# Evaluation of Model Transformation Testing in Practice

Chen, Zijun

*Award date:*
2020

Department of Mathematics and Computer Science
Software Engineering and Technology Group

# Evaluation of Model Transformation Testing in Practice

*Master Thesis*

Zijun Chen

Supervisors:
Mark van den Brand
Ivan Kurtev
Wilbert Alberts
Rob Posthumus

Eindhoven, September 2020

# Abstract

Model Driven Engineering (MDE) is a software development methodology that aims at increasing the productivity and quality of software development. MDE uses abstract models as primary artifacts to drive development. Software developers in MDE specify software systems in models. A model transformation chain can transform models into code. A model transformation chain consists of one or several model transformations. The correctness of a model transformation will significantly affect the quality of the generated software. In industry, the model transformation testing is a common method to detect faults in model transformations and to ensure the quality of the model transformation tool. However, the model transformation testing practice in the industry is facing multiple challenges demonstrated by the large workload of the processes and failures in the results of the transformations. Meanwhile, there are several approaches proposed in the literature that focus on model transformation testing. However, the suitability of these approaches is unclear. There is a knowledge gap between the industry and academia. The suitability of the approach proposed by the academics is unclear to the industry and the needs of the industry are unclear to academia. Therefore, we conducted this research to narrow the knowledge gap between the industry and academia.

To understand the challenges in testing model transformations in industry, we conducted interviews with testers in a project group called ASOME at Altran. From the interviews, we gained knowledge of the current model transformation testing process and the key needs of the testers at Altran. To search for suitable approaches in the literature, we conducted a literature review. We reviewed, analyzed, and selected the approaches that address the challenges of model transformation testing. We selected two types of approaches. The first one is to check the correctness of model transformations using formal specifications. The second one is dedicated to automatic model generation and equivalence partitioning. There are two tools for the first approach: TractsTool and Matching Table Builder. For the second approach, there are two tools: USE and Efinder. To evaluate the suitability of these tools, we proposed four criteria for suitability evaluation. We first conducted small case studies using these tools. During the small case studies, we identified the problems and limitations of the four tools. TractsTool, Matching Table Builder, and USE can not be used for industrial cases. Only Efinder showed positive results in the small case study and we further conducted several industrial case studies with Efinder. We discuss and evaluate the suitability of the four tools using the proposed criteria in the thesis. Moreover, we discuss the possible usage and suggestions for further improvements for the four tools.

# Acknowledgement

This master thesis is the last part of my Master's program Computer Science and Engineering at Department of Software Engineering and Technology, Eindhoven University of Technology. First of all, I would like to thank my supervisor, prof. Mark van den Brand for supervising this thesis project. I would like to thank my manager, Aad van Gerwen and dr. Ivan Kurtev at Altran for giving me the chance to work at Altran as an intern.
Second, I would like to thank dr. Ivan Kurtev, dr. Wilbert Alberts and Rob Posthumus for their valuable and comprehensive guidance during the project. They were very patient and thoughtful during the project. I appreciate their hard-working spirits, kindness, patience, and positivity.
Third, I would like to thank Koen Staal, Adrain Yankov, Vivek Vishal from the ASOME project group for patiently answering me questions and providing useful information. I also would like to thank other group members of the ASOME project group. Even though it was a short time working together in the same office, they created a friendly and kind atmosphere for me.
Last but not least, I would like to thank my boyfriend Jan Emrich for loving and caring for me during the coronavirus outbreak.

# Contents

**Contents** **vii**

**List of Figures** **xi**

**List of Tables** **xiii**

**Listings** **xv**

**1 Introduction** **1**

1.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Problem Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.3 Research Questions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.4 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.5 Thesis Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

**2 The study of the practice and the literature review** **5**

2.1 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.1.1 Model Transformation Development . . . . . . . . . . . . . . . . . 5

2.1.2 Model Transformation Testing . . . . . . . . . . . . . . . . . . . . 6

2.1.3 Test Quality Evaluation . . . . . . . . . . . . . . . . . . . . . . . . 6

2.1.3.1 Test Coverage . . . . . . . . . . . . . . . . . . . . . . . . 7

2.1.3.2 Mutation Analysis . . . . . . . . . . . . . . . . . . . . . . 7

2.1.3.3 Model Diversity . . . . . . . . . . . . . . . . . . . . . . . 7

2.2 Altran Practice . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.2.1 Interview Guidelines . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.2.2 Findings from Interviews . . . . . . . . . . . . . . . . . . . . . . . 9

2.2.3 Challenges at Altran . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.3 Literature Review . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.3.1 Procedure for Executing the Review . . . . . . . . . . . . . . . . . 11

2.3.2 Challenges and Open Issues Mentioned in the Literature . . . . . . . . . . . 12
2.3.3 Summary of the Approaches . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.3.3.1 M2M Transformation Testing . . . . . . . . . . . . . . . . . . . . . . 13
2.3.3.2 M2T Transformation Testing . . . . . . . . . . . . . . . . . . . . . . 18
2.3.4 Further Selection and Prioritization . . . . . . . . . . . . . . . . . . . . . . 19
2.3.5 Prioritization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.4 Reflection on Interviews and the Literature Review . . . . . . . . . . . . . . . . . . 23
2.5 Evaluation Criteria for the Selected Approaches . . . . . . . . . . . . . . . . . . . . 24

**3 TractsTool and Matching Table Builder 25**
3.1 TractsTool . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.1.1 Introduction of TractsTool . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.1.2 Case Study . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
3.1.2.1 Introduction of UML2Relational transformation case . . . . . . . 26
3.1.2.2 Implementation of the UML2Relational case study using TractsTool 27
3.1.2.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.1.3 Discussion on TractsTool . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.1.4 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.2 Matching Table Builder . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.2.1 Introduction of Matching Table Builder . . . . . . . . . . . . . . . . . . . . 32
3.2.2 Discussion on Matching Table Builder . . . . . . . . . . . . . . . . . . . . . 33
3.2.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

**4 UML-based Specification Environment and Efinder 35**
4.1 The UML-based Specification Environment . . . . . . . . . . . . . . . . . . . . . 35
4.1.1 Introduction to UML-based Specification Environment . . . . . . . . . . . . 35
4.1.2 Introduction of Classifying Terms . . . . . . . . . . . . . . . . . . . . . . . 36
4.1.3 Case Study . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.1.3.1 Invariants . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.1.3.2 Configuration in USE . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.1.3.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.1.4 Discussion on USE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.1.5 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.2 Efinder . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.2.1 Introduction of Efinder . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.2.2 Case Study: Generating multiple instance models . . . . . . . . . . . . . . . 43
4.2.2.1 Generating OCL files . . . . . . . . . . . . . . . . . . . . . . . . . 43

4.2.2.2 Configure and Run the Tool . . . . . . . . . . . . . . . . . . . . . . 45
4.2.2.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
4.2.3 Discussion on Efinder . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
4.2.4 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

**5 Conclusions** **51**

**Bibliography** **53**

**Appendix** **61**

**A ASOME Case Study Reports** **61**
A.1 Industrial case study 1: to compile and generate one instance model of data.ecore . 61
A.1.1 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
A.1.1.1 1. Remove the operations in the ecore models that Efinder can not compile . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
A.1.1.2 2. Ignore some original OCL constraints of the ASOME metamodels 63
A.1.1.3 3. Remove the OCL expressions so that there is no type conformance problems in USE . . . . . . . . . . . . . . . . . . . . . . . . . 65
A.1.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
A.2 Industrial case study 2: to generate multiple instance models using classifying terms 67
A.2.1 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
A.2.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
A.3 Problem 1 of Efinder: compilation error in OCL constraints . . . . . . . . . . . . 69
A.4 Problem 2 of Efinder: unsatisfiable proof . . . . . . . . . . . . . . . . . . . . . . . 71
A.5 Problem 3: Efinder does not support generating a pair of models . . . . . . . . . 73

**B Code** **75**
B.1 QVTo Transformation for UML2Relational . . . . . . . . . . . . . . . . . . . . . 75
B.2 uml.use generated by TractsTool . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
B.3 Generated USE Specifications By TractsTool . . . . . . . . . . . . . . . . . . . . 78
B.4 Generated USE Specifications By Efinder . . . . . . . . . . . . . . . . . . . . . . 80
B.5 The code for generating OCL files with classifying terms . . . . . . . . . . . . . . 83
B.6 The code for spliting a model into one ASOME model and one generator model . . 87

# List of Figures

2.1 The general model transformation process . . . . . . . . . . . . . . . . . . . . . . . 5
2.2 The model transformation testing process in ASOME project . . . . . . . . . . . . 9
2.3 The methodology of literature review . . . . . . . . . . . . . . . . . . . . . . . . . 11

3.1 TractsTool GUI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
3.2 uml.ecore shown in Eclipse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.3 relational.ecore shown in Eclipse . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.4 uml_model.xmi shown in Eclipse . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.5 relational_model.xmi shown in Eclipse . . . . . . . . . . . . . . . . . . . . . . . . 28
3.6 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.7 ATL Transformation Types Extractor GUI . . . . . . . . . . . . . . . . . . . . . . 33
3.8 Matching Table Builder GUI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

4.1 Classifying term dashboard shown in USE GUI . . . . . . . . . . . . . . . . . . . 37
4.2 Configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.3 Generated models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.4 the generated models shown in Eclipse . . . . . . . . . . . . . . . . . . . . . . . . 47

A.1 data.ecore . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
A.2 Unsupported operations in Efinder . . . . . . . . . . . . . . . . . . . . . . . . . . 64
A.3 helper operations defined by ASOME developers . . . . . . . . . . . . . . . . . . 65
A.4 Class 'Operation' in system.ecore . . . . . . . . . . . . . . . . . . . . . . . . . . 66
A.5 the USE specification for Class 'Operation' . . . . . . . . . . . . . . . . . . . . . 66
A.6 the generated model shown in Eclipse . . . . . . . . . . . . . . . . . . . . . . . . 66
A.7 the expected model shown in ASOME . . . . . . . . . . . . . . . . . . . . . . . . 67
A.8 the expected model shown in ASOME . . . . . . . . . . . . . . . . . . . . . . . . 70
A.9 a monolithic model generated by Efinder . . . . . . . . . . . . . . . . . . . . . . . 74
A.10 first figure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

A.11 second figure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

# List of Tables

2.1 The result of the first screening: test generation with coverage criteria . . . . . . . 14
2.2 The result of the first screening: mutation analysis . . . . . . . . . . . . . . . . . . . 15
2.3 The result of the first screening: equivalence partitioning . . . . . . . . . . . . . . 16
2.4 The result of the first screening: developing oracle functions . . . . . . . . . . . . . 17
2.5 The result of the first screening: M2T transformation testing . . . . . . . . . . . . 19
2.6 The result of the second screening: test generation with coverage criteria . . . . . . 20
2.7 The result of the second screening: equivalence partitioning . . . . . . . . . . . . . 21
2.8 The result of the second screening: M2T transformation testing . . . . . . . . . . . 22
2.9 The priority list of the 11 approaches . . . . . . . . . . . . . . . . . . . . . . . . . . 23

3.1 Evaluation results of TractsTool . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.2 Evaluation results of MTB . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

4.1 The characteristic values of partitions . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.2 Evaluation results of USE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.3 Evaluation results of Efinder . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

A.1 The deleted operations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
A.2 the test results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

# Listings

3.1 Mapping rule type2type in UML2Relational.qvto . . . . . . . . . . . . . . . . . . . . 28
3.2 Tract for type2type() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.3 Generated commands . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.1 Classifying term dashboard . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.2 constraints for Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.3 constraints for Type . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.4 constraints for Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.5 constraints for Attribute . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.6 an integer-type classifying term . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
4.7 an integer-type classifying term is rewritten into two boolean-type classifying terms 44
4.8 3 classifying terms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
4.9 The full OCL file with characteristic value 000 . . . . . . . . . . . . . . . . . . . 44
A.1 data.ocl . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
A.2 Model configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
A.3 4 classifying terms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
A.4 the complete OCL file for partition 0000 . . . . . . . . . . . . . . . . . . . . . . . . 68
A.5 Test bounds configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
A.6 OCL constraints . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
A.7 Error message . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
A.8 The invariant "nonNull_type_Collection_ordered" in model.use . . . . . . . . . . . . 71
A.9 Model configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
A.10 OCL constraints . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
A.11 Unsatisfiable proof . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
B.1 UML2Relational.qvto . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
B.2 Class.use . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
B.3 ClassifyingTerms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
B.4 generateClassifyingTermOCLFiles . . . . . . . . . . . . . . . . . . . . . . . . . . . 83

B.5 Partition7.ocl . . . . . . . . . . 85
B.6 The class ClassTest . . . . . . . . . . 86
B.7 The code for spliting a model into one ASOME model and one generator model . . 87

# Chapter 1

# Introduction

## 1.1 Background

Model Driven Engineering (MDE)[40] is a software development methodology that uses models as primary artifacts to drive development. It aims at increasing productivity and improving the quality of the software development process. Instead of writing code manually, software developers in MDE specify the software system to be developed in abstract models expressed in a suitable modeling language. Models can be analyzed for correctness so that defects in the generated software are avoided thus improving the quality of the final product. Model transformation is an important part of MDE. Model transformation automates the software development. It is the process where the transformation software transforms the source models into target models or code in the target language following certain rules.

## 1.2 Problem Statement

Like any other software artifact, model transformations need to be specified, designed, implemented, and checked for correctness. One particular way of checking is testing. MDE has been increasingly applied in the industry and testing of transformations is becoming an important task in the development process. The current testing processes in the industry are facing multiple challenges demonstrated by the large workload of the processes and failures in the results of the transformations. A number of approaches have been proposed in the literature to improve the model transformation testing processes. However, there is still insufficient information about the suitability of these approaches in an industrial context. Therefore, the problem is how suitable these approaches are and why these approaches fail if they are not suitable.

## 1.3 Research Questions

The focus of this thesis is on implementing the existing approaches for testing model transformations and evaluating their suitability in an industrial context. The project is executed at Altran Netherlands, where several industrial projects are using MDE techniques for software development including model transformations. The following research questions are formulated:

**RQ1: What are the main challenges in the practice of model transformation testing in the industry?**

- Objective 1: Describe the current model transformation testing process in the industry.
- Objective 2: Identify the major problems and key needs in the current model transformation testing processes.
- Objective 3: Describe the current model transformation testing processes at Altran and identify the major problems and key needs in the model transformation testing process at Altran.

**RQ2: What are the existing approaches that can improve the testing process in the industry?**

- Objective 1: A description of the existing approaches about model transformation testing.
- Objective 2: Select the approaches that can be used to improve the current testing practice at Altran for further investigation and motivate the choices.

**RQ3: How suitable are these approaches?**

- Objective 1: Implement the selected approaches on small cases and evaluate their suitability. If the selected approach is considered suitable in small cases, implement the selected approaches for ASOME cases. [1]
- Objective 2: Analyze and report the suitability of the selected approaches.

## 1.4 Methodology

For RQ 1, we intend to assume that Altran sufficiently represents the industry for this thesis. Other companies may have different challenges. However, the scope in this thesis is on Altran

[1]ASOME cases are referred to the case studies for testing the model transformations of the ASOME tool.

practices. If we want to enlarge the scope then another method should be selected: interviews and cases from multiple industries and companies. We conducted several interviews with developers at Altran to gain a better understanding of the current testing process in the industry. We interviewed developers of model transformations, testers, and test architects at Altran to understand the major problems of the model transformation testing. We summarized the problems by discussing with the testers.
For RQ2, we conducted a literature review. We classified the approaches. We listed the tools and languages used in the literature and analyzed their suitability in the given context. We selected some of the approaches for further investigation and motivated our choices.
For RQ3, we implemented the selected approaches on small cases and further on industrial cases inspired by the ASOME tool.

## 1.5 Thesis Outline

Chapter 2 introduces the concepts of model transformation testing, the model transformation testing process at Altran and the results of the literature review. Chapter 3 and Chapter 4 introduces 4 model transformation tools. Since TractsTool[6] and Matching Table Builder[68] have similar purposes, they are introduced in Chapter 3. Chapter 4 introduces UML-based Specification Environment (USE)[23] and Efinder because Efinder and USE have the same purpose and Efinder is based on USE. Both Chapter 3 and 4 provide the case studies and the discussions of the tools. The industrial case study reports are shown in Appendix A.

Chapter 2

# The study of the practice and the literature review

## 2.1 Definitions

### 2.1.1 Model Transformation Development

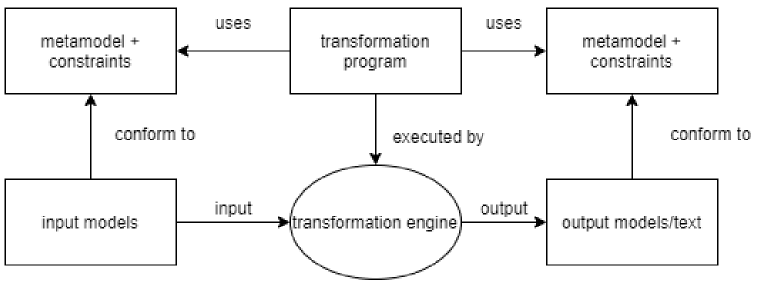


Figure 2.1: The general model transformation process

**Model** In MDE, a model is referred to as a domain model that describes a software system in a formal way.

**Metamodel** A metamodel is a model of a modeling language[15]. A metamodel specifies a set of rules and constraints that define a modeling language. The metamodel is the definition of the abstract syntax of the modeling language. Every model expressed in that modeling language conforms to the metamodel.

**Model Transformation** Model transformation is an automated process to generate models or text. It takes a model as input and generates a model or text as output. Figure 2.1 shows the general model transformation process. As can be seen from Figure 2.1, input models conform to a metamodel and a set of constraints. The transformation program uses the input metamodel and the output metamodel. It defines how elements in the input model are transformed into the output model. The transformation program can be implemented in a general-purpose language like Java or a dedicated model transformation language such as QVTo[1], ATL[2], etc. A transformation engine executes the transformation program and generates possibly multiple output models that conform to their metamodels. There are two types of model transformation: model-to-model (M2M) transformation and model-to-text (M2T) transformation. An M2M transformation generates models conforming to metamodels, whereas an M2T transformation results in text that can be anything: program code, documents, plain text, etc. Typically, when the difference between an input language and an output language in the level of abstraction becomes large, monolithic transformations may have some inherent problems, such as little reuse opportunities, bad scalability, etc [65]. A monolithic M2T transformation can be composed into a model transformation chain[65]: a sequence of smaller sub-transformations which enables isolated testing of individual model transformations[42].

### 2.1.2 Model Transformation Testing

We first present four basic concepts used throughout this report.

**Fault and Failure** In the context of this study, a fault is an error of a model transformation. Failure is an unexpected result of a transformation. Failure is an indication of a fault. The purpose of testing is to detect faults in the transformation and manage risks of failures. In general, it is impossible to detect all faults. Moreover, it is not the responsibility of testing to locate, clear faults, or analyze the cause of faults.

**Oracle Function** In model transformation testing, oracle functions are used to check if the result of the transformation is correct. Typically, oracle functions can use user expectations, transformation specifications, previous versions of the same program, comparable products, etc [66][16].

### 2.1.3 Test Quality Evaluation

Testers need to measure the quality of a test suite. This measurement process is called test quality evaluation. Test coverage and mutation analysis are commonly adopted in the literature to qualify a test suite[67]. Model diversity is less common but was also used in the literature[54].

[1]https://wiki.eclipse.org/QVTo
[2]https://www.eclipse.org/atl/

#### 2.1.3.1 Test Coverage

In this study, a set of test cases using a set of test models as input is referred to as a test suite. A good test suite should be large enough to satisfy certain criteria but small enough to avoid redundant testing. Redundant testing means using structurally similar test models. A coverage criterion aims at measuring the quality of a test suite and deciding when testers can stop testing so that the model transformations are qualified[62]. There are three types of coverage commonly adopted in the state of the art of model transformation testing: transformation coverage, specification coverage, and metamodel coverage.

**Transformation Coverage** Transformation coverage measures how much of the transformation implementation is used by a test suite. For example, Schönböck et al.[52] adopted *rule coverage* to ensure every transformation rule is invoked at least once by a test suite.

**Specification Coverage** The specifications are formalized requirements of correctness of transformations. Specification coverage measures how many specifications and combinations of these specifications are covered by a set of test models[31].

**Metamodel Coverage** Metamodel coverage measures how much of a source metamodel is used by a test suite. For example, [18] and [19] adopted *class coverage*. The class coverage criterion states that for each concept defined in the metamodel, at least one concrete instance can be found in the test models.

#### 2.1.3.2 Mutation Analysis

Mutation analysis can show the sensitivity of a test suite[53]. Mutation analysis mutates units in the transformation implementations to create faults. This faulty version of a transformation is called *mutant*[43]. If at least one test model can detect the mutant, then the mutant is called a *killed* mutant [43]. The tester usually creates a set of mutants to check whether a test suite can detect and kill all the mutants. The proportion of the killed mutants in the total non-equivalent mutants are called *mutation score*[43]. The *mutation operators* are used to generate mutants[53]. If a test suite gets a high *mutation score*, it indicates that this test suite is able to detect man-made faults of the system under test.

#### 2.1.3.3 Model Diversity

Model diversity measures the diversity of a test suite. An efficient test suite should be diverse to prevent redundant testing. Therefore, the model diversity can indicate the quality of a test suite.

Studies [54], [51], [35] and [64] gave different definitions of model diversity metrics. The defined model diversity metrics were further applied in *equivalence partitioning* techniques.

## 2.2 Altran Practice

To study the Altran practice, we will focus on the ASML Software Modeling Environment (ASOME) project. The ASOME project is aimed at modeling and construction of software systems that adhere to the Data-Control-Algorithm (DCA) architecture pattern. A tool also called ASOME has been developed for modeling and generating a part of the ASML software in this project. In particular, the tool allows the modeling of data structures and generating software repositories for managing the data.
To understand the current practice of model transformation testing and its main challenges in the context of Altran, we conducted interviews with three testers. A testing process flowchart was formed according to the answers of the testers. Moreover, an explanation of the flowchart is presented. Both the flowchart and the description were confirmed by the testers.

### 2.2.1 Interview Guidelines

In this section, we present the interview questions for ASOME testers. These questions are in the context of the ASOME project.

**Questions about the model transformation chain**

- What is the structure of the model transformation chain?
- How is the software development process using this model transformation chain?

**Questions about the model transformation testing process**

- Please describe the process of model transformation testing.
- How do you verify the results of M2M/M2T transformation?
- Is there any formal specification for the model transformation testing?
- Is there any automatic test model generation?

