that are labelled. The package also contains an exact algorithm for determining Graph Isomorphism.

We propose the following procedure. For all enabled transitions, compute for the resulting state the hash value of the corresponding state graph. Then, check via a dictionary whether the hash has been seen before and determine the group of enabled transitions leading to a state of which the hash has never been seen before (thus leading to a novel state) and pick a random transition out of the group. If there are no novel states reachable, determine the group of reachable states of which the corresponding hashes have been visited least often, and pick a random transition from the group. We will refer to the method of state coverage modulo state equivalency with State Coverage 2 (SC2).

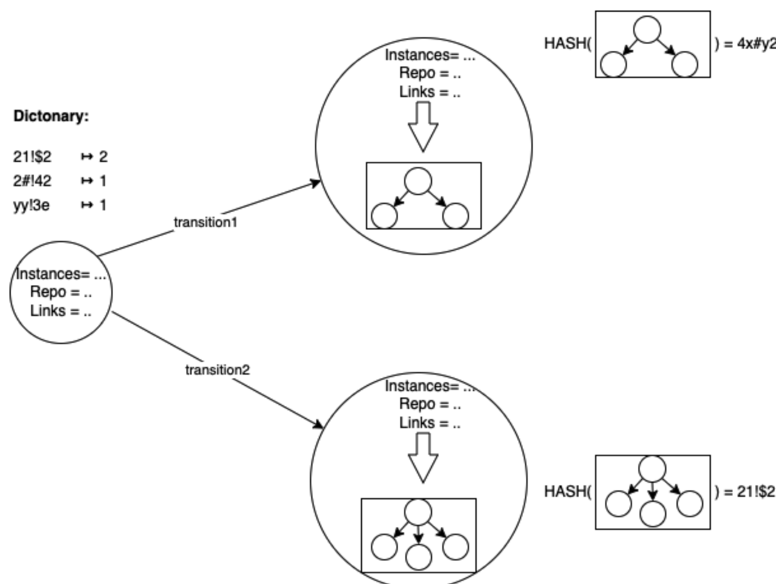


Figure 8.5: Visualization of State Coverage 2. In the figure transition1 would be the preferred transition.

Model	Hash revisits in SC1	False positives in SC1	Hash revisits in SC2	False positives in SC2
AME1437	11	0	0	0
bench	18	1	0	0
IPCDv2	49	0	2	0
SIRE01	99	0	2	0
TestOptional	120	0	5	0
TM003	73	8	2	0
TM020	85	0	2	0
TM033	12	0	0	0
TM065	17	0	0	0
TM10001	68	0	3	0

Table 8.1: Table indicating the number of hash revisits in generating a test sequence of lenght 250 using SC1 and SC2

8.2.5.4 State Coverage 2 vs. State Coverage 1

One could wonder if the notion of state equivalency introduced in SC2 is really interesting. Perhaps the notion in which the `all_identifiers` and `highest_id` variables are ignored in state comparison suffices (SC1), and visiting equivalent states does not happen often in practice. We apply the following procedure:

In sections 8.4.2 we will introduce a set of test models. Visualizations of these models are provided in appendix B. We run a single test sequence of transition length 250 using strategy SC1 and SC2 procedure. In "Hash revisits in SC1" the number of states visited, where for the corresponding state graph, the hash value was seen earlier in the test sequence. Two states having the same hash *suggests* that corresponding states are equivalent states. The hash function does not guarantee that two states with the same hash are equivalent states. States that have equal hashes, but are not equivalent states, we will refer to as false positives. We can confirm whether a false positive occurs using the exact isomorphism algorithm.

As can be seen in table 8.1 in 'False positives in SC1' there are not a lot of false positives, showing that the hashing function works well in determining whether two graphs are isomorphic, and thus wether two states are equivalent. It is clear that visiting equivalent states happens often using SC1. Only for TM003 the number of false positives was relatively high. But even for this model, still, $73 - 8 = 65$ of the states visited, were non-novel states. SC2 managed to avoid most of the equivalent states. Only in rare situations, PyModel needed to take a transition leading to a state of which an equivalent state was already visited earlier in the sequence. Especially for smaller models such as TestOptional, the reduction is significant. In both test runs, 250 states were visited. In SC1 120/250 states were non-novel states, whereas in SC2 only 5/250 were non-novel states. This shows the strong need for this notion in smaller models.

Analysis on 'Extensive Model' We will now run and keep statistics of how many duplicate states are visited during the execution of a test sequence on the model called 'Extensive Model' of figure 8.9. After executing three test sequences of 250 transitions with SC1, we observe that on average 121 states are visited of which the corresponding hash was already seen earlier in the trace. What was interesting to observe is that in this model there were lots of false positives. There were in fact on average 80 false positives. This means that there are actually much fewer equivalent states visited. There were on average 41 equivalent states visited using SC1. For SC2 we observe that on average 29 states are visited of which the corresponding hash was already seen earlier in the trace. On average there were 12 false positives, and thus the average number of equivalent states visited was 17. It is interesting that this specific model gives quite a lot of false positives. It is expected that some of the induced state graphs that are typical for this kind of DMDSL model to occur are a type of graph in which the Weisfeiler-Lehman graph hash will give

equal hashes. Further research is required here.

The linear data (averaged) has been plotted in figure 8.6 We can see that for the SC1 in the

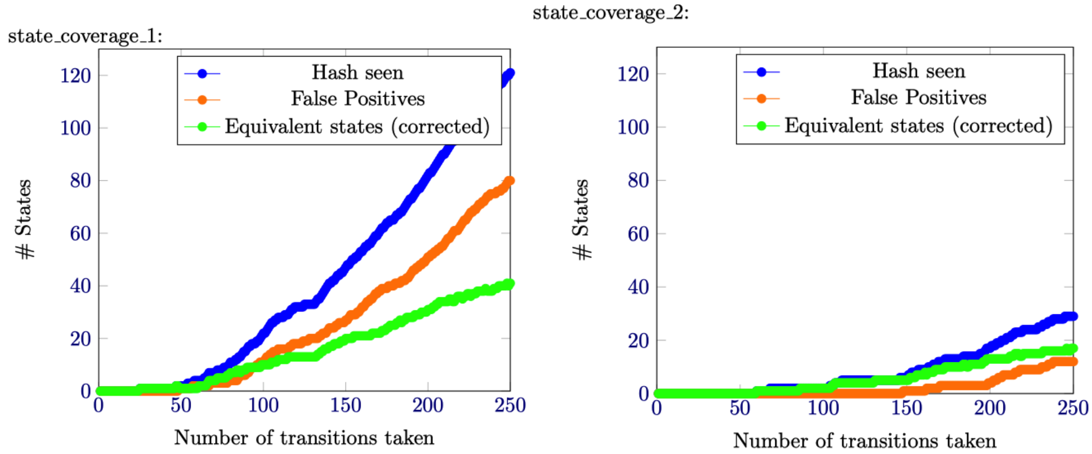


Figure 8.6: Plot of equivalent states visited for a test sequence of size 250, on EXTENSIVE_MODEL with STATE COVERAGE 1 and STATE COVERAGE 2 strategy.

first 50 transitions, not a lot of duplicate states are visited. For SC2 visiting states with a known hash, could be postponed to roughly 100 transitions. It is not very surprising that such a section exists in the beginning, since all states are initially unknown. When more states are visited, it becomes increasingly more likely that a state is visited of which no equivalent state has been seen before. Even though this particular model does not seem to be ideal for our procedure, the SC2 still is significantly better than SC1 in revisiting novel states. The growth in visits of equivalent states visited is also much less steep in SC2 when compared to SC1.

8.2.5.5 Chains of Entities Continued

We will also try to apply SC2 on the model in figure 8.2 three times with a test sequence of 250 transitions. This gives a creation average of 5 Entity4 instances, 5 Entity3 instances, 5 Entity2 instances, and 2 Entity1 instances. All of these instances are also added to the repository. As a matter of fact, this is the theoretical Create maximum for each of the Entities. In all three runs, it managed to produce the theoretical maximum. It turns out that the incline of avoiding equivalent states works very well in exploring the chain example. Also, since creation is done on arbitrary moments (and not only in a creation phase), the domains of the create actions are more often finitized, increasing the probability that at some point a successful create action is possible. The fact that the notion of state equivalence seems to be able to cope with chains in the creation process, and tries to avoid visiting equivalent states, makes this strategy appealing.

8.2.6 State Coverage 3

It should be noted that SC2 will pick a state it has not seen before. Therefore, it will practically always take a Create, Update, Delete or Add action, since these will result in modification of the state variables. The Read action will thus practically never be executed. This seems undesirable since it is relevant to know if instances that should not be in the repository, are not in it and if instances that should be there, are in fact there. A simple fix that we can add is to simply add some of the enabled Read actions to the set containing the actions leading to novel/least visited states. Since the newly performed action will be chosen from this set, we will have that sometimes a Read action will occur (and as a result the next state will then be the same state). We will in this way thus slightly deviate from the State Coverage strategy of reaching as many different states as possible, but as a result, sometimes execute Read actions.

Another problem that we can observe is that some actions are overrepresented in the test sequence due to the high number of parameters that are possible for the specific action. One could wonder whether it is really more likely for problems to occur in such a function when more arguments are possible. Furthermore, we finitized the domains, so in practice, some of the less represented actions in the test sequence might have more possible arguments making the representation rather arbitrary. Hence, instead, we will give equal priority to each of the actions, so that at least equal chances are given for the execution of each of the CRUD+A actions. We can do this by grouping the enabled transitions on the basis of the corresponding action. We randomly pick a CRUD+A action of which the corresponding group is non-empty. Then randomly pick a transition out of the corresponding group. In this way, all of the CRUD+A actions will be more equally represented in the test sequence.

We refer to this strategy in which Read actions are included, and equal priority is given to all CRUD+A actions as State Coverage 3 (SC3)

8.2.7 State Coverage 4

Finally, when observing the test sequence of SC1,2,3, we observe that typically only `Success` output messages occur. This is actually as you would expect since when error output messages occur, the state variables do not change. We of course also would like to consider behavior that should induce error messages. A way to cope with this is to consider the State Coverage strategy as seen in SC2, but every second transition will be a random enabled transition (with equal priority for each of the enabled CRUD+A actions, as seen in SC3). In this way, we will explore the state space, but for a significant part of the states discovered, also have a self-loop executed and some error output message tested. We will refer to this strategy as State Coverage 4 (SC4)

8.2.8 Coverage for output messages

8.2.8.1 Output Coverage

We have distinguished several output messages. To test in an efficient manner whether these output messages are implemented correctly by the IUT, we can introduce the notion of Output Coverage. We simply maintain in a dictionary which of the output messages we have already seen. Then, for all enabled transition we pick a random transition that results in an output message that is seen least often.

8.2.8.2 Output-Action Coverage

While the proposed strategy might seem tempting, one should be aware that some of the output actions such as `Success` or `Instance_NotInRepository` are shared by different actions. It is clear that these actions will have different implementations. The fact that the IUT managed to produce output x for action y, does not mean that output x will also correctly be produced for action z (where $y \neq z$). It, therefore, makes sense to make the output coverage dependent on the action that is taken. We can change this by using the Action name+output message as key in the dictionary.

On DMDSL models we can see that these strategies focussing on output are not of practical relevance. There will be an emphasis on giving different output messages, with as a result that the model will not really be discovered. The `Success` output message will be largely under-represented.

8.3 Strategy Analysis

In the previous section, we considered different strategies. We considered Random Strategy, Exhaustive Strategy, SC3, SC4 to be the most interesting strategies. In this section, we will analyze the different strategies on a model and point out some of the identified benefits and

drawbacks of applying them on a large model. Furthermore, we will propose refinements to SC3 called SC3.1 and SC3.2 and to the domain of identifiers. The analysis will happen without actually executing them on the implementation. In this subsection we only care about the generation of abstract tests and how the different strategies compare in this.

8.3.1 Bench

We will have a close look at the Bench model in appendix B.2. It is used to perform benchmark analysis by Capgemini. As a matter of fact, it is a realistic example of a DMDSL model that could be used by ASML. The exact names are therefore disclosed. As one can observe from the figure, we have indeed long chains of entities in the figure. E.g., to create an Entity1 instance, one needs to provide an Entity2 instance, to create an Entity2 instance one needs to provide an Entity4 instance, etc. This shows that models in practice suffer from the chain issue. We will do a more careful analysis to see how the different strategies perform on such an extensive model. We create test sequences of 300 transitions with the different strategies. The spread in the Creation and Adding of instances is visualized in figure 8.7. This gives a notion to what extent the model is explored. If some entities are not created or added, some features of the model might not be tested. The spread in output messages is visualized in figure 8.8. This gives a notion if for all actions sufficient `Success` output messages occur, and gives some insight to what extent all other output messages are covered. The runtime of the strategies is indicated in table 8.2.

