A Novel Approach to Mutation Operator Design for MDE Languages

Faisal Haji M. Alhwikem

A thesis submitted in fulfillment
of the requirements for the degree of
Doctor of Philosophy

University of York

Computer Science

December 2019

Abstract

Due to the increasing reliance on the software of systems, such as enterprise systems, a wide array of approaches has been found to facilitate the development of software for such systems. The growth of system complexity, however, has provoked concerns about the quality of the software. One approach that copes with complexity is model driven engineering that uses models containing only essential domain concepts, in order to represent complex systems. With some level of automation, models are then maintained by programs that are prone to error, as they are man-made. In order to find errors in programs, software engineers use mutation testing to build strong test inputs by introducing faults into the tested software using mutation operators. They then study if the introduced faults are detected by the test inputs. There have been few attempts to design mutation operators for model driven languages, which have common metamodeling language models, compared with traditional programming languages. This thesis presents an effective language-agnostic approach for mutation operator design for the rapidly emerging model driven engineering languages by considering metamodeling languages. The approach produces generic operators that can be instantiated to generate concrete ones for a given language model, which can be used to mutate program models that conform to the language model. The approach and generic operators are evaluated using empirical mutation analysis experiments over programs written in the ATL and EOL languages. The results show that the generic operators generated from the approach are instantiatable over ATL and EOL metamodels and have produced low proportions of invalid and equivalent mutants that can impact negatively on any mutation testing process. Also, the generic operators have produced useful mutants such as live and not trivially detected kinds of mutants.

For my family.

Contents

Abstract iii

Contents xi

List of Tables xiii

List of Figures xv

List of Algorithms xv

List of Listings xix

Acknowledgements xix

Declaration of Authorship xxi

1 Introduction 1
- 1.1 Motivation and Research Hypothesis . . . 2
- 1.2 Research Methodology . . . 3
- 1.3 Thesis Contributions . . . 5
- 1.4 Thesis Structure . . . 5

2 Literature Review 7
- 2.1 Model Driven Engineering . . . 7
  - 2.1.1 Models, metamodels and modeling languages . . . 8
  - 2.1.2 Model management languages . . . 10
  - 2.1.3 MDE in practice . . . 13
  - 2.1.4 Benefits of MDE . . . 17
- 2.2 Mutation Testing . . . 18
  - 2.2.1 Software testing: concepts and characteristics . . . 19
  - 2.2.2 Mutation testing and structural coverage metrics . . . 21
  - 2.2.3 Mutation operators and mutants . . . 22
  - 2.2.4 Mutation operator injection . . . 29
  - 2.2.5 Challenges to mutation testing . . . 30
- 2.3 Testing and Mutation Testing in MDE . . . 31
  - 2.3.1 Testing in MDE . . . 32
  - 2.3.2 Mutation Testing in MDE . . . 37
- 2.4 Chapter Summary . . . 42

3 Analysis and Hypothesis **43**
3.1 Problem Analysis . . . 43
3.2 Research Hypothesis . . . 45
3.3 Research Objectives . . . 45

4 A Mutation Operator Design Approach **47**
4.1 A Design Approach using a Generic Meta-metamodel . . . 48
4.1.1 Conformance restriction . . . 50
4.2 Abstract Mutation Operators (AMO) . . . 50
4.2.1 AMO-single . . . 52
4.2.2 AMO-multiple . . . 53
4.3 Adaptation of the Approach to Ecore . . . 54
4.3.1 Conformance restriction . . . 55
4.3.2 Further restrictions for Ecore . . . 56
4.3.3 Adaptation of AMOs to Ecore . . . 57
4.4 Instantiation of AMOs . . . 58
4.4.1 Example: a replacement CMO for property `condition` . . . 59
4.4.2 Example: an addition CMO for property `statements` . . . 60
4.4.3 Example: a deletion CMO for property `statements` . . . 61
4.4.4 Example: a replacement CMO for property `method` . . . 62
4.5 Chapter Summary . . . 63

5 Epsilon Mutator (EMU) **65**
5.1 Mutant Integration Layer . . . 66
5.2 Abstract Syntax . . . 67
5.3 Concrete Syntax . . . 68
5.4 Execution Semantics with Examples . . . 70
5.4.1 CMO implementation of Example 4.4.1 . . . 70
5.4.2 CMO implementation of Example 4.4.2 . . . 71
5.4.3 CMO implementation of Example 4.4.3 . . . 71
5.4.4 CMO implementation of Example 4.4.4 . . . 73
5.5 Chapter Summary . . . 73

6 Evaluation: Empirical Mutation Analysis **75**
6.1 Experiment Questions . . . 76
6.2 Experimental Approach . . . 77
6.3 Candidate Programs . . . 79
6.4 Test Models . . . 81
6.5 Concrete Mutation Operators . . . 83
6.5.1 Systematic mutation operator definition process . . . 83
6.5.2 ATL and EOL overlap mutation operators . . . 85
6.5.3 ATL concrete mutation operators . . . 87
6.5.4 EOL concrete mutation operators . . . 94
6.6 Results . . . 103
6.6.1 Empirical results of ATL programs . . . 103
6.6.2 Empirical results of EOL programs . . . 110
6.6.3 Research Hypothesis Evaluation . . . 118

7 Conclusion **121**
7.1 Thesis Contributions 122
7.1.1 Systematic Mutation Design 122
7.1.2 AMOs 122
7.1.3 Evaluation 123
7.2 Future Work 124
7.2.1 Further experiments and evaluations 124
7.2.2 Additional constraints 125
7.2.3 Independent EMU syntax 125

A Mutation Operators **127**
A.1 ATL Mutation Operators 127
A.2 EOL Mutation Operators 138

B Test Models **143**
B.1 Test Models for ATL Candidate Programs 143
B.2 Test Models for EOL Candidate Programs 147

C Mutation Analysis Results **155**
C.1 ATL Complete Results 155
C.2 EOL Complete Results 156

Bibliography **159**

List of Tables

2.1 Total mutation operators for programming languages 27
2.2 Mottu et al. [96, 97] generic mutation operators 38
2.3 Khan and Hassine [100] ATL mutation operators 39
2.4 Troya et al. [101] ATL mutation operators 40
2.5 Cuadrado et al. [102] ATL mutation operators 41

4.1 A list of EDatatypes provided by Ecore 58

6.1 ATL metamodel coverage and model instances of candidate programs . . 80
6.2 EOL metamodel coverage and model instances of candidate programs . . 80
6.3 Valid and invalid mutants of mutating ATL programs (stage one) 104
6.4 Overall results of executing ATL mutants (stage two) 104
6.5 Mutation operators that only generated invalid mutants during stage one 105
6.6 Mutation operators that only generate invalid mutants during stage two . 106
6.7 Mutation operators that contributed to live mutants 107
6.8 Mutation operators that contributed to killed mutants 109
6.9 Valid and invalid mutants of mutating EOL programs (stage one) 111
6.10 Overall results of executing EOL mutants (stage two) 111
6.11 Mutation operators that contributed to invalid mutants in stage one . . . 112
6.12 Mutation operators that contributed to invalid mutants in stage two . . . 113
6.13 Mutation operators that contributed to live mutants 116
6.14 Mutation operators that contributed to killed mutants 117
6.15 Mutation operators that contributed to equivalent mutants 118

C.1 ATL mutation operators and their produced mutants 156
C.2 EOL mutation operators and their produced mutants 157

List of Figures

1.1 Outline of thesis research methodology 4

2.1 Metamodeling hierarchy and relations between models and modeling languages (from [2]) 9
2.2 An example of abstract syntax of Listing 2.1 10
2.3 A typical MDE management task 11
2.4 The core modeling concepts in Ecore from [15] 14
2.5 A tree-based metamodel (Ecore) of the example in Fig. 2.2 14
2.6 The architecture of Epsilon [18] 15
2.7 Abstract syntax of EPL taken from [18] 16
2.8 Abstract syntax of EVL taken from [17] 17
2.9 Different software development phases and testing levels, from Figure 1.2 in [12] 19
2.10 Typical mutation testing process adapted from [29] 23

3.1 Typical MDE metamodeling level and modeling levels 44

4.1 Typical MDE metamodeling hierarchy and generic meta-metamodel . . . 48
4.2 A generic metamodel 49
4.3 AMOs and CMOs 51
4.4 A representation of core elements of Ecore taken from [15] 55
4.5 The abstract syntax of the MiniLang metamodel 59
4.6 A fragment of an instance model of the MiniLang metamodel 4.5 used for the instantiation example 4.4.1 60
4.7 A fragment of an instance model of the MiniLang metamodel 4.5 used for the instantiation example 4.4.2 61
4.8 A fragment of an instance model of the MiniLang metamodel 4.5 used for the instantiation example 4.4.3 62
4.9 A fragment of an instance model of MiniLang metamodel 4.5 used for the instantiation example 4.4.4 62

5.1 Epsilon *IModel* abstraction and Mutant Integration Layer 67
5.2 The abstract syntax of EMU 68
5.3 A fragment of the metamodel in Fig. 4.5 71
5.4 A fragment of the metamodel in Fig. 4.5 72
5.5 A fragment of the metamodel in Fig. 4.5 73
5.6 A fragment of the metamodel in Fig. 4.5 73

6.1 Stage one – mutants production process 78

6.2 Stage two – mutants execution process 79
6.3 Test models generation process 81
6.4 ATL Metamodel – ATL module concept 88
6.5 ATL Metamodel – ATL–OCL related concepts 89
6.6 EOL Metamodel – EOL module 95
6.7 EOL Metamodel – EOL expressions 97
6.8 EOL Metamodel – EOL statements 100
6.9 EOL Metamodel – EOL types 102

B.1 Book metamodel 143
B.2 HTML metamodel 145
B.3 Make metamodel 147
B.4 Table metamodel 148
B.5 DirectedGraph metamodel 149

List of Algorithms

1 Candidate programs selection 81
2 Systematic process for generating concrete mutations for a metamodel . . . 84

List of Listings

2.1 An example of concrete syntax 10
2.2 An Emfatic textual view of the metamodel in Fig. 2.2 14

4.1 An example of code that is modeling in Fig. 4.6 60

5.1 EMU concrete syntax 68
5.2 A CMO implementation for Example 4.4.1 71
5.3 A CMO implementation for Example 4.4.2 72
5.4 A CMO implementation for Example 4.4.3 73
5.5 A CMO implementation of Example 4.4.4 73

6.1 Original EOL metamodel fragment expressed using EMFatic 114
6.2 Improved EOL metamodel fragment of Listing 6.1 114

A.1 `CMO-single-REP(IfStat condition)` 127
A.2 `CMO-Single-REP(VariableDeclaration type)` 129
A.3 `CMO-multiple-DEL(Operation parameters)` 134
A.4 `CMO-multiple-REP(Module elements)` 135
A.5 `CMO-multiple-REP(MatchedRule children)` 135
A.6 `CMO-single-REP(Binding propertyName)` 136
A.7 `CMO-single-REP(BindingStat source)` 136
A.8 `CMO-single-REP(OclFeatureDefinition feature)` 137
A.9 `CMO-multiple-DEL(OperationDefinition parameters)` 138
A.10 `CMO-single-REP(Expression resolvedType)` 138
A.11 `CMO-multiple-REP(Block statements)` 140
A.12 `CMO-single-ADD(ModelElementType modelName)` 141
A.13 `CMO-single-REP(ModelElementType modelName)` 142
A.14 `CMO-multiple-REP(MethodCallExpression arguments)` 142

B.1 EMG generator code for Book metamodel 143
B.2 EMG generator code for HTML metamodel 144
B.3 EMG generator code for Make metamodel 145
B.4 EMG generator code for Table metamodel 146
B.5 EMG generator code for DirectedGraph metamodel 147
B.6 EMG generator code for Ecore metamodel of EuGENia 149
B.7 EMG generator code for Ecore metamodel of Incremental EVL 152

Acknowledgements

First and foremost, my great gratitude goes to my supervisors Prof. Richard Paige and Dr. Rob Alexander. Without their fruitful support, guidance and patience, this work would never be completed. I will be forever grateful to Richard for accepting me as his PhD student and to Rob for his valuable support, discussions and comments. I also have to mention and thank my previous co-supervisor Dr. Louis Rose for his support and help during my early years of my PhD. Also, my great thanks go to Prof. Fiona Polack and Dr. Ibrahim Habli for their important assessments and valuable insights about my work.

I must give sincere gratitude to Prof. Dimitris Kolovos, Dr. Horacio H. Rodriguez, Dr. Antonio García-Domínguez and Dr. Ran Wei for their advice and support during my studies. I am also most grateful to all my colleagues and friends in the Computer Science Department and its visitors with whom I appreciate the various fruitful discussions and a friendly working environment. I would like to make special mentions to Dr. Adolfo Sanchez-Barbudo Herrera, Dr. Simos Gerasimou, Dr. Colin Paterson, Dr. Konstantinos Barmpis and Dr. Athanasios Zolotas.

My deepest thanks go to whom I love the most, my family; and to whom I dedicate this work. To both my mother and father for their endless loving care and encouragement that made me who I am today, I would be lost without them. My deepest thanks also go to my wonderful life partner, Dalal, for her unlimited love and support, and to my dearest children for being always around me whenever I need motivation. I must give also my great gratitude to my brothers and sisters for their encouragement.

Last but not least, I must pay great thanks to Qassim University for its financial support to my PhD scholarship under umbrella of the Ministry of Higher Education of the Kingdom of Saudi Arabia.

Declaration of Authorship

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Part of the work described in this thesis has been previously published by the author in:

- F. Alhwikem, R. Paige, L. Rose, R. Alexander. A Systematic Approach for Designing Mutation Operators for MDE Languages. In: Proceedings of the 13th Workshop on Model-Driven Engineering, Verification and Validation co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), October 3, 2016, pp. 54–59.

Chapter 1

Introduction

The use of software is ubiquitous in everyday life. It has helped us to increase productivity by reducing the amount of labor that is usually dedicated to repetitive and tedious operations. It has also allowed us to build software systems, such as business systems, safety systems, healthcare systems and many more.

With the increasing demands on software reliability and the quality of complex systems (enterprise systems), however, software engineering has stepped up to provide principles and practices that facilitate and guide the development of quality software for such systems. Model-Driven Engineering (MDE) is a unique software engineering practice that tries to solve issues related to complex systems by allowing software engineers to create models that describe the systems at different levels [1]. It uses models, which contain the essential domain concepts of a complex system, as the main artifacts for activities, such as (semi-)automated development, process engineering, interoperability, reverse engineering and modeling languages. Also, MDE provides the principles that make models amenable to automated analysis, processing and management due to their conformance to a set of formal concepts defined in metamodels. These are specification models of a higher level of abstraction, containing the necessary modeling elements and constraints to create and validate models.

Models are not working artifacts in themselves because of their level of abstraction and lack of implementation details. Therefore, they are managed by MDE programs and transformed into new types of artifact. They are queried for and modified with data, and verified and validated with respect to particular specifications, among others. These management tasks are facilitated by rapidly emerging MDE languages, each of which is tailored to a specific task. As such, they are referred to as task-specific management languages. Since anything can be represented as a model [2], languages in model driven engineering are sometimes represented as models created with metamodeling languages.

Therefore, models of languages also lend themselves to automated analysis and processing.

MDE programs, just like any software, are extensively researched in order to improve their quality and reliability on a par with the constantly and rapidly evolving high complexity software systems. One approach is to perform testing sequences by executing programs with a set of test data and then examining their outputs with respect to the relevant software specifications.

A testing technique that powerfully can build and assess test inputs for a piece of software is mutation testing [3]. This involves intentionally introducing faults into the implementation of software using mutation operators, which are predefined rules that target specific language concepts defined in a language specification or grammar. This produces a list of modified versions of the software that are then executed against a test set in order to determine their adequacy in detecting the introduced faults. If any modified version is not detected, additional tests need to be built. This will enhance the quality of the test set in forcing the detection of the modified versions, consequently improving confidence in the target software.