**Questions about faults and failures of the model transformation**

- Do you have a classification for the failures? If yes, what are the types? (e.g. the generated code can not compile or the program has unexpected behaviors? Can you name them all?)
- Why were the faults not detected during testing?

**Questions about test quality**

- Do you measure the metamodel/transformation/specification/code coverage? If yes, how do you measure? How much is the metamodel/transformation/specification/code coverage? If no, why not?
- Have you made duplicate or structurally similar test models or have you seen duplicate or structurally similar test models made by other testers?

### 2.2.2 Findings from Interviews

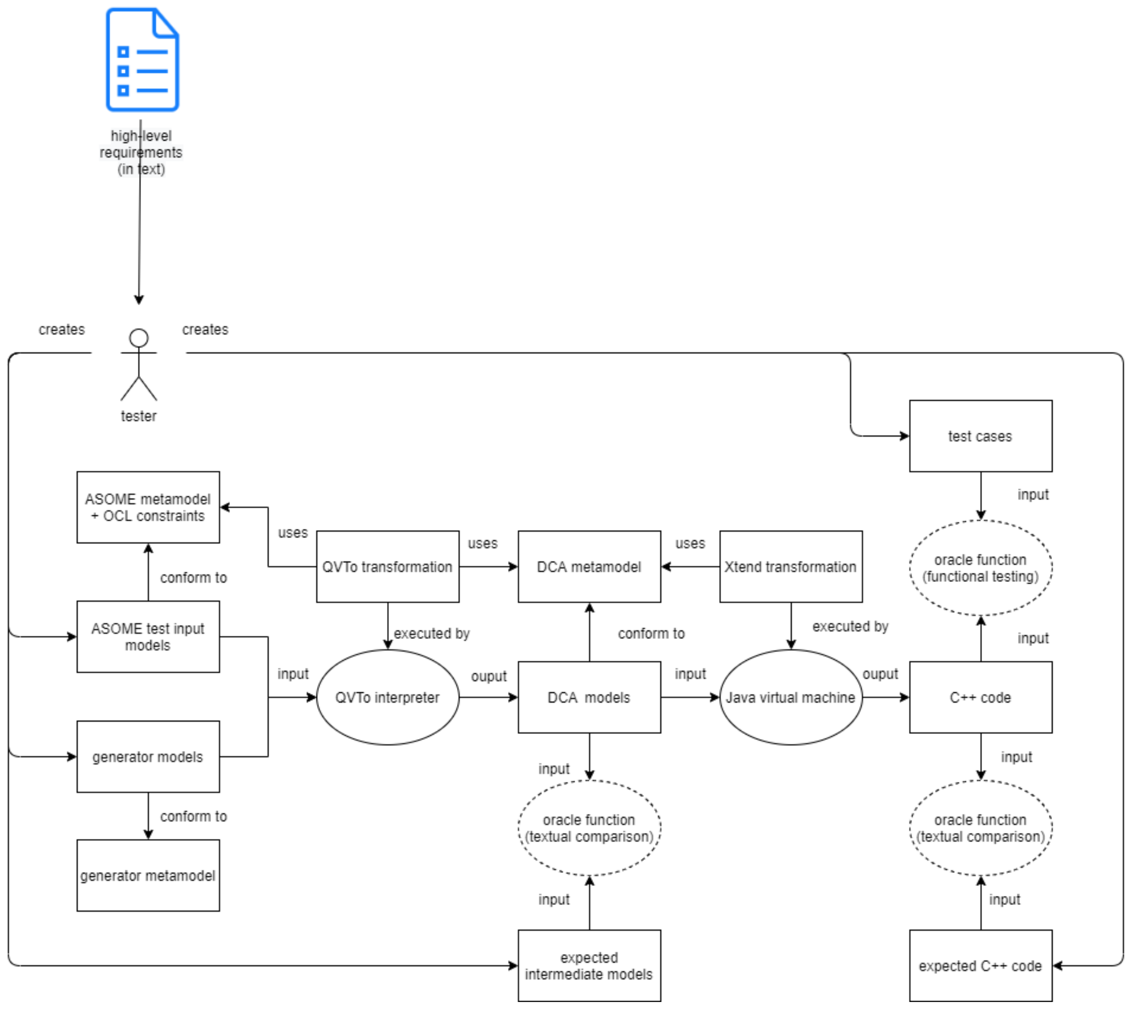


Figure 2.2: The model transformation testing process in ASOME project

Figure 2.2 shows the transformation chain and the testing process in the ASOME project of Altran. The model transformation chain of the ASOME project is composed of two model transformations. The first model transformation is an M2M model transformation in QVTo. The

second model transformation is an M2T model transformation in Xtend. The ASOME test models together with the generator models will be transformed into intermediate models by the QVTo transformation. The intermediate models are DCA models that conform to the DCA metamodel. DCA is the central architectural pattern for the software systems at ASML. DCA is referred to as three aspects: Data, Control, and Algorithm. The QVTo transformation is executed by the QVTo interpreter. Further, the DCA models are transformed into C++ code by the Xtend transformation which is executed by the Java virtual machine. The high-level requirements are often recorded in text informally by ASML architects/toolsmiths. The tester manually creates test models and generator models according to the informal requirements. There are no low-level requirements or formal specifications shared among the testers.
Testers use oracle functions to verify the results. The oracle functions return the results of a comparison between the actual output and the expected output. The results are binary answers, which indicate whether the actual output is acceptable. There are two types of oracle functions. One is to compare the code of the model/generated code line by line. The other one is to evaluate the execution results of the generated C++ code with test cases made by the testers. Initially, there are no available expected DCA models. The tester needs to manually check the correctness of the actual output. Once the actual output is considered as correct, they will be set as the expected output. When the transformation implementation changes, the tester will give the same input and manually compare the actual output with the expected output again. There is no formal specification made for the testing and there is no automatic test model generation.
There are reports about the failures including types and occurrences. However, there is no report about the faults in the transformations which cause the failures and no analysis of why these faults were not detected during testing. Most of the faults occurred in the M2T transformation.
There is no measurement for test quality. Duplicate test models were found during testing.

### 2.2.3 Challenges at Altran

In summary, the challenges of model transformation testing reported by the testers in the ASOME project at Altran are:

**The difficulty to generate a qualified test suite** The high-level requirements are the informal descriptions for the expected features for the generated C++ code. Moreover, the transformation does not produce the correct result for a new feature of an input model since the new feature has not been used by any input models during testing. There is no metamodel coverage or transformation coverage measurement. Therefore, there is no indicator of the test suite quality to assist the testers in completing a test suite. Moreover, the manual generation of test models is inefficient.

**The difficulty to generate an efficient test suite** An efficient test suite should be diverse to prevent redundant testing. It is often the case that testers make duplicate or structurally similar test models without awareness. It is a challenging task for testers to make sure duplicate or structurally similar test models are avoided by identifying and distinguishing the input models manually.

## 2.3 Literature Review

### 2.3.1 Procedure for Executing the Review

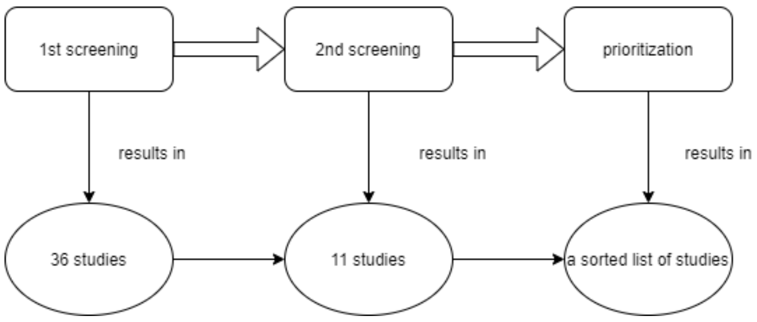


Figure 2.3: The methodology of literature review

Figure 2.3 shows the methodology of the literature review. There were two rounds of screening and a prioritization.
For the first round of screening, we searched for the studies that aim at solving problems related to model transformation testing. The result of the first round screening is 36 studies. A description of the 36 approaches is presented in Section 2.3.3.1 and Section 2.3.3.2. There are three main search strategies adopted during the first screening.

- First, we used keywords to search studies (written in English) in Google Scholar search engine[3].

```
"model transformation" AND/OR "model to model" AND/OR "model to text"
AND "testing"
```

- Second, we briefly read the content of all the studies after the first screening and excluded those that did not discuss (M2M and M2T) model transformation testing. Then we searched for the references in the related work section of the selected studies.

[3]https://scholar.google.com/

- Third, those cited by the model transformation testing studies are also likely to talk about model transformation testing. Therefore, these studies with the right contents were also considered.
- Fourth, we avoided overlapping of primary studies. This means if several studies are conducted as an evolution of the same testing approach, then only the latest study will be included in the result of the first screening. The information about the studies in the result of the first screening is further summarized and presented in the tables. These tables will be used to analyze the feasibility of the proposed approaches in the given context.

The first screening resulted in 36 papers. For the second round of screening, we reviewed the 36 studies and identified 11 studies that may solve or alleviate the problems at Altran (Section 2.2.3). We motivated our choices in Section 2.3.4. Due to the time limitation, we can not explore all 11 approaches. Taking the interests of the stakeholders into consideration, we prioritized the approaches and made a priority list of the 11 approaches in Section 2.3.5.

### 2.3.2 Challenges and Open Issues Mentioned in the Literature

**Automated generation of a qualified and efficient test suite** Most of the literature reviews[13][3][44][49][4][53][31] about model transformation testing pointed out two major concerns. The first one is the automated generation of a qualified and efficient test suite. A good test suite generation should be automatic since manual input model generations will limit the productivity of testers. A good test suite generation approach should also be qualified so that the test suite can ensure the model transformations meet the requirements. A good test suite should also be efficient, which means duplicate or similar tests can be identified and avoided.

**The availability of a suitable oracle** The second most discussed challenge is the availability of a suitable oracle. Therefore, some studies aim at providing alternative oracles when the testers do not know how oracles should be like.

**The definition of test criteria** Gerpheide et al.[22] pointed out the need to identify thresholds for the quality metrics of a test suite. The thresholds can also be called criteria. The definitions of the criteria are important since they should guide the testers to stop testing at the right time.

**The justification of the choices of test quality metrics** The justification of the choice of a test suite quality metric is lacking in the literature. For example, Guerra et al.[31] proposed to use transformation specification coverage criteria instead of metamodel coverage criteria. According to Guerra et al.[31], the purpose of model transformation testing is to test the intention of the transformation. Therefore, specification coverage criteria are more suitable than metamodel coverage criteria. However, this claim was not elaborated in this study and there is a lack of sufficient study about the comparison between the proposed coverage criteria.

### 2.3.3 Summary of the Approaches

In this section, we will classify and discuss the 36 studies we found after the first screening.

#### 2.3.3.1 M2M Transformation Testing

Different classifications of studies in M2M transformation testing have been proposed in the literature. [66] sorted the state of the art into 9 categories according to the techniques: *graph search algorithms*, *random testing*, *evolutionary testing*, *constraint solving*, *model checking*, *static analysis*, *abstract interpretation*, *partition testing*, and *slicing.* [4], [53], and [31] classified studies according to three phases of model transformation testing: *generating test models*, *defining coverage criteria* and *developing oracle functions.* [3], [44] and [49] merged *generating test models* and *defining coverage criteria* into one classification *generating test models.* The first classification is not suitable for comparison because it is unclear which approach improves which part of the testing process. The last classification is more reasonable because the approaches are grouped by their purpose. Moreover, since the coverage criteria are used as a stopping condition for the phase of test model generation, these two activities belong to one phase of testing. Therefore, in this study, we classify studies into two categories: *test model generation* and *developing oracle functions.*

**A. Test Model Generation** There is a significant amount of proposed approaches that focus on automatic test model generation. These test model generation approaches can be classified into three categories: test coverage criteria, mutation analysis, and equivalence partitioning based on different test quality evaluations. Studies in Section **A.1 Test Model Generation with Coverage Criteria** aim at reaching high test coverage criteria. Studies in Section **A.2 Mutation Analysis** aim at generating a test suite that can achieve high mutation scores. Studies in Section **A.3 Equivalence Partitioning** aim at obtaining a test suite with as many diverse models as possible.

**A.1 Test Model Generation with Coverage Criteria** Table 2.1 shows the result of the first screening of the studies which aim at reaching the chosen test coverage criteria. As can be seen from Table 2.1, the majority of the test coverage is metamodel coverage and transformation coverage. Brottier et al.[5] proposed an algorithm to generate a test suite that can satisfy metamodel coverage with a test generation tool called *OMOGEN.* Sen et al.[55][56] proposed a tool called *Cartier* to generate test models to reach metamodel coverage criteria. The tool *Cartier* can

Table 2.1: The result of the first screening: test generation with coverage criteria

| Year | Title | Test Criteria | Model Transformation Language | Used Technologies | Evaluation for the Approach |
|---|---|---|---|---|---|
| 2006 | Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool[5] | Metamodel coverage | Not mentioned | MOF, OCL | Mutation analysis |
| 2009 | Automatic model generation strategies for model transformation testing[56] | Metamodel coverage | Not mentioned | Alloy | Mutation analysis |
| 2012 | Formal specification and testing of model transformations[63] | Not mentioned | ATL, QVT, RubyTL, JTL | ASSL language, USE | Correctness of output models checked by using USE |
| 2012 | ATLTest: A White-Box Test Generation Approach for ATL Transformations[26] | Transformation coverage | ATL | OCL | Not mentioned |
| 2013 | TETRABox - A Generic White-Box Testing Framework for Model Transformations[52] | Transformation coverage | Not mentioned | PAMOMO, QVT-Relations, OCL | Only observations |
| 2014 | A Search Based Test Data Generation Approach for Model Transformations[38] | Transformation coverage | ATL | MOTTER | Not mentioned |
| 2015 | Specification-driven model transformation testing[31] | Specification coverage | ATL/ETL | PAMOMO, Eclipse, Z3 solver, USE, OCL | Mutation analysis |
| 2017 | Translating target to source constraints in model-to-model transformations[14] | Metamodel coverage | ATL, ETL, QVT, TGGs | OCL, anATLyzer | Measured solving time and total time, precision and recall metrics |
| 2019 | On Analyzing Rule-Dependencies to Generate Test Cases for Model Transformations[46] | Rule-dependency coverage | TGG | OCL, USE framework and USE model validator | Rule dependencies |

make use of the *OCL constraints* on the *Ecore* metamodels. The *OCL constraints* on the *Ecore* metamodels are used to expressed invariants and model transformation pre-conditions. Then *Cartier* invokes the *Alloy API* and then launch a SAT solver to generate models. However, the *OCL constraints* need to be transformed to *Alloy facts* manually. This study [56] adopted two *Input-domain Partition based Strategies* as metamodel coverage criteria from [18] to guide model generation, called *AllRanges Criteria* and *AllPartitions Criteria.* Vallecillo et al.[63] proposed to use *OCL invariants* as the specifications of the transformation and use *ASSL(A Snapshot Sequence Language)* to generate test models. The test models can either satisfy all constraints or violate at least one constraint. But coverage criteria of the specifications were not discussed in [63]. The framework TETRABox[52] proposed by Schonbock et al. also requires formalized requirements of transformation. TETRABox can generate a test suite that ensures a certain level of transformation coverage. Jilani et al.[38] proposed an OCL-based approach for test model generation, which adopted transformation coverage criteria. Guerra et al.[30] presented a visual, declarative language called PAMOMO to specify behavioral contracts. They also presented the PAMOMO Contract-Checker which assists in the compilation of PAMOMO into QVT-Relations. Based on the PAMOMO language, Guerra et al.[31] proposed a framework called specification-driven model transformation testing. In this testing framework, the requirements are formalized into specifications in PAMOMO. The PAMOMO specifications can derive oracle functions and test models automatically. The generated test models can cover all the specifications using SAT solving techniques. This study also proposed 7 levels of specification coverage criteria. Cuadrado et al.[14] presented a method which can also be used to generate test models considering the constraints over the target models. This method translates OCL constraints to the source meta-model using the anATLyzer tool. Hermann et al.[32] proves that rule dependencies directly impact the quality properties of a model transformation for TGG. TGG is a formal, declarative, bidirectional model transformation language[41][33]. Based on this proof, Nguyen et al.[46] proposed a method to discover rule dependencies of TGG and automate test model generation based on the rule-dependency coverage.

Table 2.2: The result of the first screening: mutation analysis

| Year | Title | Mutation Operators | Model Transformation Language | Evaluation for the Approach |
|---|---|---|---|---|
| 2006 | Mutation analysis testing for model transformations[43] | Semantic operators | Kermeta, Tefkat | Compared with MuJava |
| 2015 | Towards systematic mutations for and with ATL model transformations[60] | Syntactic operators | ATL | Not mentioned |
| 2015 | Towards an automation of the mutation analysis dedicated to model transformation[1] | Semantic operators | Kermeta | Result analysis |
| 2016 | Static analysis of model transformations[13] | Typing operators | ATL | Usefulness and performance |
| 2019 | Towards effective mutation testing for ATL[28] | Zoo operators | ATL | Applicability, resilience, stubbornness |

**A.2 Mutation Analysis** The concept of mutation analysis is explained in Section 2.1.3.2. Table 2.2 shows the result of the first screening of studies about mutation analysis. The studies about mutation analysis in model transformation testing focused on automatically generating qualified mutants and reducing computational costs. However, most of the mutation analysis approaches are language-dependent since the definition of a mutation operator requires knowledge of the language. Therefore, the mutation analysis technique used for model transformation testing is not as developed as the one used for C, C++, and Java. Troya et al.[60] proposed *syntactic operators* which can create, delete, or update the elements of the ATL metamodel. However, syntactic mutation operators do not mimic real developer mistakes. Mottu et al.[43] and Aranega et al.[1] proposed *semantic operators* on ATL and Kermeta that take semantics of model transformation into account. Cuadrado et al.[13] focused on typing errors and proposed *typing operators* that aim at testing anATLyzer (ATL static analyzer). Guerra et al.[28] proposed *zoo operators* that emulate the most common errors in ATL zoo[4]. *Mutation analysis* is computationally costly. A few studies discussed *sufficient operators*[2][47][45]. Their understanding may be valuable for mutation testing on model transformations even though they focused on imperative languages like C, C++,

[4]https://www.eclipse.org/atl/atlTransformations/

and Java. Furthermore, *mutation analysis* is commonly used to evaluate the quality of a test suite generated by other testing approaches (See Table 2.1).

Table 2.3: The result of the first screening: equivalence partitioning

| Year | Title | Equivalence Partitioning Techniques | Model Transformation Language | Used Technologies | Evaluation for the Approach |
|---|---|---|---|---|---|
| 2014 | Test data generation for model transformations combining partition and constraint analysis[27] | OCL constraints analysis | Not mentioned | OCL, OCLBBTesting EMF2CSP | Not mentioned |
| 2015 | Employing classifying terms for testing model transformations[25] | Classifying Terms | Not mentioned | ASSL, USE, OCL | Not mentioned |
| 2017 | Testing transformation models using classifying terms[7] | Classifying Terms | Not mentioned | OCL, USE | Performance analysis |
| 2018 | Testing models and model transformations using classifying terms[34] | Classifying Terms | Medini-QVT, JTL | USE, OCL, ASSL, Kodkod | Usability, performance and scalability |
| 2018 | Test Model Generation using Equivalence Partitioning[36] | Equivalence classes definition in EPL | Not mentioned | EMG, EOL, EPL, KodKod | Not mentioned |
| 2018 | Iterative Generation of Diverse Models for Testing Specifications of DSL Tools[54] | Neighborhood shapes | Not mentioned | Alloy, VIATRA-Generator | Mutation analysis |

**A.3 Equivalence Partitioning** Table 2.3 shows the result of the first screening of the studies which tried to avoid structurally similar tests. This technique is called *equivalence partitioning*. The main idea of *equivalence partitioning* is to identify structurally similar models and put them into the same partition. Only one model in the same partition needs to be tested. This idea has been considered by several studies[27][25][34][7][36]. Gonzalez et al.[27] proposed a mechanism to fine-tune the partitions by systematically analyzing OCL constraints in the source metamodels. Some studies[25][34][7][36] focus on *classifying terms*, which is a kind of *equivalence partitioning* technique. *Classifying terms* is an instrument that can identify the structural properties of models. Models in the same partitions are considered equivalent. Therefore, testers only need to test one model from the same partition instead of all models. Hilken et al.[34] and L. Burgueño[7] proposed to combine *classifying terms* with *tracts*. Vallecillo et al.[63] introduced the concept *tracts*. *Tracts* is a special kind of model transformation contract. *Tracts* can specify the constraints for a model transformation in OCL. Additionally, every *tract* provides a set of test models that satisfy a corresponding set of constraints. The generation of test models is done by ASSL (A Snapshot Sequence Language) within the USE (UML-based Specification Environment) environment[5]. Semeráth et al.[54] proposed *diversity metrics* based on *neighborhood shapes* and a model generation technique which aims at deriving structurally diverse models.

[5]http://useocl.sourceforge.net/w/index.php/Main_Page#Download

Table 2.4: The result of the first screening: developing oracle functions

| Year | Title | Proposed Techniques | Model Transformation Language | Used Technologies | Evaluation for the Approach |
|---|---|---|---|---|---|
| 2007 | Matching model-snippets[50] | A generic pattern framework | graph MT languages | Kermeta, EMF, Flora-2 | Not mentioned |
| 2008 | Model transformation testing: oracle issue[44] | 6 types of oracles | Not mentioned | MOF, OCL | Not mentioned |
| 2010 | A visual specification language for model-to-model transformations[29] | A specification language for oracle | QVT, ATL, ETL | OCL, GMF, eol | Not mentioned |
| 2013 | Using Meta-model Coverage to Qualify Test Oracles[17] | An oracle quality metric | Not mentioned | Not mentioned | Mutation analysis |
| 2013 | Partial Test Oracle in Model Transformation Testing[16] | Partial oracles | ATL | EMFCompare, EMF, XMI, VIATRA, CSP | Not mentioned |
| 2014 | Testing model transformation programs using metamorphic testing[37] | Metamorphic testing | ATL | Not mentioned | Empirical Evaluation |
| 2018 | Automated inference of likely metamorphic relations for model transformations[61] | Automated inference of MRs | ATL | ATL/EMFTVM, Java, EMT | Precision measure |

**B. Developing Oracle Functions** Table 2.4 shows the result of the first screening of studies on the development of oracle functions. In general, an oracle function is a predicate that states if the result of the test is correct. Some oracle functions compare the test results with the expected results. It is also possible to evaluate some expressions on the produced result. Mottu et al. [44] presented six kinds of oracle functions. However, oracle functions can still be difficult to develop. There are two main challenges of developing oracle functions[49].