Strategy	Average runtime
Exhaustive Create/Add	696 sec.
Random	92 sec.
State Coverage 3	159 sec.
State Coverage 4	130 sec.
State Coverage 3.1	230 sec.
State Coverage 3.2	63 sec.

Table 8.2: Runtime of different strategies on B.2.

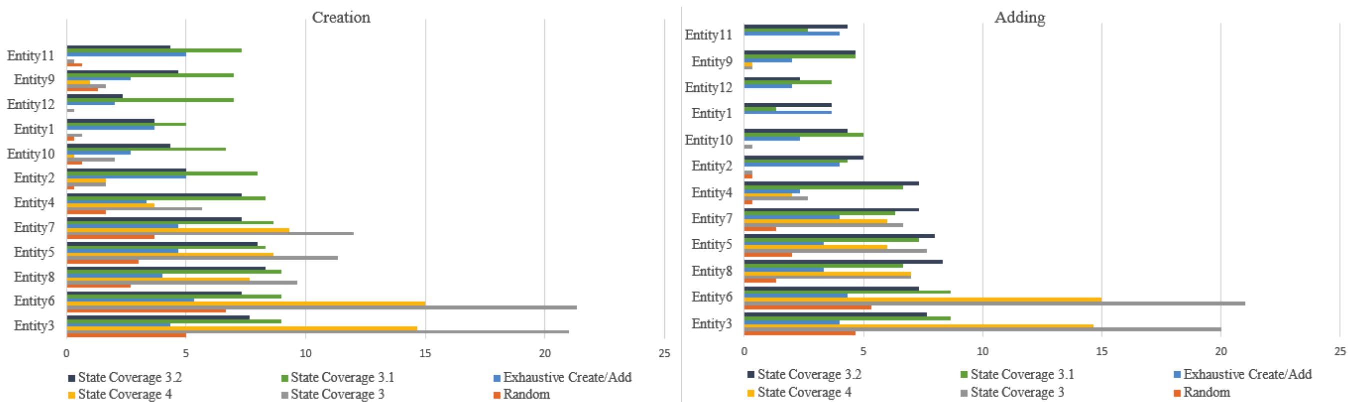


Figure 8.7: Number of instances created and added split on Entity type.

Random Strategy We observe that the Create action is overrepresented in the Random Strategy. Many of the Create attempts resulted in an Error output message. This is not very surprising since the chances are not so high that with the random strategy all requirements of the Create action will be fulfilled. This also limits the domains of the remaining CRUD+A

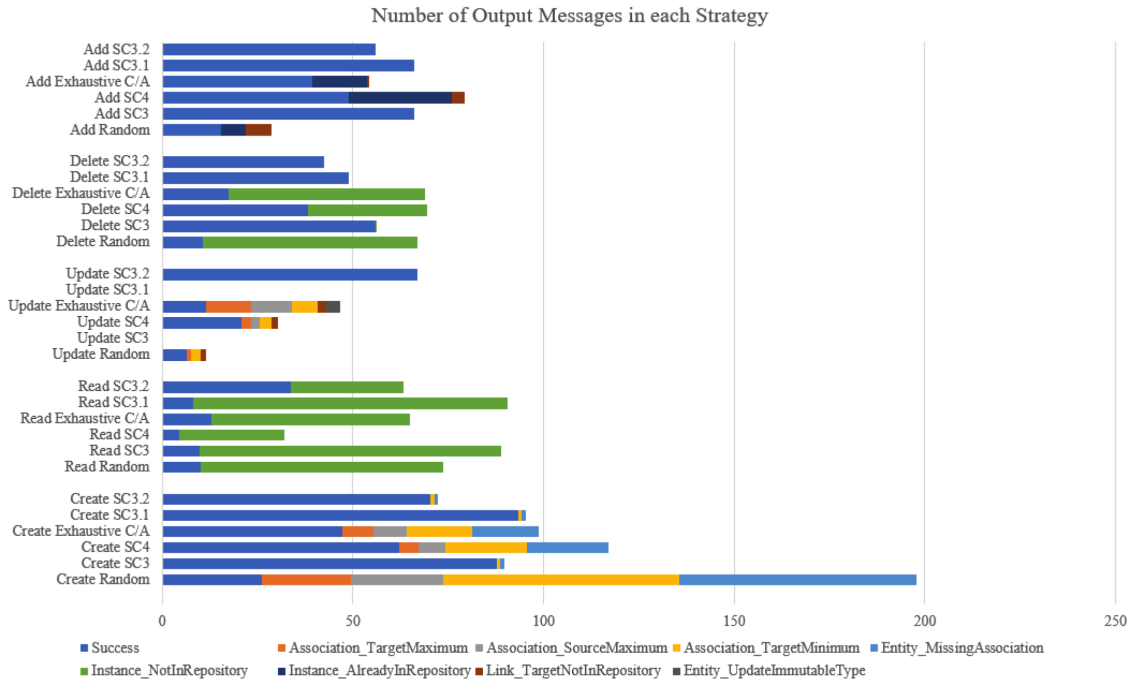


Figure 8.8: Output messages per strategy.

actions. There are, for example, fewer instances that one could add to the repository. As a result, the Create actions will be over-represented in the set of enabled transitions, and in random selection of an enabled action the chances are higher a Create action will be picked. We can also observe that the strategy does not do well in covering the different Entities in the Create and Add actions. For example, Entity4 is not even created, and thus also not added in the test sequence due to the chaining problem. The runtime of this strategy is fortunately relatively low.

Exhaustive Create Add We observe that in Exhaustive Create Add fewer Create actions are done in comparison to random testing, and more of them result in a `Success`. Furthermore, we can observe that in Exhaustive Create Add these are better spread out over the entities, which is preferable. In terms of output messages, it behaves similarly to Random Testing since after the Create and Add phase it switches to this strategy. It may be undesirable to have specific phases in which these actions occur. It might actually be desirable to have these actions happen at arbitrary moments instead. A big downside to this strategy is that the runtime is rather slow. This is due to the fact that the finitized domains are rather large, after the Create and Add phases.

State Coverage 3 We observe that in SC3, the number of entities created and added is rather large. Unfortunately, on such a large model this is not properly spread out over the different Entity types. In, for example, the entity Entity1, the strategy does not manage to add instances to the repository. We expect it works less well here than in the chain example seen in figure 8.2 because now the Entities are now deletable. As a result, there are more variations of states possible by adding and removing instances of Entities 'in the front of the chain'. We can see this in for example Entity6, which does not require any links at all in the creation, that lots of deletions and adds are done on this type of instance. We expect that for these large models a much more extensive testing process with SC 2/3/4 is needed to get better coverage. Furthermore, it should be noted that almost all output messages are `Success` output messages since only these lead to state changes. Fortunately, it is good that the notion of state equivalence does not add a large burden on the runtime showing that the hashing procedure indeed had the desired effect.

Proposal State Coverage 3.1: We noticed that even though the SC3 strategy tries to find a state in which no equivalent states have been seen, it does not evenly spread out the Create and Add actions due to the fact that the *chaining problem* still exists on large models in which the entities are deletable. We, therefore, propose a refinement. In the State Coverage 3.1 strategy, in the case of a Create/Add action it does not randomly pick a corresponding Enabled transition but instead, picks the transition that applies it for an Entity seen least often in such an action.

State Coverage 3.1 (SC3.1) We observe in figure 8.7 that the strategy indeed manages to get a good spread in both the creation and adding of instances of each entity type. Hence, this refinement helps the SC3 strategy cope with the chain issue. In terms of output messages, it behaves very similarly to SC3 as expected. As one would expect, the maintenance of keeping track of which entity types have been seen least often does put a burden on the runtime. Fortunately, the runtime is still significantly less than the Exhaustive Create Add strategy, making it very appealing.

Proposal State Coverage 3.2: What is also problematic in SC3 and SC3.1 is the fact that no Update action is executed. There are multiple reasons why this occurs, but the most significant reason is the fact that for many models it is quite hard to take an Update action leading to a state that is not equivalent to some other state. Even though the result of the Update could lead to a state that is not unique, it still is important to test whether the change of links happens properly. We, therefore, propose to add Update actions that have `Success` as output to the set containing the actions leading to new/least visited states.

A general issue that has become clear from figure 8.8 is that `Instance_NotInRepository` is overrepresented in the Read action. This is actually not very surprising since the identifiers are taken from the `all_identifiers` list as specified in option 1 in 8.1.2. For large test sequences, this list will get very large, causing an incremental burden on the runtime, where the majority of the identifiers correspond to instances that either do not exist or are not in the repository. We resolve this by changing the domain to option 2 instead. As we will use State Coverage 3.2 as testing procedure in the next section using option 2, it is important to do the analysis using this new domain.

State Coverage 3.3 (SC3.2) We indeed observe that besides a good spread in both the creation and adding of instances of each entity type, it now also involves successful executions of the Update action. Furthermore, now for approximately half of the Read actions, we get a `Success`, and for the other half, we get an `Instance_NotInRepository`. The new domain, indeed now makes sure that we properly test the reading of instances that should be in the repository. Furthermore, since we limited the domain of identifiers the runtime also significantly improved.

State Coverage 4 The strategy performs similarly to the original SC3 in the spread of creation and adding of instances, which unfortunately is not very good. Since every second transition taken is a random action, we do now consider all the error outputs. Furthermore, it even managed to (successfully!) execute Update actions (without the refinement proposed in SC3.2), which were missing in the SC3 strategy. Since every second transition taken is a random transition, the runtime is a bit less than SC3.

8.4 Testing a Model

We will now put MBT into practice on a model, and start with actually testing some generated code. We will consider the following manually crafted model called Extensive_Model. This model is visualized in figure 8.9

Figure 8.9: Visualization of Extensive Model.

The model contains entities that are deletable, undeletable, undestructable, mutable, immutable, uneditable. All entities are constructable since we are working under the context of a single domain-interface. The modeling constraints do not allow modeling an entity that is not unconstructable if there is not a second domain interface in which it is actually constructable. The model has several association relations all with source cascade deletion enabled. There are several variations of this in the model. We have a mandatory association from Entity1 to Entity3. We have an optional association from Entity1 to Entity4, and we have a required association relation from Entity1 to Entity2 that might contain more than 1 target element. There is also an association source maximum of 2 for the association between Entity1 and Entity3. The remainder of the association relations has an association source maximum of 1.

8.4.1 Finding Issues

Execution command: We will execute the test command on the Pymodel Model by making a call `pmt -n 200 -i stepper Model_Translation` to start the PyModel tester. In particular, we make a call to `pmt` (PyModel Tester) with argument `-n 200` indicating that we would like to take 200 transitions for the test sequence, and argument `-i stepper` indicating that `pmt` should use the `stepper.py` file to communicate with the IUT.

First time running *execution command* gives:

```
Create(("Entity4", ()), 'instance0', 0) / '{Success}'
```

```
Create(("Entity3", ("instance1,entity3entity4Assoc,instance0", "
    instance1,entity3entity4Assoc,instance0")), 'instance1', 1) / '{
    Success}'

Create(("Entity3", ("entity3entity4Assoc,instance0", "
    entity3entity4Assoc,instance0", "entity3entity4Assoc,instance0")
    ), 'instance2', 2) / '{Association_SourceMaximum,
    Association_TargetMaximum}'

  0. Failure at step 3, there is a problem. implementation gave
     Success where {Association_SourceMaximum,
     Association_TargetMaximum} was expected
```

In the third step, the tool tries to create an instance of Entity3, with as targets for the **entity3entity4** association, three times `instance0`. According to the model, this would be incorrect. After the second step `instance0` has two incoming `entity3entity4` links. If the third step would be successful, three extra `entity3entity4` links would have `instance0` as a target. The `entity3entity4` association allows an arbitrary instance of Entity4 to have at most two incoming `entity3entity4` links. Hence, the `Association_SourceMaximum` applies. Furthermore, the creation of `instance2` would result in three outgoing links from `instance2`. This is not allowed by the association relation that states that it needs to have at most 2 outgoing links. Hence, `Association_TargetMaximum` applies. The implementation does not implement these checks and returns `Success`. This bug is not very surprising, since we could not map these output messages since these are not statically enforced. It is good to see that so quickly the MBT tool found this problem. We conclude: The `Association_TargetMaximum` and `Association_SourceMaximum` are not always enforced.