This thesis presents a novel approach that enables users (mainly test engineers) to define mutation operators for MDE languages, which are task-specific languages for model management, in an effective and efficient manner. Section 1.1 of this chapter touches upon the motivation behind this thesis along with the research hypothesis. The research methodology that was used to validate the hypothesis and fulfill its research objectives is presented in Section 1.2. The contributions of the thesis are highlighted in Section 1.3, followed by a presentation of its structure in Section 1.4.

1.1 Motivation and Research Hypothesis

Traditionally, computational cost has been the main challenge behind mutation testing. This is an issue that originates in the poor design of mutation operators. Poor design implies the definition of a set of mutation operators that, when applied, produces useless mutants (modified versions of software that do not challenge test developers to improve their test set). For instance, invalid mutants that do not conform to the language and fail to execute against any test set are considered useless. Another category of useless mutants is equivalent mutants that produce the same output as the original program but cannot be detected. In fact, it is impossible to build test inputs that would force their detection.

Furthermore, poor mutation operator design undermines the definition of a good set of operators that, when applied, produces live mutants (mutants that are not detected but are “killable” if more tests are added). Tackling the results of poor mutation operator design has been the object of intensive focus in traditional programming languages but has received limited attention as regards MDE languages.

In addition to the shortcomings associated with poor mutation operator design, an additional drawback can be found in the MDE context. Due to the rapid emergence of MDE languages in both the industry and academia, mutation operator design has become a tedious and unproductive task, especially since MDE languages are mostly built on common metamodeling languages. In the literature on mutation testing, test developers define mutation operators for a given language and then use best practices and solutions to control them for best results; detecting or preventing the production of equivalent mutants; and preventing invalid mutants from being produced.

The limited attention of mutation analysis in model driven engineering has received is unfortunate, as it may be a particularly good candidate for effective mutation design. This thesis explores this path of integrating mutation testing and mutation operator design in an MDE context using the key principles of model driven engineering – that everything is a model – in order to derive *systematically* mutation operators. In particular, for MDE languages that are built on common metamodeling languages, such as MOF or EMF/Ecore, it is feasible to generate a set of *generic* mutation operators. These can thereafter be instantiated to provide *concrete* mutation operators for specific MDE language models (or metamodels).

A further aim of this thesis is to address a solution for tackling poor mutation operator design. To this end, it will ensure that mutations are checked in order to reduce the risk of generating useless (invalid or equivalent) mutants. In terms of valuable mutants, this thesis will validate that live mutants (i.e. those detectable with added test inputs) are produced with the proposed generic mutation operators. Since MDE languages are emerging in quick succession, the tedious task of designing mutation operators for every language separately along with solutions for poor design will be eliminated using the generic mutation operators derived from metamodeling languages.

1.2 Research Methodology

The research methodology adopted to investigate the research hypothesis is presented in this section and is illustrated in Figure 1.1 (research phases are represented in shaded boxes and inputs/outputs in plain boxes). It follows a typical software development engineering life cycle. In order to enhance the quality of the research, the process is designed to be sequentially repetitive and consists of four distinct phases: beginning with analysis through to design, implementation, evaluation and then back to analysis in order to resolve weaknesses and improve the original proposal.

In the analysis phase, current mutation operator design approaches and model driven engineering metamodeling concepts and principles are analyzed with the aim of proposing a set of requirements for a suitable design approach for current and newly emerging MDE languages. The requirements take into account current solutions of tackling useless mutants, as well as improvements to identified weaknesses. Another outcome of the analysis phase is a proposal for an appropriate methodology and a requirement set for fault injection (that is, mutation operator application) in the context of MDE.

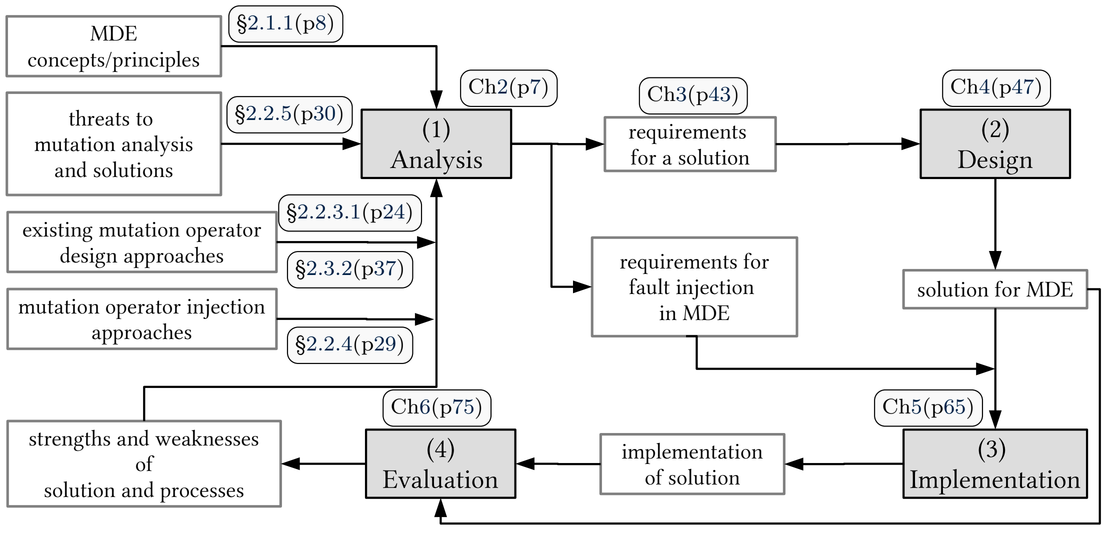
MDE concepts/principles
§2.1.1(p8)
threats to mutation analysis and solutions
§2.2.5(p30)
Ch2(p7)
(1) Analysis
Ch3(p43)
requirements for a solution
Ch4(p47)
(2) Design
existing mutation operator design approaches
§2.2.3.1(p24)
§2.3.2(p37)
mutation operator injection approaches
§2.2.4(p29)
requirements for fault injection in MDE
solution for MDE
Ch6(p75)
Ch5(p65)
strengths and weaknesses of solution and processes
(4) Evaluation
implementation of solution
(3) Implementation

Figure 1.1: Outline of thesis research methodology

In the design phase, the set of requirements identified in the analysis are processed to propose a suitable mutation operator design approach for MDE languages, as per the main research objective of the thesis. Solutions for effective mutation design are integrated into the proposed design approach in which issues of useless mutants are addressed. The proposed design approach is structured in such a way as to make its adaptation and implementation amenable to test developers of MDE programs.

The implementation phase proposes an implementation of the solution of mutation operator design taking into account requirements for fault injection. It is motivated by existing practices used in traditional programming languages.

The evaluation phase applies an experiment to evaluate the proposed mutation design approach. The outcomes of this phase include an assessment of the proposed solution design, its strengths and its weaknesses. All outcomes are input again into the analysis phase for an iterative and improved solution.

1.3 Thesis Contributions

This thesis provides a novel mutation operator design approach for the rapidly emerging MDE languages and its contributions are outlined as follows:

- A novel solution for a design approach to mutation operators for MDE languages. The approach has produced a set of generic mutation operators with constraints that reduce the likelihood of producing useless mutants (Chapter 4).
- An implementation of the proposed solution in which the generic operators can be used to generate concrete ones for a given MDE language, which then can be used to mutate programs of that language (Chapter 5).
- An evaluation of the approach and the generic operators by conducting an empirical mutation analysis against a set of ATL and EOL programs by defining and using a collection of concrete mutation operators for the languages (Chapter 6).

1.4 Thesis Structure

The structure of the thesis is as follows:

- Chapter 1 – Introduction: gives an overview of the thesis and subject disciplines; the motivation for carrying out the research and formulating the hypothesis; thesis research methodology and structure.
- Chapter 2 – Literature review: presents a field review on MDE and mutation testing including related concepts and principles; overviews some of the benefits of and challenges to MDE and mutation testing; surveys related work that attempts to integrate mutation testing into an MDE context and presents its limitations.
- Chapter 3 – Analysis and hypothesis: offers a problem analysis of current mutation operator design approaches and outlines their specific limitations; clarifies the research hypothesis of this thesis, as well as the research objectives to test it.
- Chapter 4 – A mutation operator design approach: presents the novel solution to the research problem outlined in Chapter 3.
- Chapter 5 – Epsilon mutator (EMU): describes an implementation of the novel solution as a prototype language that is metamodel-agnostic and can be used to mutate models. The prototype provides extended functionality to be used in any metamodeling language; presents examples of using the language.

- Chapter 6 – Evaluation: analyzes an empirical mutation; evaluates the proposed mutation operator design approach through the solution implementation.
- Chapter 7 – Conclusion: provides the conclusions of the thesis; presents areas for future work.

Chapter 2

Literature Review

This thesis presents a novel approach for defining mutation operators for MDE languages in a systematic and efficient manner. In order to contextualize this thesis and its objectives, a detailed review of model driven engineering and mutation testing including related concepts, principles, tool support and challenges is presented in this chapter. In addition, a critical discussion of current research in the area of integrating mutation testing in an MDE context is included, focusing on the key limitations that this thesis will try to address.

Accordingly, this chapter is organized into three main sections. Section 2.1 presents a field review on model driven engineering and its related concepts and principles including modeling, metamodeling and tool support. Section 2.2 provides a background on mutation testing, its related concepts and its challenges. Finally, Section 2.3 delves into the body of work that has so far attempted to integrate mutation testing into the context of MDE, providing a critical analysis of its key limitations.

2.1 Model Driven Engineering

Software engineering is a discipline that provides standards, methodologies and tools to facilitate the process of creating quality artifacts. The software development life cycle is a software engineering methodology for building artifacts. Traditionally, software engineers and developers have used models for documentation purposes during the software development process, whereas in other engineering disciplines, models play a key role and are used to understand complex problems by raising the level of abstraction and hiding implementation details [4]. In the early twenty-first century, model driven engineering emerged as a new software development approach treating models as primary artifacts for many activities within the software development process.

Since models, however, are abstract and contain only domain concepts, they do not apply as working software. Hence, model driven engineering relies on automated model management tasks to convert models into other forms of artifact to be applied at multiple stages of the software development process. These management tasks essentially involve model transformation from one or multiple forms of artifact to others, such as models or texts. The stated objectives of such automated tasks are to increase productivity and reliability [1].

The remainder of this section presents the principles and concepts of model driven engineering. Section 2.1.1 provides a detailed overview of models and modeling languages, and the relationship between them. Then, Section 2.1.2 reviews the model management languages used in this thesis. Section 2.1.3 examines MDE in practice, including support for tools relevant to this thesis. Finally, Section 2.1.4 presents some of the benefits of model driven engineering.

2.1.1 Models, metamodels and modeling languages

A model is a representation of a real world phenomenon. This can be a problem, a task, a process or a software system. In model driven engineering, the term representation can refer to either textual or graphical syntaxes and it describes a selected domain concept of a software system under development [5]. Since models are maintained using automated model management tasks, they must be formal and conform to a well-defined set of syntactic and semantic notations. These sets of notations are defined in a *modeling language*. Model conformance studies the well-formedness of models using a set of constraints defined in a modeling language [1]. The purpose of this task is to avoid inconsistency in models when they are maintained.

A modeling language is a specification of notations and constraints used to produce models. Normally, modeling languages are described using models [6]. Therefore, the term *metamodel* refers to a model that describes a modeling language in which specifications and constraints are defined using entities, properties and the relationships between entities. An instance of a metamodel is a model. A model is said to conform to its metamodel if the constraints defined in it are not violated by the model. This relationship is similar to the one between a program and the programming language in which it is written.

Yet, metamodels are also produced using modeling languages referred to as metamodeling languages. This raises the level of abstraction and modeling [1]. In model driven engineering, the term metamodeling hierarchy is used to refer to metamodeling based on language levels. Figure 2.1 presents the metamodeling hierarchy of the relationships between modeling languages and models at one level, and the relationship between the modeling languages and models of one level and those of a higher level.

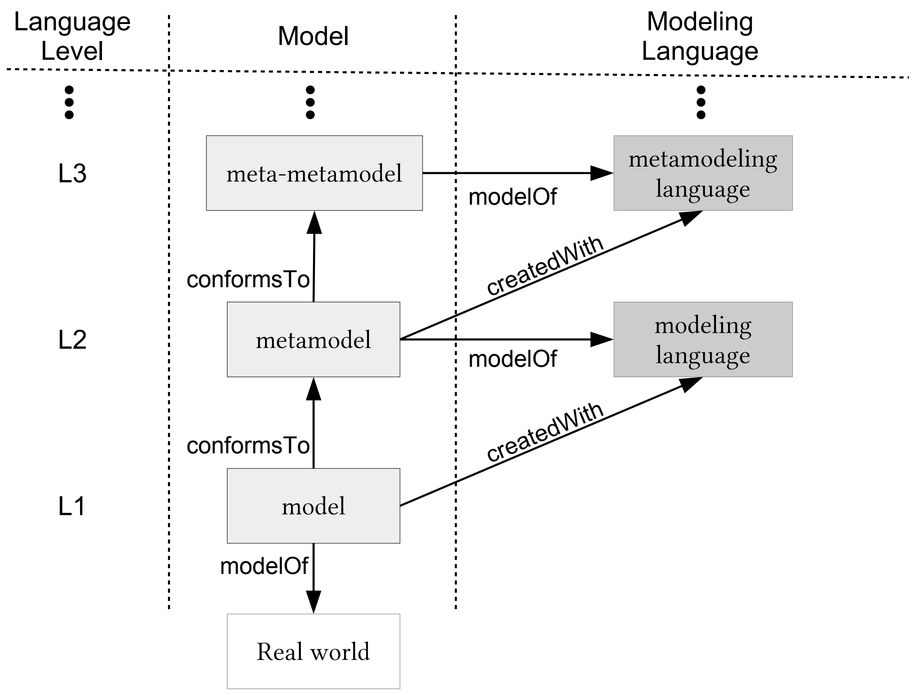

FIGURE 2.1: Metamodeling hierarchy and relations between models and modeling languages (from [2])

There is no specified limit to metamodeling levels. Developing abstraction models of concepts that provide essential elements for users (mainly modelers) to implement lower level models is sufficient. This abstraction model is referred to as a meta-metamodel, namely a model of a metamodeling language, which can be expressed by itself [5, 1]. In practice, a well-known and commonly referenced meta-metamodel is Meta Object Facility (MOF) [7]: a standard metamodeling language introduced by the Object Management Group.[1] A number of the standards provided by OMG facilitates the adaptation of model driven engineering and the definition of models of languages (metamodels).

All aforementioned principles of models, metamodels and modeling languages make model driven engineering a good candidate for mutation testing, widely regarded to build strong tests. As metamodels are encoded in common standard formats, such as MOF, there is ample opportunity for reuse across metamodels: an analysis technique that applies to MOF models can, in principle, be applied to any MDE language model. Analysis techniques that are applied to metamodels can be systematic and automated. Hence, they can be used to derive mutation operators in a straightforward manner. Mutation operators are mutation rules used to inject errors into software as part of the mutation testing technique.