**B.1 How to obtain oracles?** It is difficult for the tester to produce expected models for all test models. *Metamorphic testing*[11] can provide an alternative when the expected output of a test model is unknown. *Metamorphic testing* checks whether multiple executions of a program under test have certain properties, which are called *metamorphic relations (MRs).* Jiang et al.[37] demonstrated the application of *metamorphic testing* in the context of model transformation testing and empirically proved its effectiveness. However, the MRs are manually constructed by a domain expert. Troya et al.[61] followed the work of Jiang et al.[37] and proposed an approach to generate likely MRs automatically and improved the feasibility of *metamorphic testing.*

**B.2 How to compare the results of a transformation with the oracles?[49]** There are three techniques to implement oracle functions: model comparison, contracts, and pattern match[44].

- **Model comparison:** Model comparison requires available expected models to compare with the output model. However, it can be difficult for a tester to know all the properties that an expected model should have due to the complexity of specifications. Finot et al.

[16] presumed that the output models can be divided into a predictable part and a non-predictable part. They proposed *partial oracle* to compare the predictable part. However, the model comparison is still challenging. Comparing expected models and output models can be done syntactically or semantically. The model comparison is a graph isomorphism problem since a model can be considered as a graph. However, this can be problematic since two models can be semantically equivalent while syntactically different[49]. Additionally, checking graph isomorphism is computationally costly[54].

- **Contracts:** Contract-based comparison requires knowledge of the semantics of the target language[49]. The goal of a contract is to define what to expect. Therefore, the second technique is to use contracts as oracle functions. Carious et al. [9] used OCL contracts to define oracles. Guerra et al.[29] proposed a visual specification language called PAMOMO, which can be compiled into OCL, to simplify the specification of oracles.
- **Pattern match:** Pattern match, focus on checking the presence of a set of model elements in the target models[50].

#### 2.3.3.2 M2T Transformation Testing

Table 2.5 shows the result of the first screening of studies in M2T transformation testing. Stuermer et al.[57] proposed a systematic testing architecture for code generators. The tool *ModeSSa* generates test models that are targeted at the optimization of the code generator. Polack et al.[48] proposed a unit testing framework for both M2M and M2T transformation. Fraternali et al.[20] proposed a model transformation framework for model-driven web application testing (a kind of M2T transformation). This paper focuses on solving the testing environment issues that come with this framework. Tiso et al.[58] proposed definitions of three adequacy criteria based on the concrete syntax of the input models. This study[58] also classified transformation tests into three categories: conformance tests, semantic tests, and textual tests. Chavez et al.[10] proposed an automated testing approach called CCUJ to test whether the Java implementations conform to their UML class models. Tiso et al.[59] proposed a unit testing framework for U-OWL M2T transformation. García et al. [21] proposed a debugging tool called HandyMOF for MOFScript transformation. Although it focuses on debugging, the tool can be used to measure the transformation coverage obtained by a test model suite. Wimmer et al.[68] proposed a generic metamodel for text to transform an M2T/T2M transformation specification problem into an M2M transformation specification problem. Based on this study, Burgueno et al.[8][6] extended the *tracts* approach from M2M transformation testing and defined a mechanism based on *matching tables*[6]. The *matching tables* aligns a model transformation implementation with its *tracts* which are the

[6]http://atenea.lcc.uma.es/projects/MTB.html

Table 2.5: The result of the first screening: M2T transformation testing

| Year | Title | Test Criteria | Model Transformation Language/Tool | Used Technologies | Evaluation for the Approach |
|---|---|---|---|---|---|
| 2007 | Systematic Testing of Model-Based Code Generators[57] | Model coverage and code coverage | TargetLink | ModeSSa, UML, TargetLink | Not mentioned |
| 2008 | Unit Testing Model Management Operations[48] | Not mentioned | EGL | Epsilon, EUnit | Not mentioned |
| 2010 | Multi-level Testes for Model Driven Web Applications[20] | Not mentioned | Groovy | BPMN, WebML, Canoo | Not mentioned |
| 2013 | A Method for Testing Model to Text Transformations[58] | Conformance, textual, semantic | Acceleo | UML | Not mentioned |
| 2013 | An Approach to Testing Java Implementation against Its UML Class-Model[10] | Branch-coverage | RSA | OCL, UML, RSA, Java | Compared with other approaches |
| 2014 | Unit Testing of Model to Text Transformations[59] | Not mentioned | U-OWL | MeDMoT, OWL | Not mentioned |
| 2014 | Testing MOFScript Transformations with HandyMOF[21] | Transformation coverage | MOFScript | Eclipse, | Not mentioned |
| 2014 | Back-To-Back Testing of Model-Based Code Generators[39] | Not mentioned | Genesys | Genesys | Not mentioned |
| 2015 | Testing M2M/M2T/T2M Transformations[6] | Constraint/rule/relatedness of constraints and rules coverage | ATL | TractsTool, Matching Tables Builder, USE | Precision and recall |

specifications of this model transformation. The *matching tables* can both detect and locate the fault in the transformation. Jörges et al.[39] proposed a back-to-back testing approach for the *Genesys* code generator. The main idea of the back-to-back testing is to execute an input model and its generated code and check whether the outputs of both executions are the same. If the outputs are the same, then it shows that the semantics of the input models are preserved. It requires that the input models can be executed by a model execution tool. However, this is not always available.

### 2.3.4 Further Selection and Prioritization

The second screening is to review the 36 studies in detail. We selected 11 studies that are worth further case studies. In this section, We summarize and classify the content of the 11 studies. We discuss the possibility of each group of approaches to improve the current testing practice at Altran. We prioritize the 11 studies based on the needs in the Altran practice.

**Test Coverage Criteria** The result of the second screening for Table 2.1 is shown in Table 2.6. The input test models of the ASOME project are created manually. There is no measurement on the metamodel coverage nor the model transformation coverage. There is no specification available for the M2M transformation. There is no coverage measurement on the M2T transformation either. This leaves the approaches using metamodel/transformation/specification coverage as possible improvement approaches. The test generation approaches in Table 2.1 are promising to alleviate this testing problem.

Some approaches in Table 2.1 do not directly apply to the ASOME project at Altran. The study [14] requires constraints on the target metamodel. The model transformation chain in the ASOME project does not have constraints on the target metamodels. The stakeholders decided not to create constraints for the target metamodels. Therefore, the study [14] will not be considered for further selection. The approach proposed by the study [46] depends on the language TGG. However, we do not use TGG in the model transformation chain at Altran. Using TGG can be time-consuming. Therefore, the study [46] will not be considered.

Table 2.6: The result of the second screening: test generation with coverage criteria

| Year | Title | Test Criteria | Model Transformation Language | Used Technologies | Evaluation for the Approach |
|---|---|---|---|---|---|
| 2006 | Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool[5] | Metamodel coverage | Not mentioned | MOF, OCL | Mutation analysis |
| 2009 | Automatic model generation strategies for model transformation testing[56] | Metamodel coverage | Not mentioned | Alloy | Mutation analysis |
| 2012 | ATLTest: A White-Box Test Generation Approach for ATL Transformations[26] | Transformation coverage | ATL | OCL | Not mentioned |
| 2013 | TETRABox - A Generic White-Box Testing Framework for Model Transformations[52] | Transformation coverage | Not mentioned | PAMOMO, QVT-Relations, OCL | Only observations |
| 2014 | A Search Based Test Data Generation Approach for Model Transformations[38] | Transformation coverage | ATL | MOTTER | Not mentioned |
| 2015 | Specification-driven model transformation testing[31] | Specification coverage | ATL/ETL | ocl2smt, PAMOMO, Eclipse, Z3 solver, USE, OCL | Mutation analysis |

**Equivalence Partitioning** The result of the second screening for Table 2.3 is shown in Table 2.7. The partitioning equivalence techniques aim at identifying similar test models and helping testers create an efficient test suite automatically. The study [36] will not be considered because this study does not explain how to define equivalence classes in Epsilon Pattern Language (EPL). The study [54] will not be considered because the VIATRA-Generator[7] used in the study [54] does not incorporate OCL constraints. However, Altran uses OCL invariants with the metamodels. This shows low compatibility with the input models at Altran.

[7]https://github.com/viatra/VIATRA-Generator/wiki

Table 2.7: The result of the second screening: equivalence partitioning

| Year | Title | Equivalence Partitioning Techniques | Model Transformation Language | Used Technologies | Evaluation for the Approach |
|---|---|---|---|---|---|
| 2014 | Test data generation for model transformations combining partition and constraint analysis[27] | OCL constraints analysis | Not mentioned | OCL, OCL-BBTesting EMF2CSP | Not mentioned |
| 2015 | Employing classifying terms for testing model transformations[25] | Classifying Terms | Not mentioned | ASSL, USE, OCL | Not mentioned |
| 2017 | Testing transformation models using classifying terms[7] | Classifying Terms | Not mentioned | OCL, USE | Performance analysis |
| 2018 | Testing models and model transformations using classifying terms[34] | Classifying Terms | Medini-QVT, JTL | USE, OCL, ASSL, Kodkod | Usability, performance and scalability |

**Mutation Analysis** The mutation analysis in Model transformation testing is used to assess the sensitivity of a test suite. There are three types of mutation operators proposed in the literature: syntactic operators, semantic operators, and ATL zoo operators. All of them require rich knowledge about the transformation language, ATL. However, the transformation languages used in the ASOME tool are QVTo and Xtend. Therefore, no mutation operators are available for QVTo and Xtend.

**Developing Oracle Functions** The availability of test oracles is a challenge in the literature but it has not been complained by the testers in the ASOME group. The ASOME project does not focus on testing the intermediate model transformation but the whole generator chain and the correctness of the generated C++ programs. Metamorphic testing in model transformation testing proposed in the literature is a new approach for the availability problem of test oracles. It aims at providing alternative oracles when there is no available oracle to verify the output. However, this technique is not mature enough for model transformation testing. There is a lack of studies on the applicability of this technique. Moreover, the oracle problem has not been requested with a high priority by the stakeholders. Therefore, this group of studies will not be considered for further investigation for now. However, if during the future experiment the oracle problem occurs, then we may reconsider the studies listed in Table 2.4.

**M2T Transformation Testing** Most of the M2T transformation testing in the literature focuses on the single M2T transformation, which means there is no transformation chain which includes M2M transformation. In these studies, the M2T transformation is also called a code generator. The code generator testing mentioned in these studies is close to the idea of compiler testing. The code generator testing approach [57] focuses on the optimization features of the generated code. However, code optimization is not part of the M2T transformation at Altran. The second difference is that the back-to-back testing[39] for the code generator does not apply to the M2T transformation at Altran because the input ASOME models are not executable. The M2T approach[6] extended the *tract*[63] approach from M2M transformation testing by creating a general metamodel for JAVA. According to the testers, when there is a failure in the generated code, the fault causing this failure often occurs in the M2T transformation. The M2T approach[6] looks into the structure of the generated code, whereas the practice in Altran is to run functional tests on the generated code. The general metamodel of JAVA is not practical for industrial cases of M2T transformations. But this will be considered for further investigation of its suitability on M2M transformations.

Table 2.8: The result of the second screening: M2T transformation testing

| Year | Title | Test Criteria | Model Transformation Language/Tool | Used Technologies | Evaluation for the Approach |
|---|---|---|---|---|---|
| 2015 | Testing M2M/M2T/T2M Transformations[6] | Constraint/rule/relatedness of constraints and rules coverage | ATL | TractsTool, Matching Tables Builder, USE | Precision and recall |

## 2.3.5 Prioritization

In this step, we prioritize the approaches from the last section considering the interests of all stakeholders and the feasibility of the studies. Table 2.9 shows the priority list of the 11 studies. Due to the time limitation, we can only implement 2 approaches in detail. We will focus on approaches 1 and 2 first and the other approaches in the list will be alternative options.

Approach 1 is aimed at testing transformations using tracts. The tracts can specify what to expect in the target model according to the source model using OCL. This addresses the need for increasing the quality of testing at Altran.

Approach 2 proposed and used the idea of classifying terms to generate a set of models automatically. This approach addressed the problem of manual modeling at Altran. Moreover, this addresses the problem of redundant testing. Classifying terms define partitions of generated models. Each partition will have exactly one satisfiable model.

If approaches 1 and 2 are found out to be completely infeasible during the early implementation (e.g. the tool is damaged or the example provided does not work), we will report this and try the next approach in the list.

Table 2.9: The priority list of the 11 approaches

| Order | Keywords | Year | Title | Used Technologies |
|---|---|---|---|---|
| 1 | tracts | 2015 | Testing M2M/M2T/T2M Transformations [6] | TractsTool, Matching Tables Builder, USE |
| 2 | classifying terms | 2015 | Employing classifying terms for testing model transformations [25] | ASSL, USE, OCL |
| | | 2017 | Testing transformation models using classifying terms [7] | OCL, USE |
| | | 2018 | Testing models and model transformations using classifying terms [6] | USE, OCL, ASSL, Kodkod |
| 3 | specification-driven testing | 2015 | Specification-driven model transformation testing [31] | ocl2smt, PAMOMO, Eclipse, Z3 solver, USE, OCL |
| 4 | source constraint analysis | 2014 | Test data generation for model transformations combining partition and constraint analysis [27] | OCL, OCLBBTesting, EMF2CSP |
| 5 | transformation coverage | 2014 | A Search Based Test Data Generation Approach for Model Transformations[38] | MOTTER |
| 6 | | 2013 | TETRABox - A Generic White-Box Testing Framework for Model Transformations[52] | PAMOMO, QVT-Relations, OCL |
| 7 | | 2012 | ATLTest: A White-Box Test Generation Approach for ATL Transformations[26] | OCL |
| 8 | metamodel coverage | 2009 | Automatic model generation strategies for model transformation testing[56] | Alloy |
| 9 | | 2006 | Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool[5] | MOF, OCL |

## 2.4 Reflection on Interviews and the Literature Review

Based on the interviews at Altran and the literature review, we have several observations.

**Lack of industrial examples in the literature** Most of the cases are simple examples. However, the industrial model transformation chain is way more complicated than those simple examples in the literature. This can be an obstacle for testers in the industry to adopt the approaches in literature because the simple examples do not address some problems that may occur in industrial testing. For example, all the test model generation approaches only consider simple M2M transformation examples which only take one type of test model as input. However, at Altran, the M2M transformation takes both the ASOME model and the generator model as input. No studies have proposed a solution taking this into consideration.

**Lack of knowledge about the causes of faults** There is a lack of knowledge of why these faults were not detected during testing. If we have this knowledge, we can know better which approaches in the literature to choose and the researchers may have a clearer research direction. For example, if we know the most common faults, then we can adapt the mutation operators in [28].

**Lack of knowledge about the current test coverage at Altran** There is no formal measurement of the test coverage, which makes it difficult to choose an appropriate approach from the literature. Since there is no coverage information, we do not know whether it is the low test coverage that hurts the effectiveness of the current testing at Altran. For example, if the metamodel coverage is actually high but the testing still misses out a significant number of faults, then using the metamodel coverage approaches will be insufficient.

## 2.5 Evaluation Criteria for the Selected Approaches

We define four criteria to evaluate the suitability of the selected approaches. These criteria are not quantifiable yet since there is no tool that can be directly applied to industrial cases. We can only qualitatively analyze the suitability of the selected approaches with these criteria.

- Criterion 1: The tool used in the proposed approach(es) can run without errors in small case studies.
- Criterion 2: The tool can handle the industrial-size model transformations.
- Criterion 3: The total time required by the proposed approaches should be less than the time required by the current practice of model transformation testing at Altran. For the same set of features of a model transformation, the proposed approach should require less time compared to the current model transformation process described in Section 2.2.3. The total time consists of two parts.

  $$T_{total} = T_{preparation} + T_{execution}$$

  The first part is the time spent in preparation before running the tool. The second part is the execution time for the tool. The preparation time is usually the time spent in making the input files acceptable for the tool. The execution time is the duration from starting the execution of the tool until getting the results.
- Criterion 4: The proposed approach has the potential to improve the current practice of model transformation testing. There are two ways to improve current testing practice. First, according to the interview results with the Altran testers, it is a challenge to generate a qualified test suite. The current testing method often fails to detect faults in the transformations and these faults cause failures in the generated code. Therefore, the proposed approach is expected to assist testers in detecting faults in the transformations that human testers fail to detect. Second, according to the interview results, testers create structurally similar test models by mistake. Moreover, manual modeling is a time-consuming job. Therefore, the proposed approach should improve the efficiency of the current practice.

# Chapter 3

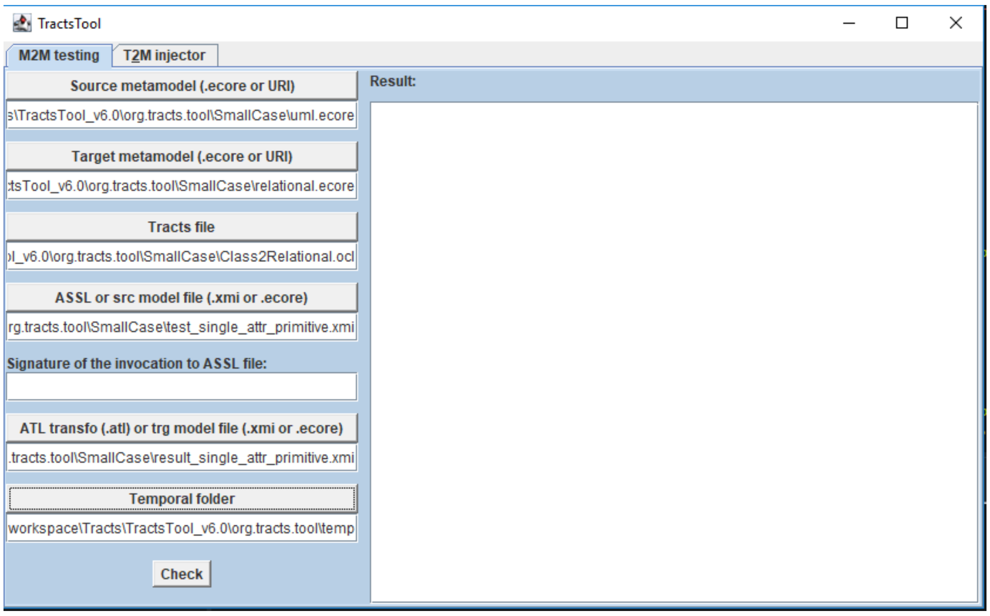
# TractsTool and Matching Table Builder

In this chapter, we will introduce and evaluate two tools, TractsTool and Matching Table builder. These tools make use of the same approach, tract[6][24]. A tract is a transformation contract. It describes the logic of transformations and allows testers to verify M2M transformation formally. For both TractsTool and Matching Table Builder, testers need to specify a tract constraint file that makes use of the knowledge of source metamodels and target metamodels.

## 3.1 TractsTool

### 3.1.1 Introduction of TractsTool

TractsTool is a tool that checks the correctness of the implementation of an M2M transformation with the tracts. Tracts specify what to expect in the target model according to the source model. The tracts are specified in OCL. Figure 3.1 shows the GUI of TractsTool. TractsTool is language independent. To test ATL transformations, TractsTool needs to take the source and target metamodels, tracts, an ATL transformation, and an ASSL script as input. For ATL transformations, TractsTool can generate source models using the ASSL script. For non-ATL transformations, TractsTool needs to take the source and target metamodels, tracts, source, and target models as input. The main idea of TractsTool is to transform the source metamodel and target metamodel into one single USE (UML-based Specification Environment) specification file. TractsTool also needs to generate source and target models in USE. TractTool uses the USE tool as an engine for the evaluation of the OCL expressions in the tracts and checks if the models satisfied the tracts. TractsTool will then return evaluation results.



Source metamodel (.ecore or URI)
s\TractsTool_v6.0\org.tracts.tool\SmallCase\uml.ecore

Target metamodel (.ecore or URI)
:tsTool_v6.0\org.tracts.tool\SmallCase\relational.ecore

Tracts file
ol_v6.0\org.tracts.tool\SmallCase\Class2Relational.ocl

ASSL or src model file (.xmi or .ecore)
rg.tracts.tool\SmallCase\test_single_attr_primitive.xmi

Signature of the invocation to ASSL file:

ATL transfo (.atl) or trg model file (.xmi or .ecore)
.tracts.tool\SmallCase\result_single_attr_primitive.xmi

Temporal folder
workspace\Tracts\TractsTool_v6.0\org.tracts.tool\temp

Figure 3.1: TractsTool GUI

### 3.1.2 Case Study

In this section, a case study of applying them on a small transformation called UML2Relational, and the evaluation of TractsTool's suitability will be present.

#### 3.1.2.1 Introduction of UML2Relational transformation case

We defined one source metamodel called UML, One target metamodel called Relational, and a QVTo transformation. This small case was used in the small case studies of the four tools. The source language is a small set of UML and the target language is a simple relational database. The QVTo transformation can transform a UML model into a relational database model. The transformation rules are defined in the QVTo transformation shown in Appendix B.1.

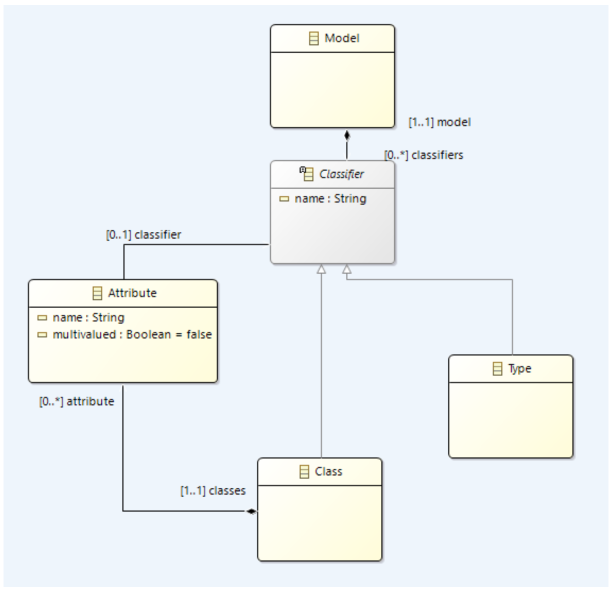


Figure 3.2: uml.ecore shown in Eclipse

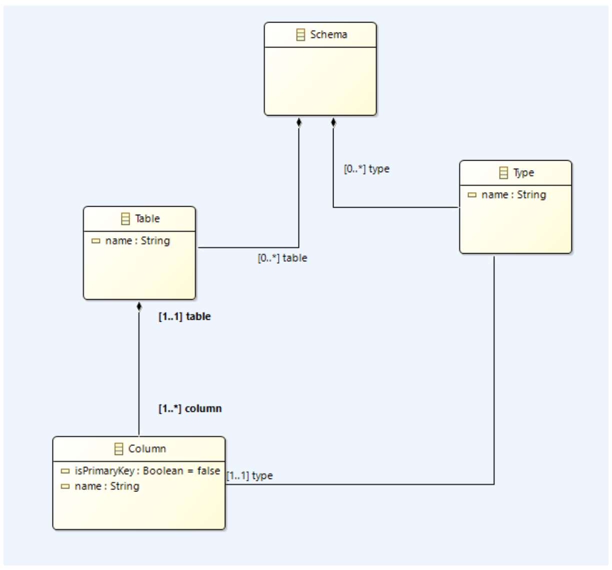


Figure 3.3: relational.ecore shown in Eclipse

#### 3.1.2.2 Implementation of the UML2Relational case study using TractsTool

We first need to check whether this tool can work without errors and detect the failure in this QVTo transformation. We need to write tracts to capture the logic of mapping rules and then inject defects in the target model. The tool should return a negative result.