Let's see if running the command again, points out a different bug. Second time running *execution command* gives:

```
Create(("Entity4", ()), 'instance0', 0) / '{Success}'
Create(("Entity2", ()), 'instance1', 1) / '{
    Entity_MissingAssociation, Association_TargetMinimum}'
Create(("Entity2", ("instance1,entity2entity4Assoc,instance0",)), '
    instance1', 1) / '{Success}'
Create(("Entity3", ()), 'instance2', 2) / '{
    Entity_MissingAssociation, Association_TargetMinimum}'

0. Failure at step 4, there is a problem. implementation gave
    Success where {Entity_MissingAssociation,
    Association_TargetMinimum} was expected
```

In step 3, an attempt is made to create an instance of Entity3, called `instance2` without any outgoing links. The model specified there should be precisely two `entity3entity4Assoc` links. Hence, we would violate the minimum number of associations, and thus `Association_TargetMinimum` applies. Furthermore, a required association is missing, and thus `Entity_MissingAssociation` applies. Such test sequences require one to think more carefully about the output messages `Entity_MissingAssociation` and `Association_TargetMinimum`. In practice, we see that when the former output message appears, the latter output appears as well. One could wonder whether the `Entity_MissingAssociation` is thus in fact redundant. Indeed, analysis of the formulas confirms this. We conclude: `Entity_MissingAssociation` implies `Association_TargetMinimum` making `Entity_MissingAssociation` redundant. Furthermore, `Association_TargetMinimum` is not always enforced.

Let's see if running the command again, points out a different bug. Third time running *execution command* gives:

```
Create(("Entity1", ()), 'instance0', 0) / '{
    Association_TargetMinimum, Entity_MissingAssociation}'
Create(("Entity4", ()), 'instance0', 0) / '{Success}'
Create(("Entity1", ()), 'instance1', 1) / '{
    Association_TargetMinimum, Entity_MissingAssociation}'
Create(("Entity4", ()), 'instance1', 1) / '{Success}'
Create(("Entity2", ("instance2,entity2entity4Assoc,instance1",)), '
    instance2', 2) / '{Success}'
Add("instance0",) / '{Success}'
Read(0,) / '{Success}'
Delete(2,) / '{Instance_NotInRepository,
    Entity_DeleteOnUndestructable}'
Add("instance2",) / '{Link_TargetNotInRepository}'
  0. Failure at step 9, there is a problem. implementation gave
     Success where {Link_TargetNotInRepository} was expected
```

When trying to add `instance2` to the repository, the implementation does not complain. However, `instance2` has as `entity2entity4Assoc` target `instance1`. This instance was never added to the repository. The semantics does not allow for adding instances to the repository that have a link to an instance that is not in the repository. We conclude: The `Link_TargetNotInRepository` is not always enforced.

It is now much harder to find traces that induce different kinds of bugs. We are aware that the above issues exist, but we can unfortunately not instantly solve these (this is also not really our task as a tester). To continue the testing process, we can avoid traces containing `Link_TargetNotInRepository` as output for the Add operation and `Entity_MissingAssociation, Association_TargetMinimum`, `Association_SourceMaximum, Association_TargetMaximum` as output for the Create operation. We can do this by specifying in the guard of corresponding action that it is disabled if one of the above messages will be output of executing the action.

Let's see if running the command again, points out a different bug. Fourth time running *execution command* gives:

```
Create(("Entity4", ()), 'instance0', 0) / '{Success}'
Create(("Entity2", ("instance1,entity2entity4Assoc,instance0",)), '
    instance1', 1) / '{Success}'
Create(("Entity4", ()), 'instance2', 2) / '{Success}'
Add("instance2",) / '{Success}'
Read(2,) / '{Success}'
Read(0,) / '{Instance_NotInRepository}'
Create(("Entity2", ("instance3,entity2entity4Assoc,instance2",)), '
    instance3', 3) / '{Success}'
Delete(2,) / '{Entity_DeleteOnUndestructable}'
  0. Failure at step 8, there is a problem. implementation gave
     Success where {Entity_DeleteOnUndestructable} was expected
```

We observe that something went wrong with the deletion using identifier 2. This is the identifier of `instance2`, which is of type `Entity4`. PyModel indicates that `Entity_DeleteOnUndestructable` is expected. `Entity4` is, however, destructable. 'Why is this happening?' one might wonder. Note that `instance3` was successfully created with `instance2` as a target instance. Now, when deleting `instance2`, we have that due to source cascade deletion the `instance3` gets deleted. This instance is in fact undeletable. It turns out that in this way it is possible to delete instances that are undeletable, even though we thought this would not be possible within the context of a single domain-interface. The semantic representation of deletability as a boolean value, as done in the semantics in the work of Derasari is thus not right. We conclude: There is in fact a difference

between undestructable and undeletable within the context of a single domain-interface. Either the semantics need to be adjusted, or models where *undeletable* entities that have an outgoing association with source-cascade deletion enabled should be rejected by the OCL constraints. The latter happens for entities that are *undestructable*.

Let's see if running the command again, points out a different bug. Fifth time running *execution command* gives:

```
Create(("Entity5", ()), 'instance0', 0) / '{Success}'
Create(("Entity5", ()), 'instance1', 1) / '{Success}'
Create(("Entity5", ()), 'instance2', 2) / '{
   Entity_MultiplicityMaximum}'
  0. Failure at step 3, there is a problem. implementation gave
     Success where {Entity_MultiplicityMaximum} was expected
```

In this trace, two instances of `Entity5` are created. When an attempt is made to create a third instance of `Entity5`, the implementation gives a `Success` where we would expect an `Entity_MultiplicityMaximum`. We conclude: The implementation does not respect the maximum multiplicity boundary.

If we also avoid traces that output `Entity_MultiplicityMaximum` of the Create action, and `Entity_Undestructable` for the Delete action, we sometimes finish without problems. However, when generating a test sequence multiple times, it is also be pointed out that `Association_TargetMinimum`, `Association_TargetMaximum`, `Association_SourceMaximum`, and `Link_TargetNotInRepository` are not enforced on the Update operation. When also avoiding traces containing these output messages for the Update operation, PyModel is able to successfully execute the tests.

Most of the issues shown here are due to the fact that the constraints specified in the model are simply not enforced in the executable code. While these kinds of issues are not the type of bugs where the true power of MBT lies, the methodology still provided a structured approach in finding issues in code of which the exact workings is unknown. After studying the code and talking to the developers we conclude from the above problems the following set of issues:

- The `Association_TargetMaximum` is not enforced for association target multiplicity > 1 on both the Create and Update action.
- The `Association_SourceMaximum` is not enforced on both the Create and Update action.
- `Association_TargetMinimum` is not enforced for association target multiplicity > 1 on both the Create and Update action.
- `Entity_MissingAssociation` implies `Association_TargetMinimum`, making it a redundant output message.
- `Link_TargetNotInRepository` is not enforced on both the Add and Update action.
- There is a distinction between an instance that is undeletable and undestructable in the context of a single domain interface, even though the semantics do not make this distinction.
- `Entity_MultiplicityMaximum` is not enforced on the Create action.

8.4.2 Applying PyModel on Capgemini Test Models

We will now use PyModel to test the generated code of the test models developed by Capgemini. Since we have identified the generated code suffers from mostly issues when the preconditions of actions are not satisfied, we will now focus on testing if the code conforms to the specification for behavior in which the preconditions are not violated. In other words, the behavior conforms, if the programmer uses the generated code as intended. This is still very interesting. It should provide confidence in things as

- Whether the implementation implements source cascade deletion in all situations properly.
- Whether instances that should be in the repository, are actually in the repository.
- Whether the Update action works properly, and it is respected by source cascade deletion
- etc.

The strategy that tests this behavior properly is the STATE COVERAGE 3.2 strategy. Except for the Read action, typically all actions will have `Success` as output since these induce state changes.

> Note: Initially no hash is in the dictionary, and therefore it is not uncommon that the first action produces output different from `Success`. Furthermore, in rare cases, it is possible that all enabled transitions lead to only non-novel states. In such cases, it could also be that output different from `Success` is produced. Since Read actions are additionally added, these can also give output different from `Success`.

In table 8.3, a large number of the test models used by Capgemini are listed. These are all adjusted (removal of data attributes, removal of specialization relations, making all multiplicities finite) so that they are covered by the formal semantics we use, and we can thus apply our MBT approach. These test models all have some interesting aspects. Some information about these models is listed in the table. Furthermore, in appendix B, these test models are visualized. Unfortunately, since we only cover a subset of the semantics, and we restrict ourselves to reference-based repositories with only intraprocess communication, some of the models show their feature of interest mostly in aspects not covered. Nevertheless, they touch on different aspects of the semantics that are covered, making them an interesting set of models to apply the MBT approach on.

Inspired by	#entities	#constr.	#mut.	#delet.	#assoc.	shape	figure
AME1437	5	5	2	4	3	3-outgoing	B.1
Bench	12	12	10	12	11	Long chain & cluttered	B.2
IPCDv2	4	4	2	4	3	Zig-zag	B.3
SIRE01	4	4	2	4	3	Zig-zag	B.4
TM003	5	5	2	4	4	Diamond	B.6
TM020	4	4	0	4	4	Diamond	B.7
TM033	7	7	0	7	6	Pyramid	B.8
TM065	7	7	0	6	7	Saucepan/Square	B.9
TM10001	5	5	2	5	4	Small chain	B.10

Table 8.3: Summary of the properties of the test models.

We will explore each of the listed test models. We will run test sequences of different sizes so that for each entity type at least one instance is created and added to the repository (ideally more than 1). In this way, we have notion if the data model has been discovered to a sufficient depth. Furthermore, for each Action, we will list the number of actions taken.

Experiment 1

Test Model: AME1437

Testing Strategy: State Coverage 3.2

Number of transitions: 300

Execution Time: 59.86 seconds

Number of instances created per entity: EntityA: 18, EntityB: 18, EntityC: 18, EntityD: 19, EntityChainStart: 4

Number of instances added per entity: EntityA: 13, EntityB: 15, EntityC: 16, EntityD: 16, EntityChainStart: 4

Action output:

- `Success`: Create 77, Read 30, Update 37, Delete 52, Add 64
- `Instance_NotInRepository`: Read 40

Generated code passes: YES

Experiment 2

Test Model: Bench

Testing Strategy: State Coverage 3.2

Number of transitions: 500

Execution Time: 149.37 seconds

Number of instances created per entity: Entity1: 7, Entity2: 10, Entity3: 10, Entity4: 10, Entity5: 10, Entity6: 10, Entity7: 10, Entity8: 10, Entity9: 10, Entity10: 10, Entity11: 10

Number of instances added per entity: Entity1: 2, Entity2: 6, Entity3: 10, Entity4: 6, Entity5: 9, Entity6: 10, Entity7: 9, Entity8: 9, Entity9: 8, Entity10: 8

Action output:

- `Success`: Create 116, Read 59, Update 109, Delete 66, Add 86
- `Instance_NotInRepository`: Read 63
- `Association_TargetMinimum`: Create: 1
- `Entity_MissingAssociation`: Create 1

Generated code passes: YES

Experiment 3

Test Model: IPCDv2

Testing Strategy: State Coverage 3.2

Number of transitions: 300

Execution Time: 22.66 seconds

Number of instances created per entity: EntityA1: 29, EntityA2: 6, EntityB1: 9, EntityB2: 27

Number of instances added per entity: EntityA1 26, EntityA2 3, EntityB1 3, EntityB2 27

Action output:

- `Success`: Create: 71, Read: 41, Update; 45, Delete: 52, Add: 59
- `Instance_NotInRepository`: Read: 31
- `Association_TargetMinimum`: Create: 1
- `Entity_MissingAssociation`: Create: 1

Generated code passes: YES

Experiment 4

Test Model: SIRE01

Testing Strategy: State Coverage 3.2

Number of transitions: 300

Execution Time: 35.67 seconds

Number of instances created per entity: EntityA1: 24, EntityA2: 22, EntityB1: 22, EntityB2: 23

Number of instances added per entity: EntityA1: 21, EntityA2: 13, EntityB1: 5, EntityB2: 21

Action output:

- `Success`: Create: 91, Read: 38, Update: 42, Delete: 48, Add: 60
- `Instance_NotInRepository`: Read: 20
- `Association_TargetMinimum`: Create: 1
- `Entity_MissingAssociation`: Create: 1