[1]http://www.omg.org

In concluding this section, it is worth mentioning that a modeling language consists of three elements:

- **Concrete syntax:** describes a specific representation of a modeling language by using its notation. It can be a textual representation (as in programming languages) or a graphical representation (as in Unified Modeling Language notation). An example of concrete syntax can be similar to the code below, where `ArithmeticExpression` and `Expression` are non-terminal rules:

```
ArithmeticExpression:
  Expression ('+'|'-'|'*'|'/') Expression
;
```

Listing 2.1: An example of concrete syntax

- **Abstract syntax:** describes the structure and concepts (entities) of a language, and their relationships. This is a similar description to a metamodel. In fact, the term abstract syntax is used interchangeably with the term metamodel. An abstract syntax equivalent of the concrete syntax example above is a notation similar to the following:

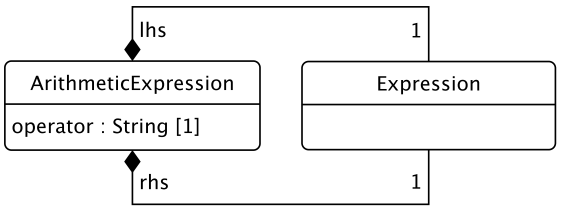

Figure 2.2: An example of abstract syntax of Listing 2.1

- **Semantics:** defines the meaning of concepts of a language (whether in abstract syntax or in concrete syntax).

2.1.2 Model management languages

As mentioned above, since models in model driven engineering contain only selected domain concepts of a real problem or task, they do not qualify as working software systems. Therefore, they are maintained for certain objectives throughout the development life cycle. This management is achieved using languages tailored to a specific purpose. There are specific languages for model transformation of artifact(s) to (new) artifact(s), model validation with respect to a set of metamodel constraints (normally used for checking semantics), model query to fetch and obtain data from models, and many more. Although that the term task-specific languages is used to refer to these management language [5], this thesis uses the term MDE languages to also refer to task-specific model management languages. The typical process of an MDE language is illustrated in Figure 2.3.

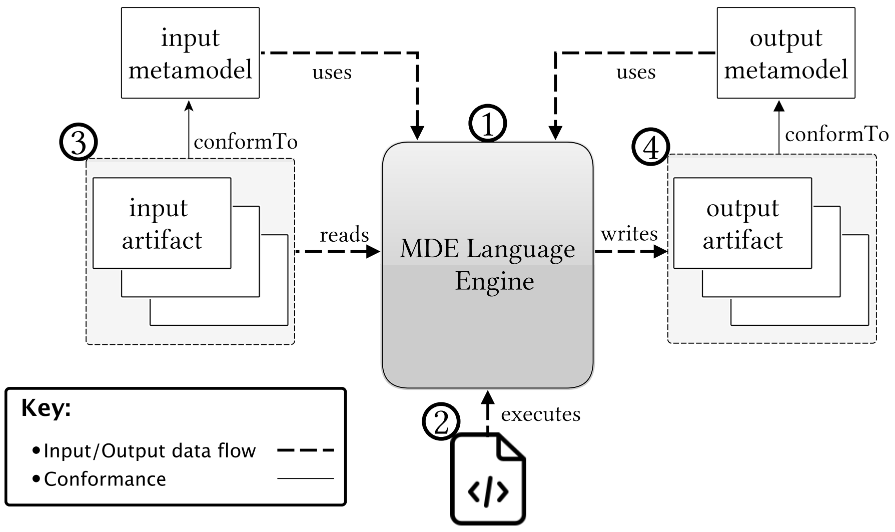

Figure 2.3: A typical MDE management task

An MDE language engine (indicated by ①) executes a given program (as indicated by ②) when the text of the program is processed by a number of elements of the engine, for example the lexical analyzer, the parser or the compiler of the subject language. The engine then reads the input artifacts (whether model or text) indicated by ③ and may use input and/or output metamodels to produce output artifacts (indicated by ④).

Just like any software, MDE programs (indicated by ② in Figure 2.3) are prone to errors. In order to increase confidence in their quality, software engineers use a number of techniques, such as static analysis and software testing. Since model driven engineering lends itself for mutation testing as explained above, the particulars will be introduced in some detail in Section 2.2.

The following sections present the model management languages that are relevant to this thesis, namely model transformation, model validation and model verification.

2.1.2.1 Model transformation

Model transformation is considered a core model management task in MDE [1]. It is used to transform a model, or a set of models, to other useful artifacts. There are three types of model transformations: model-to-model, model-to-text and text-to-model. The process of model transformation requires reading and navigating against input models. It maps specific elements on input models in order to generate (with perhaps additional information) new elements in output models. The process is guided by the so-called *transformation definition*, which is a set of *transformation rules*. A transformation rule describes how a certain set of elements of the source model should be mapped to a set of elements in the target model upon the respective metamodel. There are many languages that allow the definition and execution of model transformation, such as the Atlas Transformation Language (ATL) [8], the Epsilon Transformation Language (ETL) [5], and Query/View/Transformation Relations and Operational (QVTr and QVTo) [9]. There are also languages that support model-to-text transformation. Examples of such languages include Epsilon Generation Language (EGL) [5] and Acceleo [2].

2.1.2.2 Model validation and model verification

Generally, software engineers perform two main tasks in order to increase confidence in software quality. These are validation and verification [10]. This thesis follows Boehm [11], who considers the former as the process of building the right product and the latter as the process of building the product right. Since the essential artifacts in MDE are models, the terms model validation and model verification are deemed appropriate to refer to the verification and validation processes.

Model validation involves the checking of models with respect to their metamodel constraints. In other words, it is the operation that studies a model's validity, as implied by a set of predefined constraints as per its metamodel. The validity of models (that is, their conformance) is important because an invalid model may trigger an implementation to fail when it is processed by an MDE language engine, for example, when navigating to a model element that is not structured correctly according to its metamodel constructs and constraints.

Model verification aims at determining whether, following a model management task (for example, model transformation), models are correct outputs as expected. Incorrect artifacts result from a failure in the program and its implementation [12],which arises when a fault is executed. Faults are the result of errors committed by a software engineer during the implementation phase of the software's development and model verification is the process of determining these faults. In the related literature, there are many testing approaches that have been proposed in order to detect errors committed by programmers. Section 2.3.1 elaborates on verification and testing in model driven engineering.

[2]`https://eclipse.org/acceleo/`

2.1.3 MDE in practice

A very well-known standard that is usually discussed in MDE is Model Driven Architecture, which is a proposed guideline defined by OMG for applying MDE practices in software development [1]. Model Driven Architecture has influenced the software industry through important standards such as MOF, which is used as a meta-metamodel for defining language models (metamodels), and XMI (XML Metadata Interchange) [13], which is a standard format for loading, storing and exchanging models instantiated using MOF. These standards allow modelers to define metamodels and encode them in standard formats that are amenable to systematic and automated analysis. Such automated analysis makes model driven engineering a good candidate for mutation testing, as metamodels can be processed and analyzed in a straightforward manner in order to define mutation operators for models conforming to the analyzed metamodels. The process is easier than analyzing (manually) the grammar of languages within a traditional programming context.

Furthermore, OMG standards facilitate interoperability across different modeling languages, frameworks and tools that are produced by software and language engineers for modeling purposes. Among previously mentioned model transformation languages discussed in Section 2.1.2.1, there is a number of other tools and languages that use these standards as well, including MoDisco for reverse engineering [14], XPand [3] for code generation and Graphical Modeling Framework (GMF) [4] for generating graphical editors for model representation. This section presents the essential tooling support relevant to this thesis, namely the Eclipse Modeling Framework (EMF) [15] and Epsilon [5].

2.1.3.1 Eclipse Modeling Framework (EMF)

This is a framework founded by the Eclipse Foundation[5] that supports model driven engineering and its practices. It provides a number of facilities for modeling and model interchange [16]. EMF uses Ecore as a metamodeling language and provides an implementation of the MOF2.0 standard that helps in the production and management of models and metamodels with a range of modeling concepts, for example `EClass` to define entities, `EAttribute` to define attributes for entities, and `EReference` to define associations between entities. In addition, Ecore provides data types such as `EString`, `EFloat`, `EInt` etc., which are affiliated with Java data types and objects for defining properties of entities (that is, attributes). Figure 2.4 presents the core modeling concepts of Ecore as a metamodeling language and an implementation of MOF2.0.

[3]`https://www.eclipse.org/modeling/m2t/?project=xpand`

[4]`http://www.eclipse.org/modeling/gmp/`

[5]http://www.eclipse.org

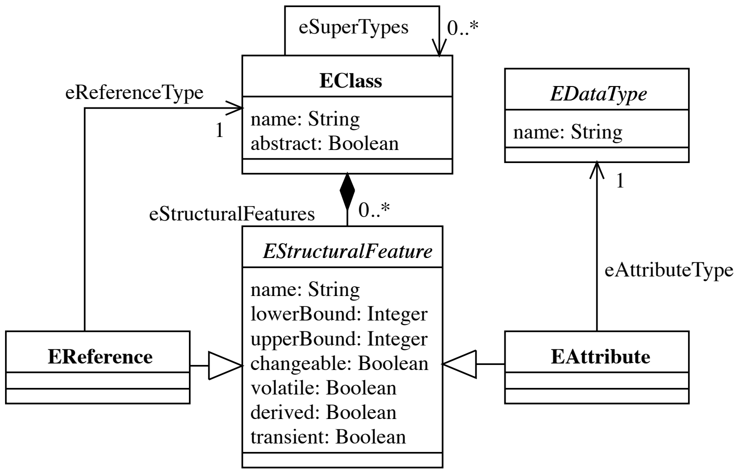

FIGURE 2.4: The core modeling concepts in Ecore from [15]

Furthermore, EMF provides a useful functionality for constructing and building metamodels. Modelers can define their metamodels using a tree-based editor that facilitates the addition and deletion of modeling concepts (entities and properties, including attributes and associations) from a list of predefined meta-classes (as seen in Figure 2.4). Also, the tree-based functionality provides information fields for modeling concepts, through which modelers can edit the values of some meta-properties in a convenient manner. Figure 2.5 shows a tree-based view of the metamodel example discussed in Figure. 2.2.

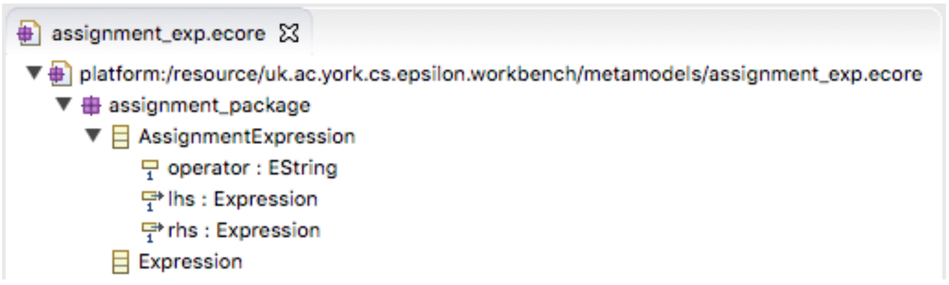

FIGURE 2.5: A tree-based metamodel (Ecore) of the example in Fig. 2.2

Another tool for defining metamodels using Ecore is the EMFatic[6] text editor that relies on EMF and allows modelers to construct metamodels using a textual syntax that is similar to annotated Java language. Modelers can define Ecore model elements as they would do using the tree-based editor. Listing 2.2 gives an EMFatic textual representation of the same metamodel illustrated in Figure. 2.2.

```
@namespace(
 uri="www.cs.york.ac.uk/epsilon/workbench/metamodels/assignment_exp", prefix="assignment_exp"
)
package assignment_package;
```

[6]http://www.eclipse.org/emfatic/, http://wiki.eclipse.org/Emfatic

```

class AssignmentExpression {
 attr String[1] operator;
 val Expression[1] lhs;
 val Expression[1] rhs;
}

class Expression { }
```

Listing 2.2: An Emfatic textual view of the metamodel in Fig. 2.2

2.1.3.2 Epsilon platform

Epsilon is a platform of integrated task specific languages for model management, including model transformation, model merging, model validation and many more. In order to boost interoperability across these different model management tasks, Epsilon provides two essential components: Epsilon Model Connectivity (as illustrated in Figure 2.6), which provides a uniform interface for accessing models conforming to various concrete modeling technologies, such as EMF, XML, CSV etc., and Epsilon Object Language (EOL), which is briefly descripted in the following subsections along with other languages that are related to this thesis. A full descrition of Epsilon languages can be found in Epsilon Book [17].

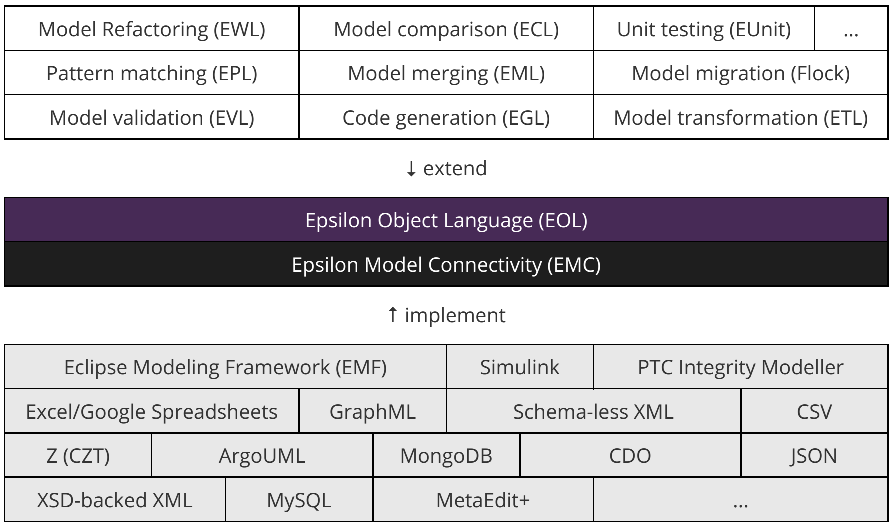

Figure 2.6: The architecture of Epsilon [18]

Epsilon Object Language (EOL) is an imperative expression language useful for navigating, modifying and creating model elements. It provides a mixture of usual programming concepts similar to JavaScript and Object Constraint Language (OCL) [19], such as variable definitions, control flow statements (for example, *if*, *while* or *for*, *switch*), and collection querying functions also known as first order logic operations (such as *select*, *collect* etc.). In addition, EOL support a number of various expressions such as literal expression for supported built-in types, model-feature navigation expression, common binary arthimetical, comparison, and logical expressions.

Furthermore, EOL privides a list of built-in types that extend a super type `Any`. These types are primitive types (String, Boolean, etc.), Collection types (Sequence, OrderedSet, etc.), Map type and model element type. All these built-in types support a number of appropritate operations including, but not limited to, ***`isDefined()`***, ***`isTypeOf()`*** for `Any` types ***`startsWith()`***, ***`toUpperCase()`*** for `String` types and many more.

Epsilon Pattern Language (EPL) is another language of Epsilon platform that provides seamless language constructs to pattern matching, which is often an initial step in most model management operation [18]. EPL built atop of EOL for expression and its abstract syntax is depicted in Figure 2.7.

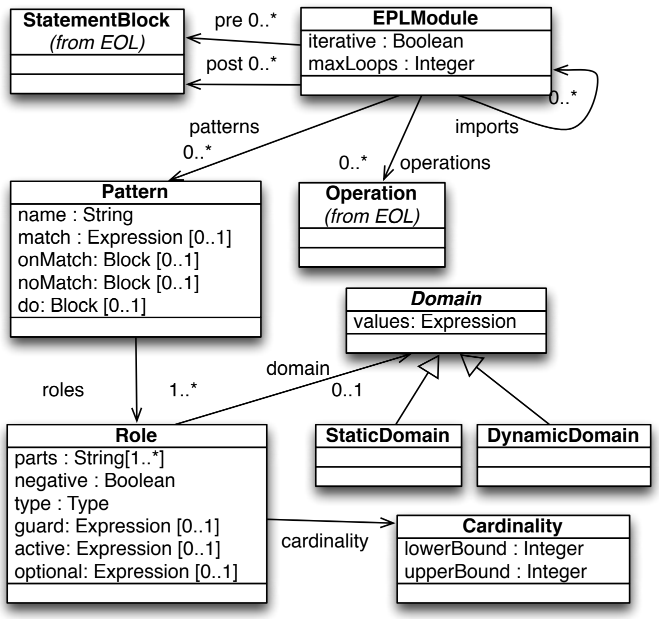

Figure 2.7: Abstract syntax of EPL taken from [18]

An EPL module may contain one or more *pattern*, *pre* (can be used to prepare some elements before pattern execution) and *post* (can be used to check the models after execution) blocks, and/or *operations* or functions. In order to do the pattern matching, a *pattern* must have at least one binding role that is used to query models and to obtain model instances. It is possible to do combination of binding roles each of which target a specific model type. The result of the query, which are instances of model types, can then be used to do matching action by specifying an EOL expression to *match* language concept. The output of this matching can be after that manipulated by *onmatch*, on which elements evaluated positively to the match expression, *nomatch*, on which element evaluated negatively to the match expression), and/or *do* EOL blocks, which can contain arbitrary EOL statements that can be executed on elements obtained by the binding roles and regardless of output of the match expression.