To simplify the problem, a rule called `type2type()` in the QVTo transformation is taken as an example (The code is listed in Listing 3.1). This transformation rule maps each primitive type in the source model to a primitive type in the target model.

```
mapping UML::Type::type2type() : RELATIONAL::Type{
    name := self.name;
}
```

Listing 3.1: Mapping rule type2type in UML2Relational.qvto

Therefore, in the tracts file (The code is listed in Listing 3.2), we define that all instances of the source types should find at least one type with the same name in the target model. The naming of elements in the tract is different from the QVTo mapping rule. To distinguish the source metamodel from the target metamodel, TractsTool will add prefixes to the names of the elements in both metamodels. For example, TractsTool will rename `Type` in the source metamodel to `src_Type` and the `Type` in the target metamodel as `trg_Type`. The tract for the `type2type()` is shown in Listing 3.2.

```
context src_Type inv Type2Type:
src_Type.allInstances->forAll(t_src|trg_Type.allInstances->exists(t_trg|
t_trg.name = t_src.name))
```

Listing 3.2: Tract for type2type()

Fig. 3.4 shows a source model and a target model. As can be seen in Fig. 3.5, the types in the target model (relational.xmi) are different from the ones in the source model (uml.xmi). We added this defect deliberately and we expected that the TractsTool could detect this failure.

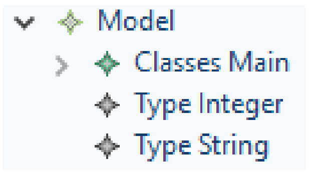


Figure 3.4: uml_model.xmi shown in Eclipse

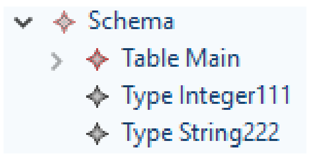
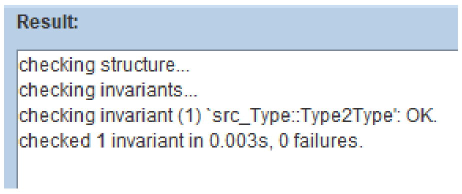


Figure 3.5: relational_model.xmi shown in Eclipse

#### 3.1.2.3 Results

The result in Figure 3.6 shows that the TractsTool failed to detect the failure. Listing 3.3 shows the generated commands. These commands are for transforming the source model and the target model into models in USE. After analyzing the commands generated by the tool, we found that the tool invoked USE and used commands to generate 2 source types and 2 target types. The two source types are named as "Type_102" and "Type_103". The two target types are named as "Type_108" and "Type_109".



```
Result:
checking structure...
checking invariants...
checking invariant (1) `src_Type::Type2Type': OK.
checked 1 invariant in 0.003s, 0 failures.
```

Figure 3.6: Results

However, the names of the source and target types are all 'null'. We traced back to the commands that TractsTool generated. We found that all the attributes were assigned with null as a value by the commands. This is the reason why the source type and the target type were considered the same and the tool failed to detect the transformation fault. This also means TractsTool can not detect other kinds of faults as well since it will set every attribute value as 'null'.

```
!new src_Model ('Model_99')
!new src_Classes ('Classes_100')
!new src_Attribute ('Attribute_101')
!new src_Type ('Type_102')
!new src_Type ('Type_103')
!insert (@Model_99,@Classes_100) into src_classes_Classes_Model
!insert (@Model_99,@Type_102) into src_type_Model_Type
!insert (@Model_99,@Type_103) into src_type_Model_Type
!@Classes_100.name := null
!insert (@Classes_100,@Attribute_101) into src_attribute_classes_Attribute_Classes
!@Attribute_101.name := null
!@Attribute_101.multivalued := null
!insert (@Attribute_101,@Type_103) into src_classifier_Attribute_Classifier
!@Type_102.name := null
!@Type_103.name := null
!new trg_Schema ('Schema_104')
!new trg_Table ('Table_105')
!new trg_Column ('Column_106')
!new trg_Column ('Column_107')
!new trg_Type ('Type_108')
!new trg_Type ('Type_109')
!insert (@Schema_104,@Table_105) into trg_table_Schema_Table
!insert (@Schema_104,@Type_108) into trg_type_Schema_Type
!insert (@Schema_104,@Type_109) into trg_type_Schema_Type
!@Table_105.name := null
!insert (@Table_105,@Column_106) into trg_column_table_Column_Table
!insert (@Table_105,@Column_107) into trg_column_table_Column_Table
!@Column_106.isPrimaryKey := null
!@Column_106.name := null
!insert (@Column_106,@Type_108) into trg_type_Column_Type
```

```
!@Column_107.isPrimaryKey := null
!@Column_107.name := null
!insert (@Column_107,@Type_109) into trg_type_Column_Type
!@Type_108.name := null
!@Type_109.name := null
check -d
```

Listing 3.3: Generated commands

### 3.1.3 Discussion on TractsTool

**TractsTool's capability to detect faults depends on the quality of the tracts given by testers** The challenge that the testers in the industry are facing is that an unacceptable number of transformation faults are not detected during the model transformation testing. It means that human testers are not able to make a comprehensive and qualified test set so that all the faults can be detected. However, TractsTool's capability of fault detection is completely based on the quality (such as completeness and correctness) of the tracts given by testers. If the user makes a simple tract file, TractsTool will return positive results. But it does not necessarily mean that the transformation under test is free of faults. Therefore, TractsTool's capability to detect faults depends on the quality of the tracts given by testers.

**Inconsistent naming in the tract file.** There is a naming inconsistency problem in the tract file. The generated USE file for UML2Relational case is shown in Appendix B.3. It will add a prefix to the name of every element in the Ecore metamodel to distinguish between the source metamodel and the target metamodel. Therefore, testers need to be careful about this naming difference among generated USE specifications and Ecore metamodels when writing a tract. It is a problem from a practical point of view but can be solved by further development.

**TractsTool still contains defects.** TractsTool will generate a command file to generate runtime instances in USE according to the source models and target models. However, it assigned null to all the attributes as a value. Therefore, the current version of TractsTool still contains defects.

**TractsTool may solve the oracle problem in M2M transformations** As mentioned in Section 2.2.2, the oracle function will compare the generated target model with an expected model. TractsTool may be useful when there is no available expected model or the available expected model is not reliable. The expected model is verified manually in the testing. This model will be used as the expected transformation result. But it is only verified manually. However, if the tester makes a mistake and takes a wrong model as an expected model, the whole testing process will have problems and the faults in the transformation may remain undetected. Therefore, TractsTool may serve a role in checking the models formally so that the testing can be less error-prone.

### 3.1.4 Conclusions

Table 3.1 summarizes the suitability evaluation of TractsTool. As can be seen from the table, TractsTool did not pass the small case study and we do not conduct a further industrial case study on TractsTool. Therefore, TractsTool does not satisfy the first two criteria and for criterion

3 we do not have the answer since there is no further industrial case study. For criterion 4, the result is positive because TractsTool has the potential to solve the oracle problem.

Table 3.1: Evaluation results of TractsTool

| No. | Criteria | Results (Yes or No) | Reasons |
|---|---|---|---|
| 1 | The tool used in the proposed approach(s) can run without errors in the small case studies. | No | TractsTool still contains defects in its program. |
| 2 | The tool can handle industrial-size model transformations | No | TractsTool does not satisfy criterion 1. |
| 3 | We did not use TractsTool for industrial case studies. Therefore, we did not measure. | NA | We do not know what will take longer: writing a set of tracts or creating a set of input test model and expected result model. |
| 4 | The proposed approach has the potential to improve the current practice of model transformation testing. | Yes | TractsTool may solve the oracle problem in the M2M transformations. |

## 3.2 Matching Table Builder

### 3.2.1 Introduction of Matching Table Builder

The Matching Table Builder (MTB) is an ATL-dependent model transformation testing tool. Like TractsTool, the goal of MTB is also to check the correctness of transformations. The logic of transformations also needs to be expressed in OCL constraints. These constraints are similar to tracts in TractTool. What is different from TractsTool is that it does not generate models in USE. Instead, it requires the user to first extract information of the source and target metamodels, called *types*, in the ATL Transformation Types Extractor (shown in Figure 3.7). Since the ATL Transformation Types Extractor can only work for ATL transformations, we can not directly apply it to our QVTo transformations. Then the information of the source and target metamodels should be taken as the input of the MTB. The GUI is shown in Figure 3.8. The MTB will check the ATL rules according to the OCL constraints.

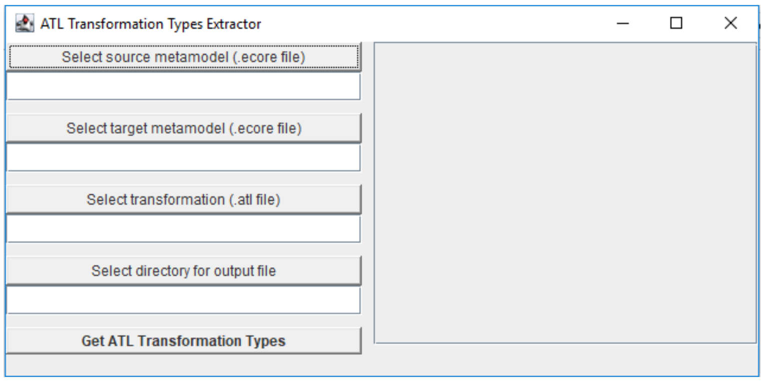
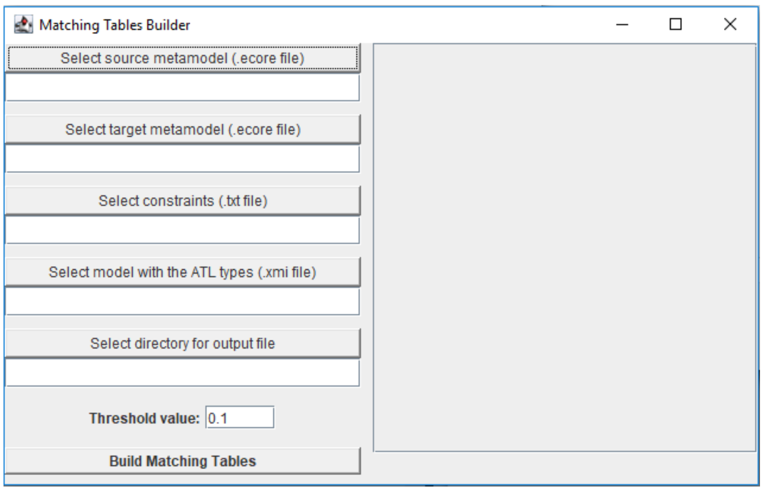

ATL Transformation Types Extractor
Select source metamodel (.ecore file)
Select target metamodel (.ecore file)
Select transformation (.atl file)
Select directory for output file
Get ATL Transformation Types

Figure 3.7: ATL Transformation Types Extractor GUI


Matching Tables Builder
Select source metamodel (.ecore file)
Select target metamodel (.ecore file)
Select constraints (.txt file)
Select model with the ATL types (.xmi file)
Select directory for output file
Build Matching Tables

Threshold value: 0.1

Figure 3.8: Matching Table Builder GUI

### 3.2.2 Discussion on Matching Table Builder

**MTB works with ATL transformations but there are no conceptual and technical obstacles to adapt it to QVTo** In the ASOME tool, the M2M transformation language is QVTo. Therefore, we can not conduct a simple case study nor the industrial study. MTB can not be directly applied to non-ATL transformations but in principle can be adapted to QVTo because we have traces produced as a result of the transformation execution. From these traces, we can extract QVTo transformation types.

**MTB's capability to detect faults depends on the quality of OCL constraints** In the context of model transformation testing, the purpose of testing is to detect faults in the model transformations. The tool should help the tester to detect faults in the transformations which are hard to detect. However, whether MTB can detect faults in the transformation fully depends on the quality of the OCL constraints. If the tester writes a simple OCL constraint, MTB will return the results as positive while the transformation still contains faults. The MTB can check whether the transformation is expected by the tester.

### 3.2.3 Conclusion

In conclusion, we will not conduct an industrial case study with MTB. Similarly like TractsTool, MTB uses OCL contraints to verify transformations formally. It can be a solution of the oracle problems but the adaption of the tool is needed. Table 3.2 shows the suitability evaluation results of MTB according to the four criteria and the reasons are listed in the table.

Table 3.2: Evaluation results of MTB

| No. | Criteria | Results (Yes or No) | Reasons |
|---|---|---|---|
| 1 | The tool used in the proposed approach(s) can run without errors in the small case studies. | NA | The example provided by the author can run without errors as shown in the tool's webpage[1]. But MTB does not support QVTo transformations. Therefore, we can not make a judgment on this criterion. |
| 2 | The tool can handle industrial-size model transformations | NA | MTB does not support non-ATL transformations. Therefore, we can not make a judgment on this criterion. |
| 3 | The total time required by the proposed approaches should be less than the time required by the current practice of model transformation testing at Altran. | NA | MTB does not support non-ATL transformations. Therefore, we can not make a judgment on this criterion. |
| 4 | The proposed approach has the potential to improve the current practice of model transformation testing. | Yes | MTB can be a solution for oracle problems and it can be adapted for QVTo transformations. |

[1]http://atenea.lcc.uma.es/projects/MTB.html

# Chapter 4

# UML-based Specification Environment and Efinder

In this chapter, we will introduce and evaluate two tools that realize automatic model generation for equivalence partitioning. We will first introduce USE and then Efinder. Efinder extends USE and makes it more suitable to Eclipse Modeling Framework.

## 4.1 The UML-based Specification Environment

### 4.1.1 Introduction to UML-based Specification Environment

The UML-based Specification Environment (USE) is a system that can generate models by solving OCL constraints automatically. USE is based on a subset of Unified Modeling Language (UML). Main features[1] of USE are:

- The user can create UML models on the GUI by drag-and-drop or by commands or by an ASSL script.
- USE can evaluate OCL expressions for the model and return values of the OCL expressions.
- USE has a plug-in called "model validator". Although the name of this plug-in is "model validator", its main function is to generate instance models according to the OCL constraints specified by the user.

With these features, USE can generate and partition models using classifying terms. The concept of classifying terms is introduced in the following section.

[1] http://useocl.sourceforge.net/w/index.php/Quick_Tour

### 4.1.2 Introduction of Classifying Terms

Classifying terms are OCL expressions that describe features of models. Classifying terms with values are OCL constraints. These values construct a characteristic value. Characteristic values are used to identify partitions. OCL invariants and classifying terms with characteristic values are both OCL constraints.
The number of partitions is

$$2^N,$$

where N is the total number of classifying terms. In the following test case, N is 3. The number of partitions is 8. Studies[6][7][25] proposed the idea of classifying terms based on USE. Classifying terms are used to construct a characteristic value that identifies an equivalence partition. For each partition, USE can generate an instance model.

### 4.1.3 Case Study

As we are interested in the model validator of USE, we conducted a simple case study to check if the model validator could generate a set of instance models with classifying terms. Given Ecore metamodels, we used the tool to generate instance models such that every partition will have one model. We used the previously-defined source metamodel from UML2Relational case (see Figure 3.2).

**Implementation steps:**

- Translate the Ecore metamodels into USE specifications'. The complete USE specification file consists of the information of metamodels, OCL invariants shared by all instance models, and classifying terms with characteristic values.
- Specify the OCL invariants of the metamodel in the USE specification file. These OCL invariants are shared by all generated models. They are not classifying terms.
- Configure the model properties in the GUI of the Model Validator.
- Start the generation and get the instance model in the object diagram view.

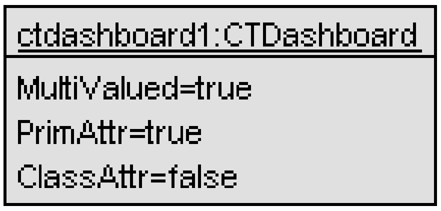
**USE Specifications** USE only takes USE specifications as input. Therefore, the source metamodel and the OCL constraints were translated into USE specifications in one single USE file (shown in Appendix B.2). USE does not have this translation function. This translation was done by TractsTool.

**Classifying term** In this small case, we define three boolean Classifying terms: MultiValued, PrimAttr, and ClassAttr. A combination of the boolean values constructs a characteristic value for an equivalence partition. If MultiValued is true, it means there exists at least one multivalued attribute. If PrimAttr is true, it means there exists at least one attribute whose type is a primitive type. If ClassAttr is true, it means there exists at least one attribute whose type is a class type. We defined a classifying term dashboard in the USE specification.

```
class CTDashboard
attributes
  MultiValued: Boolean
  PrimAttr: Boolean
  ClassAttr: Boolean

operations
  MultiValued_OP(): Boolean = Attribute.allInstances->exists(a | a.multivalued =
      true)
  PrimAttr_OP(): Boolean = Attribute.allInstances->exists(a | a.classifier.
      oclIsTypeOf( Type))
  ClassAttr_OP(): Boolean =  Attribute.allInstances->exists(a | a.classifier.
      oclIsTypeOf( Classes))
end

context CTDashboard inv CT_Operation :
  MultiValued = MultiValued_OP() and PrimAttr = PrimAttr_OP() and ClassAttr =
      ClassAttr_OP()
```

Listing 4.1: Classifying term dashboard

This dashboard is simply an indicator of characteristic value and will indicate which equivalent partition that the model belongs to. Figure 4.1 shows an example how the dashboard present in the GUI. In this example, the characteristic value is 110 (binary).


ctdashboard1:CTDashboard

MultiValued=true
PrimAttr=true
ClassAttr=false

Figure 4.1: Classifying term dashboard shown in USE GUI

#### 4.1.3.1 Invariants

The OCL invariants of the metamodel should also be included in USE specifications. We specified some basic constraints. The explanations are shown as follows. some of the invariants are part of the language definition (for example, type names are unique) whereas some are just for experimentation (for example, there must be at least 2 types).

- An generated instance should have exactly one `Model` .

```
context Model inv Model_Size:
  Model.allInstances->size() = 1
```

Listing 4.2: constraints for Model

- There should be at least 2 `Type`s.
- There should be no `Type`s with the same name.

```
context Type
  inv Type_Size:
    Type.allInstances()->size() >= 2
  inv Type_Name:
    Type.allInstances()->exists( a, b | (a.name = b.name) and a <> b) = false
```

Listing 4.3: constraints for Type

- There should be at least one `Classes`.
- There should be no `Classes` with the same name.
- There should be no `Type` with the same name as any `Classes`.
- There should be no `Attributes` with the same name as any `Classes`.

```
context Classes
inv Classes_Size:
Classes.allInstances()->size() >= 1
inv Unique_ClassesName_AmongClasses:
Classes.allInstances()->exists( a, b| (a.name = b.name) and a <> b) = false
inv Unique_ClassesName_AmongType:
Classes.allInstances()->exists( a| Type.allInstances()->exists(b|a.name = b.
    name)) = false
inv Unique_ClassesName_AmongAttribute:
Classes.allInstances()->exists( a| Attribute.allInstances()->exists(b|a.name =
     b.name)) = false
```

Listing 4.4: constraints for Classes

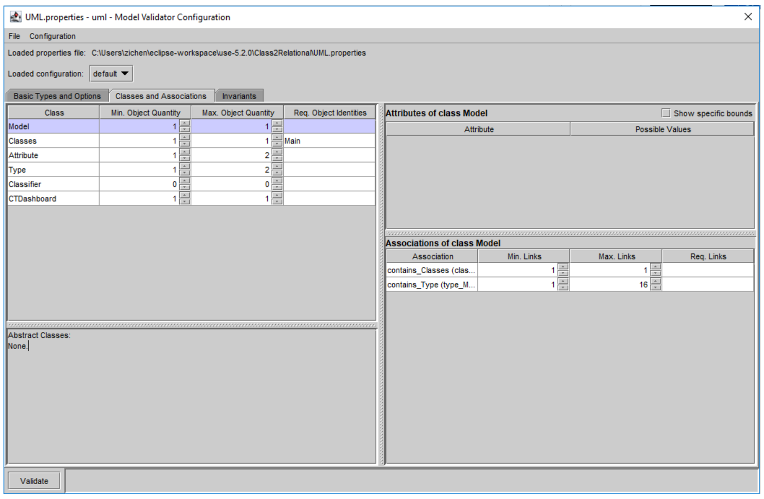
- The number of `Attributes` should be no less than the number of `Classes`.
- There should be no `Attributes` with the same name.

```
context Attribute
    inv Attribute_Size:
      Attribute.allInstances->size() >= Classes.allInstances->size()
    inv Unique_AttrName_AmongAttr:
        Attribute.allInstances()->exists( a, b| (a.name = b.name) and a <> b)
            = false
```

Listing 4.5: constraints for Attribute

### 4.1.3.2 Configuration in USE

USE can generate instance models based on the specified constraints and boundaries. USE provides a GUI for the user to configure the generation of all elements. If the expected value of a `Classes` is "Main" as shown in Fig.4.2, then USE will generate the Classes with the name 'Main'. Multiple expected values can be specified in the configuration. If the field is left blank, then USE will generate random strings.


UML.properties - uml - Model Validator Configuration

Loaded properties file: C:\Users\zichen\eclipse-workspace\use-5.2.0\Class2Relational\UML.properties

Loaded configuration: default

| Class | Min. Object Quantity | Max. Object Quantity | Req. Object Identities |
|---|---|---|---|
| Model | 1 | 1 | |
| Classes | 1 | 1 | Main |
| Attribute | 1 | 2 | |
| Type | 1 | 2 | |
| Classifier | 0 | 0 | |
| CTDashboard | 1 | 1 | |

Abstract Classes:
None

Attributes of class Model

| Attribute | Possible Values |
|---|---|

Associations of class Model

| Association | Min. Links | Max. Links | Req. Links |
|---|---|---|---|
| contains_Classes (clas... | 1 | 1 | |
| contains_Type (type_M... | 1 | 16 | |

Figure 4.2: Configuration

### 4.1.3.3 Results

USE generated 8 solutions shown in Fig. 4.3. Each solution belongs to a partition. The characteristic values of each partition are shown in Table 4.1. This case shows that USE can generate instance models according to the OCL constraints. However, this tool can only export the generated models as PDF files. Therefore, we did not conduct an industrial case study on USE.