Generated code passes: YES

Experiment 5

Test Model: TestOptional

Testing Strategy: State Coverage 3.2

Number of transitions: 300

Execution Time: 61.25 seconds

Number of instances created per entity: Entity1: 35, Entity2: 31

Number of instances added per entity: Entity1: 30, Entity2: 29

Action output:

- `Success`: Create: 66, Read: 57, Update: 0, Delete: 47, Add: 59
- `Instance_NotInRepository`: Read: 64, Delete: 1
- `Association_TargetMinimum`: Create: 1
- `Entity_MissingAssociation`: Create: 1
- `Instance_AlreadyInRepository`: Add: 1
- `Entity_UpdateImmutableType`: Update: 3
- `Association_TargetMaximum`: Create: 1
- `Association_SourceMaximum`: Create: 1

Generated code passes: YES

Experiment 6

Test Model: TM003

Testing Strategy: State Coverage 3.2

Number of transitions: 300

Execution Time: 76.16 seconds

Number of instances created per entity: Entity1: 22, Entity2: 21, Entity3: 21, Entity4: 20, Entity5: 4

Number of instances added per entity: Entity1: 20, Entity2: 18, Entity3: 17, Entity4: 5, Entity5: 4

Action output:

- `Success`: Create: 88, Read: 30, Update: 54, Delete: 36, Add: 64
- `Instance_NotInRepository`: Read: 27
- `Entity_UpdateImmutableType`: Update: 3

Generated code passes: YES

Experiment 7

Test Model: TM020

Testing Strategy: State Coverage 3.2

Number of transitions: 300

Execution Time: 45.75 seconds

Number of instances created per entity: Entity1: 26, Entity2: 25, Entity3: 24, Entity4: 22

Number of instances added per entity: Entity1: 23, Entity2: 16, Entity3: 17, Entity4: 6

Action output:

- `Success`: Create: 97, Read: 45, Update: 0, Delete: 48, Add: 62
- `Instance_NotInRepository`: Read: 46
- `Entity_UpdateImmutableType`: Update: 1
- `Association_TargetMinimum`: Create: 1
- `Entity_MissingAssociation`: Create: 1

Generated code passes: YES

Experiment 8

Test Model: TM033

Testing Strategy: State Coverage 3.2

Number of transitions: 400

Execution Time: 61.83 seconds

Number of instances created per entity: Entity1: 19, Entity2: 19, Entity3: 18, Entity4: 18, Entity5: 18, Entity6: 17, Entity7: 20

Number of instances added per entity: Entity1: 19, Entity2: 16, Entity3: 14, Entity4 9, Entity5: 9, Entity6: 8, Entity7: 18

Action output:

- `Success`: Create: 92, Read: 41, Update: 21, Delete: 49, Add: 64
- `Instance_NotInRepository`: Read: 32
- `Association_TargetMinimum`: Create: 1
- `Entity_MissingAssociation`: Create: 1

Generated code passes: YES

Experiment 8

Test Model: TM065

Testing Strategy: State Coverage 3.2

Number of transitions: 600

Execution Time: 424.19 seconds

Number of instances created per entity: Entity1: 47, Entity2: 48, Entity3: 16, Entity4: 48, Entity4B: 13, Entity6: 5 , Entity7: 7

Number of instances added per entity: Entity1: 45, Entity2: 33, Entity3: 7, Entity4: 33, Entity4B: 2, Entity6: 5 , Entity7: 7

Action output:

- `Success`: Create: 184, Read: 102, Update: 0, Delete: 96, Add: 132
- `Instance_NotInRepository`: Read: 81, Delete: 1
- `Instance_AlreadyInRepository`: Add: 2
- `Association_TargetMinimum`: Create: 2
- `Entity_MissingAssociation`: Create: 2

Generated code passes: YES

Experiment 9

Test Model: TM10001

Testing Strategy: State Coverage 3.2

Number of transitions: 300

Execution Time: 43.85 seconds

Number of instances created per entity: EntityA0 19:, EntityA1: 18, EntityA2: 15, EntityB1: 19, EntityB2: 21

Number of instances added per entity: EntityA0: 19, EntityA1: 14, EntityA2: 8, EntityB1: 5, EntityB2: 18

Action output:

- `Success`: Create: 92, Read: 41, Update: 21, Delete: 49, Add: 64
- `Instance_NotInRepository`: Read: 32
- `Association_TargetMinimum`: Create: 1
- `Entity_MissingAssociation`: Create: 1

Generated code passes: YES

All experiments passed. The result is not surprising since the code generator is a mature code generator that is in use for approximately five years. It would be of interest to see when more peculiar features are used if the code generator still works correctly. E.g., the usage of multiple domain interfaces, customized service realizations, and code generation for interprocess communication. This has been left out of scope.

8.5 Discussion

We have shown different strategies to test DMDSL Models. It turned out that it is not easy to create good strategies. An analysis is necessary to point out deficiencies of strategies. We have seen that strategies can perform well for a certain kind of model, and less well on a different kind of model. For example, the SC3 strategy worked well on the chain issue in 8.2 but not so well on the issue in figure 8.9. One could try to fine-tune the strategy to a specific type of model, but still, it is hard to draw general conclusions about the testing strategy. The creation, fine-tuning, and analysis of strategies cost quite some time and effort. It is not easy to assess whether sufficient analysis is done. It is also hard to get a good notion of the types of traces it produces, and it requires research to investigate why the strategy induces certain behavior. It may depend on your testing goal which strategy is appropriate. Sometimes, a combination of strategies might actually be a good idea. There are uncountably many ideas possible in the creation of these testing strategies. When the language or test models are changed, the strategies and analysis should be reconsidered. At some point, this would open the debate to which extent MBT is still contributing to a testing procedure that is less labor-intensive than writing manual unit tests.

We have seen that MBT was helpful in pointing out some of the issues that happen in the translation process. The bugs discovered happened mostly when the pre-conditions of the Action calls were not satisfied. We, therefore, finished by testing the test models with the State

Coverage 3.2 strategy to see if the generated code does conform to the specification and if the preconditions of the action calls are satisfied. All experiments passed. While this is not a proof of the absence of bugs, it does provide us with more confidence that the code generator works correctly under this assumption.

Chapter 9

Conclusions and Future Work

We will now conclude our work by answering the research questions, and then discuss future work.

9.1 Answers to Research Questions

RQ1. *How is Model-Based Testing applied to improve the testing of code generators?* Based on the discussion of existing techniques, MBT can be considered an appealing strategy in the testing of code generators. We have seen that MBT is able to generate testcases and their oracles automatically. The techniques of MBT have the potential for an extensive testing procedure. Not only in the generation of test models, but also in testing whether a test model is correctly translated, MBT can be applied. The latter we have shown extensively in this work in the context of the DMDSL language. By using MBT we tested whether generated code of a test model behaves according to the semantics of the model. The architecture of applying MBT in this context was visualized and addressed in chapter 4.

RQ2. *How to assess which Model-Based Testing tool is suitable for a code generator?* We have proposed a practical procedure in the selection of an MBT tool. The procedure involved identifying the modeling paradigm of the test models, identifying which formal notations exist for this paradigm, and researching which formalism captures the dynamic semantics of the test models in the best manner. It also involved the investigation of the underlying theory and notation more carefully to identify whether it is possible to capture the dynamic semantics on the desired abstraction level. The step involved encoding a small model in the specification language of Model-Based Testing tools that rely on the selected formalism, to see whether the specification language is sufficiently expressive and whether the Model-Based Testing tool offered sufficient features. Furthermore, experimentation was needed to get a notion if the tool is powerful enough to explore a test model.

RQ2.1 *Which Model-Based Testing tool is suitable for the problem setting?* We have identified that in the context of the DMDSL language, PyModel is an MBT tool that can be used to test the DMDSL code generator. This tool allows encoding the model in Python. Since, some of the DMDSL actions have quite some complexity in their execution semantics, it turned out that expressing these in the Python language was very convenient. Furthermore, PyModel allowed for specifying state-dependent finite domains, which was very useful in our context where sensible arguments for the operations are dependent on the state variables. We also considered the Axini and TorXakis tools. Both of these tools relied on the LTS/STS formalism which is a transition-based paradigm. We saw that using locations did not contribute in specifying DMDSL models in a sensible manner. Furthermore, we figured that Axini was not sufficiently expressive to encode DMDSL models properly. We saw that TorXakis was sufficiently expressive, however due to the underlying programming language, some of the operations could not

be encoded in an efficient manner. Since a constraint solver runs in the background, this put a burden on the tool that made exploration of the translated DMDSL models in practice not feasible.

RQ3 *How to apply a given Model-Based Testing tool to a code generator?* To use PyModel to test whether test models are correctly translated, we needed to to be able to express these test models as a PyModel Model.

RQ3.1 *How to make test models compatible with a Model-Based Testing tool?* We answered this question by expressing how the AST constructs of DMDSL and dynamic semantics of a DMDSL test model can be translated into a specification language supported by an MBT tool, the PyModel tool in particular. A model-to-text transformation from DMDSL models to a PyModel specification has been written so that this can be done automatically.

To continue with answering RQ3, we used PyModel to generate test sequences using the PyModel model of a DMDSL test model as input. To manipulate how PyModel generates the test sequences, we created different testing strategies. We saw how these could be written in such a way that the tool applies Create operations on all entities, and tries to Add instances of each Entity type to the repository. If the test sequence manages to reach both goals, this gave some guarantee that the DMDSL model is explored until sufficient depth.

RQ3.2 *How to create an adapter that can apply abstract test cases to generated code?* To apply these abstract test cases on generated code, we created an adaptor for DMDSL code. By generating the adaptor for the provided DMDSL test model, we managed to connect the adaptor to the correct interfaces, making the adaptor able to translate abstract actions to concrete actions. Furthermore, the adaptor related concrete output messages to corresponding abstract output actions. Using the adaptor, the MBT tool PyModel was able to apply its abstract test cases on the IUT, and was able to decide on a verdict of the test cases

RQ4 *What are the benefits and drawbacks of applying Model-Based Testing on a code generator in an industrial context?* We can now answer this question by listing the benefits and drawbacks encountered:

+ Forces defining semantics explicitly: The process of applying MBT started off by discussing the semantics of the DMDSL language. It turned out that it was sometimes rather hard to express what aspects of a DMDSL model are supposed to mean, and even misconceptions among employees existed in that regard. The fact that applying MBT requires formalizing the semantics can be rather beneficial. We have seen in section 8.4.1 that many of the identified issues are actually a result of not enforcing the constraints at runtime. If the semantics were properly defined from the beginning, and MBT would have been applied, these could not have been unwillingly neglected. Before the software would have gone into production, the MBT tool would have pointed this out. Implementing these constraint checks afterwards is not easy. In fact, users of the generated code may in fact now rely on the fact that multiplicity checking is not enforced. Hence, if it would be changed post-factum, it might break code of the users. This problem is in fact an argument for applying MBT straight from the beginning of the implementation of a code generator.

- Translating models to a different specification language is hard: We observed that expressing the semantics of a DMDSL model required thoughts on which parts of the semantics of these models should be covered in the translation, and how to express them in a different specification language. In the context of PyModel, it even required creativity on the finitization of the domains of the actions.

+ Allows for very extensive testing: We observed that if the MBT architecture has been created, it is possible to generate test cases for arbitrary test models without any mental effort. Furthermore, it is possible to generate a large number of tests and their oracles in a short time period.

- Guiding an MBT tool in generating test sequences is hard: We observed that to explore DMDSL models properly, in order to generate interesting test cases, a specialized strategy needed to be created. We saw how difficult it was to come up with strategies, and how drawing general conclusions turned out to be rather hard. A strategy may perform well on a certain kind of model but perform not so well on a different kind of model. The creation and analysis of strategies cost some real effort. If the test strategies are not general it might be the case that when new test models are created or the language is changed, the strategies need to be adapted and analysis should again take place. If the strategies are really general, the test sequences produced might be of less interests.

9.2 Future Directions and Conclusion

This thesis showed a structural approach to testing an industrial code generator using MBT. The aspect that is not researched in this work is the generation of test models and integration in the testing procedure. Currently the creation of test models requires creativity from the tester. It is hard to manually get a good coverage of the modeling language, so that all transformation steps of the code generator are properly tested. An interesting approach is to take Model-Based Testing one step further and investigate the automatic generation of test models using the meta-model of the input language. There has already been research in this area, but it would be interesting to integrate this technique in the proposed MBT approach.