Epsilon Validation Language (EVL) is a model validation language that can be used to define further constraints and semantics, atop of a given metamodel, against which the user want to evaluate models [17]. It is abstract syntax is represented in Figure 2.8.

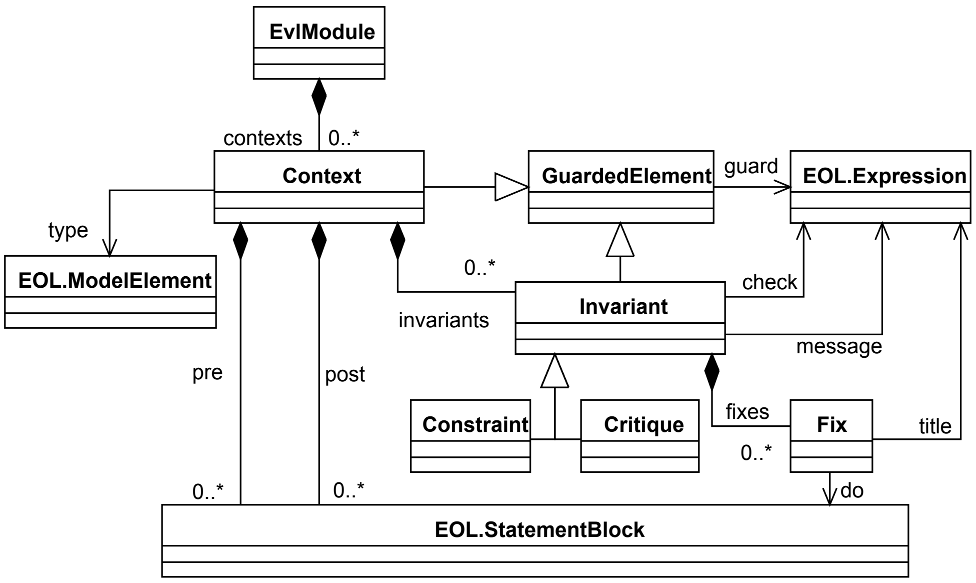

FIGURE 2.8: Abstract syntax of EVL taken from [17]

The user of the language can specify a collection of *invariant* (whether it is a constraint or a critique) that is defined for, or is to be evaluate against, contexts that are specific model element types. Instances of a *context* can be filtered using a *guard* expression to obtain a subset of instances that can be evaluated against invariants. Also, invariant can have a *check* expression that to be evaluated. The user of the language can specify a feedback *message* to be displayed and a collection of *fixes* in case the checking fails.

2.1.4 Benefits of MDE

There are many benefits to using model driven engineering, as identified in the literature. One of these is understandability [20]. Since MDE advocates raising the level of abstraction by using models, it is argued that models are easier to be specified, understood and maintained, compared with other types of artifact that do not use abstraction. Mohagheghi et al. [21] found that the use of models facilitates comprehension among non-technical staff. The same benefit was also reported by Hutchinson et al. [22]. The reason behind this is that by using models, where technical and implementation details are omitted, it is easier for non-experts to understand the system and give feedback.

Furthermore, MDE practices are seen to improve productivity [23, 24]. As models are intended to be maintained automatically, software development artifacts (for example, models and code) are re-generated using model transformation in any desired system. Therefore, unless the implementation of model transformations is complex and tedious, the runtime of the development process is minimized, which can improve productivity. Moreover, Hutchinson et al. [22] found that model driven engineering can reduce the time necessary to respond to the change of requirements and automate code generation by using transformation, both of which are key activities to increasing maintainability and productivity respectively. In addition, Mohagheghi and Dehlen [25] reported that portability can be considered as another beneficial aspect of applying MDE as a software engineering paradigm. With tool support and by writing targeted model transformation programs, models can be re-hosted to a new platform.

2.2 Mutation Testing

The previous section has provided some background on model driven engineering and its related concepts and principles, in order to substantiate why this software engineering paradigm makes such a good candidate for mutation testing. In particular, one of its founding principles, namely that everything is a model (even for an MDE language), enables modelers to analyze metamodels automatically, in order to generate mutation operators. This is because models must conform to predefined specifications of constraints and notations. Since the thesis broadly focuses on mutation testing in the context of MDE, this section covers those concepts and principles, some of which are generally related to software testing and some specifically related to mutation testing.

Accordingly, this section is structured as follows. Section 2.2.1 reviews testing and its related concepts and characteristics, including definitions of faults and failures in a software context, functional and structural testing, and test cases and oracles. Section 2.2.2 provides specific background information on mutation testing, as well as a comparison with other related testing techniques. Section 2.2.3 provides a survey on current mutation operator design approaches for programming languages, and mutation operators and their purposes. Section 2.2.4 provides a detailed overview of mutation operator injection approaches and the available tool support. Finally, Section 2.2.5 highlights drawbacks of mutation testing, a few of which this thesis tries to address.

2.2.1 Software testing: concepts and characteristics

Software testing is an essential process in the software development life cycle [10]. It involves executing software under test with a set of inputs, and then checking the outputs of the execution against expectations. The purpose of conducting software testing is to raise confidence about the quality of a particular software and assess whether it conforms to its intended specifications or not. Ideally, software testing should be involved in all stages of software development (for example, during the requirements and specifications phase, the design phase etc.), as each phase contributes to test design information. A typical software testing model widely used in the literature is the v-model (illustrated in Figure 2.9). The model shows software development phases on the left and the corresponding testing levels on the right. Since the purpose of this thesis is to apply mutation testing (which belongs to the implementation phase) in an MDE context, the lowest test level (i.e. unit test) is the primary focus.

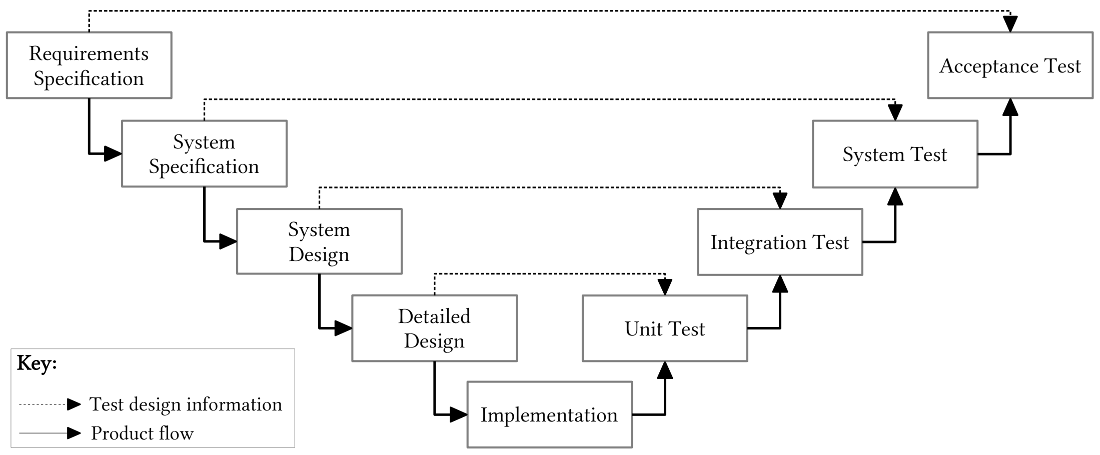

FIGURE 2.9: Different software development phases and testing levels, from Figure 1.2 in [12]

2.2.1.1 Faults and failures

The process of testing is undertaken by test developers using a set of designed input values. The execution results are then evaluated to reveal *faults* in the implementation of the program tested. A fault (a bug or a defect) in a program usually results from misusing the language to which the program conforms. If not resolved, a fault can lead to the incorrect behavior of a piece of software. Incorrect behavior, with respect with expected behavior (according to a set of program specifications), is known as software failure. There are three conditions for noticing (observing) a failure [12], which play a key role in the mutation testing technique, as will be discussed later on in the thesis. The failure conditions are:

- Reachability: an observable failure must be triggered by a reachable fault (or faults). In some cases, a fault can be inaccessible based on its location at the source code of a program. Examples of such cases include a mistaken implementation at an unreachable statement/expression, an uncalled function/operation or resource etc.
- Infection: an accessible fault must lead to an infected state in a program when the fault is executed.
- Propagation: an infected state in a program must propagate to produce an unexpected result at the end of the execution of the program.

2.2.1.2 Functional and structural testing

The core objective of testing is to identify a set of test cases that evaluate a given software. Traditionally, there are two approaches for test input identification: functional testing and structural testing [12]. The former treats the tested software as a black box (i.e. it is not necessary to know about its implementation) and identifies test cases based on its specifications. The testing process involves the evaluation of the software based on a comparison of the actual and expected outputs of input values. In contrast with the functional (black box) testing approach, structural testing (also known as white box testing) takes into account the implementation of the software, where test developers determine which parts of the code are actually tested. This determination is usually computed using test *coverage metrics* [26], which measure the reachability of each software item (part) in the implementation. This is useful for revealing faults in a program and increasing confidence that the implementation parts work as expected.

There are many coverage metrics (also referred to as coverage criteria) that are widely used to conduct structural testing: statement coverage metrics, which check that every statement in a source code is reached (at least once); condition coverage metrics, which ensure that every Boolean condition (two Boolean expressions interconnected by a single binary comparison operator such as $>$, $\geq$, $<$ etc.) is reached at least once; and decision (or branch) coverage metrics, which check that every possible decision (sequence of conditions interconnected by logical operators like AND, OR, XOR etc.) is reached.

There is no explicit preference between these two main approaches for use during test case determination. Each approach considers different inputs for constructing test cases. On the one hand, black box testing is good for constructing test cases using software requirements and specifications, because white box testing does not detect whether an implementation of a specification is missed. On the other hand, black box testing does not detect the implementation of an undesirable specification of the software (for example, a fault at a specific part of implementation), which is something picked up on by white box testing. Therefore, a mixture of test cases using both approaches is recommended for test developers, who wish to perform software testing not only at implementation level but at other testing levels as well (testing levels are illustrated in Figure 2.9).

2.2.1.3 Test cases and oracles

A test case, as defined by Ammann and Offutt [12], contains some elements that are essential for the complete execution and evaluation of a tested software. These elements are:

- Input values. These are necessary for completing an execution of the software. The test inputs can generally vary and depend on which functionality or parts of the software a test developer wishes to evaluate.
- Expected outputs, when a particular test input is executed.
- Prefix values. These inputs ready the software to receive the input values, for example, values to prepare the software under test to a state to be tested.
- Postfix values. These inputs are submitted to the software after the input values, for example, values to terminate or to inspect the result of test execution.

The mechanism of checking the correctness of a particular software behavior according to test cases is known as a test oracle [27]. A set of test cases is used to execute the tested software and the results of the execution are evaluated (or compared) with the expected outputs specified in the test cases. The evaluation declares whether a test case has passed or failed. A test case is successful when the expected output is equal (or identical) to the actual output of a given test input. Otherwise, it has failed.

2.2.2 Mutation testing and structural coverage metrics

Mutation testing (also known as mutation analysis) is a white box testing technique used not only for identifying test cases but also for evaluating them [12]. It involves the deliberate introduction of faults into the tested program (using *mutation operators*) in order to generate faulty versions (known as *mutants*) of the tested program, which are then used to evaluate the quality of a given test suite (a set of test cases). The evaluation is conducted by examining the ability of the used test cases to detect the faulty versions [12]. If a faulty version was successfully executed and not detected by any test case, then test developers are challenged to add more test cases to force its detection.

There are two hypotheses behind mutation testing [28]: the *competent programmer* and the *coupling effect*. The former accepts that a programmer is competent and, thus, more likely to produce a program that is correct or almost correct. Hence, an incorrect program can be corrected with a minor modification to its syntax. The latter states that test cases, which can distinguish between programs that are marginally different, are more sensitive to discerning programs with major differences. Both hypotheses imply that minor modifications in programs are sufficient to reveal complex faults.

Although structural testing and mutation testing both consider the implementation of software for identifying and evaluating test cases (functional testing is intentionally omitted, as it applies to a different testing level), there is a main difference between the two. Looking back at the conditions of observable failure in Section 2.2.1.1, structural testing requires that a sufficient part of the implementation is reached by an input test, regardless of whether that part is executed or not. In other words, structural testing does not require an observable failure (i.e. to reach a fault); hence, there is no need for test cases and test inputs alone are sufficient. On the contrary, an observable failure is required in mutation testing, whereby a reachable fault is executed and propagated to produce an incorrect output.

In fact, the literature discusses different forms of mutations, which require that a fault is reached and executed "locally", putting the tested program in infection state earlier during the execution of a mutant, not waiting until it propagates to produce an odd output. Such mutations are known as weak and firm mutations [12]. For the purposes and objectives of this thesis, only classical mutations are targeted because the propagation of faults to actual failures plays an important role in finding mutation operators that divert wrong outputs from expected outputs for a given software of an MDE language.

2.2.3 Mutation operators and mutants

A mutant is a version of a program that has been introduced with a syntactic change following the typical mutation process illustrated in Figure 2.10. A tested program (P) is used to generate mutants by applying a set of mutation operators – mutation rules that mimic errors programmers are likely to make when using a language. Each generated mutant (P') is executed against a set of test cases. If the mutant is detected by any test case then it is classified as killed, and as not killed if otherwise.

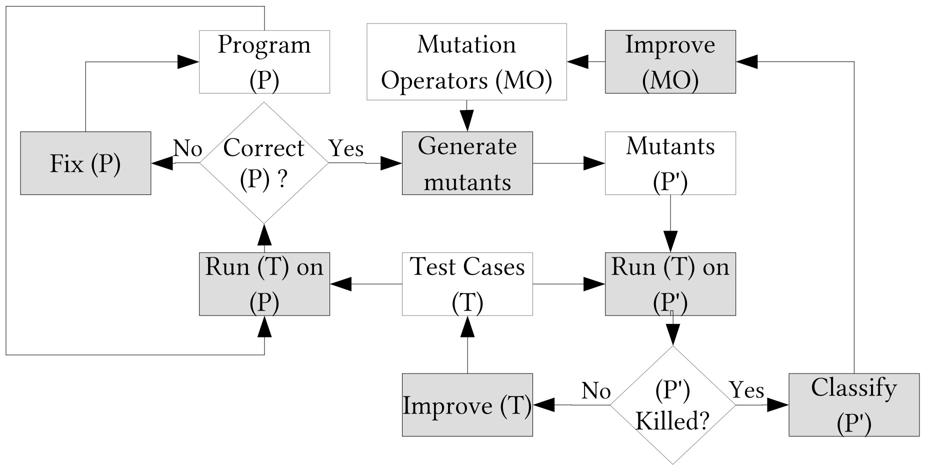

FIGURE 2.10: Typical mutation testing process adapted from [29]

Generally, mutants can be valid when they conform to the language specification (i.e. the grammar or metamodel) and invalid when they do not (the literature uses the term stillborn). Valid mutants can be:

- Killed mutants – As mentioned above, killed mutants are detected by at least one test case. Generally, killed mutants, especially if they are plausible and based on errors likely to be committed by programmers, are an indication that the used test set is a good set. Mutation operators that generate killed mutants are considered useful against low quality test sets.
- Trivial mutants – These are killed mutants, detectable (killable) by all test cases. Trivial mutants do not challenge test developers in adding or enhancing their test set and, hence, are sometimes overlooked.
- Live mutants – These are not-killed mutants, which produce the same output as the original program but that are killable if test developers enhance their test set by adding more test cases. Therefore, they are considered valuable from a test developer's perspective.
- Equivalent mutants – They are also not-killed mutants but, unlike live mutants, cannot be killed by any test case.