Table 4.1: The characteristic values of partitions

| MultiValued | PrimAttr | ClassAttr |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

### 4.1.4 Discussion on USE

**USE can generate instance models according to the OCL constraints** The most important feature is that USE can give a solution for a set of OCL constraints and generate run-time instances. This feature is useful when testers need to make test models with combinations of different features.

**Limitations of USE** The metamodel needs to be translated into USE specifications first but USE does not have this feature. Moreover, USE can only save the instance models as a PDF file therefore there is no native export mechanism to EMF.

### 4.1.5 Conclusions

Table 4.2 summarizes the evaluation results of USE and the reasons. In conclusion, USE has the potential to improve the current testing process at Altran. We did not experience major errors. But due to the two problems mentioned above and the fact that a tool named Efinder[12] has been recently developed, we decided not to conduct a further industrial case study with USE. We decided to explore Efinder's suitability instead because it mitigated the limitation of USE by offering better integration with EMF.

Table 4.2: Evaluation results of USE

| No. | Criteria | Results (Yes or No) | Reasons |
|---|---|---|---|
| 1 | The tool used in the proposed approach(s) can run without errors in the small case studies. | Yes | According to the small case study, USE can perform without errors. |
| 2 | The tool can handle industrial-size model transformations | No | USE does not accept Ecore metamodels and can not export models as Ecore models. |
| 3 | The total time required by the proposed approaches should be less than the time required by the current practice of model transformation testing at Altran. | NA | We can not make a judgment on this criterion because we can not apply it to the industrial case study. |
| 4 | The proposed approach has the potential to improve the current practice of model transformation testing. | Yes | The classifying term approach using USE model validator is to reduce the workload of manual modeling so that it can be easier for testers to generate a qualified test suite. The idea of classifying terms can help avoid structurally similar test models. |

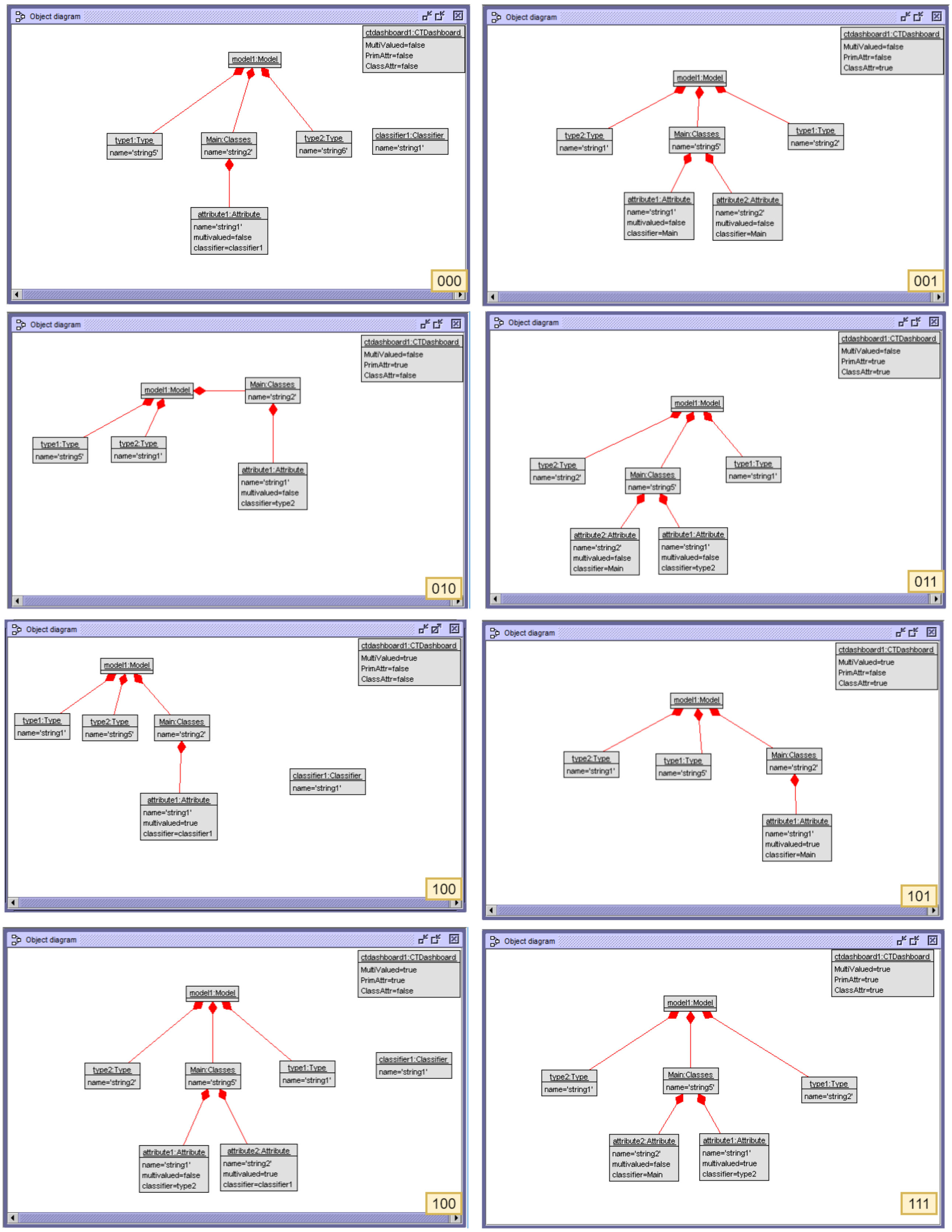


Figure 4.3: Generated models

## 4.2 Efinder

### 4.2.1 Introduction of Efinder

USE is a tool that can be used to solve constraints and generate UML models. As shown in Section 4.1, USE can not export the models as XMI files and does not accept Ecore metamodels. Efinder[12] has solved these two problems of USE. Efinder is a USE-based tool. It has two new features. The first one is that it can transform Ecore metamodels and OCL constraints into USE specifications. An example of the translated USE specifications is shown in Appendix B.4. The second one is that it can save the instance models as XMI data.

### 4.2.2 Case Study: Generating multiple instance models

The goal of this case study is to generate a set of models using classifying terms. We used the UML metamodel from the UML2Relational case that we previously defined (shown in Figure 3.2). Each model should belong to a partition identified by a characteristic value.
In general, the implementation of this case study contains two steps :

- Generate a set of OCL files using the script.
- Run Efinder with each OCL file and generate an instance model for a partition if satisfiable.

#### 4.2.2.1 Generating OCL files

For each execution, Efinder takes an OCL file as input. It will then automatically transform the corresponding Ecore metamodels into USE specifications and generate exactly one model. Therefore, to generate multiple models, we need to prepare multiple OCL files. Each OCL file is corresponding to a partition. The number of partitions is the same as the number of OCL files. The number of OCL files is the same as the number of partitions. In this test case, The number of OCL files to be prepared is 8. The OCL files share the same OCL invariants. Different OCL files have the same classifying terms but with different characteristic values. To automate this generation, a script was developed (in Appendix B.5) to generate OCL constraints using boolean-type classifying terms.
There are different types of classifying terms. If a classifying term does not have a boolean value, we formulate it in a way that it ends with boolean values. For example, Listing 4.6 shows an integer-type classifying term. If we need to generate partitions for models that their entity sizes are from entity size 1 to entity size 1 to 3. In the meantime, there are other types of classifying terms. To simply the generation of OCL files, we need to write integer-type classifying terms like boolean-type classifying terms. Listing 4.7 shows two boolean-type classifying terms, `Entity_Size1` with

boolean value `true` and `Entity_Size2` with boolean value `false`. The characteristic value (binary) is 01. The script in Appendix B.5 will assign these two classifying terms characteristic values from 00 to 11. When the characteristic value is 11, the model generation will fail because the number of entities can not be true at the same time. In this case, Efinder will skip the generation. Only satisfiable models will be generated.

```
Context Entity
 inv Entity_Size:
    Entity.allInstances()->size() = 1
```

Listing 4.6: an integer-type classifying term

```
Context Entity
 inv Entity_Size1:
    Entity.allInstances()->size() = 1 = true
 inv Entity_Size2:
    Entity.allInstances()->size() = 2 = false
```

Listing 4.7: an integer-type classifying term is rewritten into two boolean-type classifying terms

All classifying terms will be given a boolean value during this OCL file generation. All classifying terms under the same context need to be specified in the same text file line by line. For example, Listing 4.8 shows the content of classifying terms. These classifying terms will be given a boolean value and will be inserted into an ocl file by the script.

```
Attribute.allInstances()->exists(a | a.multivalued = true)
Attribute.allInstances()->exists(a | a.classifier.oclIsTypeOf( Type))
Attribute.allInstances()->exists(a | a.classifier.oclIsTypeOf( Class))
```

Listing 4.8: 3 classifying terms

Listing 4.9 shows an example of a full OCL file. It specifies the file path of the Ecore metamodel, invariants, and classifying terms with a characteristic value. The characteristic value is 000 (binary) in this file. The "- -" is the comment symbol in OCL.

```
import 'uml.ecore'
package uml
context Model inv Model_Size:
  Model.allInstances()->size() = 1

context Type
  inv Type_Size:
    Type.allInstances()->size() >= 2
  inv Type_Name:
    Type.allInstances()->exists( a, b | (a.name = b.name) and a <> b) = false -- there should be no types with the same name

context Class
  inv Class_Size:
    Class.allInstances()->size() >= 1
  inv Unique_ClassName_AmongClass:
    Class.allInstances()->exists( a, b| (a.name = b.name) and a <> b) = false -- there should be no Class with the same name.
  inv Unique_ClassName_AmongType:
    Class.allInstances()->exists( a| Type.allInstances()->exists(b|a.name = b.name)) = false -- there should be no Type with the same name as any Class.
  inv Unique_ClassName_AmongAttribute:
    Class.allInstances()->exists( a| Attribute.allInstances()->exists(b|a.name = b.name)) = false -- there should be no Attribute with the same name as any Class.

context Attribute
  inv CT1:
  Attribute.allInstances()->exists(a | a.multivalued = true) = false
  inv CT2:
  Attribute.allInstances()->exists(a | a.classifier.oclIsTypeOf( Type)) = false
  inv CT3:
  Attribute.allInstances()->exists(a | a.classifier.oclIsTypeOf( Class)) = false
  inv Attribute_Size:
    Attribute.allInstances()->size() >= Class.allInstances()->size()
  inv Unique_AttrName_AmongAttr:
      Attribute.allInstances()->exists( a, b| (a.name = b.name) and a <> b) = false -- there should be no Class with the same name.

endpackage
```

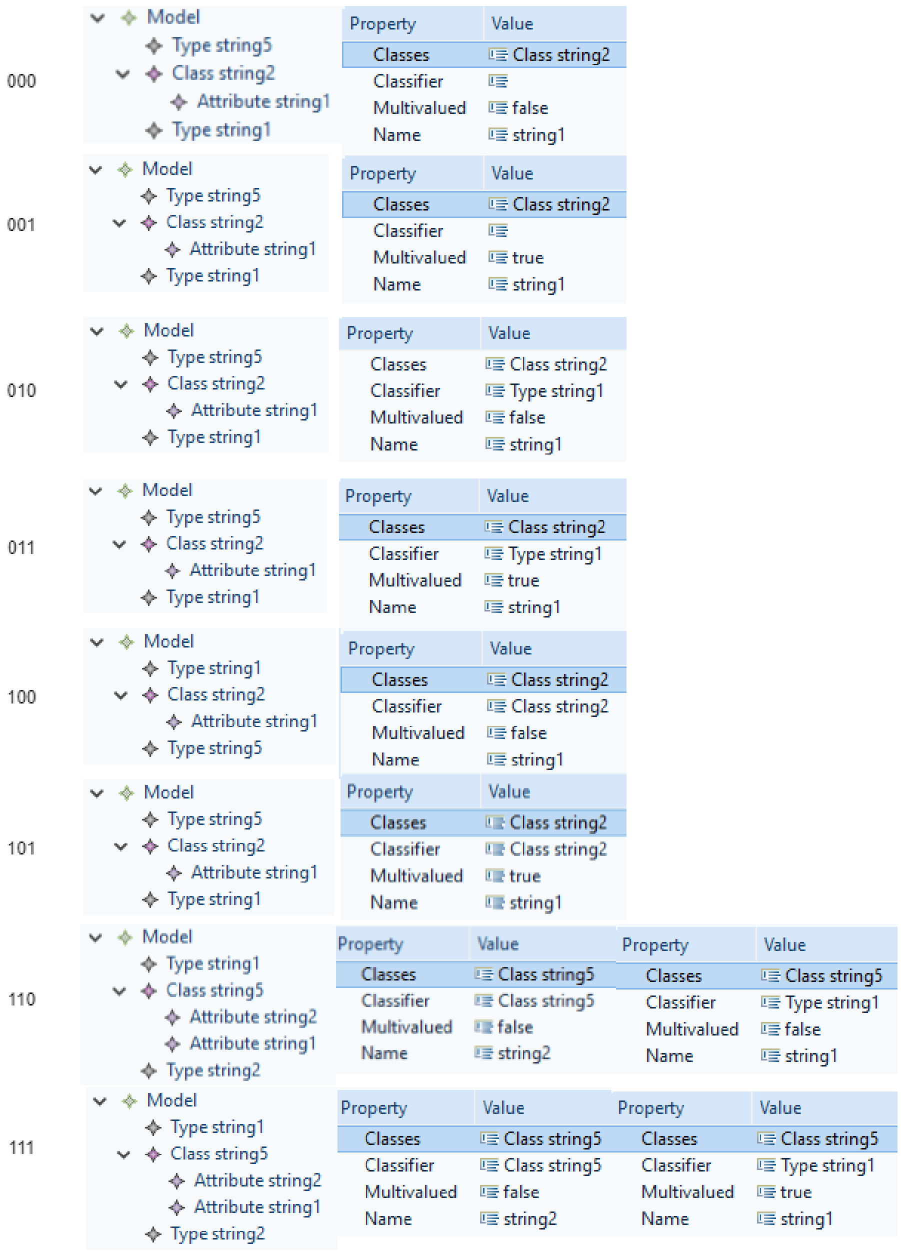
Listing 4.9: The full OCL file with characteristic value 000

#### 4.2.2.2 Configure and Run the Tool

The main test code is shown in Listing B.6. The model configurations are located from line 26 to line 30 in Listing B.6. The user needs to configure the expected minimum values and the maximum values of the elements.

#### 4.2.2.3 Results

Efinder executed 8 times and generated a model for each of the 8 partitions. Figure 4.4 shows the 8 generated instance models and their properties. The numbers of the first row are the characteristic values. They are used to identify different partitions. This test case shows that Efinder can generate a set of instance models of which each one belongs to a partition. We can conclude that we can use Efinder to generate a set of models using classifying terms. Therefore, we conducted industrial case studies on Efinder based on ASOME metamodels. The reports of the case study are shown in Appendix A. The reader can first read reports in Appendix A and then the discussion on Efinder in Section 4.2.3.



000

```
Model
  Type string5
  Class string2
    Attribute string1
  Type string1
```

| Property | Value |
|---|---|
| Classes | Class string2 |
| Classifier | |
| Multivalued | false |
| Name | string1 |

001

```
Model
  Type string5
  Class string2
    Attribute string1
  Type string1
```

| Property | Value |
|---|---|
| Classes | Class string2 |
| Classifier | |
| Multivalued | true |
| Name | string1 |

010

```
Model
  Type string5
  Class string2
    Attribute string1
  Type string1
```

| Property | Value |
|---|---|
| Classes | Class string2 |
| Classifier | Type string1 |
| Multivalued | false |
| Name | string1 |

011

```
Model
  Type string5
  Class string2
    Attribute string1
  Type string1
```

| Property | Value |
|---|---|
| Classes | Class string2 |
| Classifier | Type string1 |
| Multivalued | true |
| Name | string1 |

100

```
Model
  Type string1
  Class string2
    Attribute string1
  Type string5
```

| Property | Value |
|---|---|
| Classes | Class string2 |
| Classifier | Class string2 |
| Multivalued | false |
| Name | string1 |

101

```
Model
  Type string5
  Class string2
    Attribute string1
  Type string1
```

| Property | Value |
|---|---|
| Classes | Class string2 |
| Classifier | Class string2 |
| Multivalued | true |
| Name | string1 |

110

```
Model
  Type string1
  Class string5
    Attribute string2
    Attribute string1
  Type string2
```

| Property | Value | Property | Value |
|---|---|---|---|
| Classes | Class string5 | Classes | Class string5 |
| Classifier | Class string5 | Classifier | Type string1 |
| Multivalued | false | Multivalued | false |
| Name | string2 | Name | string1 |

111

```
Model
  Type string1
  Class string5
    Attribute string2
    Attribute string1
  Type string2
```

| Property | Value | Property | Value |
|---|---|---|---|
| Classes | Class string5 | Classes | Class string5 |
| Classifier | Class string5 | Classifier | Type string1 |
| Multivalued | false | Multivalued | true |
| Name | string2 | Name | string1 |

Figure 4.4: the generated models shown in Eclipse

### 4.2.3 Discussion on Efinder

The discussion summarizes all the problems we encountered during the small case study and industrial case studies.

**Efinder has the potential to increase the overall productivity in model transformation testing at Altran.** Both the small case study in Section 4.2.2 and the industrial case study in Appendix A have shown that Efinder has two useful features compared to USE. It can transform

Ecore metamodel, OCL invariants, and Classifying term values into USE specification. Efinder can also save generated instances as Ecore models. Efinder can generate a set of instance models according to the defined classifying terms. It will generate an instance model for each partition identified by the classifying terms. This feature reduces the effort in manually generating instance models.
It also avoids the duplicate or structurally similar test models made by the testers while creating ASOME models. When manually generating test models, the ASOME tester may create models belonging to the same equivalence partition. This problem has been complained about by the testers at Altran. Efinder can avoid the problem since it will generate different models identified by a characteristic value.

**The quality of the testing depends on the classifying terms given by testers.** For the current version of Efinder, the user not only needs to write classifying terms but also needs to specify other invariants so that a solution is satisfiable. The quality of the testing depends on the classifying terms specified by testers.

**Several problems need to be considered in the future work of Efinder** First, there are some OCL operations that are still not supported including "matches", "size", "at", "toUpper", "toUpperCase", "toLower", "subString", and "oclType". This limits the constraints that a user can specify. In the industrial case study, we need to ignore the original OCL invariants of the ASOME metamodels so that Efinder can compile.
Second, an operation defined by the developers of the ASOME projects, called "asError", is not supported by Efinder. This is a helper operation. In OCL, all failures of invariants are errors. The ASOME developers introduced these helpers to distinguish errors and warnings. Efinder can not compile them. However, these operations are not significant and can be neglected. If there is an error in an ASOME model, then the result of the OCL expression of the invariant will be evaluated as null and then will be transformed to null. This will cause a problem for Efinder. Efinder then may produce invalid models.
Third, Efinder does not support interaction between OCL constraints and manually-written Java operations in Ecore metamodels. If the testers in the ASOME project group want to benefit from Efinder, they need to use OCL for constraints instead of Java.
Fourth, from the industrial case study on the ASOME tool in Appendix A.4, we found that the message of an unsatisfiable proof is not clear for the user to interpret.
Fifth, Efinder does not support generating a pair of connected models. It is expected that the tool can provide this feature for the user because the ASOME transformation chain takes two models as input. We solved that by writing a program to split a monolithic model into one ASOME model

and one generator model.

**Efinder's classifying term plug-in** Although the classifying term plug-in has not been developed yet, we have some expectations for it. The plug-in for classifying terms should ask the user for the information of classifying terms according to the type of the classifying term. It should have an engineer-friendly interface. It should automatically run Efinder to generate a model for each partition according to the characteristic values. For example, for an integer type of classifying term, the integer value is the characteristic value that identifies a partition. The user may expect this integer value to vary from 1 to 100. Then this plug-in should ask the range that the partition and generate OCL files automatically.

### 4.2.4 Conclusion

Efinder solved two major problems of USE. Efinder makes the idea of classifying terms more suitable to the current model transformation testing in the industry. Table 4.3 shows the evaluation results of Efinder. Despite the fact that Efinder still has some problems mentioned above, Efinder with the idea of classifying terms has the potential to improve the current practice at Altran.

Table 4.3: Evaluation results of Efinder

| No. | Criteria | Results (Yes or No) | Reasons |
|---|---|---|---|
| 1 | The tool used in the proposed approach(s) can run without errors in the small case studies. | Yes | According to the small case study, Efinder can perform without errors. |
| 2 | The tool can handle industrial-size model transformations | No | In the industrial case studies, the compilation of OCL files and Ecore models failed. Some OCL operations and Java operations are not supported. |
| 3 | The total time required by the proposed approaches should be less than the time required by the current practice of model transformation testing at Altran. | NA | We can not measure it since the current version of the tool still has problems in the industrial case studies. |
| 4 | The proposed approach has the potential to improve the current practice of model transformation testing. | Yes | Efinder extends USE. Therefore, Efinder also can implement classifying terms and bring the benefits of classifying terms to model transformation testing. |

# Chapter 5

# Conclusions

In this section, we first answer the three research questions and then discuss future work.

**RQ1: What are the main challenges in the practice of model transformation the industry?**

In this study, we took the practice of model transformation testing at Altran as an example. We interviewed the testers at Altran. From the interview results, we summarized that the current model transformation practice of model transformation testing at Altran confronts two main challenges. One is the difficulty to generate a qualified test suite. The other is the difficulty to generate an efficient test suite.

**RQ2: What are the existing approaches that can improve the testing process in the industry?**

We conducted a literature review to search for the approaches proposed to improve model transformation testing. We found 36 studies on model transformation testing approaches. We present a description of the studies. After reviewing and discussing the studies, we selected 11 studies out of the 36 studies considering the model transformation testing practice at Altran. We then prioritized the 11 studies taking the interests of the stakeholders of the ASOME project group into account.

**RQ3: How suitable are these approaches?**

We defined four criteria for the suitability evaluation of the selected tools. We explored 4 tools: TractsTool, Matching Table Builder, USE, and Efinder. We explored the tools and implemented small case studies on them. If the tool can run in a small case without major problems, we conducted industrial case studies with ASOME examples. We evaluated the suitability of them in terms of the defined criteria.

**The suitability of TractsTool and Matching Table Builder**

TractTool can be used to solve oracle problems. It uses tracts to verify the transformation results.