In the context of the DMDSL language, it is of interest to extend the formal semantics so that domain interfaces are addressed in the formal semantics. Code generation using multiple domain interfaces is not extensively tested by Capgemini. During regular meetings with Wilbert Alberts, Niels Brouwers, and Ivan Kurtev, it has become clear that it is quite easy to come up with DMDSL models, having multiple domain interfaces, of which it is actually unclear what the corresponding behavior should be. Sometimes, the generated code was even uncompilable. The fact that some obscure models lead to uncompilable code shows that also in the context of the DMDSL language, the automatic generation of test models is an important research field.

In the work of Derasari, the dynamic semantics of DMDSL are checked for preserving Repository Consistency using the Alloy tool. The fact that Alloy is not able to provide a counterexample does not prove that the semantics are sound with respect to model consistency. Alloy relies on the *small-scope hypothesis*. It would be of interest to mathematically prove that the semantics are sound with respect to model consistency. By proving that model consistency is respected for all possible initial states (Base), and showing that for an arbitrary state that respects model consistency, all outgoing transitions lead to a state that preserves model consistency (Step), one would potentially be able to prove inductively that the semantics are sound w.r.t. model consistency.

We have seen that MBT could be a valuable tool to test code generators. Even when applying MBT post-factum we managed to find some issues for a code generator that is mature. We expect that when MBT is applied directly from the beginning of the development process, it will be a very valuable tool in preventing bugs from an early stage. Further research is required to determine whether the costs of applying MBT is economically beneficial. Analysis needs to be done on the time employees of Capgemini spend on the testing process, and how much time is needed to apply MBT. Furthermore, research is required to find out what bugs could have been prevented if MBT was applied from the beginning, by for example testing an earlier version of the tool. Research could be done to determine what the ‘cost’ of these bugs are to answer the question whether applying MBT is sensible from an economic perspective. This is not an easy analysis. Answering this question is out of scope for this thesis, but is of interest for a further study to better understand the benefits and drawbacks of testing a code generator in an industrial context.

Bibliography

[1] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the "small scope hypothesis". 10 2002. 13

[2] Kent Beck. *Test Driven Development. By Example (Addison-Wesley Signature)*. Addison-Wesley Longman, Amsterdam, 2002. 4

[3] Axel Belinfante. Jtorx: exploring model-based testing, 9 2014. 38

[4] Jan Olaf Blech and Sabine Glesner. A formal correctness proof for code generation from ssa form in isabelle/hol. *Informatik 2004, Informatik verbindet, Band 2, Beiträge der 34. Jahrestagung der Gesellschaft für Informatik eV (GI)*, 2004. 6

[5] Tom Britton, Lisa Jeng, Graham Carver, Tomer Katzenellenbogen, and Paul Cheak. Reversible debugging software "quantify the time and cost saved using reversible debuggers", 11 2020. 1

[6] Frederick P. Brooks. *The mythical man-month – Essays on Software-Engineering*. Addison-Wesley, 1975. 1

[7] Zijun Chen, Wilbert Alberts, and Ivan Kurtev. Testing code generators: a case study on applying use, efinder and tracts in practice. In *STAF Workshops*, 2021. 22

[8] Raj Derasari. Adding formal specifications to a legacy code generator, 11 2021. 12, 34

[9] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. Test generation based on symbolic specifications. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing*, pages 1–15, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. 34, 38

[10] Mark Frenken. Code generation and model-based testing in context of oil, 2019. 5

[11] Yuri Gurevich. Logic and the challenge of computer science. 01 1988. 28

[12] Fazilat Hojaji, Tanja Mayerhofer, Bahman Zamani, Abdelwahab Hamou-Lhadj, and Erwan Bousse. Model execution tracing: a systematic mapping study. *Software and Systems Modeling*, 18:3461–3485, 12 2019. 5

[13] Papadopoulos Ioannis. Testing code generators against definitional interpreters, 2018. 5

[14] Daniel Jackson. Alloy: A language and tool for exploring software designs. *Commun. ACM*, 62(9):6676, aug 2019. 12

[15] Jonathan. Jacky. *Model-based software testing and analysis with C#*. Cambridge University Press, 2008. 28, 47, 49

[16] Jonathan Jacky. Pymodel: Model-based testing in python. pages 48–52, 01 2011. 28, 47, 49

[17] Sven Jörges and Bernhard Steffen. Back-to-back testing of model-based code generators. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 425–444, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. 5

[18] Willibald Krenn, Rupert Schlick, Stefan Tiran, Bernhard K. Aichernig, Elisabeth Jöbstl, and Harald Brandl. Momut::uml model-based mutation testing for uml. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–8, 2015. 28

[19] Philippe Lalanda. Shared repository pattern. In *Proc. 5th Annual Conference on the Pattern Languages of Programs*. Citeseer, 1998. 9, 22

[20] Bruno Legeard Mark Utting. *Practical Model-Based Testing: A Tools Approach*. 1 edition, 2006. 6, 7, 27, 28, 29, 37, 64, 73, 74

[21] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, Heidelberg, 1982. 30

[22] Alexandre Petrenko. Checking experiments for symbolic input/output finite state machines. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 229–237, 2016. 28

[23] A. C. Rajeev, Prahladavaradan Sampath, K. C. Shashidhar, and S. Ramesh. Cogente: A tool for code generator testing. pages 349–350, 2010. 8

[24] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, page 717740, New York, NY, USA, 1972. Association for Computing Machinery. 5

[25] Muhammad Shafique and Yvan Labiche. A systematic review of model based testing tool support. 04 2010. 25

[26] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *J. Mach. Learn. Res.*, 12:2539–2561, 2011. 88

[27] Thomas Stahl and Markus Völter. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, Chichester, UK, 2006. 4

[28] I. Stürmer and M. Conrad. Test suite design for code generation tools. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 286–290, 2003. 8

[29] Ingo Stürmer and Mirko Conrad. Code generator testing in practice. code generator testing in practice, 2004. 6, 8

[30] Ingo Stürmer, Mirko Conrad, Heiko Doerr, and Peter Pepper. Systematic testing of model-based code generators. *IEEE Transactions on Software Engineering*, 33(9):622–634, 2007. 6

[31] G.J. Tretmans and Hendrik Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering*, pages 31–43, December 2003. null ; Conference date: 11-12-2003 Through 12-12-2003. 28, 38

[32] GJ Tretmans and Piërre van de Laar. Model-based testing with torxakis: The mysteries of dropbox revisited. 2019. 28, 38

[33] Jan Tretmans. *Model Based Testing with Labelled Transition Systems*, pages 1–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. 28, 31, 33

[34] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer, 2007. 28, 47, 49

[35] MA Vouk. Back-to-back testing. *Information and Software Technology*, 32(1):34–45, 1990. Special Issue on Software Quality Assurance. 5

[36] Tobias Werth, Alexander Dreweke, Marc Wrlein, Ingrid Fischer, and Michael Philippsen. Dagma: Mining directed acyclic graphs. *First publ. in: IADIS European Conference on Data Mining 2008, Amsterdam, The Netherlands, 24. - 26. July 2008. IADIS Press, 2008, pp. 11-18*, 01 2008. 83