As the focus of this thesis is on mutation operator design for MDE languages, the following section provides a detailed presentation of the currently available designed mutations and their design approaches.

2.2.3.1 Mutation operators for programming languages

In mutation testing literature, a number of studies has developed mutation operators both for general purpose and domain specific programming languages. A general purpose language is a programming language that supports various application domains and provides a wide range of language concepts that are not specific to a particular domain (such as C, C# and Java). On the contrary, a domain specific language targets a particular application domain and provides a limited range of language concepts that are only suitable for a certain domain. The following languages have received mutation testing attention.

Fortran was a subject language for early mutation testing experiments back in 1977 by Budd et al. [30]. They proposed a set of 25 mutation operators for Fortran IV that were then implemented into the PIMS mutation tool [31]. Those early studies were followed by DeMillo et al. [32], who proposed a set of 22 mutation operators for Fortran 77, applied into the Mothra mutation tool [33]. In both studies ([30, 32]), the mutation design approaches were motivated by the programmers' experience of the type of faults expected from Fortran language users.

C has received a number of mutation definitions. Agrawal et al. in [34] proposed 77 mutation operators based on possible errors committed by programmers. These mutation operators were implemented into the Proteum mutation tool [35] and CSaw, which provides implementation of some of those operators [36]. Furthermore, there is a set of eight mutation operators inspired by possible faults of format String bugs, for example, faults that arise from misusing format string parameters of C functions that accept String formats like $printf$,$fprintf$ etc. The operators were implemented into the MUFORMAT mutation tool [37].

Ada has also been subject to mutation testing. Offutt et al. [38] proposed a set of 65 mutation operators drawing from their previous experience designing mutations for Fortran 77 [32, 33] and C [34]. They argued that they had designed a complete set of mutation operators for Ada that also considered its language specifications.

Java has been the focus of a number of research studies. Kim et al. [39] proposed an interesting approach for defining mutation operators for Java by using HAZOP analysis (Hazard and Operability Studies). This is a technique used to examine in a systematic way a particular process for critical and safety purposes. Their approach used a set of *guide words* and applied them by analyzing the deviation of the Java language constructs and its attributes found in the Java language definitions. Furthermore, Ma et al. in [40] proposed a set of class level mutation operators for Java that were later refined by Offutt et al. in [41]. These class level mutation operators are related to specific features of Object Oriented Java (such as inheritance and polymorphism). Following that, a design of a set of method level mutations was presented in [42] that includes operators related to arithmetic operator replacement, comparison operator replacement and many more. In addition, extra mutation operators (mainly deletion operators) for Java were designed in [43, 44]. All of the proposed mutation operators (in [41, 42, 43, 44]) were based on faults originating in the misuse of Java language concepts and were implemented into the MuJava tool. Finally, Ji et al. [45] introduced five mutation operators for exception handling of Java language features (mainly catching blocks) by modeling likely faults of Java programmers.

C# is a language that has received little attention. Derezińska et al. in [46] proposed a set of 40 mutation operators that were studied in [47, 48]. In their latest study, Derezińska and Rudnik [49] used 18 mutation operators out of those initial 40 with an extra eight traditional mutation operators (for example, arithmetic operator replacement, comparison operator replacement etc.) that were implemented in the CREAM mutation tool [47]. All of the mutation operators were designed based on an analysis of the C# language specification/grammar and mutation operators for Java, as defined in [39, 40].

C++ has also been subjected to mutation testing experiments. Delgado-Pérez et al. [50] designed 37 mutation operators for class level mutation (such as inheritance, polymorphism, access control of class members etc.) that have been extensively evaluated in works such as [51, 52, 53]. The designed operators were based on common faults of C++ programmers and were implemented in the MuCPP mutation tool. For the concurrency feature of C++, Kusano and Wang [54] defined a total of 29 mutation operators (with no explicit motivation or source for their operators) that were later implemented in the CCmutator mutation tool.

SQL has received a number of mutation operator definitions. Chan et al. [55] defined seven mutation operators based on the standard types of constraints in the EER (Enhanced Entity–Relationship) model. The purpose of their study was to test the constraints of EER and whether they were mapped properly in SQL. In addition, Tuya et al. [56] examined the syntax and semantics elements of the SQL language, and defined mutation operators accordingly that were later integrated and implemented in the SQLMutation mutation tool [57].

WS-BPEL (Web Services Business Process Execution Language), which is an executable language for specifying business process actions based on Web Services compositions [58], has also received mutation operator definitions. Estero-Botaro et al. in [59] designed a set of 26 mutation operators that were mainly derived from and modeled on the faults likely to occur from the misuse of the WS-BPEL language specifications. Such operators were reviewed and implemented in the GAmera mutation tool [60].

XSLT, which is a language for transforming an XML document into another format, for example HTML, has also received some mutation operator definitions. Lonetti and Marchetti [61] used a combination of approaches to design 23 mutation operators. One approach was to design operators by adapting typical mutation operators defined for general purpose programming languages (mainly Fortran and Java). Another approach was to design operators by examining XSLT specifications and then modeling faults that were likely to arise from the misuse of the language. All mutation operators were implemented in the X-MuT mutation tool.

A summary of all design mutation operators of the aforementioned programming languages is given in Table 2.1 that also shows the types of mutations (addition, deletion and replacement) and the number of mutations for each type.

2.2.3.2 Mutation operator design discussion

It is possible to conclude from the aforementioned definitions and mutation designs that there are two aspects involved in mutation operator design. One aspect is the kind of design approach that is followed and the other is the type of action or behavior of each intended mutation operator. Regarding the first aspect, there are three distinct design approaches. Mutation operators can be designed by (1) examining the definition/specification of a subject language, (2) examining the error taxonomy of users of the language, and (3) adapting from mutation operators of similar languages.

There is no obvious choice as to which mutation operator design is best, as each approach is appropriate in a particular situation. The approach that involves designing mutation operators from a language specification is preferable for immature, emerging languages. This consists in going through the concepts of the selected language and generating, if applicable, mutation operators for each concept by analyzing the type of

Table 2.1: Total mutation operators for programming languages

Language	Mutation Operator Definition	Number of Mutation Operators		
		Addition	Deletion	Replacement
Fortran	DeMillo et al. [32]	2	1	19
C	Agrawal et al [34]	-	-	77
	Shahriar [37]	1	2	5
Ada	Offutt et al. [38]	6	1	58
Java	Kim et al. [39]	6	4	36
	Ma et al. [40, 42]			
	Offutt et al. [41],			
	Deng et al. [43],			
	Delamaro et al. [44]	9	18	22
	Ji et al. [45]	1	1	3
C#	Derezińska et al. [46]	3	10	23
C++	Delgado-Pérez et al. [50]	6	11	20
	Kusano and Wang [54]	-	6	23
SQL	Chan et al. [55]	-	-	7
	Tuya et al. [56]	2	6	28
WS-BPEL	Estero-Botaro et al. [59]	-	12	14
XSLT	Lonetti and Marchetti [61]	1	3	19
	Total	37	75	354

errors that may occur when the concept is misused by programmers. The consequence of using this approach may be a large mutation operator set when the complete set of language concepts is considered, which is a reasonable outcome for new languages [12]. A large mutation operator set may be viewed as a limitation, since it leads to a large set of mutants that may prolong the mutation testing process. This can be resolved using mutation operator reduction techniques (mainly the selective mutation operators approach), as discussed in Sect. 2.2.5.

The second approach of mutation operator design involves examining the error taxonomy of the users (or programmers) of a language. Since operators are designed to mimic faults that are likely to be produced by programmers, the error taxonomy can be an ideal source of mutation operators. The error taxonomy of a language, however, can sometimes also be *incomplete* as it only covers those language concepts that are actually exercised or instantiated by programmers, a fact which may unintentionally lead to overlooking crucial concepts worthy of investigation.

The third and final approach for mutation design is adapting mutation operators from similar languages. For general purpose programming languages, this approach can be practical as the adapted mutation operators likely require minor adjustments to the target language. For instance, the class level mutation operators for C++ and C# defined in [50, 46] respectively were copied over, with minor modifications, from Java mutation operators defined in [40]. For domain specific programming languages, however, the approach of adapting from similar languages can be sometimes impractical. This is because concepts of a domain specific language are defined in a particular application domain, and mutation operators for those concepts are likely not adaptable to other application domains. Since this thesis targets MDE languages, which are mostly task specific languages, this mutation operator design approach is impractical for defining a complete set of mutation operators.

Regarding the type of action or behavior intended for a subject mutation operator, the syntactic changes introduced by mutation operators mainly involve the addition, deletion or replacement of information. Concerning deletion operators, the action of a simple deletion is followed by most mutation operator design approaches. Examples of deletion operators include the removal of the keyword *this* in object oriented classes (as in [39, 40, 50]), the removal of $else - if$ or *else* statement blocks (as in [59]), and the removal of order-by-expression (for example, ASC and DESC) from an SQL clause (as in [56]). A few mutation operators were found to involve complex deletions, although from a language's concrete syntax standpoint, complex can also be simple. Examples of such actions include deleting a block of statements or deleting a method/function.

Concerning addition operators, these are designed to insert *simple* changes to a piece of software. It is permissible for test developers to instantiate multiple language concepts in order to introduce a complex change into an implementation of a software by a single addition operator that will, however, break the competent programmer hypothesis. A simple syntactic change, instantiated from a single language concept, is less likely to undermine this hypothesis. For this reason, the addition operators of programming languages in the reviewed literature are simplified. Examples of such operators include inserting the *continue* or *break* keywords to looping structures (like *while* and for), inserting a unary operator ('-') to numerical expressions or literals, inserting the keyword *this* to variables of methods of object oriented classes, and many more. Although there are no examples of complex addition operators found in the literature for programming languages, a complex operator may involve a complete insertion of a Boolean condition with two operands interconnected with one binary operator, or a variable declaration with name and type.

Concerning replacement operators, the same principle discussed for addition operators was followed by most of the replacement mutation operators reviewed in the literature. The new values used in replacement operators are usually obtained from compatible instances of language concepts of the same categories of the target mutated concepts. For instance, the most frequently used mutation operator is the binary comparison operator replacement (found in [39, 42, 56, 59]). It involves a replacement of a binary comparison operator (such as $>$, $\geq$ etc.) with another binary comparison operator from the same category. Another example is a replacement of a variable super class with another super class in the multiple inheritance class in C++ (found in [50]), for example $A :: var1$ is mutated to $B :: var1$ where A and B have the same type-kind. The design of complex operators that require the instantiation of multiple language concepts was not found in the literature because complex changes are likely to be detected by any test case, consequently undermining the competent programmer hypothesis as with complex addition operators.

The total number of mutation operators reviewed in Sect. 2.2.3.1 and summarized in Table 2.1 amounts to 466 mutation operators out of which 37 are addition operators, 75 deletion operators and 354 replacement operators. It is clear that replacement operators are the most frequently used mutations, followed by deletion and addition operators.

2.2.4 Mutation operator injection

One of the essential elements of mutation testing is the application of mutation operators to introduce syntactic changes into a tested program. In the literature, there are two distinct ways in which mutation operators can be injected. In the first, the source code is used for injecting mutation operators, and in the second an intermediate representation of the source code is used for injection. The former situation uses the string of the source code for making changes, and produces versions of the string, each of which contains a single modification, without needing to check whether the original or modified versions conform (at least syntactically) to the language of the source code. Mutation tools that apply mutation operators directly onto the source code include CSaw [36], MuClipse[7] and GAmera [60].

The second method for injecting faults into a program, as mentioned above, involves the use of intermediate representation of the tested program. The intermediate representation can be obtained from parsing (syntactically analyzing) the source code of a program according to a set of syntactic rules defined in the grammar of the language to which the program conforms. Unlike the practice of injecting faults directly into the source code of the program, the target source code is checked explicitly to ascertain whether it is valid or not by the language compiler/interpreter or any other means of static analysis. Thus, the risk of generating invalid mutations from unverified strings is less in this situation compared with injecting faults directly into the source code. Examples of intermediate representations found in the literature include a parsed tree, as

[7] `http://muclipse.sourceforge.net`

is the case with the CREAM mutating tool in [49]), the abstract syntax tree (AST), as is the case with MuCpp in [51] and CCmutator in [54], and the byte code after parsing and compiling the source code of Java programs, as with the MuJava [42], PIT[8] and Jumble[9] mutation tools.

In an MDE context, where everything is a model [2], MDE programs can be expressed as models and as yet another intermediate representation for mutation operator injection. The expressed models must conform to the models of languages (that is, metamodels) to which the subject programs conform. These expressed models can be obtained directly from the source code using text2model transformation (automation tools such as Xtext[10]) or another intermediate representation like Concrete Syntax Tree (CST) (used in the ATL transformation engine), AST (used in Epsilon transformation languages) or any other representation using model2model transformation.

2.2.5 Challenges to mutation testing

Mutation testing suffers from two main challenges that make its adaptation unattractive to some test developers [62]. These are the high computational cost and equivalent mutant management. The former emerges when a large set of mutation operators is applied, which consequently produces a large number of mutants that may require a long execution runtime against a test set. The latter arises when intensive human effort is required to distinguish equivalent mutants from not-killed mutants, which is time consuming. In order to tackle these two challenges, a number of techniques has been proposed, briefly discussed below.

2.2.5.1 Computational cost

In order to manage the problem of computational cost, test developers typically use mutation sampling [63, 64] and/or mutation selection reduction [65].

1. Mutation sampling: a small subset of mutants is selected randomly from an entire set of mutants to be used for mutation testing. The rate in which mutants are sampled has been a subject of extensive research in [32, 33, 66]. The process of random sampling, however, may require more intelligent approaches than currently available in order to select those mutants that are important for the quality of the mutation testing.

[8]http://pitest.org

[9]http://jumble.sourceforge.net/index.html

[10]Xtext is a language workbench and framework for developing domain specific languages https://www.eclipse.org/Xtext/

2. Mutation selection: a minimum set of mutation operators is selected without significantly compromising the effectiveness of the test. Such a reduction technique has been widely used in [67, 68, 69, 70, 43], rendering it a more popular technique than mutation sampling.

2.2.5.2 Equivalent mutant management

Managing useless equivalent mutants is another mutation testing challenge. There are two main actions related to equivalent mutants: detection and prevention. As mentioned earlier in Sect. 2.2.3, a not-killed mutant produces the same outputs as the original program. It can refer to a live mutant when it is detected by stronger additional input tests or to an equivalent when no input test can detect it. Since equivalent mutants are syntactically different from the original program, human effort is usually required to distinguish them [71], which is considered a limitation of mutation testing. Several solutions have been proposed in the relevant literature in order to overcome this drawback.

One solution to detect equivalent mutants is to use compiler optimization techniques aimed at deriving heuristics by examining intermediate representations (e.g. flow graphs) of the mutants themselves (as proposed by Baldwin and Sayward in [72]. Such a technique was implemented in the Mothra mutating tool. Another detection technique is through execution profiles (such as execution time or memory usage), which detects semantically different equivalent mutants (as applied by Ellims et al. in [36]). Yet another technique is to use test coverage metrics as in [73].