It may avoid the mistakes caused by the manual result checking in the current testing process at Altran. But there are two problems with TractsTool to be considered. First, the quality of the testing using TractsTool depends on the quality of the tracts made by testers. Second, the current version of TractsTool still contains errors but this problem can be solved by further improvement. Matching Table Builder (MTB) can not directly be applied to the model transformations at Altran but the tool can be adapted. MTB also uses OCL constraints like tracts. Therefore, the quality of the testing using MTB also depends on the quality of OCL constraints made by testers.

**The suitability of the USE and Efinder**

We consider Efinder as a suitable solution to improve the current testing process at Altran. USE and Efinder both can generate test models automatically according to the defined classifying terms. However, USE is not adequate to improve the current practice at Altran. There are two main drawbacks of USE. One is that it does not accept Ecore metamodels. The other is that it can not export models as Ecore models.

Efinder extends USE with two new features, translating Ecore metamodels into USE specifications and saving generated models as XMI files. These two new features make it possible to improve the current model transformation testing at Altran. There are still some problems with this tool including unsupported OCL and Java operations. Moreover, Efinder is expected to generate a pair of connected models and have a classifying terms plug-in. Overall, the idea of classifying terms has the potential to improve the current model transformation testing. It can not only generate test models automatically but also avoid structurally similar tests. Therefore, we consider Efinder is suitable and worth further development.

For future work, there are two possible research directions for extending this study. One is to explore the rest of the list shown in Table 2.9. Especially for the third study [31] in the list, the idea of it is similar to USE and Efinder. Instead of using OCL, [31] uses a specification language called PAMOMO. The second direction is to improve the current version of Efinder and mitigate its problems mentioned in this thesis.

# Bibliography

[1] Vincent Aranega, Jean-Marie Mottu, Anne Etien, Thomas Degueule, Benoit Baudry, and Jean-Luc Dekeyser. Towards an automation of the mutation analysis dedicated to model transformation. *Software Testing, Verification and Reliability*, 25(5-7):653–683, 2015. 15

[2] Ellen Francine Barbosa, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. Toward the determination of sufficient mutant operators for c. *Software Testing, Verification and Reliability*, 11(2):113–136, 2001. 15

[3] Benoit Baudry, Trung Dinh-Trong, Jean-Marie Mottu, Devon Simmonds, Robert France, Sudipto Ghosh, Franck Fleurey, and Yves Le Traon. Model transformation testing challenges. 2006. 12, 13

[4] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon, and Jean-Marie Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 53(6):139–143, 2010. 12, 13

[5] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *2006 17th International Symposium on Software Reliability Engineering*, pages 85–94. IEEE, 2006. 13, 14, 20, 23

[6] Loli Burgueno. Testing m2m/m2t/t2m transformations. In *SRC@ MoDELS*, pages 7–12, 2015. 3, 18, 19, 22, 23, 25, 36

[7] Loli Burgueño, Frank Hilken, Antonio Vallecillo, and Martin Gogolla. Testing transformation models using classifying terms. In *International Conference on Theory and Practice of Model Transformations*, pages 69–85. Springer, 2017. 16, 21, 23, 36

[8] Loli Burgueño, Javier Troya, Manuel Wimmer, and Antonio Vallecillo. Static fault localization in model transformations. *IEEE Transactions on Software Engineering*, 41(5):490–506, 2014. 18

[9] Eric Cariou, Raphaël Marvie, Lionel Seinturier, and Laurence Duchien. Ocl for the specification of model transformation contracts. In *OCL and Model Driven Engineering, UML 2004 Conference Workshop*, volume 12, pages 69–83, 2004. 18

[10] Hector M Chavez, Wuwei Shen, Robert B France, and Benjamin A Mechling. An approach to testing java implementation against its uml class model. In *International Conference on Model Driven Engineering Languages and Systems*, pages 220–236. Springer, 2013. 18, 19

[11] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong . . . , 1998. 17

[12] Jesús Sánchez Cuadrado and Martin Gogolla. Model finding in the emf ecosystem. *Journal of Object Technology*, 19(2), 2020. 40, 43

[13] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Static analysis of model transformations. *IEEE Transactions on Software Engineering*, 43(9):868–897, 2016. 12, 15

[14] Jesús Sánchez Cuadrado, Esther Guerra, Juan de Lara, Robert Clarisó, and Jordi Cabot. Translating target to source constraints in model-to-model transformations. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 12–22. IEEE, 2017. 14, 15, 20

[15] Jean-Marie Favre. Foundations of meta-pyramids: Languages vs. metamodels–episode ii: Story of thotus the baboon1. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2005. 5

[16] Olivier Finot, Jean-Marie Mottu, Gerson Sunyé, and Christian Attiogbé. Partial test oracle in model transformation testing. In *International Conference on Theory and Practice of Model Transformations*, pages 189–204. Springer, 2013. 6, 17, 18

[17] Olivier Finot, Jean-Marie Mottu, Gerson Sunyé, and Thomas Degueule. Using meta-model coverage to qualify test oracles. 2013. 17

[18] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, and Yves Le Traon. Towards dependable model transformations: Qualifying input test data. 2007. 7, 14

[19] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, and Yves Le Traon. Qualifying input test data for model transformations. *Software & Systems Modeling*, 8(2):185–203, 2009. 7

[20] Piero Fraternali and Massimo Tisi. Multi-level tests for model driven web applications. In *International Conference on Web Engineering*, pages 158–172. Springer, 2010. 18, 19

[21] Jokin García, Maider Azanza, Arantza Irastorza, and Oscar Díaz. Testing mofscript transformations with handymof. In *International Conference on Theory and Practice of Model Transformations*, pages 42–56. Springer, 2014. 18, 19

[22] Christine M Gerpheide, Ramon RH Schiffelers, and Alexander Serebrenik. Assessing and improving quality of qvto model transformations. *Software Quality Journal*, 24(3):797–834, 2016. 12

[23] Martin Gogolla, Fabian Büttner, and Mark Richters. Use: A uml-based specification environment for validating uml and ocl. *Science of Computer Programming*, 69(1-3):27–34, 2007. 3

[24] Martin Gogolla and Antonio Vallecillo. Tractable model transformation testing. In *European Conference on Modelling Foundations and Applications*, pages 221–235. Springer, 2011. 25

[25] Martin Gogolla, Antonio Vallecillo, Loli Burgueno, and Frank Hilken. Employing classifying terms for testing model transformations. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 312–321. IEEE, 2015. 16, 21, 23, 36

[26] Carlos A González and Jordi Cabot. Atltest: a white-box test generation approach for atl transformations. In *International Conference on Model Driven Engineering Languages and Systems*, pages 449–464. Springer, 2012. 14, 20, 23

[27] Carlos A González and Jordi Cabot. Test data generation for model transformations combining partition and constraint analysis. In *International Conference on Theory and Practice of Model Transformations*, pages 25–41. Springer, 2014. 16, 21, 23

[28] Esther Guerra, Jesús Sánchez Cuadrado, and Juan de Lara. Towards effective mutation testing for atl. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 78–88. IEEE, 2019. 15, 23

[29] Esther Guerra, Juan De Lara, Dimitris Kolovos, and Richard Paige. A visual specification language for model-to-model transformations. In *2010 IEEE symposium on visual languages and human-centric computing*, pages 119–126. IEEE, 2010. 17, 18

[30] Esther Guerra, Juan de Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Automated verification of model transformations based on visual contracts. *Automated Software Engineering*, 20(1):5–46, 2013. 14

[31] Esther Guerra and Mathias Soeken. Specification-driven model transformation testing. *Software & Systems Modeling*, 14(2):623–644, 2015. 7, 12, 13, 14, 20, 23, 52

[32] Frank Hermann, Hartmut Ehrig, Ulrike Golas, and Fernando Orejas. Formal analysis of model transformations based on triple graph grammars. *Mathematical Structures in Computer Science*, 24(4), 2014. 15

[33] Stephan Hildebrandt, Leen Lambers, Holger Giese, Jan Rieke, Joel Greenyer, Wilhelm Schäfer, Marius Lauder, Anthony Anjorin, and Andy Schürr. A survey of triple graph grammar tools. *Electronic Communications of the EASST*, 57, 2013. 15

[34] Frank Hilken, Martin Gogolla, Loli Burgueño, and Antonio Vallecillo. Testing models and model transformations using classifying terms. *Software & Systems Modeling*, 17(3):885–912, 2018. 16, 21

[35] Ethan K Jackson, Gabor Simko, and Janos Sztipanovits. Diversely enumerating system-level architectures. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 1–10. IEEE, 2013. 8

[36] Sorour Jahanbin and Bahman Zamani. Test model generation using equivalence partitioning. In *2018 8th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 98–103. IEEE, 2018. 16, 20

[37] Mingyue Jiang, Tsong Yueh Chen, Fei-Ching Kuo, Zhiquan Zhou, and Zuohua Ding. Testing model transformation programs using metamorphic testing. 2014. 17

[38] Atif Aftab Jilani, Muhammad Zohaib Iqbal, and Muhammad Uzair Khan. A search based test data generation approach for model transformations. In *International Conference on Theory and Practice of Model Transformations*, pages 17–24. Springer, 2014. 14, 20, 23

[39] Sven Jörges and Bernhard Steffen. Back-to-back testing of model-based code generators. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 425–444. Springer, 2014. 19, 21

[40] Stuart Kent. Model driven engineering. In *International Conference on Integrated Formal Methods*, pages 286–298. Springer, 2002. 1

[41] Alexander Königs. Model transformation with triple graph grammars. In *Model Transformations in Practice Satellite Workshop of MODELS*, page 166, 2005. 15

[42] Jochen M Küster, Thomas Gschwind, and Olaf Zimmermann. Incremental development of model transformation chains using automated testing. In *International Conference on Model Driven Engineering Languages and Systems*, pages 733–747. Springer, 2009. 6

[43] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Mutation analysis testing for model transformations. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 376–390. Springer, 2006. 7, 15

[44] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Model transformation testing: oracle issue. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 105–112. IEEE, 2008. 12, 13, 17

[45] Akbar Siami Namin, James Andrews, and Duncan Murdoch. Sufficient mutation operators for measuring test effectiveness. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 351–360. IEEE, 2008. 15

[46] Thi-Hanh Nguyen, Duc-Hanh Dang, and Quang-Trung Nguyen. On analyzing rule-dependencies to generate test cases for model transformations. In *2019 11th International Conference on Knowledge and Systems Engineering (KSE)*, pages 1–6. IEEE, 2019. 14, 15, 20

[47] A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996. 15

[48] Dimitrios S Polack, Richard F Paige, Louis M Rose, and Fiona AC Polack. Unit testing model management operations. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 97–104. IEEE, 2008. 18, 19

[49] Lukman Ab Rahim and Jon Whittle. A survey of approaches for verifying model transformations. *Software & Systems Modeling*, 14(2):1003–1028, 2015. 12, 13, 17, 18

[50] Rodrigo Ramos, Olivier Barais, and Jean-Marc Jézéquel. Matching model-snippets. In *International Conference on Model Driven Engineering Languages and Systems*, pages 121–135. Springer, 2007. 17, 18

[51] Stuart C Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Proceedings Fourth International Software Metrics Symposium*, pages 64–73. IEEE, 1997. 8

[52] Johannes Schönböck, Gerti Kappel, Manuel Wimmer, Angelika Kusel, Werner Retschitzegger, and Wieland Schwinger. Tetrabox-a generic white-box testing framework for model transformations. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 1, pages 75–82. IEEE, 2013. 7, 14, 20, 23

[53] Gehan MK Selim, James R Cordy, and Juergen Dingel. Model transformation testing: The state of the art. In *Proceedings of the First Workshop on the Analysis of Model Transformations*, pages 21–26, 2012. 7, 12, 13

[54] Oszkár Semeráth and Dániel Varró. Iterative generation of diverse models for testing specifications of dsl tools. In *FASE*, volume 18, pages 227–245, 2018. 6, 8, 16, 18, 20

[55] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. On combining multi-formalism knowledge to select models for model transformation testing. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 328–337. IEEE, 2008. 13

[56] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. Automatic model generation strategies for model transformation testing. In *International Conference on Theory and Practice of Model Transformations*, pages 148–164. Springer, 2009. 13, 14, 20, 23

[57] Ingo Stuermer, Mirko Conrad, Heiko Doerr, and Peter Pepper. Systematic testing of model-based code generators. *IEEE Transactions on Software Engineering*, 33(9):622–634, 2007. 18, 19, 21

[58] Alessandro Tiso, Gianna Reggio, and Maurizio Leotta. A method for testing model to text transformations. In *AMT@ MoDELS*, 2013. 18, 19

[59] Alessandro Tiso, Gianna Reggio, and Maurizio Leotta. Unit testing of model to text transformations. In *AMT@ MoDELS*, pages 14–23, 2014. 18, 19

[60] Javier Troya, Alexander Bergmayr, Loli Burgueno, and Manuel Wimmer. Towards systematic mutations for and with atl model transformations. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–10. IEEE, 2015. 15

[61] Javier Troya, Sergio Segura, and Antonio Ruiz-Cortés. Automated inference of likely metamorphic relations for model transformations. *Journal of Systems and Software*, 136:188–208, 2018. 17

[62] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach.* Elsevier, 2010. 7

[63] Antonio Vallecillo, Martin Gogolla, Loli Burgueño, Manuel Wimmer, and Lars Hamann. Formal specification and testing of model transformations. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 399–437. Springer, 2012. 14, 16, 22

[64] Dániel Varró, Oszkár Semeráth, Gábor Szárnyas, and Ákos Horváth. Towards the automated generation of consistent, diverse, scalable and realistic graph models. In *Graph Transformation, Specifications, and Nets*, pages 285–312. Springer, 2018. 8

[65] Jens von Pilgrim, Bert Vanhooff, Immo Schulz-Gerlach, and Yolande Berbers. Constructing and visualizing transformation chains. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 17–32. Springer, 2008. 6

[66] Stephan Weißleder. *Test models and coverage criteria for automatic model-based test generation with UML state machines.* PhD thesis, Humboldt University of Berlin, 2010. 6, 13

[67] Martin Wieber, Anthony Anjorin, and Andy Schürr. On the usage of tggs for automated model transformation testing. In *International Conference on Theory and Practice of Model Transformations*, pages 1–16. Springer, 2014. 6

[68] Manuel Wimmer and Loli Burgueño. Testing m2t/t2m transformations. In *International Conference on Model Driven Engineering Languages and Systems*, pages 203–219. Springer, 2013. 3, 18

# Appendix A

# ASOME Case Study Reports

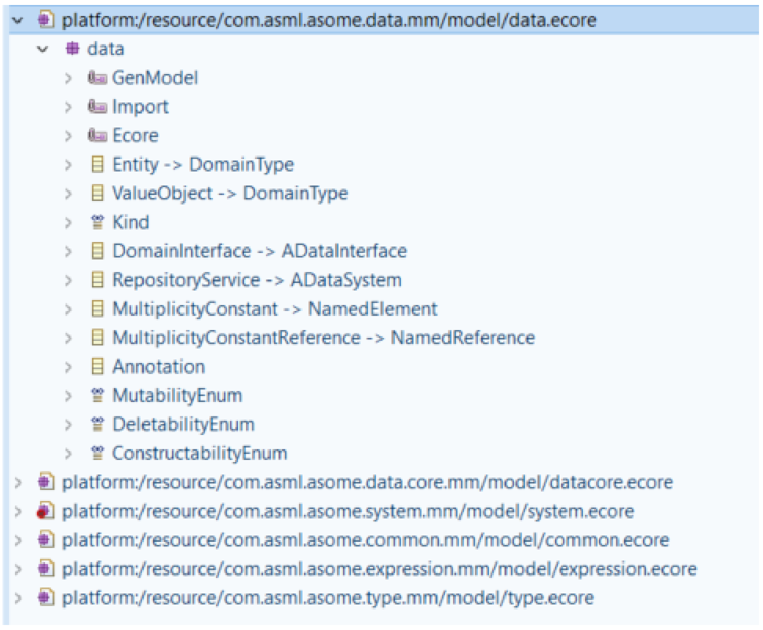
Here we present the industrial case studies using the ASOME metamodels. To check if Efinder can generate one model, we first tested Efinder using ASOME metamodels with a simple invariant (see industrial case 1 in Section A.1). Then we checked if Efinder can generate a set of ASOME models using classifying terms (see the industrial case 2 in Section A.2). Since the ASOME tool takes an ASOME model and a generator model as input. In Section A.3, A.4 and A.5, we reported three problems of Efinder. After reading these reports, the reader can go back to Section 4.2.3. We summarize the discussion on Efinder in Section 4.2.3.

## A.1 Industrial case study 1: to compile and generate one instance model of data.ecore

The purpose of this test case is to check whether Efinder can transpile the metamodels of ASOME and generate a simple ASOME model. A domain interface with several entities is a typical design pattern of ASOME models. In this case study, the expected model is a domain interface with two entities. This report shows how to use Efinder so that it can generate an instance model satisfying certain constraints. It explains what an ASOME developer needs to do in order to successfully compile ASOME metamodel and generate an ASOME model.

### A.1.1 Implementation

Figure A.1 shows the content of the data.ecore file. The data.ecore is the ecore metamodel of ASOME. As can be seen from Figure A.1, the data.ecore refers to five other ecore files.



platform:/resource/com.asml.asome.data.mm/model/data.ecore
data
GenModel
Import
Ecore
Entity -> DomainType
ValueObject -> DomainType
Kind
DomainInterface -> ADataInterface
RepositoryService -> ADataSystem
MultiplicityConstant -> NamedElement
MultiplicityConstantReference -> NamedReference
Annotation
MutabilityEnum
DeletabilityEnum
ConstructabilityEnum
platform:/resource/com.asml.asome.data.core.mm/model/datacore.ecore
platform:/resource/com.asml.asome.system.mm/model/system.ecore
platform:/resource/com.asml.asome.common.mm/model/common.ecore
platform:/resource/com.asml.asome.expression.mm/model/expression.ecore
platform:/resource/com.asml.asome.type.mm/model/type.ecore

Figure A.1: data.ecore

We defined a simple OCL invariant in data.ocl. This OCL invariant requires the instance model to be a domain interface with two entities. The code in data.ocl is listed in Listing A.1.

```
import 'data.ecore'

package data

context DomainInterface

inv CT1:
  elements->select(e | e.oclIsKindOf(Entity))->size() = 2
endpackage
```

Listing A.1: data.ocl

To simplify the case, we limit the number of domain interfaces to 1. The configuration code is shown in Listing A.2.

```
TestBoundsProvider boundsProvider = new TestBoundsProvider()
        .withInterval("DomainInterface", 1, 1);
```

Listing A.2: Model configuration

There were several problems when Efinder compiled data.ecore and data.ocl into USE specifications. We ignored the following problems for now to explore the suitability of Efinder. To be able to use Efinder, one should implement the following steps.

#### A.1.1.1 1. Remove the operations in the ecore models that Efinder can not compile

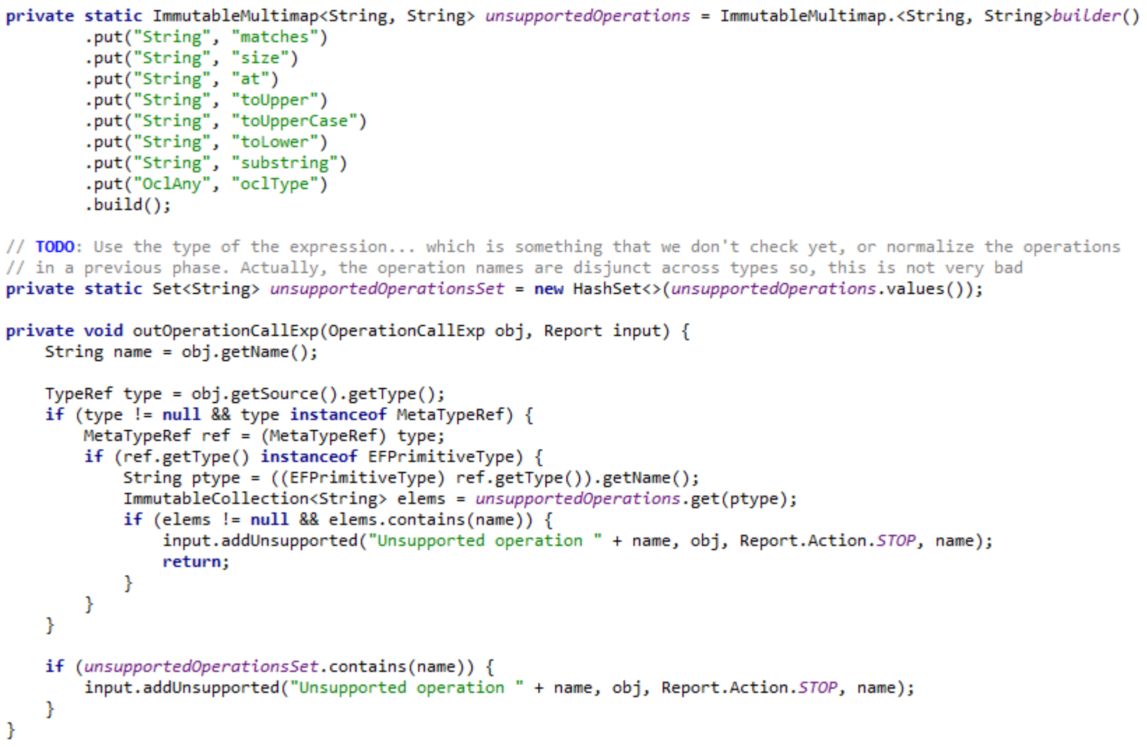
There are some Java operations in the ASOME metamodels not recognized by Efinder. We needed to delete these operations so that the metamodels could be compiled. But if the developers of the ASOME project want to use Efinder, they can express the operations in OCL instead.