Appendix A

TorXakis

A.1 Model Translation

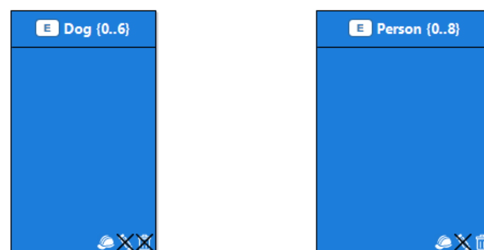


Figure A.1: Simple Model used for the TorXakis transformation in listing A.1

```
CHANDEF Channels ::=
    CreateInput :: CreateOperation;
    ReadInput :: ReadOperation;
    DeleteInput :: DeleteOperation;
    AddInput :: AddOperation;
    Output:: OutputMessage
ENDDEF


TYPEDEF
  Multiplicity ::= Tuple {min :: Int; max :: Int}
ENDDEF


TYPEDEF
  AssociationEndProperty ::= AssociationEndProp {cascade :: Bool;
     multiplicity :: Multiplicity}
ENDDEF

TYPEDEF
  Association ::= Assoc {source :: Entity; associationName ::
     ValidAssociationName; target :: Entity; sproperty ::
```

```
    AssociationEndProperty; tproperty :: AssociationEndProperty}
ENDDEF

TYPEDEF
  AssociationList ::= Nil | Cons { hd :: Association; tl ::
    AssociationList}
ENDDEF

TYPEDEF
  InstanceList ::= Nil | Cons { hd :: Instance; tl :: InstanceList}
ENDDEF

TYPEDEF
  LinkList ::= Nil | Cons { hd :: Link; tl :: LinkList}
ENDDEF

TYPEDEF
  EntityList ::= Nil | Cons { hd :: Entity; tl :: EntityList}
ENDDEF


TYPEDEF
  Entity ::= Ent {enityName :: ValidEntityName; isConstructable ::
    Bool; isMutable :: Bool; isDeletable :: Bool; multiplicity ::
    Multiplicity}
ENDDEF

TYPEDEF
  Link ::= Triple { source :: Instance; associationName ::
    ValidAssociationName; target :: Instance}
ENDDEF

TYPEDEF
  Instance ::= Instance { entityName :: ValidEntityName; id :: Int}
ENDDEF

TYPEDEF
  CreateOperation ::= Create {entityName :: ValidEntityName; id ::
    Int}
ENDDEF

TYPEDEF
  ReadOperation ::= Read{id :: Int}
ENDDEF


TYPEDEF
  DeleteOperation ::= Delete {id :: Int}
ENDDEF

TYPEDEF
  AddOperation ::= Add {instance :: Instance}
ENDDEF
```

```
TYPEDEF
  LinkTarget ::= LinkTarget{association :: Association; target ::
     Instance}
ENDDEF

TYPEDEF
  LinkTargetList ::= Nil | Cons{hd :: LinkTarget; tl ::
     LinkTargetList}
ENDDEF



TYPEDEF
  OutputMessage ::= Success | Entity_MultiplicityMaximum |
     Entity_MultiplicityMinimum | Entity_Unconstructable |
     Entity_Immutable | Entity_Undestructable |
     Entity_UnexpectedAssociation | Entity_MissingAssociation |
     Association_SourceMaximum | Association_TargetMaximum |
     Association_TargetMinimum | Link_TargetNotInRepository |
     Instance_NotInRepository | Instance_AlreadyInRepository
ENDDEF


TYPEDEF
  ValidEntityName ::= Dog | Person
ENDDEF

TYPEDEF
  ValidAssociationName ::= Aowner
ENDDEF




CONSTDEF
  dog :: Entity ::= Ent(Dog, True, False, False, Tuple(0,6));
  person :: Entity ::= Ent(Person, True, False, True, Tuple(0,8))
ENDDEF


CONSTDEF
  associationsGlobal :: AssociationList ::= Cons(Assoc(
     getEntityByEntityName(Dog), Aowner, getEntityByEntityName(
     Person), AssociationEndProp(True, Tuple(0,1)),
     AssociationEndProp(False, Tuple(1,1))), Nil)
ENDDEF


STAUTDEF DmdslModel [CreateInput :: CreateOperation;
                      DeleteInput :: DeleteOperation;
                      AddInput :: AddOperation;
                      ReadInput :: ReadOperation;
```

```
                  Output :: OutputMessage]() ::=
STATE initialState, outputState, Entity_Unconstructable_State,
   Entity_MultiplicityMaximum_State,
Instance_NotInRepository_State_Delete,
   Instance_NotInRepository_State_Read,
   Entity_Undestructable_State,
Instance_AlreadyInRepository_State,
   Entity_MultiplicityMinimum_State,
successfulCreationState, successfulDeletionState,
   successfulAdditionState, successfulReadState

VAR   instances :: InstanceList; repo :: InstanceList; links ::
    LinkList; highestId :: Int

INIT  initialState{ instances := Nil; repo := Nil; links := Nil
   ; highestId := 0 }
TRANS


{-BEGIN: CREATION-}
  {-ERROR CREATION -}
  initialState ->
  CreateInput ? createInput
  [[isEntityUnConstructableError(getEntityByEntityName(
     entityName(createInput))) /\ (id(createInput) == highestId
     )]]
  {  }
  -> Entity_Unconstructable_State


  initialState ->
  CreateInput ? createInput
  [[isEntityMaximumMultiplicityError(getEntityByEntityName(
     entityName(createInput)), instances) /\ (id(createInput)
     == highestId)]]
  {  }
  -> Entity_MultiplicityMaximum_State


  {-SUCCESFUL CREATION -}
  initialState ->
  CreateInput ? createInput
  [[
  not( isEntityMaximumMultiplicityError(getEntityByEntityName(
     entityName(createInput)), instances) )
  /\
  not( isEntityUnConstructableError(getEntityByEntityName(
     entityName(createInput))) )
  /\ (id(createInput) == highestId)
  ]]
  { instances := Cons(Instance(entityName(createInput),
     highestId), instances); highestId := highestId + 1 }
  -> successfulCreationState
{-END: CREATION-}
```

```
{-BEGIN: DELETION-}
  {-ERROR DELETION -}

  initialState ->
  DeleteInput ? deleteInput
  [[
  isEntityDeleteOnUndestructableError(id(deleteInput),
     instances) /\ isIdInInstanceList(id(deleteInput),
     instances)
  ]]
  {  }
    -> Entity_Undestructable_State


  initialState ->
  DeleteInput ? deleteInput
  [[
  isInstanceNotInRepositoryError(id(deleteInput), repo)
  ]]
  {  }
    -> Instance_NotInRepository_State_Delete


  initialState ->
  DeleteInput ? deleteInput
  [[
  isEntityMultiplicityMinimumError(id(deleteInput), repo) /\
     isIdInInstanceList(id(deleteInput), repo)
  ]]
  {  }
    -> Entity_MultiplicityMinimum_State



  {-SUCCESFUL DELETION -}
  initialState ->
  DeleteInput ? deleteInput
  [[
  not( isInstanceNotInRepositoryError(id(deleteInput), repo) )
     /\
  not( isEntityDeleteOnUndestructableError(id(deleteInput),
     instances) /\ isIdInInstanceList(id(deleteInput),
     instances) ) /\
  not( isEntityMultiplicityMinimumError(id(deleteInput),
     instances) /\ isIdInInstanceList(id(deleteInput),
     instances) )
  ]]
  { repo := removeInstanceFromInstanceList(id(deleteInput),
     repo);
```

```
      instances := removeInstanceFromInstanceList(id(deleteInput)
         , instances) }
    -> successfulDeletionState
  {-END: DELETION-}



  {-BEGIN: ADDITION-}
    {- ERROR ADDition-}

    initialState ->
    AddInput ? addInput
    [[
    isInstanceAlreadyInRepositoryError(instance(addInput), repo)
    ]]
    { }
    -> Instance_AlreadyInRepository_State


    {- SUCCESFUL ADD-}

    initialState ->
    AddInput ? addInput
    [[
    instanceInList(instance(addInput), instances)
    /\
    not(isInstanceAlreadyInRepositoryError(instance(addInput),
       repo))
    ]]
    { repo := Cons(instance(addInput),repo)}
    -> successfulAdditionState
  {-END: ADDITION-}


  {-BEGIN: READ-}
    {-ERROR READ -}
    initialState ->
    ReadInput ? readInput
    [[
    isInstanceOfIdDoesNotExistError(id(readInput), repo)
    ]]
    { }
    -> Instance_NotInRepository_State_Read

    {-SUCCESFUL READ -}
    initialState ->
    ReadInput ? readInput
    [[
    not( isInstanceOfIdDoesNotExistError(id(readInput), repo) )
    ]]
    { }
    -> successfulReadState
```

```
    {-END: READ-}




    Entity_Unconstructable_State -> Output ! Entity_Unconstructable
        -> initialState
    Entity_MultiplicityMaximum_State -> Output !
       Entity_MultiplicityMaximum -> initialState
    Entity_MultiplicityMinimum_State -> Output !
       Entity_MultiplicityMinimum -> initialState
    Instance_NotInRepository_State_Delete -> Output !
       Instance_NotInRepository -> initialState
    Entity_Undestructable_State -> Output ! Entity_Undestructable
       -> initialState
    Instance_AlreadyInRepository_State -> Output !
       Instance_AlreadyInRepository -> initialState
    Instance_NotInRepository_State_Read -> Output !
       Instance_NotInRepository -> initialState


    {- SUCCESS STATES -}
    successfulCreationState -> Output ! Success -> initialState
    successfulDeletionState -> Output ! Success -> initialState
    successfulAdditionState -> Output ! Success -> initialState
    successfulReadState -> Output ! Success -> initialState


ENDDEF


MODELDEF Model ::=
    CHAN IN   CreateInput,  DeleteInput, AddInput, ReadInput
    CHAN OUT  Output
    BEHAVIOUR DmdslModel [CreateInput, DeleteInput, AddInput,
       ReadInput, Output]()
ENDDEF




{- CREATE ERRORS -}

FUNCDEF isEntityUnConstructableError(entity :: Entity) :: Bool
::=
  IF isConstructable(entity) THEN False
  ELSE True
  FI
ENDDEF

FUNCDEF isEntityMaximumMultiplicityError(entity :: Entity;
   instances :: InstanceList) :: Bool
::=
```

```
  IF numberOfTimesEntityInList(entity, instances) >= max(
     multiplicity(entity)) THEN True
  ELSE False
  FI
ENDDEF



{- DELETE ERRORS -}

FUNCDEF isEntityDeleteOnUndestructableError(id :: Int; instances ::
    InstanceList) :: Bool
::=
  IF instances == Nil THEN False
  ELSE
    IF id(hd(instances)) == id THEN not(isConstructable(
       getEntityByEntityName(entityName(hd(instances)))))
    ELSE isEntityDeleteOnUndestructableError(id, tl(instances))
    FI
  FI
ENDDEF

FUNCDEF isInstanceNotInRepositoryError(id :: Int; repo ::
   InstanceList) :: Bool
::=
  IF isIdInInstanceList(id, repo) THEN False
  ELSE True
  FI
ENDDEF

FUNCDEF isEntityMultiplicityMinimumError(id :: Int; instances ::
   InstanceList) :: Bool
::=
 isEntityMultiplicityMinimumErrorAux(id, instances, instances)
ENDDEF



{- ADD ERRORS -}

FUNCDEF isInstanceAlreadyInRepositoryError(instance :: Instance;
   repo :: InstanceList) :: Bool
::=
  IF instanceInList(instance, repo) THEN True
  ELSE False
  FI
ENDDEF


{- READ ERRORS -}

FUNCDEF isInstanceOfIdDoesNotExistError(identifier :: Int; repo ::
   InstanceList) :: Bool
::=
```

```
  IF repo == Nil THEN True
  ELSE
    IF id(hd(repo)) == identifier THEN False
    ELSE isInstanceOfIdDoesNotExistError(identifier, tl(repo))
    FI
  FI
ENDDEF




{- AUXILIARY FUNCTIONS -}

{-
FUNCDEF getEntityByEntityName(entityName :: ValidEntityName) ::
   Entity
::=
  IF  entityName == Dog THEN Ent(Dog, True, False, False, Tuple
     (0,1))
  ELSE
    IF entityName == Person THEN Ent(Person, True, False, True,
       Tuple(0,2))
    ELSE Ent(Person, True, False, True, Tuple(0,2))
    FI
  FI
ENDDEF
-}

FUNCDEF getEntityByEntityName(entityName :: ValidEntityName) ::
   Entity
::=
  IF  entityName == Dog THEN dog
  ELSE person
  FI
ENDDEF



FUNCDEF numberOfTimesEntityInList(entity :: Entity; instanceList ::
     InstanceList) :: Int
::=
  IF instanceList == Nil THEN 0
  ELSE
    IF getEntityByEntityName(entityName(hd(instanceList))) ==
       entity THEN 1 + numberOfTimesEntityInList(entity, tl(
       instanceList))
    ELSE numberOfTimesEntityInList(entity, tl(instanceList))
    FI
  FI
ENDDEF
```

```
FUNCDEF isIdInInstanceList(id :: Int; instances :: InstanceList) ::
     Bool
::=
  IF instances == Nil THEN False
  ELSE
    IF id(hd(instances)) == id THEN True
    ELSE isIdInInstanceList(id, tl(instances))
    FI
  FI
ENDDEF





FUNCDEF removeInstanceFromInstanceList(id :: Int; instanceList ::
   InstanceList) :: InstanceList
::=
        IF instanceList == Nil THEN Nil
        ELSE
                IF id(hd(instanceList)) == id THEN
                   removeInstanceFromInstanceList(id, tl(
                   instanceList))
                ELSE Cons(hd(instanceList),
                   removeInstanceFromInstanceList(id, tl(
                   instanceList)))
                FI
        FI
ENDDEF

FUNCDEF instanceWithIdInInstanceList(identifier :: Int; instances
   :: InstanceList) :: Bool
::=
  IF instances == Nil THEN False
  ELSE
    IF id(hd(instances)) == identifier THEN True
    ELSE instanceWithIdInInstanceList(identifier, tl(instances))
    FI
  FI
ENDDEF



FUNCDEF entityUndestructable(entity :: Entity) :: Bool
  ::=
    IF not(isDeletable(entity)) THEN True
    ELSE False
    FI
ENDDEF


FUNCDEF instanceInList(instance :: Instance; instanceList ::
   InstanceList) :: Bool
  ::=
```

```
    IF instanceList == Nil THEN False
    ELSE
      IF hd(instanceList) == instance THEN True
      ELSE
        instanceInList(instance, tl(instanceList))
      FI
    FI
ENDDEF

FUNCDEF isEntityMultiplicityMinimumErrorAux(id :: Int;
   instanceIterator :: InstanceList; instances :: InstanceList) ::
   Bool
::=
  IF instanceIterator == Nil THEN False
  ELSE
    IF id(hd(instanceIterator)) == id THEN (
       numberOfTimesEntityInList(getEntityByEntityName(entityName(
       hd(instanceIterator))), instances)
                                        == min(multiplicity(
                                           getEntityByEntityName(
                                           entityName(hd(
                                           instanceIterator))))) )
    ELSE isEntityMultiplicityMinimumErrorAux(id, tl(
       instanceIterator), instances)
    FI
  FI
ENDDEF
```

Listing A.1: Simple TorXakis Model

A.2 Source cascade deletion

```
FUNCDEF ComputeDeletionSet(instances :: InstanceList; links ::
   LinkList; returnList :: InstanceList) :: InstanceList
::=
IF instances == Nil THEN returnList
ELSE
  concatenate(concatenate(returnList,SCDPredecessorOfInstances(
     instances, links, Nil)),
  ComputeDeletionSet(SCDPredecessorOfInstances(instances, links,
     Nil), links, Nil))
FI
ENDDEF

FUNCDEF SCDPredecessorOfInstances(instances :: InstanceList; links
   :: LinkList; returnList :: InstanceList) :: InstanceList
::=
  IF instances == Nil THEN returnList
  ELSE
    concatenate(concatenate(returnList, SCDPredecessorOfInstance(hd
       (instances), links, Nil)),  SCDPredecessorOfInstances(tl(
       instances), links, Nil))
```

```
  FI
ENDDEF

FUNCDEF SCDPredecessorOfInstance(instance :: Instance; links ::
    LinkList; returnList :: InstanceList) :: InstanceList
::=
  IF links == Nil THEN returnList
  ELSE
    IF target(hd(links)) == instance /\ ( cascade(sproperty(
        getAssociationByAssociationName(associationName(hd(links))))
        ) == True )
      THEN SCDPredecessorOfInstance(instance, tl(links), Cons(
          source(hd(links)), returnList))
    ELSE SCDPredecessorOfInstance(instance, tl(links), returnList)
    FI
  FI
ENDDEF



FUNCDEF concatenate(instances1 :: InstanceList; instances2 ::
    InstanceList) :: InstanceList
::=
  IF instances2 == Nil THEN instances1
  ELSE
    concatenate(Cons(hd(instances2), instances1), tl(instances2))
  FI
ENDDEF
```

Listing A.2: Proof of concept for determining deletion set in source cascade deletion.

```
FUNCDEF UnionInstanceLists(firstInstanceList :: InstanceList;
    secondInstanceList :: InstanceList) :: InstanceList
::=
  IF secondInstanceList == Nil THEN firstInstanceList
  ELSE
    IF not(instanceInList(hd(secondInstanceList), firstInstanceList
        )) THEN UnionInstanceLists(Cons(hd(secondInstanceList),
        firstInstanceList), tl(secondInstanceList))
    ELSE UnionInstanceLists(firstInstanceList, tl(
        secondInstanceList))
    FI
  FI
ENDDEF

FUNCDEF instanceInList(instance :: Instance; instanceList ::
    InstanceList) :: Bool
  ::=
    IF instanceList == Nil THEN False
    ELSE
      IF hd(instanceList) == instance THEN True
      ELSE
        instanceInList(instance, tl(instanceList))
      FI
```

```
    FI
ENDDEF
```