In order to prevent equivalent mutations from being generated, a set of predefined rules or conditions for mutation operators are used. For instance, DeMilli and Offutt [74] proposed a technique in which mutation operators are associated with necessary input value constraints, which are computed from data flow analysis with algebraic transformations, which prohibits their mutation testing tool (called Godzilla) from encountering equivalent mutants. Prevention of equivalent mutants has a potential positive impact on the overall mutation testing process, as it produces a low number of mutants that need to be executed, hence reducing the overall processing time.

2.3 Testing and Mutation Testing in MDE

The previous section covered those concepts and principles related to testing and mutation testing. It has also presented a literature review on mutation operator design approaches and injections. Of the design approaches discussed in Section 2.2.3.2, examining the definition of a language and adapting mutations from similar languages, where applicable, seem the most appropriate for emerging domain specific languages, like MDE languages. In terms of the most appropriate mutation operator injection method in an MDE context, where everything can be expressed as a model, errors can be injected to program models of MDE languages obtained directly from the source code of an MDE program using text2model transformation or from other types of model like CST or AST using model2model transformation. Solutions to the problems of high computational costs and equivalent mutant management have already been addressed above.

This section extends the discussion on the principles of testing in the context of traditional engineering paradigms (see Section 2.2) by addressing the challenges of mutation testing in the MDE paradigm. First, it provides a review of relevant studies in the field of testing in model driven engineering in Section 2.3.1, focusing specifically on mutation testing in MDE in Section 2.3.2.

2.3.1 Testing in MDE

Verification is an essential practice in software development [10]. It raises confidence in software quality with respect to its specifications. Since this thesis is also concerned with the verification of MDE programs, this section provides an overview of testing attempts in model driven engineering, as well as some of the related challenges.

2.3.1.1 Test input models

As discussed in Sect. 2.2.1.3, one of the essential elements related to test case definition is test inputs. Since models are the primary artifacts in model driven engineering, test inputs are, therefore, models conforming to metamodels. Manually creating test models consumes valuable resources and time from the point of view of test developers. Therefore, a number of approaches has been proposed to tackle the challenge of generating test models with some level of input metamodel coverage in an automated manner. One approach of metamodel coverage is the generation of all sets of test models for the entire metamodel. Another approach is to define the metamodel coverage by considering the tested MDE program that will use the generated models (for example, those parts of the input models that are actually referred to by the MDE program).

Wang et al. [75] proposed the concept of an effective metamodel, which is constructed from fragments of the input metamodel actually acted upon by a model transformation.

Their metamodel was constructed from certain definitions of the core MOF constructs, for example, class, feature, inheritance and association. By using coverage criteria defined by Andrews et al. [76], they generated test data from effective sources of metamodels that bound the input scope of model transformation. In addition, they presented a framework (built for EMF) for generating test inputs for model transformation written in the Tetkat transformation language.

A similar approach of using coverage criteria was proposed by Fleurey et al. [77], who also included partition analysis of the data in [78] in order to construct test models. More specifically, they defined a set of adequacy criteria based on the partitioning of an input metamodel and its properties, in which a set of model fragments that has to be covered by test models is defined by each criterion. They also developed a prototype that computes a set of model fragments for a given metamodel that they named Meta Model Coverage Checker (MMCC). Sen et al. [79] extended a part of this work [77] by proposing a set of strategies for model generation along with a supported tool (named Cartier) that synthesizes the Alloy model, which contains a set of Alloy constraints imposed by each strategy.

Cuerra and Soeken in [80] used a contract-based specification language (PaMoMo), proposed in [81], to generate test inputs and oracles for model transformation using transformation requirements. The language allows users to define model transformation at the metamodel level in terms of pre-conditions (requirements to which input models must conform), post-condition (requirement to which output models must conform), invariants (requirements to which expected models must satisfy). In [80], the authors used the precondition and invariant specifications and select a level of metamodel coverage from the specifications to generate automatically, using SAT-solver, test input models. The test inputs are then used to build new assertions for model transformation taking into account new invariants and postconditions built accordingly for each test input.

Although the aforementioned proposals attempt to address model generation in an automated manner, fully automated techniques are not always needed and test developers may value some involvement in the model generation process via parameters or configurations. A partially automated approach is proposed by Rose and Poulding [82], who used a search-based algorithm (that accepts few parameter settings) in order to derive an optimal input profile. The profile represents a context-free grammar produced by a set of rules (constraints) obtained from the input metamodel, and weights for the rules that specify a probability distribution over the language defined by the grammar. The optimal resulting profile, however, needs to be validated manually before it is used and sampled to produce test models. Furthermore, user involvement in the process is limited to setting the search-based algorithm parameters, while involvement in selecting the weights of probability distribution can be a preferable feature.

Another partially automated approach is facilitated by Epsilon Model Generation, initially proposed by Popoola et al. [83] and integrated later into the Epsilon Platform. The generator allows users to provide weights that specify a profitability distribution for selected modeling constructs (or modeling entities) that can be instantiated. It also allows users to assign values (optionally random) for properties (modeling attributes and associations) defined in each entity in the input metamodel. Furthermore, the generator is very flexible in that it allows users to include random data from other types of input models, such as CSV files, EMF models etc., to be selected during the model generation process.

2.3.1.2 Test expected models

Finding or obtaining expected output models of MDE programs has been a challenge for test developers. According to Mottu et al. [84], test oracles can be obtained from a number of methods. One of these methods is finding a reference model from a previous correct (and accepted) output of an MDE program. This method is not always true for new MDE program implementation. Another method is to obtain expected models from an inverse model that results from transforming (or managing with other activities) the test output model of a transformation with an inverse transformation in order to produce the same input model. This particular method may be considered ineffective in some cases due to the fact that some model transformation (or management) languages are not bi-directional. In other words, a model program may include the addition or omission of information that may (or may not) be used for the inverse action.

Another method for defining oracles, as proposed by Mottu et al. [84], is to define them from scratch. Defining expected models for complex domains, however, is usually considered a difficult task [85]. Models can sometimes be complex in nature and defining them requires listing all expected properties by considering numerous concepts and relationships between different entities in metamodels. In order to overcome this problem, Finot et al. [86] suggested that a test developer may be able to predict part of the output models of concepts that are more likely to be produced by the model transformation. They proposed an approach that compares the actual output of model transformation with a partially predicted model, and any differences can be filtered using unpredictable parts of the model. One limitation of their approach is that the manual definition of a part of a complex model (whether it involves predicted or unpredictable parts) can still be a difficult task for large software systems.

Contracts are also another method found in the literature to define oracles for model transformation testing. They can be constraints and/or a set of invariants expressed in OCL or any other validation language. For the purpose of test oracles, Gogolla et al. in [87] proposed a contract-based approach for model-to-model transformation testing. Th ey specified properties for model transformation based on source models, target models and the relationships between them using OCL validation constraints. Contracts are demonstrated by generating automatically input models and executing them against ATL programs. The output models are then checked against the set of contracts defined for the model transformation using USE (UML Specification Environment).

Guerra et al. [88] used the PaMoMo in [81] for contract-based specification of transformation requirements that allows defining specifications at the metamodel level. More specifically, it allows the definition of specifications as pre-conditions, post-conditions and a set of invariants for model transformation. Those specifications are then translated to QVTr and executed before the transformation in order to (1) validate the source models against pre-conditions, (2) validate relationships between the source and target models, and (3) validate target models with respect to post-conditions.

Transformation trace is another approach to defining oracle functions discussed in the literature. In this particular method, which mainly relates to model transformation, the definition of expected models can be replaced by obtaining valid transformation trace links between input and output models, as proposed by Kessentini et al. [89]. They identified, however, a limitation of the approach as it relies on the availability and correctness of the transformation examples. This work is based on a framework proposed by Matragkas et al. [90] that captures the transformation data of a transformation engine in terms of a trace model that conforms to a traceability metamodel defined using the Traceability Metamodeling Language (TML) [91]. This trace model can be used to generate trace links, which are formed from mappings between elements of the input models, their corresponding elements in the output models, and the transformation rule that creates those elements in the output models. It was argued that any transformation errors can be identified from those generated traces that do not conform to the defined traceability metamodel or that violate the constraints imposed by it.

2.3.1.3 Model comparison

Model comparison is a task that aims to find similarities and/or differences between models. Generally speaking, a comparison process consists of two main phases. The first finds matches between model elements by studying their identity. A match is marked when model elements have *equal* identities. The equality of identities can be found by applying matching strategies. Kolovos et al. [92] identify four matching strategies that can be used in this context. These are:

- Static identity-based matching: this strategy relies on unique identifiers assigned (by the underlying modeling language) to each model element in models. For oracle purposes, where elements of actual output artifacts receive new IDs at each execution of an MDE program, this matching strategy would give every time unmatched elements of the actual model with elements of the expected model. Hence, this strategy must be avoided for testing purposes.
- Signature-based matching: this strategy relies on a combination of feature values of model elements to conduct a comparison. Unlike identity-based matching, this strategy can be used for oracle purposes.
- Similarity-based matching: this strategy finds similarities between model elements based on name equivalences or property equivalences. This strategy can also be used for oracle purposes.
- Language specific matching: this strategy relies on matching rules to define matching model elements.

The second phase of the comparison is finding differences between two model element pairs (that correspond to each other or are found to match in the matching phase). In practice, there are four sets of actions or operations for each difference: added, deleted, moved and updated. For example, if a model element (along with its property values) is organized to a different part in the corresponding model, the moved keyword is noted in the corresponding model assuming that this is the revised version.

In the related literature, a number of comparison frameworks has been developed that provide comparison mechanisms. Kolovos [5] created a dedicated comparison rule-based language (Epsilon Comparison Language (ECL)) that allows users to define comparison rules to identify pairs of matching elements between a source model and a target model. Küster and Abd-El-Razik [93] used a similar approach to compare model elements using constraints.

For EMF models (discussed in Sect. 2.1.3.1), there are tools that facilitate model comparison using the aforementioned model matching strategies and differences. One such tool, developed by SiDiff [94], involves a metamodel agnostic approach for model comparison. It supports EMF but it is also a configurable framework that can be adapted to any modeling language and provide support to other model representations including those in graph-like structures. Another tool is EMFCompare[11] that enhances the different matching strategies described above with additional configurations for difference detection between models.

2.3.2 Mutation Testing in MDE

The technique of mutation testing has been used in a model driven engineering context. This section presents, with analysis and discussion, the current approaches to mutation operator design, as well as attempts at performing mutation testing for MDE languages (whether metamodels or task specific and model management languages).

To begin with, Semeráth et al. [95] designed a set of mutation operators to evaluate automatically synthesized models of DSLs. produced using modelling tools that used for critical complex systems engineering. The mutation operators were specific and designed by examining target DSLs; and targeted well-formedness constraints and design rules of two industrial case studies. However, this thesis targets MDE task-specific and model management languages which is a different scope and contribution.

For task specific MDE languages, Mottu et al. [96, 97] proposed a set of generic mutation operators that are applicable to model transformation. Such operators were designed by analyzing core activities that are likely undertaken during model transformation: navigation of models via relations between classes defined in input and output metamodels, filtration of a collection of objects, and creation of output models. Based on these core activities, they proposed a set of ten mutation operators (as given in Table 2.2). Such operators were widely used as a fundamental set of operators in [79, 98, 99]. Although most operators include mutations for navigation and filtering, which are core activities in most MDE languages, these operators require to be constructed according to the target MDE language in order to derive meaningful mutation operators, by considering language concepts to which those operators can be applicable. Also, some of these operators may require execution – especially navigation and creation operators – in order to determine some parameters needed for their adaptation to a given MDE language (although such requirement is not necessary in model transformation).

The ATL transformation language has been the focus of a number of research studies. Khan and Hassine [100] proposed a set of ten mutation operators for the ATL transformation language that were derived only from examining possible errors that are likely to emerge from programmers' misuse of the language. This is an approach that may lead to the overlooking of mutation operators for key language concepts when it is used as the only source for mutation operator design. Another drawback of their attempt is

[11]`https://www.eclipse.org/emf/compare/`

Table 2.2: Mottu et al. [96, 97] generic mutation operators

	Mutation Operator	Description
Navigation	Relation to same class change	Replaces the navigation through one association to the same class
	Relation to another class change	Replaces navigation through an association to another class
	Relation sequence deletion	Removes the last step of a navigation sequence
	Relation sequence addition	Adds a navigation step at the end of a navigation sequence
Filter	Collection filtering change with perturbation	Replaces an existing filter of a collection with another filter
	Collection filtering change with deletion	Removes the creation of a relation between two objects
	Collection filtering change with addition	Adds a filter against a collection
Creation	Class compatible creation replacement	Replaces an object creation by the creation of another object of a compatible type
	Class association creation deletion	Removes the creation of a relation between two objects
	Class association creation addition	Adds the creation of a relation between two objects

that the designed operators only work for ATL and it is probably not easy for them to be copied over to other MDE languages. This is due to the fact that most, if not all, MDE languages are domain specific, with each language targeting a specific application domain. Hence, it is likely hard (or sometimes impossible) to adapt operators tailored to a particular language concept to another domain. Also, the lack of a mechanism through which to apply the operators against ATL programs is another limitation of their work. The authors' operators are given in Table 2.3.

Another work applying mutation testing is that of Troya et al. [101], who proposed an interesting systematic approach for generating mutation operators for the ATL transformation language by examining its metamodel and applying three mutation operators, namely addition, deletion and modification, to several specific concepts defined in the language. Furthermore, they also proposed a technique for applying their operators using again the ATL language as a *High Order Transformation* to inject faults into tested ATL programs through model transformation. There are a couple of limitations, however, that need to be addressed. The first is that the defined mutation operators are specific to ATL and are likely not adaptable to other MDE languages (as is also the case with [100]). A second limitation of their work is the lack of safeguards that the produced mutants from the generated operators comply with the ATL metamodel and its constraints. This means that constraints (at least syntactical constraints) imposed

Table 2.3: Khan and Hassine [100] ATL mutation operators

Mutation Operator	Description
Matched to Lazy (M2L)	Changes a matched rule (declarative style) to a lazy matched rule (imperative style)
Lazy to Matched (L2M)	Changes a lazy matched rule to a declarative matched rule
Delete Attribute Mapping (DAM)	Removes a binding element of an out-pattern of a rule
Add Attribute Mapping (AAM)	Adds a binding element to an out-pattern element of a rule
Delete Filtering Expression (DFE)	Removes a Boolean filtering expression of an in-pattern element of a rule
Add Filtering Expression (AFE)	Adds a Boolean filtering expression to an in-pattern element of a rule
Change Source Type (CST)	Changes the source type of an in-pattern element of a rule to another
Change Target Type (CST)	Changes the target type of an out-pattern element of a rule to another
Change Execution Mode (CEM)	Changes the engine execution from mapping to in-place and vice versa
Delete Return Statement (DRS)	Removes a return statement of an action rule

by ATL metamodels are not checked explicitly when mutation operators are applied to produce mutants.

A set of twenty-seven mutation operators (as listed in Table 2.5) has been compiled by Cuadrado et al. in [102] that is more complete than what Troya et al. [101] and Khan and Hassine [100] suggested. This is mainly derived from an examination of the ATL metamodel. Some of the operators overlap with those proposed by [101] and [100]. The purpose of this set of twenty-seven operators was to test an ATL static analyzer (AnATLyzer) that the authors developed for finding errors in ATL programs. The analyzer examined all transformation programs found in the ATL examples repository (also known as ATL Zoo) on `https://www.eclipse.org/atl/atlTransformations/` and reported five common typing errors.