Table A.1: The deleted operations

| Operations | Filename |
|---|---|
| getAllDomainInterfaces | data.ecore |
| getDomainInterfaceDependencies | |
| getAllElements | |
| getAllTypes | |
| getAllConstants | |
| getAllImportedModels | system.ecore |
| getAllImportedModelsRecursive | |
| calcOperations | |
| getInterfaceDependencies | |
| multiplicityValue | |
| getAllSystems | |
| intValue | expression.ecore |
| getRepr | |
| evaluate | |
| getBasicType | type.ecore |
| getRepr | |

#### A.1.1.2 2. Ignore some original OCL constraints of the ASOME metamodels

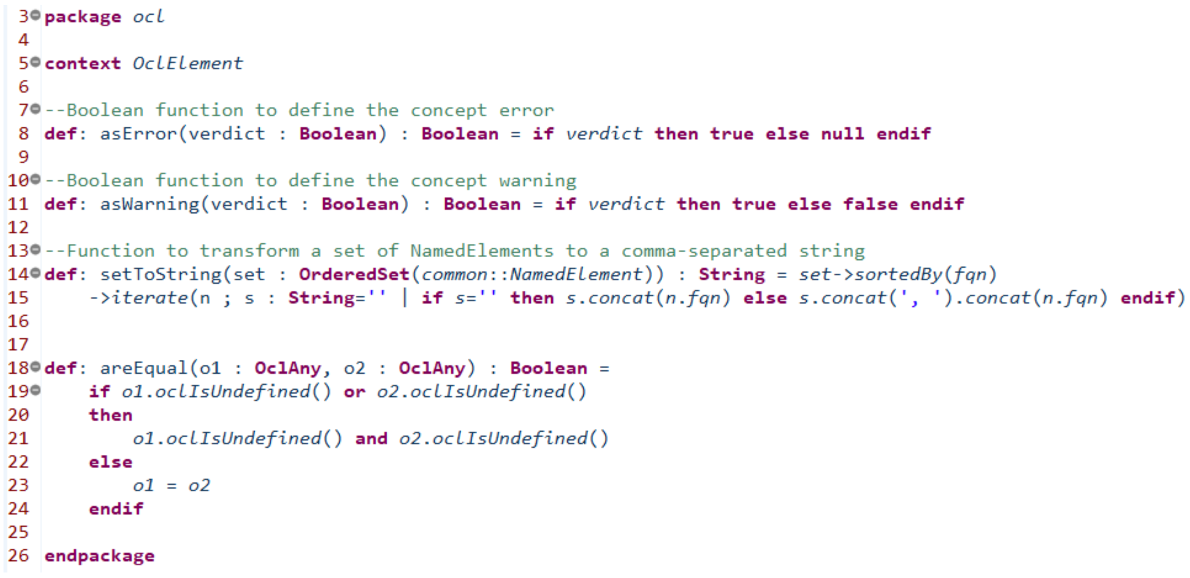
The original OCL constraints of the ASOME metamodels contain a number of expressions that Efinder currently does not support. Figure A.2 shows a part of the code of Efinder. It shows that there are 8 OCL operations that are not supported by Efinder including ”matches”, ”size”, ”at”, ”toUpper”, ”toUpperCase”, ”toLower”, ”subString”, and ”oclType”.



```java
private static ImmutableMultimap<String, String> unsupportedOperations = ImmutableMultimap.<String, String>builder()
        .put("String", "matches")
        .put("String", "size")
        .put("String", "at")
        .put("String", "toUpper")
        .put("String", "toUpperCase")
        .put("String", "toLower")
        .put("String", "substring")
        .put("OclAny", "oclType")
        .build();

// TODO: Use the type of the expression... which is something that we don't check yet, or normalize the operations
// in a previous phase. Actually, the operation names are disjunct across types so, this is not very bad
private static Set<String> unsupportedOperationsSet = new HashSet<>(unsupportedOperations.values());

private void outOperationCallExp(OperationCallExp obj, Report input) {
    String name = obj.getName();

    TypeRef type = obj.getSource().getType();
    if (type != null && type instanceof MetaTypeRef) {
        MetaTypeRef ref = (MetaTypeRef) type;
        if (ref.getType() instanceof EFPrimitiveType) {
            String ptype = ((EFPrimitiveType) ref.getType()).getName();
            ImmutableCollection<String> elems = unsupportedOperations.get(ptype);
            if (elems != null && elems.contains(name)) {
                input.addUnsupported("Unsupported operation " + name, obj, Report.Action.STOP, name);
                return;
            }
        }
    }

    if (unsupportedOperationsSet.contains(name)) {
        input.addUnsupported("Unsupported operation " + name, obj, Report.Action.STOP, name);
    }
}
```

Figure A.2: Unsupported operations in Efinder

Additionally, we found that a helper function called ”asError” caused a problem for Efinder. There are some helper operations defined by the ASOME developers. As can be seen from Figure A.3, there are two boolean functions called ”asErrors” and ”asWarning”. They are used to distinguish errors and warnings in ASOME modeling. If an invariant is violated and considered as an error, the OCL expression will be first evaluated as false and given null. This will cause a problem for Efinder because it should not produce invalid models.



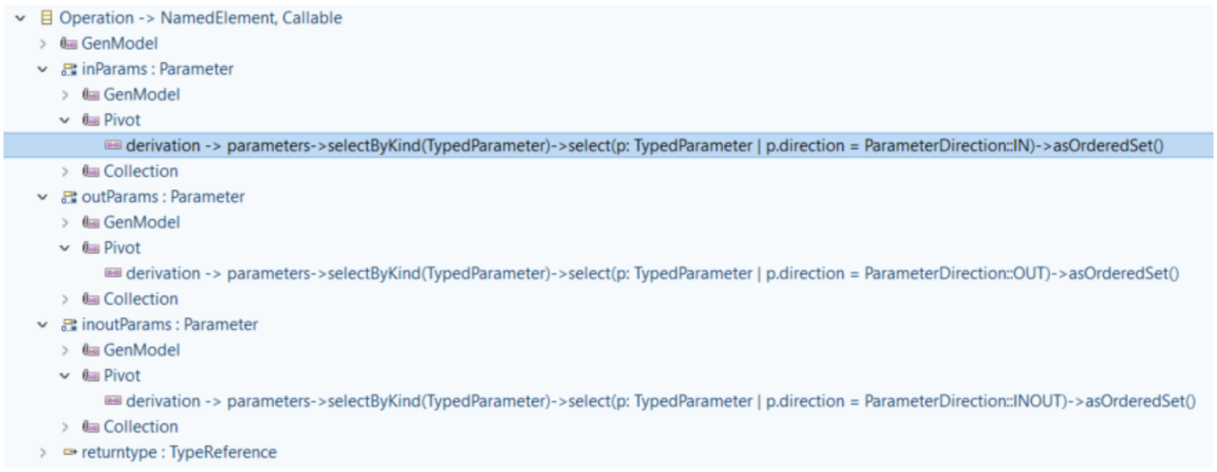
```
package ocl

context OclElement

--Boolean function to define the concept error
def: asError(verdict : Boolean) : Boolean = if verdict then true else null endif

--Boolean function to define the concept warning
def: asWarning(verdict : Boolean) : Boolean = if verdict then true else false endif

--Function to transform a set of NamedElements to a comma-separated string
def: setToString(set : OrderedSet(common::NamedElement)) : String = set->sortedBy(fqn)
    ->iterate(n ; s : String='' | if s='' then s.concat(n.fqn) else s.concat(', ').concat(n.fqn) endif)


def: areEqual(o1 : OclAny, o2 : OclAny) : Boolean =
    if o1.oclIsUndefined() or o2.oclIsUndefined()
    then
        o1.oclIsUndefined() and o2.oclIsUndefined()
    else
        o1 = o2
    endif

endpackage
```

Figure A.3: helper operations defined by ASOME developers

If we run the test with the original OCL files, Efinder will report errors and stop. Therefore, we ignored the original OCL files of the ASOME metamodels so that we could explore further with Efinder.

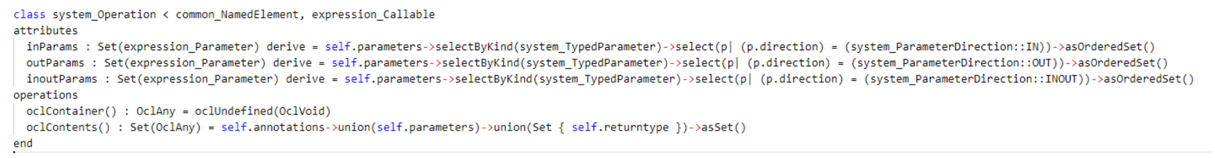
#### A.1.1.3 3. Remove the OCL expressions so that there is no type conformance problems in USE

As can be seen from Figure A.4 and A.5, the translation from ecore to USE caused a type conformance problem. The attribute 'inParams' is derived according to the expression specified in system.ecore model. The Efinder compiled the attribute 'inParams' to a set. However, the expression given in the system.ecore is an ordered set. The ordered set does not conform to the set.



```
Operation -> NamedElement, Callable
  GenModel
  inParams : Parameter
    GenModel
    Pivot
      derivation -> parameters->selectByKind(TypedParameter)->select(p: TypedParameter | p.direction = ParameterDirection::IN)->asOrderedSet()
    Collection
  outParams : Parameter
    GenModel
    Pivot
      derivation -> parameters->selectByKind(TypedParameter)->select(p: TypedParameter | p.direction = ParameterDirection::OUT)->asOrderedSet()
    Collection
  inoutParams : Parameter
    GenModel
    Pivot
      derivation -> parameters->selectByKind(TypedParameter)->select(p: TypedParameter | p.direction = ParameterDirection::INOUT)->asOrderedSet()
    Collection
  returntype : TypeReference
```

Figure A.4: Class 'Operation' in system.ecore



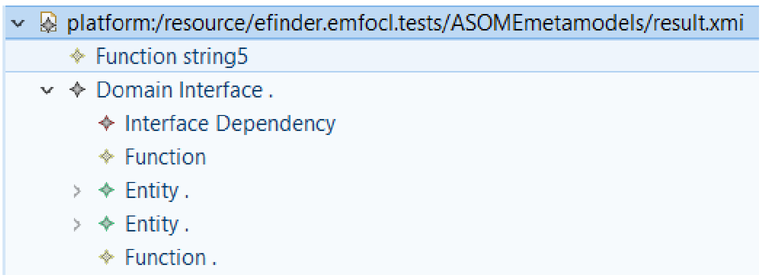
```
class system_Operation < common_NamedElement, expression_Callable
attributes
  inParams : Set(expression_Parameter) derive = self.parameters->selectByKind(system_TypedParameter)->select(p| (p.direction) = (system_ParameterDirection::IN))->asOrderedSet()
  outParams : Set(expression_Parameter) derive = self.parameters->selectByKind(system_TypedParameter)->select(p| (p.direction) = (system_ParameterDirection::OUT))->asOrderedSet()
  inoutParams : Set(expression_Parameter) derive = self.parameters->selectByKind(system_TypedParameter)->select(p| (p.direction) = (system_ParameterDirection::INOUT))->asOrderedSet()
operations
  oclContainer() : OclAny = oclUndefined(OclVoid)
  oclContents() : Set(OclAny) = self.annotations->union(self.parameters)->union(Set { self.returntype })->asSet()
end
```

Figure A.5: the USE specification for Class 'Operation'

The other two attributes "outParams" and "inoutParams" also have this problem. It could be the case that the OCL expressions specified by ASOME developers are problematic or the Efinder's compilation is problematic. However, this problem is considered as less significant as they are derived from other elements. Therefore they were removed to simplify the problem.

### A.1.2 Results

The test ran successfully and generated an instance model. The model is shown in Figure A.6. It satisfies the defined constraints.



```
platform:/resource/efinder.emfocl.tests/ASOMEmetamodels/result.xmi
  Function string5
  Domain Interface .
    Interface Dependency
    Function
    Entity .
    Entity .
    Function .
```

Figure A.6: the generated model shown in Eclipse

## A.2 Industrial case study 2: to generate multiple instance models using classifying terms

The purpose of this case study is to show how to use classifying terms to generate a set of ASOME models with Efinder. This case is derived from the requirements of the testers of the ASOME project. They want to test the combinations of possible interesting features in the input models. Figure A.7 shows one of the expected models. We try to generate two entities that are associated with each other and different ranges for their entity multiplicities.

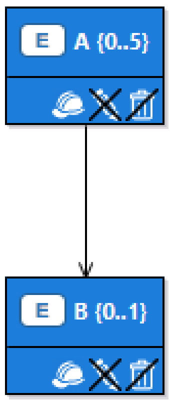


Figure A.7: the expected model shown in ASOME

### A.2.1 Implementation

For this industrial case, the 4 classifying terms about the source multiplicity and the target multiplicity are listed in Listing A.3. The script will add a boolean value at the end of each classifying term.

```
Entity.allInstances()->exists(e | (e.name = 'A') and e.associations->exists(a | a.
    typereference.type.oclAsType(Entity).name = 'B' ) and e.multiplicity.max.
    oclAsType(expression::IntExpression).value = 1 and e.multiplicity.min.oclAsType
    (expression::IntExpression).value = 0) -- CT1
Entity.allInstances()->exists(e | (e.name = 'A') and e.associations->exists(a | a.
    typereference.type.oclAsType(Entity).name = 'B' ) and e.multiplicity.max.
    oclAsType(expression::IntExpression).value = 5 and e.multiplicity.min.oclAsType
    (expression::IntExpression).value = 0) -- CT2
Entity.allInstances()->exists(e | (e.name = 'B') and e.multiplicity.max.oclAsType(
    expression::IntExpression).value = 1 and e.multiplicity.min.oclAsType(
    expression::IntExpression).value = 0) -- CT3
Entity.allInstances()->exists(e | (e.name = 'B') and e.multiplicity.max.oclAsType(
    expression::IntExpression).value = 5 and e.multiplicity.min.oclAsType(
    expression::IntExpression).value = 0) --CT4
```

Listing A.3: 4 classifying terms

For partition 0000, the complete OCL file is shown in Listing A.4. We limit the number of Entity A and B to 1.

```
import 'data.ecore'

package data

context Entity
  inv CT1:
  Entity.allInstances()->exists(e | (e.name = 'A') and e.associations->exists(a | a
      .typereference.type.oclAsType(Entity).name = 'B' ) and e.multiplicity.max.
      oclAsType(expression::IntExpression).value = 1 and e.multiplicity.min.
      oclAsType(expression::IntExpression).value = 0) = false
  inv CT2:
  Entity.allInstances()->exists(e | (e.name = 'A') and e.associations->exists(a | a
      .typereference.type.oclAsType(Entity).name = 'B' ) and e.multiplicity.max.
      oclAsType(expression::IntExpression).value = 5 and e.multiplicity.min.
      oclAsType(expression::IntExpression).value = 0) = false
  inv CT3:
  Entity.allInstances()->exists(e | (e.name = 'B') and e.multiplicity.max.oclAsType
      (expression::IntExpression).value = 1 and e.multiplicity.min.oclAsType(
      expression::IntExpression).value = 0) = false
  inv CT4:
  Entity.allInstances()->exists(e | (e.name = 'B') and e.multiplicity.max.oclAsType
      (expression::IntExpression).value = 5 and e.multiplicity.min.oclAsType(
      expression::IntExpression).value = 0)  = false
inv uniqueEntityA:
  Entity.allInstances()->select(e | e.name = 'A')->size() = 1
inv uniqueEntityB:
  Entity.allInstances()->select(e | e.name = 'B')->size() = 1



endpackage
```

Listing A.4: the complete OCL file for partition 0000

Table A.2: the test results

| Partition No. | | CT4 | CT3 | CT2 | CT1 | Satisfiable |
|---|---|---|---|---|---|---|
| Decimal | Binary | Entity B multiplicity: (0, 1) | Entity B multiplicity: (0, 1) | Entity A multiplicity: (0, 1) | Entity A multiplicity: (0, 5) | |
| 0 | 0000 | 0 | 0 | 0 | 0 | Yes |
| 1 | 0001 | 0 | 0 | 0 | 1 | No |
| 2 | 0010 | 0 | 0 | 1 | 0 | Yes |
| 3 | 0011 | 0 | 0 | 1 | 1 | No |
| 4 | 0100 | 0 | 1 | 0 | 0 | Yes |
| 5 | 0101 | 0 | 1 | 0 | 1 | Yes |
| 6 | 0110 | 0 | 1 | 1 | 0 | Yes |
| 7 | 0111 | 0 | 1 | 1 | 1 | No |
| 8 | 1000 | 1 | 0 | 0 | 0 | Yes |
| 9 | 1001 | 1 | 0 | 0 | 1 | Yes |
| 10 | 1010 | 1 | 0 | 1 | 0 | Yes |
| 11 | 1011 | 1 | 0 | 1 | 1 | No |
| 12 | 1100 | 1 | 1 | 0 | 0 | No |
| 13 | 1101 | 1 | 1 | 0 | 1 | No |
| 14 | 1110 | 1 | 1 | 1 | 0 | No |
| 15 | 1111 | 1 | 1 | 1 | 1 | No |

### A.2.2 Results

The script generated 16 OCL files and Efinder executed 16 times. Table A.2 shows the partitions for the 4 classifying terms. As can be seen from the table, this test generated 8 instances in total. Not every partition will have a satisfiable solution. For example, from partitions 1101 to 1111, there is no generated model. Because for CT3 and CT4, the maximum value of entity multiplicity can not be 1 and 5 at the same time. This is expected because of the script we use in this thesis. However, we do not know why partition 0001 fails. We can not interpret the error message. We also record this problem in Appendix A.4.

## A.3 Problem 1 of Efinder: compilation error in OCL constraints

We encountered some problems while writing OCL files. And this test case shows that Efinder still needs to be improved so that the amount of effort in writing OCL files will be acceptable. This test will focus on the source multiplicity and target multiplicity of an association between an Entity A and Entity B. As can be seen from Figure A.8, we expect to generate 2 entities, called "A" and "B". Entity A is associated with Entity B. The source multiplicity can be either zero to infinity or zero to two. The target multiplicity can be either zero to infinity or zero to one.

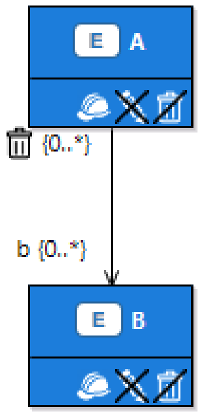


Figure A.8: the expected model shown in ASOME

Listing A.5 shows the model configuration. We defined one and only one `ImplementationModel` should be instantiated.

```
TestBoundsProvider boundsProvider = new TestBoundsProvider()
        .withInterval("ImplementationModel", 1, 1);
```

Listing A.5: Test bounds configuration

In the OCL file, we define that:

- There should be at least one `Entity` called "A".
- There should be at least one `Entity` called "B".
- `Entity` A should be associated with `Entity` B.
- The type reference of the association with `Entity` B should be `type:Collection`.

```
import 'data.ecore'

package data

context Entity
inv CT1:
  Entity.allInstances()->exists(e | e.name = 'A')
inv CT2:
  Entity.allInstances()->exists(e | e.name = 'B')
inv CT3:
  Entity.allInstances()->select(e | e.name = 'A').associations->exists(a | a.
      typereference.type.oclAsType(Entity).name = 'B')

```

```
inv CT5:
  Entity.allInstances()->exists(e | (e.name = 'A') and (e.associations->exists(a |
     a.typereference.type.oclAsType(Entity).name = 'B'
    and a.typereference.oclIsTypeOf(type::Collection)
  ) ) )
endpackage
```

Listing A.6: OCL constraints

After running the test, Efinder returned an error message and stopped. Listing A.7 shows the error message.

```
   Invariant 'type_Collection::nonNull_type_Collection_ordered' is not fulfilled in
        generated system state.
```

Listing A.7: Error message

Efinder compiled the OCL file and generated a USE file called "model.use". Listing A.8 showed where this error message came from. It is an invariant called `nonNull_type_Collection_ordered` in the generated USE file. This error message indicates that the boolean attribute `ordered` in `type::Collection` should be assigned a value in the OCL file by the user.

```
context self : type_Collection inv nonNull_type_Collection_ordered:
  not self.ordered_.isUndefined()
```

Listing A.8: The invariant "nonNull_type_Collection_ordered" in model.use

This test case shows that for the current version of Efinder, the user needs to specify not only classifying terms but also other hidden invariants required by the metamodels.

## A.4 Problem 2 of Efinder: unsatisfiable proof

For the current version of Efinder, the user may find it difficult to interpret the message given by Efinder. When Efinder can not find a solution, it will return an unsatisfiable proof but we cannot interpret the message. This case study shows an example of this problem.
Listing A.9 shows the model configuration of this test. It limits the number of `Entity` to be among 2 and 30.

```
TestBoundsProvider boundsProvider = new TestBoundsProvider()
    .withInterval("Entity", 2, 30);
```

Listing A.9: Model configuration

Listing A.10 shows three defined classifying terms. In the OCL file, we define that

- There should be at least one `Entity` called ”A”.
- There should be at least one `Entity` called ”B”.
- `Entity` A should be associated with `Entity` B. The maximum source multiplicity of `Entity` B should be 1.

```
import 'data.ecore'

package data

context Entity
inv CT1:
  Entity.allInstances()->exists(e | e.name = 'A')
inv CT2:
  Entity.allInstances()->exists(e | e.name = 'B')
inv CT3:
  Entity.allInstances()->exists(e | (e.name = 'A') and (e.associations->exists(a |
      a.typereference.type.oclAsType(Entity).name = 'B'
    and a.typereference.oclAsType(type::Collection).multiplicity.max.oclIsTypeOf(
        expression::IntExpression)= 1
  ) ) )
endpackage
```

Listing A.10: OCL constraints

For these classifying terms, it should be easily satisfied. However, Efinder considered it as ”trivially unsatisfiable” and gave the proof as shown in Listing A.11. It is difficult for the user to interpret the message and fix the OCL constraints.

```
[main] INFO  .use.kodkod.UseKodkodModelValidator  - Unsatisfiable proof:
< node: (all self: one data_Entity | ((if (data_Entity = Undefined_Set) then
    Undefined else (if (some e: one data_Entity | (((if (e = Undefined) then
    Undefined else (e . common_NamedElement_name)) = String_string3) && ((if ((if (
    e = Undefined) then Undefined_Set else (e . DomainType_associations)) =
    Undefined_Set) then Undefined else (if (some a: one (if (e = Undefined) then
    Undefined_Set else (e . DomainType_associations)) | (((if ((if ((if ((if (a =
    Undefined) then Undefined else (a . TypeReferral_typereference)) = Undefined)
    then Undefined else ((if (a = Undefined) then Undefined else (a .
    TypeReferral_typereference)) . TypeReference_type)) in data_Entity) then (if ((
    if (a = Undefined) then Undefined else (a . TypeReferral_typereference)) =
    Undefined) then Undefined else ((if (a = Undefined) then Undefined else (a .
    TypeReferral_typereference)) . TypeReference_type)) else Undefined) = Undefined
    ) then Undefined else ((if ((if ((if (a = Undefined) then Undefined else (a .
    TypeReferral_typereference)) = Undefined) then Undefined else ((if (a =
    Undefined) then Undefined else (a . TypeReferral_typereference)) .
    TypeReference_type)) in data_Entity) then (if ((if (a = Undefined) then
    Undefined else (a . TypeReferral_typereference)) = Undefined) then Undefined
    else ((if (a = Undefined) then Undefined else (a . TypeReferral_typereference))
     . TypeReference_type)) else Undefined) . common_NamedElement_name)) =
    String_string4) && (Boolean_False = Boolean_True))) then Boolean_True else
    Boolean_False)) = Boolean_True))) then Boolean_True else Boolean_False)) =
    Boolean_True)), literal: -2147483647, env: {}>
```

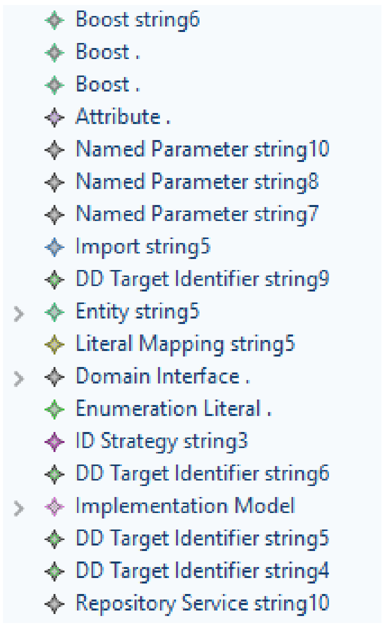
Listing A.11: Unsatisfiable proof

For now, we can not interpret the message returned by Efinder. This problem may hurt the productivity of model transformation testing.