Listing A.3: Proof of concept for union operator

Appendix B

Test Models

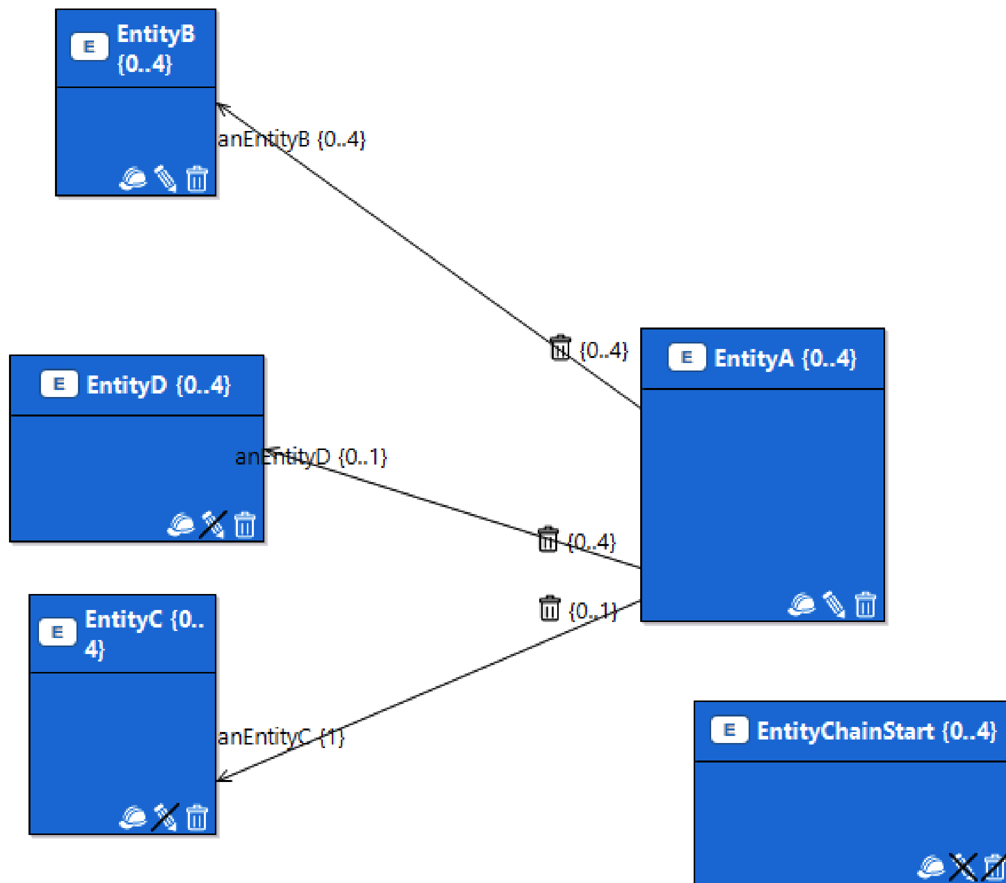


Figure B.1: The AME1437 test model.

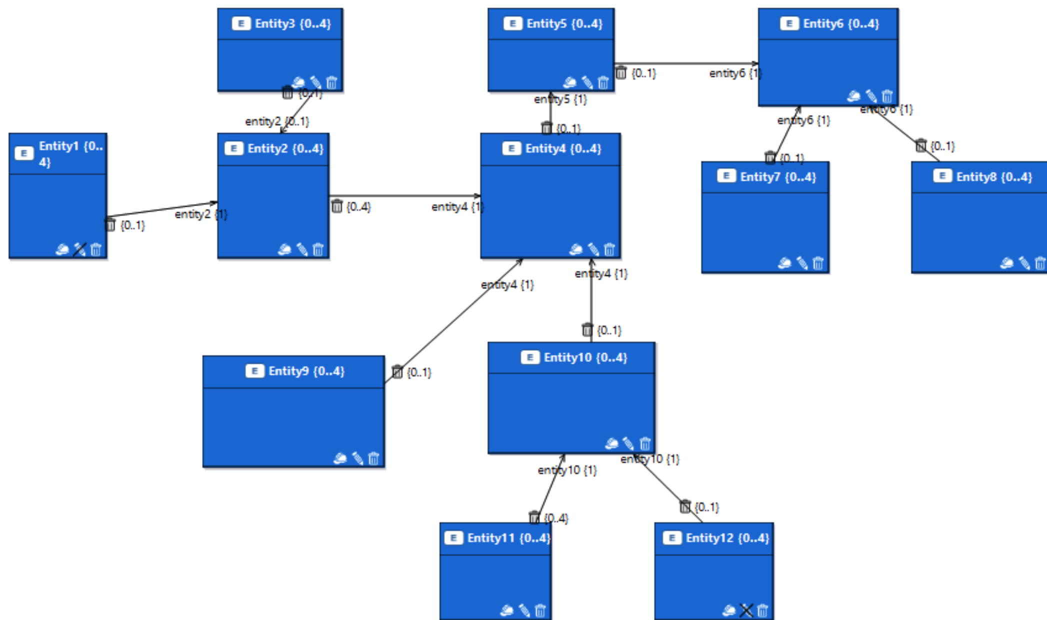


Figure B.2: The Bench test model.

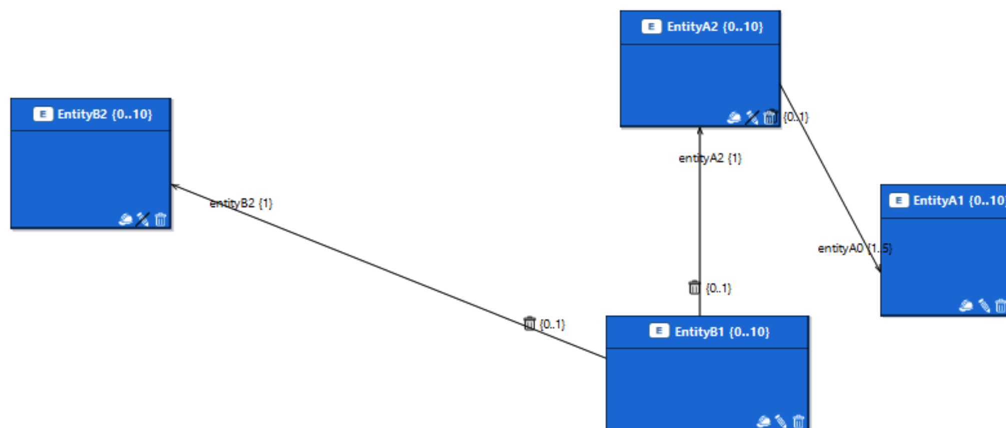


Figure B.3: The IPCDv2 test model.

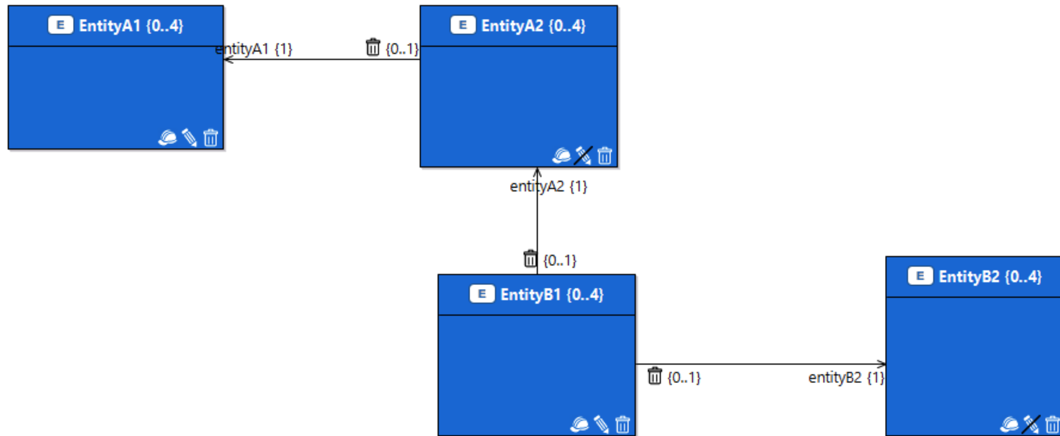


Figure B.4: The SIRE01 test model.

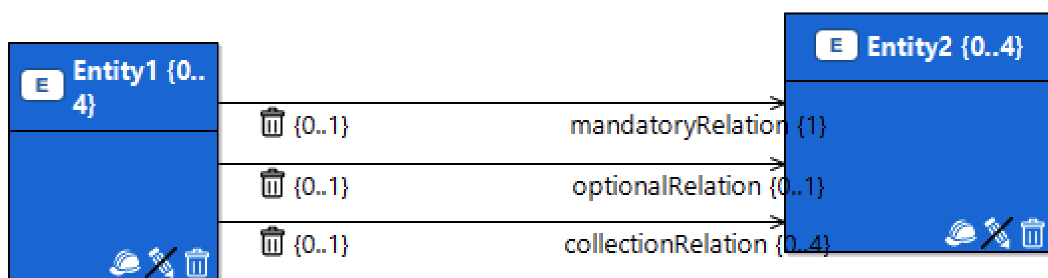


Figure B.5: The TestOptional test model.

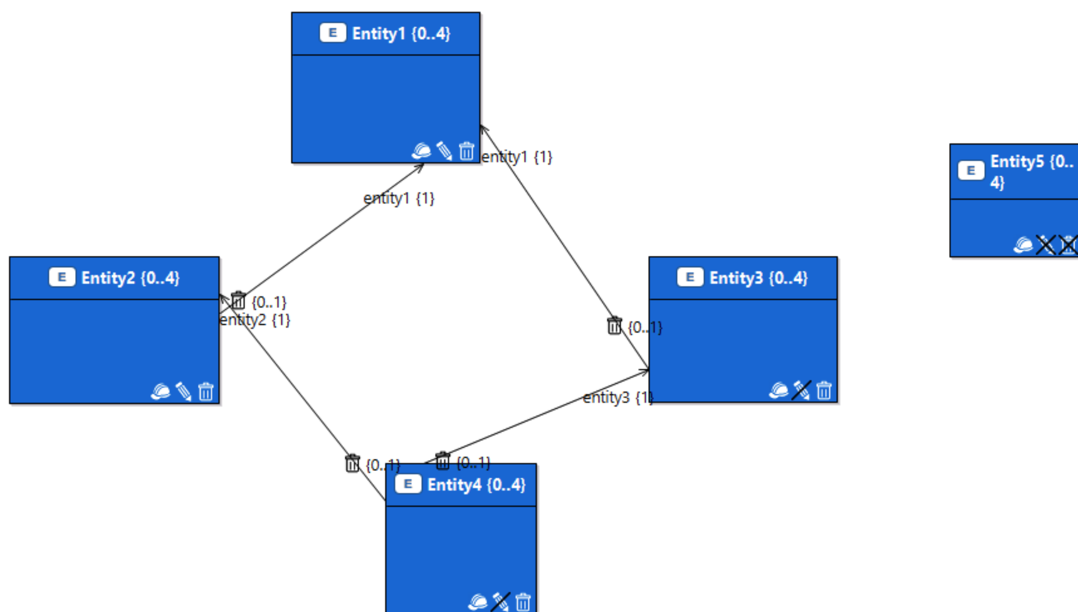


Figure B.6: The TM003 test model.