Recently, Guerra et al. [103] proposed a set of seven mutation operators (listed below) derived from common errors of ATL programs (found in [102]). They used them, along with other mutation operators found in [102, 96, 97, 101], to perform an empirical mutation analysis over six ATL programs in order to assess the quality of the used operators. This was achieved by synthesizing test models, which were generated using a random generator with metamodel coverage. The synthesis was conducted by considering the transformation program using the AnATLyzer for finding the path(s) of flow graphs to the modified location in the mutant code and then synthesizing the test models accordingly. Their design approach and mutation operators, however, are specific to ATL.

Table 2.4: Troya et al. [101] ATL mutation operators

	Mutation Operator	Description
Matched Rule	Addition	Adds a new matched rule
	Deletion	Removes a matched rule
	Name Change	Replaces the name of a matching rule with a new name
In-Pattern Element	Addition	Adds a new in-pattern element to a transformation rule
	Deletion	Removes an in-pattern element
	Class Change	Replaces the type of a variable with another type
	Name Change	Replaces the name of a variable with another name
Out-Pattern Element	Addition	Adds a new out-pattern element to a transformation rule
	Deletion	Removes an out-pattern element
	Class Change	Replaces the variable type with another type
	Name Change	Replaces the name of a variable with another name
Filter	Addition	Adds a new Boolean filter to an in-pattern element of a rule
	Deletion	Removes a Boolean filter from an in-pattern element
	Condition Change	Replaces a Boolean filter of an in-pattern element
Binding	Addition	Adds a new binding element to an out-pattern element
	Deletion	Removes a binding element from an out-pattern element
	Value Change	Replaces the value of a binding statement
	Feature Change	Replaces the feature to which a value of a binding statement is assigned

Instead, this thesis focuses on developing an approach for language-agnostic mutation design. The Guerra et al. [103] mutation operators are as follows:

- Remove binding element of compulsory feature (RBCF)
- Replace helper call parameter (RHCP)
- Remove enclosing conditional (REC)
- Navigation after optional feature (ANAOF)
- Replace feature access by subtype feature (RSF)
- Restrict rule filter (RRF)
- Delete rule (DR)

To the best of the present author's knowledge, the only work that explores a metamodel agnostic mutation operator design approach is by Gómez-Abajo et al. [104], who provide a domain specific language, Wodel, for mutating models of any given metamodel. Users of the language need to invoke one or more different mutation operators that are agnostic, in order to perform one or multiple mutations in a single model. Although the language also provides functionality for the user to include (optionally) OCL constraints that are validated against the generated mutations (models), their operators do not check explicitly the syntactic constraints imposed by the abstract syntax of the target language

Table 2.5: Cuadrado et al. [102] ATL mutation operators

	Mutation Operator	Description
Creation	Binding	Adds a new binding element to an out-pattern element of a rule
	Source Pattern	Adds a new in-pattern element to a rule
	Target Pattern	Adds a new out-pattern element to a rule
	Rule Inheritance	Adds an inheritance relation for a matching rule
Deletion	Module Element	Removes a module element (whether rule or helper)
	Binding	Removes a binding element of an out-pattern element
	Source Pattern	Removes an input-pattern element of a rule
	Target Pattern	Removes an out-pattern element of a rule
	Rule Filter	Removes a Boolean filter of an in-pattern element
	Rule Inheritance	Removes an inheritance relation between rules
	Operation Context	Removes a context (type) of an operation
	Formal Parameter	Removes formal parameters in operations or called rules
	Argument	Removes arguments in operation invocation
	Parameter	Removes a parameter in operation/called rules (in definition blocks)
	Variable Definition	Removes variable declarations in action blocks or called rules
Type Change	Source/Target Pattern	Replaces types of variables associated with in-pattern and out-pattern elements with other types
	Helper Type	Replaces the context of a helper (type) with another type
	Return Type	Replaces the return type of a helper function with another
	Type of Variable/Collection	Replaces a type of variable declaration (whether ATL collection, model or primitive type) with another type
	Parameter Type Operation	Replaces the simple parameter type in operation and called rules with another type
	Parameter Type	
Name Change	Navigation	Replaces the name of a navigation feature
	Binding Target	Replaces the name of a feature of an out-pattern binding element
	Feature Operation	Replaces the name of an operation of a feature (e.g. $x.optimize()$)
	Predefined Operator/Operation	Replaces the operator name (e.g. $+$, $>$, *and*) or operation name (e.g. $first$, ...)
	Collection Operation	Replaces the name of a predefined operation of collection (e.g. *includes*, ...)
	Iterate Method	Replaces the name of a predefined iterate method with another (e.g. *select*, ...)
	Helper Invocation	Replaces the name of an operation and attribute helpers invocation

(for example, the cardinality of the relationships between entities that constitute the metamodel or type compatibility).

2.4 Chapter Summary

This chapter has provided a literature review of core disciplines related to this thesis and its objectives, namely model driven engineering and mutation testing. It has covered the essential general concepts and principles related to MDE and mutation testing that are important for this research, the objective of which is to provide an approach for defining mutation operators for rapidly emerging MDE languages. Models of MDE languages are mostly (if not entirely) built on a common metamodeling language, MOF (and Ecore as its practical implementation), which provides an opportunity to define operators for different metamodels in a simple and efficient manner. In short, an approach could be devised to find mutation operators for a given MDE language model that can also be repeated or re-used to find operators for other metamodels. The conformance of MDE models is achieved when they meet predefined specifications of notations and constraints of MOF on a higher level in the metamodeling stack.

In regards to mutation testing, this chapter has discussed its most relevant concepts and principles, including types of mutants and the mutation operator process. It has also presented a survey on the currently defined mutation operators and their design approaches in traditional engineering and MDE paradigms, as well as mutation operator injection methods. The survey has revealed three methods of mutation operator design of which two methods are more appropriate to be used in MDE context, namely, using a language specification (in model driven this can be a metamodel) and adapting operators from similar languages. In addition, the chapter has presented current limitations of using mutation testing mainly computational cost, which can be reduced by eliminating invalid and equivalent mutants from the mutation testing process.

Finally, this chapter has reviewed in detail approaches to mutation operator design and mutation testing in model driven engineering. The limitations of this body of work fall into two categories. The first is that, with the exception of [104] and [97], most studies focus on the ATL transformation language, whereas this thesis is interested in an approach that applies to different MDE languages. The second is that all designed mutation operators do not implicitly check the conformance of operators to metamodels or MDE language models before application (querying, for example, whether operators are about to generate invalid or equivalent mutants).

The following chapter will discuss an approach that will address some of the limitations of current approaches to mutation operator design and injection techniques that have been discussed in this chapter, and will be dedicated to model driven engineering languages.

Chapter 3

Analysis and Hypothesis

The previous chapter presented a literature review of the definitions, concepts and principles related to model driven engineering and mutation testing in order to contextualize the objectives of this thesis. This chapter identifies the limitations of current mutation operator design in the context of MDE languages. It also identifies the essential elements of MDE that are useful in proposing a solution that would resolve them.

The chapter is structured as follows. Section 3.1 analyzes the problems and limitations of current approaches to designing mutation operators for MDE languages and, then, identifies key elements in model driven engineering that are worth exploiting to resolve such limitations. Section 3.2 states the research hypothesis and Section 3.3 lists the research objectives that must be fulfilled in order to verify the hypothesis.

3.1 Problem Analysis

Section 2.3 presents a number of current approaches to mutation design in the context of MDE and their limitations. In terms of the latter, there are two in particular that are very important: (1) the difficulty of designing mutation operators for the rapidly emerging MDE languages that comply with their respective rules and definitions; (2) the generation of operators that produce useless mutants (invalid or equivalent), which usually impact negatively on the overall mutation analysis process.

Regarding the first limitation, some mutation operators exist for the ATL transformation language but there is lack of an agnostic approach (with the exception of [104]) for new and rapidly emerging MDE languages. In fact, this is rarely mentioned (at least explicitly) in current approaches to design operators. In order to address this limitation, this thesis uses concepts of modeling and metamodeling to design a set of language-agnostic mutation operators that comply with different MDE languages. As discussed in Section 2.1.1, MDE languages are currently built according to the typical 3-level metamodeling hierarchy (as illustrated in Figure 3.1) and depend on MOF (and Ecore as a practical implementation of MOF) for defining their abstract syntax. Thus, there are substantial opportunities for re-use across languages: an analysis technique that applies to a target meta-metamodel (in level M3) can, in principle, be applied to models of MDE languages (in level M2).

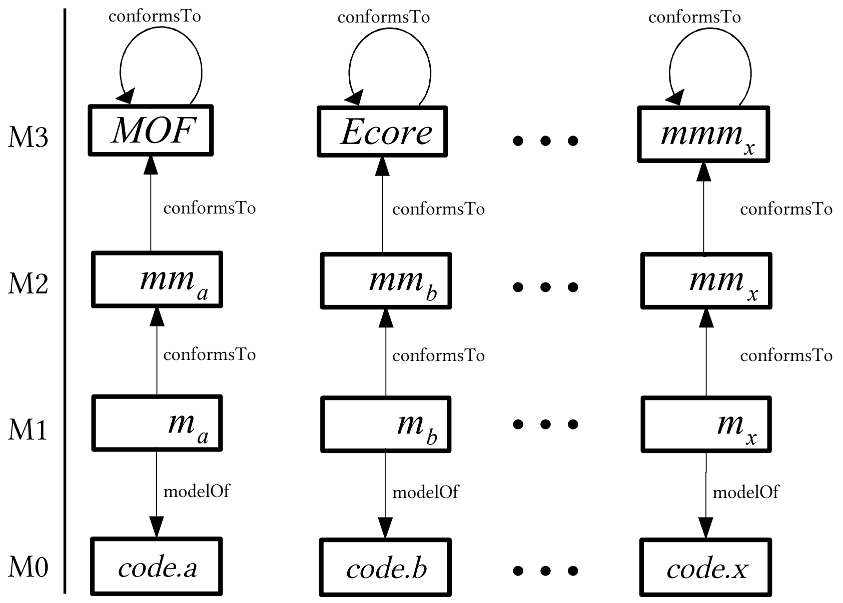

FIGURE 3.1: Typical MDE metamodeling level and modeling levels

Programs of MDE languages (as discussed in Section 2.3) can be expressed as models that conform to metamodels, which are models of MDE languages. The models of MDE programs (at M1 level in Figure 3.1) can be mutated using mutation operators derived from metamodels defined in M2. Metamodels in model driven engineering conform to another type of models know as meta-metamodels (models in level M3) in the common metamodeling architecture. M3 models are usually defined using concepts such as property and relation [2] to define a meaningful level of abstraction that can be used to express models at a lower level (i.e. M2 models). Hence, there is an opportunity to use M3 models and their modeling concepts to derive mutation operators that have a level of *abstraction* and are applicable to different metamodels that conform to M3 models.

In addition, since M3 models usually consist of elements and constraints that metamodels must conform to, the “abstract” mutation operators that could be derived from M3 models must comply to the aforementioned elements and rules. This means that the elements and rules defined in meta-metamodels should form a fundamental set of requirements that must be complied with by mutation operators.

Regarding the second limitation, solutions to the challenge of computational cost in mutation testing have been proposed for traditional programming languages, as discussed in Section 2.2.5. In particular, there are two kinds of mutants that usually contribute to computational cost and which have received limited attention: invalid and equivalent mutants. No solutions, however, have been proposed for model driven engineering languages. In addressing this limitation, this thesis uses concepts and principles of modeling and metamodeling to produce low rates of invalid and equivalent mutants in order to achieve overall low computational cost during a mutation testing process.

3.2 Research Hypothesis

The proposed solution of a flexible and adaptable set of “abstract” mutation operators for MDE languages can be used to investigate the following hypothesis:

Adaptable *sets of mutation operators for metamodels that conform to a common meta-metamodel can be derived from examining the common meta-metamodel. Any generated mutation operators will be metamodel-agnostic, produce* ***useful*** *mutants and low proportions of* ***useless*** *mutants.*

The emboldened terms in the hypothesis above are explained as follows:

Adaptable operators are defined as operators that can be re-used across multiple MDE languages, that when applied, produce actual mutants.

Useful mutants are valid mutants that conform to their metamodel and mutants that test developers can use to assess their test suite. Example of such are live and non-trivially killed mutants.

Useless mutants do not help test developers in assessing their test suite and can be valid like equivalent mutants or invalid that do not conform to the MDE language metamodel or to MDE language engine (invalid at (compile/parse)-time).

3.3 Research Objectives

In order to evaluate the above research hypothesis, this thesis has defined three main objectives derived from the hypothesis itself:

1. To propose and design a set of metamodel-agnostic (abstract) mutation operators in order to generate a new set of mutation operators that comply to a new MDE language. Any metamodel-agnostic mutation operators must be adaptable and applicable to any metamodel, including any mechanism for ensuring the compliance of mutation operators to that new metamodel and its constraints.
2. The proposed mutation operators should be instrumental in the production of:
   (a) useful mutants for test developers. This thesis defines useful mutants as live or non-trivially killed mutants. Live mutants always lead to good mutation testing, as they challenge test developers to add more test inputs or even enhance existing ones in order to detect such mutants. Non-trivially killed mutants potentially lead to live mutants if they are executed against low quality test inputs.
   (b) low rates of invalid and equivalent mutants. Invalid and equivalent mutants are considered by this thesis as useless mutants, as they (i) usually do not challenge test developers to add more tests inputs, and (ii) prolong the overall execution of mutation analysis.
3. The usefulness of the proposed mutation operators according to objective 2a and objective 2b should be evaluated using an empirical mutation analysis with representative programs and data.

Chapter 4

A Mutation Operator Design Approach

Chapter 2 has reviewed some of the limitations related to a mutation operator design in the context of MDE. In particular, it has provided an analysis of the limitations of current mutation operator design approaches in terms of adaptability to new and rapidly emerging MDE languages, especially in the case where there is a need for a mechanism to verify the compliance of mutation operators to new MDE languages. The previous chapter has suggested using models and metamodel to design mutation operators to be able to perform mutation analysis, while using the typical metamodeling hierarchy in model driven engineering to overcome the limitation of adaptability for MDE languages. Accordingly, the thesis has presented a hypothesis that considers those suggestions, along with a few research objectives to validate the research hypothesis.

This chapter proposes for the first time in Section 4.1 a generic meta-metamodel and a mutation design approach that contribute to the design of a set of *abstract* mutation operators that fulfills the objectives put forward by this thesis, taking into account some of the key elements of MDE explained in Section 2.1 and of mutation testing as presented in Section 2.2. Next, Section 4.2 gives a detailed description of the designed abstract operators. Following that, in Section 4.3, an adaptation of the approach and abstract operators to the Ecore metamodeling framework is presented, including an explanation of how elements of the approach are addressed. Finally, examples of using the abstract operators are given in Section 4.4 with an explanation on how they are instantiated to produce *concrete* mutation operators for use in a simple metamodel example.

4.1 A Design Approach using a Generic Meta-metamodel

Drawing on the typical metamodeling hierarchy in Figure 3.1 that was introduced in Section 3.1, in order to fulfill the research hypothesis and objectives 1 and 2b (introduced in Section 3.3), this thesis defines the concept of *generic meta-metamodel*. The concept (as illustrated in Figure 4.1) is a structure that captures the commonality of a few key modeling concepts existing in different MDE meta-metamodels. As stated above, meta-metamodels usually exhibit a high level of abstraction using concepts, such as properties and relations. Hence, the generic meta-metamodel refers to a collection of entities with properties and relations between entities. The concept also defines multiple indications for properties and relations of entities. Defining such a structure helps to find common ground among metamodeling technologies, which consequently facilitates the design of metamodel-agnostic mutation operators that can be flexible and adaptable to various MDE modeling technologies.