## A.5 Problem 3: Efinder does not support generating a pair of models

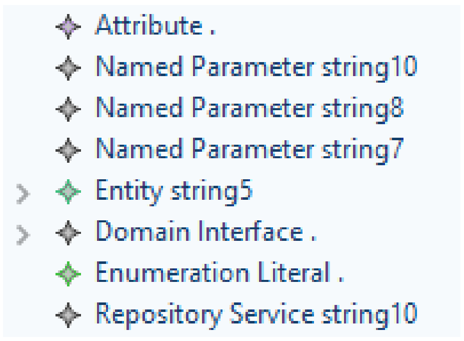
The current version of Efinder does not support generating a pair of models where one refers to another. It only support generating one monolithic model. But the ASOME tool needs to take one ASOME model and one generator model as input and the generator model should refer to the ASOME model. Therefore, we developed a script to split the model into one ASOME model and one generator model. The code is listed in Appendix B.6.

Take a monolithic model generated by Efinder as an example. The model is shown in Figure A.9. This monolithic model contains both objects of the generator metamodel and objects of the ASOME metamodel. Using the script, the model can be split into an ASOME model shown in Figure A.10 and a generator model shown in Figure A.11. This feature should be integrated into Efinder.



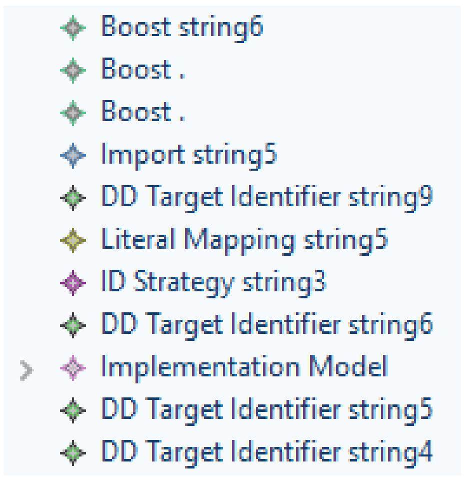
```
  Boost string6
  Boost .
  Boost .
  Attribute .
  Named Parameter string10
  Named Parameter string8
  Named Parameter string7
  Import string5
  DD Target Identifier string9
> Entity string5
  Literal Mapping string5
> Domain Interface .
  Enumeration Literal .
  ID Strategy string3
  DD Target Identifier string6
> Implementation Model
  DD Target Identifier string5
  DD Target Identifier string4
  Repository Service string10
```

Figure A.9: a monolithic model generated by Efinder



```
  Attribute .
  Named Parameter string10
  Named Parameter string8
  Named Parameter string7
> Entity string5
> Domain Interface .
  Enumeration Literal .
  Repository Service string10
```

Figure A.10: first figure



```
  Boost string6
  Boost .
  Boost .
  Import string5
  DD Target Identifier string9
  Literal Mapping string5
  ID Strategy string3
  DD Target Identifier string6
> Implementation Model
  DD Target Identifier string5
  DD Target Identifier string4
```

Figure A.11: second figure

# Appendix B

# Code

## B.1 QVTo Transformation for UML2Relational

```
modeltype UML uses 'http://www.example.org/uml';
modeltype RELATIONAL uses 'http://www.example.org/relational';

transformation class2relational(in uml : UML, out RELATIONAL);



main() {

  uml.rootObjects()[UML::Model]->map model2schema();
}
mapping UML::Model::model2schema() : RELATIONAL::Schema {
  type := self.type->map type2type();
  table := self.classes->map class2table();
  table += (self.classes.attribute
    ->select(e | e.multivalued and e.classifier.oclIsKindOf(Classes)))
    ->map multiClassAttribute2table()
  ->union(self.classes.attribute
    ->select(e | e.multivalued and e.classifier.oclIsKindOf(Type))
    ->map multiPrimAttribute2table());
}

mapping UML::Classes::class2table() : RELATIONAL::Table {
  name := self.name;
  var primaryKey := object RELATIONAL::Column {name := 'objectId'; type :=
      getIntegerType(); isPrimaryKey := true};
  column := Sequence{primaryKey}
  ->union(self.attribute
    ->select(e | not e.multivalued and e.classifier.oclIsKindOf(Classes))->map
        singleClassAttribute2columns())
  ->union(self.attribute
    ->select(e | not e.multivalued and e.classifier.oclIsKindOf(Type))->map
        singlePrimAttribute2columns());

}

mapping UML::Attribute::singleClassAttribute2columns() : RELATIONAL::Column {
    name := self.classifier.name + "Id";
    type := getIntegerType();
}

mapping UML::Attribute::singlePrimAttribute2columns() : RELATIONAL::Column {
    name := self.name;
    type := getClassifer2Type(self.classifier);
}

mapping UML::Attribute::multiPrimAttribute2table() : RELATIONAL::Table {
    name :=  self.classes.name + "_" + self.name;
    column := object RELATIONAL::Column {name := self.name; type :=
        getClassifer2Type(self.classifier)};
    column += object RELATIONAL::Column {name := self.classes.name + "Id"; type :=
        getIntegerType()};
}

mapping UML::Attribute::multiClassAttribute2table() : RELATIONAL::Table {
    name := self.classes.name + "_" + self.name;
    column := object RELATIONAL::Column {name := self.classes.name + "Id"; type :=
        getIntegerType()};
    column += object RELATIONAL::Column {name := self.name + "Id"; type :=
        getIntegerType()};
}


query getClassifer2Type(in targetType : UML::Classifier) : RELATIONAL::Type{

  return targetType.oclAsType(UML::Type).map type2type();
}

query getIntegerType() : RELATIONAL::Type {
       return UML::Type.allInstances()->select(e | e.name = "Integer")->asSequence
           ()->first().oclAsType(UML::Type).map type2type();
}
query UML::Classifier::getClassifierType() : UML::Type {
  return self.oclAsType(UML::Type)
}

mapping UML::Type::type2type() : RELATIONAL::Type{
    name := self.name;
}
mapping UML::Classifier::classifier2type() : RELATIONAL::Type{
    name := self.name;
}
mapping UML::Classes::class2type() : RELATIONAL::Type{
    name := self.name;
}
```

Listing B.1: UML2Relational.qvto

## B.2 uml.use generated by TractsTool

```
model Class

class Model
attributes
end

class Classes < Classifier
attributes
end

class Attribute
attributes
 name : String
 multivalued : Boolean
 classifier : Classifier
end

class Type < Classifier
attributes
end

class Classifier
attributes
 name : String
end

composition contains_Classes between
```

```
Model [1] role classes_Classes_Model_derived
Classes [0..*] role classes_Classes_Model ordered
end

composition contains_Type between
Model [1] role type_Model_Type_derived
Type [1..*] role type_Model_Type ordered
end

composition has_Attribute between
Classes [1..1] role classes_Attribute_Classes
Attribute [0..*] role attribute_Attribute_Classes ordered
end
```

Listing B.2: Class.use

## B.3 Generated USE Specifications By TractsTool

```
model uml

class src_Model
attributes
end

class src_Classes < src_Classifier
attributes
end

class src_Attribute
attributes
 name : String
 multivalued : Boolean
end

class src_Type < src_Classifier
attributes
end

class src_Classifier
attributes
 name : String
end

class trg_Table
```

```
attributes
 name : String
end

class trg_Type
attributes
 name : String
end

class trg_Schema
attributes
end

class trg_Column
attributes
 isPrimaryKey : Boolean
 name : String
end

composition src_classes_Classes_Model between
src_Model [0..*] role classes_Classes_Model_derived
src_Classes [0..*] role classes_Classes_Model ordered
end

composition src_type_Model_Type between
src_Model [0..*] role type_Model_Type_derived
src_Type [0..*] role type_Model_Type ordered
end

association src_classifier_Attribute_Classifier between
src_Attribute [0..*] role classifier_Attribute_Classifier_derived
src_Classifier [0..1] role classifier_Attribute_Classifier
end

composition src_attribute_classes_Attribute_Classes between
src_Classes [1..1] role classes_Attribute_Classes
src_Attribute [0..*] role attribute_Attribute_Classes ordered
end

composition trg_table_Schema_Table between
trg_Schema [0..*] role table_Schema_Table_derived
trg_Table [0..*] role table_Schema_Table ordered
end

composition trg_type_Schema_Type between
```

```
trg_Schema [0..*] role type_Schema_Type_derived
trg_Type [0..*] role type_Schema_Type ordered
end

association trg_type_Column_Type between
trg_Column [0..*] role type_Column_Type_derived
trg_Type [1..1] role type_Column_Type
end

composition trg_column_table_Column_Table between
trg_Table [1..1] role table_Column_Table
trg_Column [1..*] role column_Column_Table ordered
end

class CTDashboard
attributes
  MultiValued: Boolean
  PrimAttr: Boolean
  ClassAttr: Boolean

operations
  MultiValued_OP(): Boolean = Attribute.allInstances->exists(a | a.multivalued =
      true)
  PrimAttr_OP(): Boolean = Attribute.allInstances->exists(a | a.classifier.
      oclIsTypeOf( Type))
  ClassAttr_OP(): Boolean =  Attribute.allInstances->exists(a | a.classifier.
      oclIsTypeOf( Classes))
end

context CTDashboard inv CT_Operation :
  MultiValued = MultiValued_OP() and PrimAttr = PrimAttr_OP() and ClassAttr =
      ClassAttr_OP()
```

## B.4 Generated USE Specifications By Efinder

```
model uml

class Model
operations
  oclContainer() : OclAny = oclUndefined(OclVoid)
  oclContents() : Set(OclAny) = self.classifiers->asSet()
end

class Class < Classifier
```

```
operations
  oclContainer() : OclAny = oclUndefined(OclVoid)
  oclContents() : Set(OclAny) = self.attribute->asSet()
end

class Attribute
attributes
  name : String
  multivalued : Boolean
operations
  oclContainer() : OclAny = Class.allInstances()->select(o|o.attribute->includes(
      self))->any(true)
  oclContents() : Set(OclAny) = Set { }
end

class Type < Classifier
operations
  oclContainer() : OclAny = oclUndefined(OclVoid)
  oclContents() : Set(OclAny) = Set { }
end

abstract class Classifier
attributes
  name : String
operations
  oclContainer() : OclAny = Model.allInstances()->select(o|o.classifiers->includes(
      self))->any(true)
  oclContents() : Set(OclAny) = Set { }
end

composition Classifier_model_Model_classifiers between
  Model[1] role model_
  Classifier[*] role classifiers
end

composition Attribute_classes_Class_attribute between
  Class[1] role classes
  Attribute[*] role attribute
end

association Attribute_classifier between
  Attribute[*] role Attribute_classifier_source
  Classifier[0..1] role classifier
end

```

```
constraints

context self : Attribute inv Attribute_0:
  (Attribute.allInstances()->size()) >= (Class.allInstances()->size())

context self : Attribute inv Attribute_1:
  (Attribute.allInstances()->exists(a, b| ((a.name) = (b.name)) and ((a) <> (b))))
      = (false)

context self : Attribute inv Attribute_2:
  (Attribute.allInstances()->exists(a| (a.multivalued) = (true))) = (true)

context self : Attribute inv Attribute_3:
  (Attribute.allInstances()->exists(a| a.classifier.oclIsTypeOf(Type))) = (true)

context self : Attribute inv Attribute_4:
  (Attribute.allInstances()->exists(a| a.classifier.oclIsTypeOf(Class))) = (false)

context self : Class inv Class_5:
  (Class.allInstances()->size()) >= (1)

context self : Class inv Class_6:
  (Class.allInstances()->exists(a, b| ((a.name) = (b.name)) and ((a) <> (b)))) = (
      false)

context self : Class inv Class_7:
  (Class.allInstances()->exists(a| Type.allInstances()->exists(b| (a.name) = (b.
      name)))) = (false)

context self : Class inv Class_8:
  (Class.allInstances()->exists(a| Attribute.allInstances()->exists(b| (a.name) = (
      b.name)))) = (false)

context self : Model inv Model_9:
  (Model.allInstances()->size()) = (1)

context self : Type inv Type_10:
  (Type.allInstances()->size()) >= (2)

context self : Type inv Type_11:
  (Type.allInstances()->exists(a, b| ((a.name) = (b.name)) and ((a) <> (b)))) = (
      false)

context self : Attribute inv nonNull_Attribute_multivalued:
  not self.multivalued.isUndefined()
```

## B.5 The code for generating OCL files with classifying terms

```
public class ClassifyingTerms {
  int size = 0;//num of classifying terms (CT)
  int value = 0;// the maximum characteristic value of the classifying terms, e.g.,
       for 3 CTs, the maximum characteristic value = 111, which is 7 in decimal
  HashMap<String, List<String>> mapOfCTs = new LinkedHashMap<String, List<String
      >>();

  public ClassifyingTerms(LinkedHashMap<String, String> mapOfPaths) throws
      IOException {
    for(Map.Entry<String, String> m : mapOfPaths.entrySet()) {

      String context = m.getKey();
      String path = m.getValue();
      File fileCT = new File(path);
      List<String> CTs = new LinkedList<String>();
      BufferedReader readerCT = new BufferedReader(new FileReader(fileCT));
      String ct = null;
      while ((ct = readerCT.readLine()) != null) {
        CTs.add(ct);
        System.out.println(ct);
      }
      this.size += CTs.size();
      this.mapOfCTs.put(context, CTs);
      readerCT.close();
    }
    //calculate the maximum characteristic value
    int pow = this.size - 1;
    while(pow != -1) {
      this.value += Math.pow(2, pow--);
    }
  }
}
```

Listing B.3: ClassifyingTerms

```
public void generateClassifyingTermOCLFiles(ClassifyingTerms ctInfo, String
    Path_OCLfile) throws IOException {
    int characteristicValue = ctInfo.value;
    while (characteristicValue != -1) {
      Integer characteristicValueTmp = new Integer(characteristicValue);
      String filename = "../../../Class2Relational/" + characteristicValueTmp.
          toString() + ".ocl";
      File fileOut = new File(filename);
      FileWriter fw = new FileWriter(fileOut);
      BufferedWriter builder = new BufferedWriter(fw);
      BufferedReader reader = new BufferedReader(new FileReader(Path_OCLfile));
      String line;
      Integer numOfCT = new Integer(1);
      for(Map.Entry<String, List<String>> m : ctInfo.mapOfCTs.entrySet()) {
        String context = "context " + m.getKey();
        List<String> CTs = m.getValue();
        line = reader.readLine();
        int hasContext = line.indexOf(context);
        while(line != null && hasContext == -1) {
          builder.append(line + "\n");
          System.out.print(line + "\n");
          line = reader.readLine();
          hasContext = line.indexOf(context);
        }
        if(line.indexOf(context) != -1) {
          builder.append(line + "\n");
          for(String ct : CTs) {
            int bit = characteristicValueTmp & 1;
            characteristicValueTmp = characteristicValueTmp >> 1;
            System.out.println(line);
                builder.append("  inv " + "CT" + numOfCT.toString() + ":\n");
                builder.append("\t" + ct + " = " + map.get(bit) + "\n");
            System.out.print("  inv " + "CT" + numOfCT.toString() + ":\n" + "\t" +
                ct + " = " + map.get(bit) + "\n");
            numOfCT++;
          }
        }
      }
      line = reader.readLine();
      while(line != null ) {
        builder.append(line + "\n");
        System.out.print(line + "\n");
        line = reader.readLine();
      }

      characteristicValue--;
      builder.close();

    }

  }
```

Listing B.4: generateClassifyingTermOCLFiles

```
import uml:'uml.ecore'
package uml
context Model inv Model_Size:
  Model.allInstances()->size() = 1

context Type
  inv Type_Size:
    Type.allInstances()->size() >= 2
  inv Type_Name:
    Type.allInstances()->exists( a, b | (a.name = b.name) and a <> b) = false -- there should be no types with the same name

context Classes
  inv Classes_Size:
    Classes.allInstances()->size() >= 1
  inv Unique_ClassesName_AmongClasses:
    Classes.allInstances()->exists( a, b| (a.name = b.name) and a <> b) = false -- there should be no Classes with the same name.
  inv Unique_ClassesName_AmongType:
    Classes.allInstances()->exists( a| Type.allInstances()->exists(b|a.name = b.name)) = false -- there should be no Type with the same name as any classes.
  inv Unique_ClassesName_AmongAttribute:
    Classes.allInstances()->exists( a| Attribute.allInstances()->exists(b|a.name = b.name)) = false -- there should be no Attribute with the same name as any classes.

context Attribute
inv Attribute_Size:
  Attribute.allInstances()->size() >= Classes.allInstances()->size()
inv Unique_AttrName_AmongAttr:
    Attribute.allInstances()->exists( a, b| (a.name = b.name) and a <> b) = false -- there should be no Classes with the same name.
inv CT1_MultiValued:
  Attribute.allInstances()->exists(a | a.multivalued = true) = true --Classifying Term 1: there exists at least one attribute that is multivalued
inv CT2:
  Attribute.allInstances()->exists(a | a.classifier.oclIsTypeOf( Type)) = true  -- Classifying Term 2: there exists at least one attribute whose type is a primitive type
inv CT3:
  Attribute.allInstances()->exists(a | a.classifier.oclIsTypeOf( Classes)) = true --Classifying Term 3: there exists at least one attribute whose type is a class type
```

```

endpackage
```

Listing B.5: Partition7.ocl

```

public class ClassTest extends AbstractEmfOclTest{

   public void registerMetamodel(String pathOfEcoreModel) {
   Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
         "ecore", new EcoreResourceFactoryImpl());
   ResourceSet rs = new ResourceSetImpl();
   // enable extended metadata
   final ExtendedMetaData extendedMetaData = new BasicExtendedMetaData(rs.
       getPackageRegistry());
   rs.getLoadOptions().put(XMLResource.OPTION_EXTENDED_META_DATA,
       extendedMetaData);
   URI uriOfYourModel = URI.createURI(pathOfEcoreModel);
   Resource r = rs.getResource(uriOfYourModel, true);
   EObject eObject = r.getContents().get(0);
   if (eObject instanceof EPackage) {
       EPackage p = (EPackage)eObject;
       EPackage.Registry.INSTANCE.put(p.getNsURI(), p);
   }
  }

  public void configureAndRunTest(String pathOfOCLFile, String pathOfOutputXMI)
      throws FileNotFoundException, IOException {
   CompleteOCLStandaloneSetup.doSetup();
   Model pivot = loadOclDocumentFromURI(pathOfOCLFile);

   // Assume that in all tests there is a root class called Model
   TestBoundsProvider boundsProvider = new TestBoundsProvider()
       .withInterval("Model", 1, 1)
       .withInterval("Class", 1, 1)
       .withInterval("Type", 1, 2)
       .withInterval("Attribute", 1, 2);

   UseMvFinder finder = new UseMvFinder()
       .withBoundsProvider(boundsProvider);

   EFinderRunner runner = EFinderRunner.
       withOclModel(pivot).
       withFinder(finder);

   Result result = runner.find();
```

```
    System.out.print(result.isSat());
    assertTrue(result.isSat());

    if (result.isSat()) { //Only when the constraints are satisfiable, the model
        will be saved
      HashMap<String, Object> opts = new HashMap<String, Object>();
      opts.put(XMIResource.OPTION_SCHEMA_LOCATION, true);
      Resource model = result.getWitness().getResource();
      model.save(new FileOutputStream(pathOfOutputXMI), opts);
    }
  }

  @Test
  public void classTest() throws FileNotFoundException, IOException {
    //register the uml metamodel in EMF
    registerMetamodel("Class2Relational/uml.ecore");
    //generate classifying term info; the key is the context name, and the value is
         file path of the classifying terms
    LinkedHashMap<String, String> map = new LinkedHashMap<String, String>();
    map.put("Attribute", "Class2Relational/CTAttribute.text");
    ClassifyingTerms ctInfo = new ClassifyingTerms(map);
    String pathOCL = "Class2Relational/Class2Relational.ocl";
    //generate all OCL files
    generateClassifyingTermOCLFiles(ctInfo, pathOCL);
    //start model generation. Iterate from the maximum characteristic value to zero
    Integer characteristicValue = new Integer(ctInfo.value);
    while (characteristicValue != -1) {
      String pathOfOCLFile = "Class2Relational/" + characteristicValue.toString() +
           ".ocl";
      String pathOfOutputXMI = "outputs/result" + characteristicValue.toString() +
           ".xmi";
      configureAndRunTest(pathOfOCLFile, pathOfOutputXMI);
      characteristicValue--;
    }
  }
}
```

Listing B.6: The class ClassTest

## B.6 The code for spliting a model into one ASOME model and one generator model

```
ResourceSet metaResourceSet = new ResourceSetImpl();
```

```

  Resource dataModel = metaResourceSet.createResource(URI.createURI(
      pathOfOutputXMI1));
  Resource generatorModel = metaResourceSet.createResource(URI.createURI(
      pathOfOutputXMI2));

  EList<EObject> listOfdataModel = dataModel.getContents();
  EList<EObject> listOfgeneratorModel = generatorModel.getContents();


  while(model.getContents().size() >= 1) {
    EObject ob = model.getContents().get(0);
    String nameOfEPackage = ob.eClass().getEPackage().getName();
    System.out.println(nameOfEPackage);
    if (nameOfEPackage.equals("imp")) {
      listOfgeneratorModel.add(ob);
    }
    else {
      listOfdataModel.add(ob);
    }
  }

  dataModel.save(new FileOutputStream(pathOfOutputXMI1), opts);
  generatorModel.save(new FileOutputStream(pathOfOutputXMI2), opts);
```

Listing B.7: The code for spliting a model into one ASOME model and one generator model