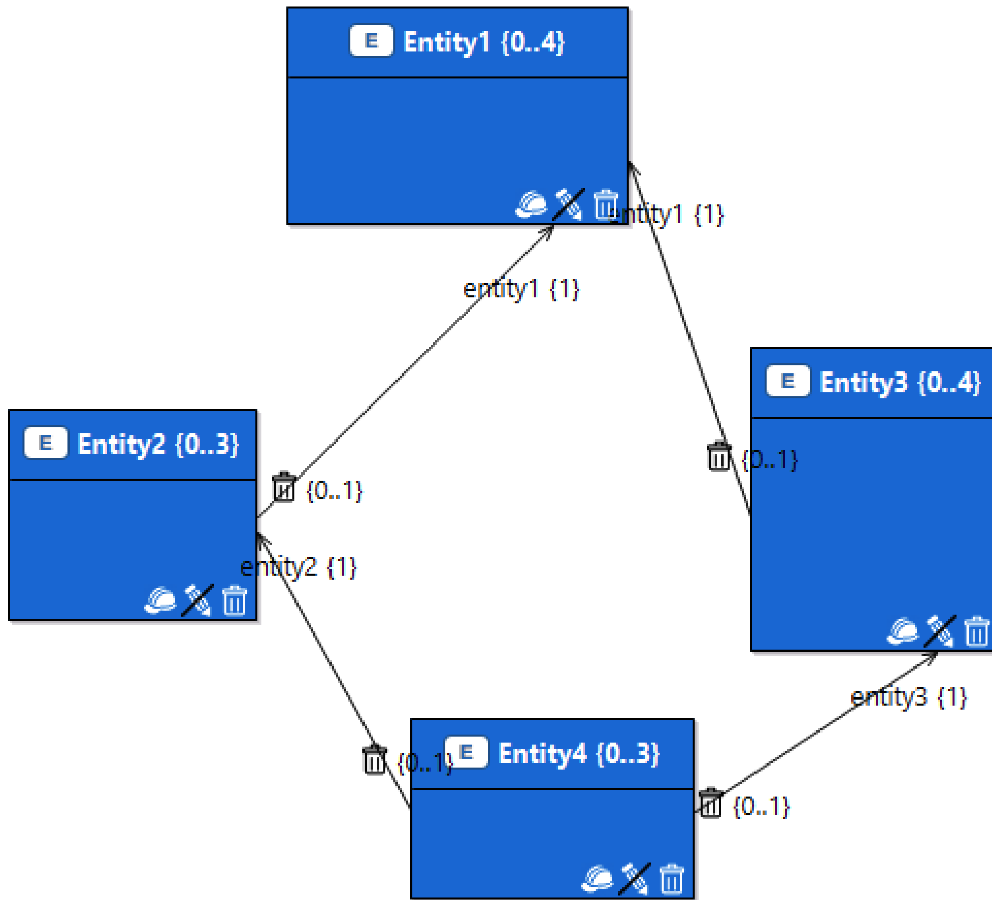


Figure B.7: The TM020 test model.

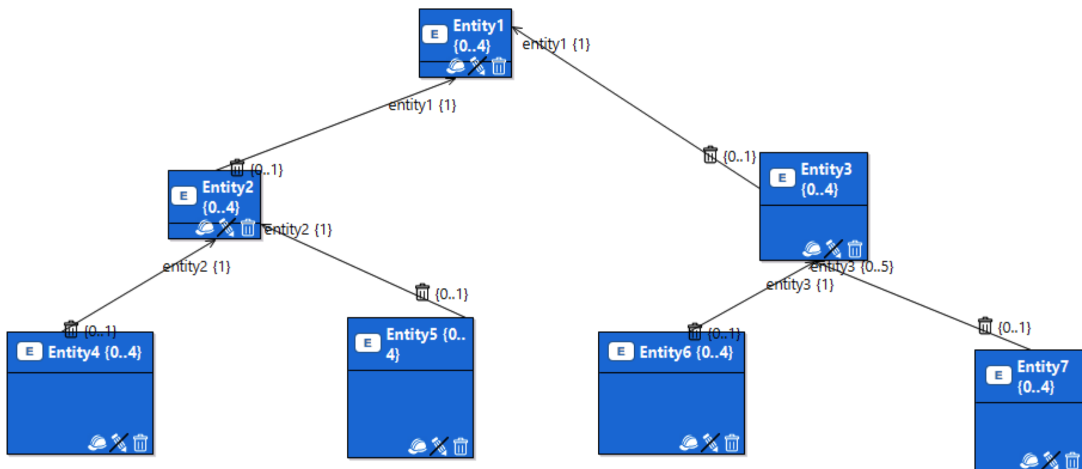


Figure B.8: The TM033 test model.

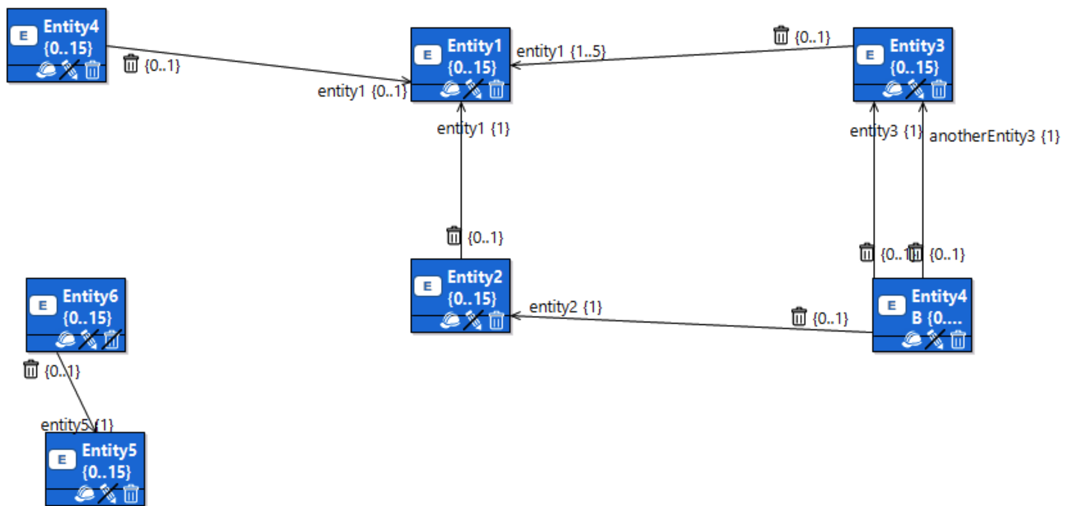


Figure B.9: The TM065 test model.

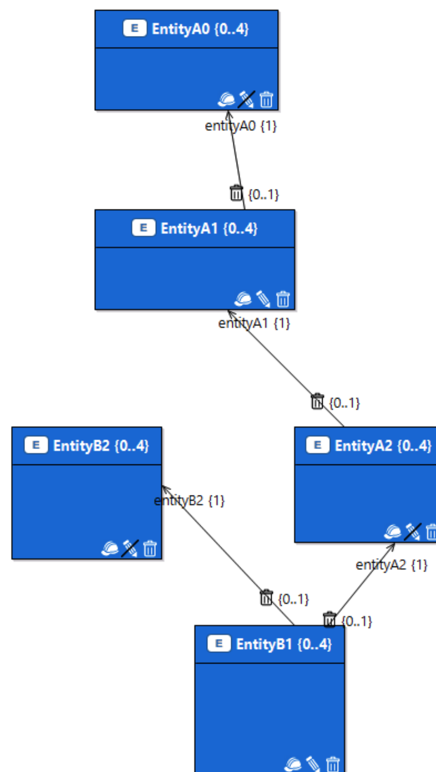


Figure B.10: The TM10001 test model.

Appendix C

Error Output

C.1 Read Output Messages

The reading of an instance using an identifier may fail. We create an error message for this, in line with the described semantics:

1. $\neg\exists_x[x \in s.REPO : x.identifier = identifier] \implies output =$ `Instance_NotInRepository`

```
def is_instance_not_in_repository_error(identifier):
    from Model_Translation import repo
    for instance in repo:
        if instance.get_id() == identifier:
            return False
    return True

```

C.2 Update Output Messages

Updating an Entity may fail. In the formal semantics it is described in what situations errors might occur, and what the corresponding error output is. We will translate these to checks for PyModel:

1. $instance.Entity.Mutability = \text{False} \implies output =$ `Entity_UpdateImmutableType`

```
def is_entity_update_immutabletype_error(instance):
    if not instance.Entity.is_mutable:
        return True
    return False

```

2. $\exists_l[l \in links : l.source = instance \wedge l.target \notin s.REPO] \implies output =$ `Link_TargetNotInRepository`

```
def is_link_target_not_in_repository_error(links):
    from Model_Translation import repo
    targets = {link.target_instance for link in links}
    if len(targets.intersection(repo)) < len(targets):
        return True
    return False

```

3. $\exists_a[a \in instance.Entity.OutgoingAssociations : |\{link \in Links \mid a = link.association\}| < a.TProperty.multiplicity.minimum] \implies output =$ `Association_TargetMinimum`

```
1  def is_association_target_minimum_error(instance, links):
2      out_associations = instance.Entity.get_outgoing_assocations()
3      for association in out_associations:
4          links_corresponding_to_association = [link for link in links if link.
       association
5  == association]
6          if len(links_corresponding_to_association) < association.
       target_association_property.multiplicity.min:
7              return True
8      return False
9
```

4. $\exists_a[a \in instance.Entity.OutgoingAssociations : |\{link \in Links \mid a = link.association\}| > a.TProperty.multiplicity.maximum] \implies output =$ `Association_TargetMaximum`

```
1   def is_association_target_maximum_error(instance, links):
2      out_associations = instance.Entity.get_outgoing_assocations()
3      for association in out_associations:
4          links_corresponding_to_association = [link for link in links if link.
       association
5  == association]
6          if len(links_corresponding_to_association) > association.
       target_association_property.multiplicity.max:
7              return True
8      return False
9
```

5. $\exists_{a,i}[a \in e.OutgoingAssociations \wedge i \in s.REPO \wedge i.Entity = a.target.Entity : |i.incomingLinks(a) \cup links(a,i)|] > a.SProperty.multiplicity.maximum \implies output =$ `Association_SourceMaximum`

```
1  def is_association_source_maximum_error(instance, new_links):
2      from Model_Translation import instances, links
3      out_associations = instance.Entity.get_outgoing_assocations()
4      links_that_remain = [link for link in links if link.source_instance !=
5  instance]
6
7      for association in out_associations:
8          for instance in instances:
9              number_of_existing_links_pointing_to_instance =
       __number_of_incoming_links_to_instance(instance,
10 association, links_that_remain)
11             number_of_new_links_pointing_to_instance =
       __number_of_incoming_links_to_instance(instance,
12 association, new_links)
13             if number_of_existing_links_pointing_to_instance +
       number_of_new_links_pointing_to_instance
14 \
15                     > association.source_association_property.multiplicity.max
       :
16                 return True
17         return False
```

C.3 Delete Output Messages

Delete Error Messages Deleting an Entity may fail. In the formal semantics it is described in what situations errors might occur, and what the corresponding error output is. Note that due to cascade deletion, a single deletion call, can result in many instances being deleted. Using a fixedpoint computation algorithm we can compute the set that should be deleted as a result of the deletion. We will translate these to checks for PyModel:

1. $instance \notin repo \implies output =$ `Instance_NotInRepository`

```
        instance = __get_instance_of_identifier_in_repo(identifier)
    if instance == None:
        return list_output({"Instance_NotInRepository"})

```

2. $\exists_y[y \in deletion_set : y.deletable = \text{False}] \implies output = \texttt{Entity_DeleteOnUndestructable}$

```
def is_entity_delete_on_undestructable_error(instances):
    for instance in instances:
        if not instance.Entity.is_deletable:
            return True
    return False

```

3. $|\{i \in s.REPO \mid i.entity = instance.Entity\} \setminus \{i \in deletion_set \mid i.Entity = instance.Entity\}| < instance.Entity.multiplicity.minimum \implies output = \texttt{Entity_MultiplicityMinimum}$

```
def is_entity_multiplicity_minimum_error(instances):
    from Model_Translation import repo
    type_to_instances_mapping = {}
    for instance in instances:
        if instance in repo:
            if instance.Entity in type_to_instances_mapping:
                type_to_instances_mapping[instance.Entity].add(instance)
            else:
                type_to_instances_mapping[instance.Entity] = {instance}

    for entity in type_to_instances_mapping:
        instance_of_entity_in_repository = set([])
        for instance in repo:
            if instance.Entity ==(entity):
                instance_of_entity_in_repository.add(instance)
        if len(instance_of_entity_in_repository) - len(
    type_to_instances_mapping[entity])
< entity.multiplicity.min:
            return True

```

C.4 Add Output Messages

Adding an instance to the repository may fail. In the formal semantics it is described in what situations errors might occur, and what the corresponding error output is.

1. $instance \in s.REPO \implies output = \texttt{Instance_AlreadyInRepository}$

```
def is_instance_already_in_repository_error(instance):
    from Model_Translation import repo
    for element in repo:
        if element == instance:
            return True
    return False
```

2. $\exists_{link}[link \in s.L : link.source = instance \land link.target \notin s.REPO] \implies output = \texttt{Link_TargetNotInRepository}$

```
def is_link_target_not_in_repository_error(instance):
    from Model_Translation import links, repo
    for link in links:
        if link.source_instance == instance and link.target_instance not
in repo:
            return True
    return False
```