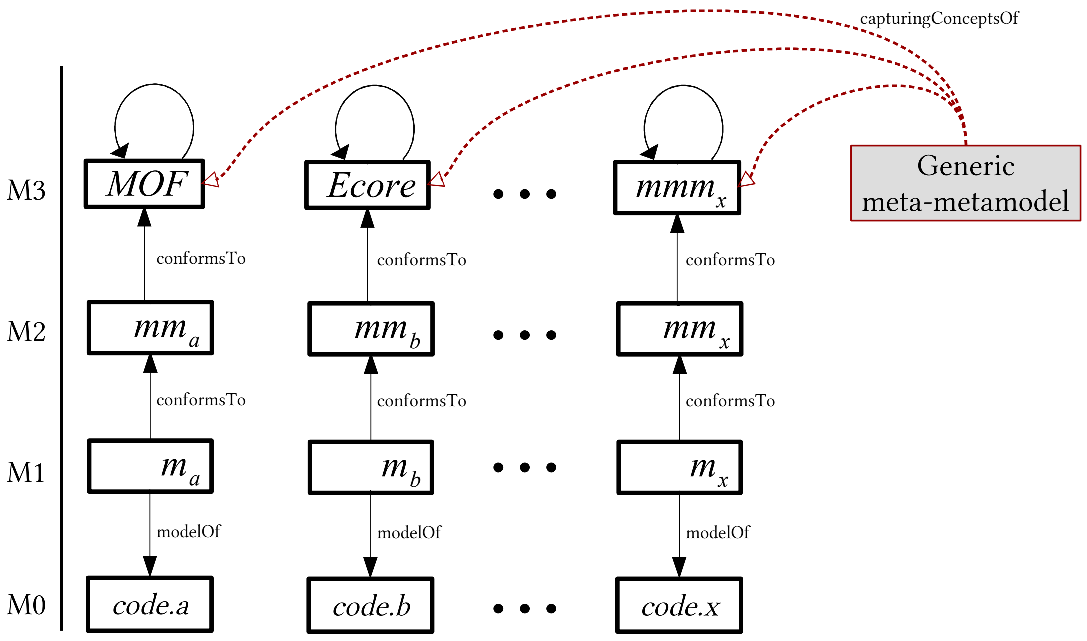

FIGURE 4.1: Typical MDE metamodeling hierarchy and generic meta-metamodel

A single instantiation of the mentioned generic meta-metamodel structure produces a metamodel in which entities are instantiated along with properties (including attribute and relations) and values. For instance, Figure 4.2 gives a metamodel of a simple assignment statement of an example language, in which entities (such as `FOLogicExp`, `BooleanExp`) are represented by UML class-like shapes; relations (such as `conditions`, `source`) by directed lines between the entities; and attributes (such as `iterator`, `name`) defined within entities.

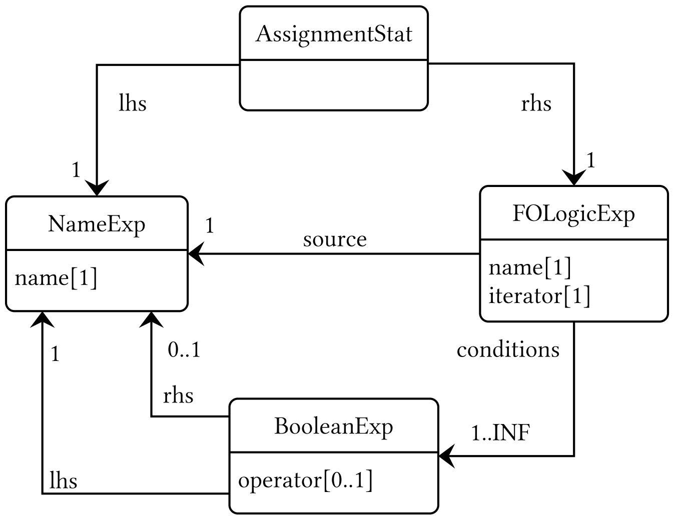

FIGURE 4.2: A generic metamodel

If an instance model exists that conforms to the represented metamodel in Fig. 4.2, such model can be modified (or mutated) so that the values of the properties of its entities are modified by addition, deletion and replacement, which are a set of *mutation actions* that are extracted by analyzing various mutation operators found in the mutation analysis literature (discussed in Sect. 2.2.3). For example, modifications may include replacing the value of the `iterator` property of an instance of `FOLogicExp` or replacing the values of `lhs` and `rhs` properties of an instance of `BooleanExp` with each other. Thus, mutation operators that facilitate the modifications of models are based on the properties of entities in the metamodel, and if those properties and modeling concepts conform to the generic meta-metamodel's concepts and constraints, there is an opportunity to control the design of mutation operators using those exact modeling concepts and constraints of the generic meta-metamodel.

In order to address objective 2b concerning useless mutants, the thesis restricts the modification of models with respect to conformance constraints defined in their metamodel to tackle the limitation of producing invalid mutants. Constraints are useful to ensure that the approach produces mutation operators that, when applied, are less likely to produce invalid mutants, which always impacts negatively on the overall performance of mutation testing. In addition, the thesis also restricts the modification of models with respect to equality and definition constraints, which are both not implementable at the level of the proposed generic meta-metamodel but should, instead, be implemented according to the underlying metamodeling technology. The former constraint is to check the values of properties and new values to be mutated that are not equal, as this would produce equivalent mutants. The latter aims to check the target values and values to be mutated which are defined, for example, as values allocated in the memory or in a resource, or using any manner of definition.

4.1.1 Conformance restriction

When a model conforms to a metamodel, this means that the model complies with the modeling concepts and their respective rules, as provided by the metamodel. Given the previous generic meta-metamodel, a conformance restriction is identified by two concepts: compatibility and multiplicity.

- **Compatibility:** the value of a property of an entity should only be allowed to be modified with a new value from the same kind, determined from the names of entities given in the metamodel. For example, from the metamodel in Figure 4.2, the property `lhs` of an instance of `BooleanExp` may be involved in a replacement of its value (which is an instance of `NameExp`) with another instance of `NameExp`. Allowing the replacement only with an instance of a compatible entity prevents the violation of the conformance constraints.
- **Multiplicity:** multiplicities of properties are determined by the *lower* and *upper* bounds. The lower bound of a property indicates the minimum number of elements that could be represented by a property, while the upper bound gives the maximum number of elements in which the invariant ($lowerBound \leq upperBound$) is always true.
  For example, the property `lhs` of a `BooleanExp` instance has the lower and upper bounds of one, which means that the property always has a single value. As a consequence, the conformance constraint allows the property to have a replacement operator, while it does not permit it to have addition or deletion operators. This is due to the multiplicity of one, which indicates a compulsory property. Adding more values or deleting the only represented value are not allowed. Therefore, obtaining mutation operators for addition and deletion for this particular property is useless, as they generate invalid mutations that break the conformance restriction. Hence, the multiplicity constraint plays an important role in reducing the number of invalid mutations and, by extent, reducing the overall cost of mutation analysis.

4.2 Abstract Mutation Operators (AMO)

The users (test developers) of the design approach presented in the previous section, need to adapt and implement the approach to its underlining metamodeling technology and verify the aforementioned restrictions – namely, restrictions of conformance, equality and definition to address objectives 1 and 2b – in order to generate mutation operators. In an effort to facilitate the adoption and implementation of the approach in an efficient manner, this section presents a *generic* and ready-to-implement set of mutations that incorporates the restrictions in the guise of *preconditions* for each operator.

The proposed set of mutations is referred to as *Abstract Mutation Operators* (AMOs). They are systematically derived from the design approach by considering typical addition, deletion and replacement mutation actions. They are organized into two main categories.

1. **`AMO-single`**: for single-valued properties with an upper bound multiplicity equal to one.
2. **`AMO-multiple`**: for multi-valued properties with an upper bound multiplicity greater than one.

One of the key novelties and benefits of the proposed AMOs (along with their integrated preconditions) is that they can be **instantiated** by a user to produce *concrete* mutation operators that are tailored to a given metamodel (at level M2, as illustrated in Figure 4.3). Such operators are called *Concrete Mutation Operators (CMOs)* and can be used in mutated models conforming to a given metamodel. During instantiation, the preconditions of AMOs are "copied over" to the resulting concrete operators, which should be implemented according to the target metamodeling technology and then evaluated whenever a concrete operator is executed or triggered.

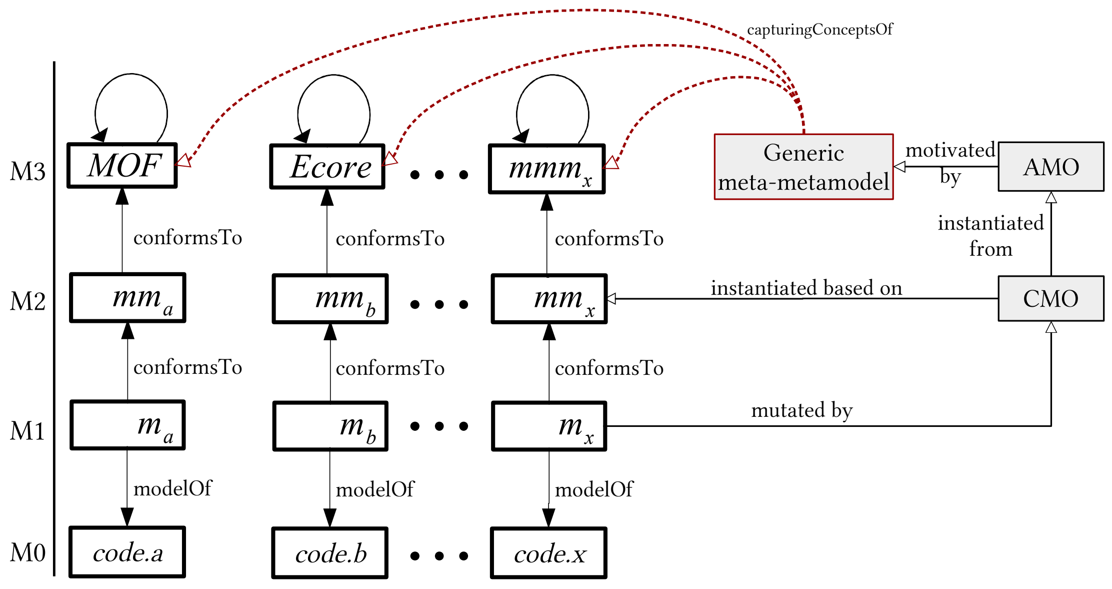

FIGURE 4.3: AMOs and CMOs

Including such preconditions into abstract operators reduces the anticipated difficulty of composing the necessary constraints and restrictions (demanded by the design approach for the purpose of mutation validations) for each property of an entity in the target metamodel. Thus, the user of AMOs can pick a desired property of any entity defined in the metamodel and, based on its multiplicity (for example, upper bound multiplicity) and kind, a target AMO can be determined and instantiated along with its specific preconditions for the same property.

In order to ensure flexibility and adaptability to different metamodels, most AMOs are designed to accept parameters. Users of such abstract operators need to provide values (one value per operator) to a few parameters during the instantiation of the operators into concrete operators. Although user configuration may seem as a limitation, AMOs provide flexibility and allow users (who are usually domain experts to the system they are testing) to decide what to mutate and with what values. It is possible to design AMOs that would require no user involvement by making assumptions of possible values based on the target metamodel. Such value assumptions, however, are not necessarily needed in all modeling languages. The option of user configuration is, thus, useful.

The following subsections present all AMOs, their descriptions and the integrated preconditions of each one.

4.2.1 AMO-single

This category comprises three abstract operators (addition, deletion and replacement operators) for the properties of entities with the upper bound multiplicity of one. The parameter `target` name can be replaced with any target property name that the user wants to mutate.

- **`AMO-single-ADD(Property target, Object` *`addValue`*`)`**: sets the value given by *`addValue`* to the modeling property `target`, when the *`addValue`* contains the value with which to be mutated.
  - $target.getValue() = \varnothing$: checks that the target property is undefined (in other words, has no value or unset) allowing the assignment of the new value given by *`addValue`* in which the function *getValue* retrieves the value that is currently withheld by the target property. If `target` has a value, then the replacement operator is more appropriate than the addition operator.
  - $addValue \neq \varnothing$: ensures that the given value by *`addValue`* is valid (i.e. is set), as the previous condition already demands that the value of the target property is unset. This condition prevents the creation of an equivalent mutant.
  - $[addValue] = [target]$: checks that the kind of *`addValue`* (as indicated by square brackets '[' and ']') is the same kind of `target`, which points to a modeled entity in the target metamodel.

- **`AMO-single-DEL(Property target)`**: removes the value that is associated with the property `target`.
  - $target.getValue() \neq \varnothing$: checks that the target property is defined so as to allow the removal of its value.
  - $target.lowerBound() = 0$: verifies that the lower bound multiplicity of `target` is equal to zero, which indicates that the target property is not a required (i.e. optional) property and, hence, its value can be removed.
- **`AMO-single-REP(Property target, Object *replaceValue*)`**: replaces the value of the property `target` with a new value given by ***`replaceValue`***.
  - $(target.getValue() \neq \varnothing) \wedge (replaceValue \neq \varnothing)$: checks that the value of property `target` and the value given by ***`replaceValue`*** are both valid (in other words, are unset) so that the replacement can take place. If the property `target` has no value, however, the addition operator is more appropriate than the replacement operator.
  - $[replaceValue] = [target]$: checks that the kind of ***`replaceValue`*** is compatible with the kind of property `target`.
  - $target.getValue() \neq replaceValue$: makes sure that the existing value of the property `target` is not equal to ***`replaceValue`***, in order to prevent the generation of equivalent mutations.

4.2.2 AMO-multiple

This category of AMOs contains a set of operators that is suitable for multi-valued properties whose upper bound multiplicity is greater that one. The value that is represented by any target property with multiplicity greater than one, is that it is a collection of items. Thus, the AMOs in this category target a collection of multi-values by adding, deleting or replacing an item in the collection.

- **`AMO-multiple-ADD(Property target, Object *newItem*)`**: adds the value given by instance ***`newItem`*** to the collection of items derived by the value of property `target`.
  - $newItem \neq \varnothing$: checks that the value of the instance ***`newItem`*** is defined and valid in order to allow the addition mutation.
  - $[newItem] = [target]$: checks that the kind of ***`newItem`*** and the kind of property `target` are compatible (have the same entity name).
  - $newItem \notin target.getValue()$: makes sure that the value of ***`newItem`*** is not contained within the value of property `target`.
  - $|target.getValue()| + 1 \leq target.upperBound()$: verifies that the size of items (indicated by the vertical bar '|') of the value of property `target` can be

increased by one additional item, in such a way that the total size of items does not exceed the value of the upper bound multiplicity.

- **`AMO-multiple-DEL(Property target, Object item)`**: deletes `item` from the collection of items given by the value of property `target`.
  - $item \neq \varnothing$: checks that the value of the instance `item` is valid.
  - $item \in target.getValue()$: makes sure that the value of property `target` contains the instance `item`.
  - $|target.getValue()| - 1 \geq target.lowerBound()$: verifies that the size of items (indicated by the vertical bar '|') of the value of property `target` is greater than or equal to the lower bound multiplicity after deleting the element `item`.
- **`AMO-multiple-REP(Property target, Object oldItem, Object newItem)`**: replaces the `oldItem` with `newItem` from the collection of items given by the value of property `target`.
  - $(oldItem \neq \varnothing) \wedge (newItem \neq \varnothing)$: ensures that the given values of `oldItem` and `newItem` are valid.
  - $oldItem \in target.getValue()$: verifies that the old value of `oldItem` (the one to be mutated) is contained by the collection value of property `target`.
  - $newItem \notin target.getValue()$: ensures that the value of property `target` does not contain the value `newItem`.
  - $[newItem] = [target]$: checks that the kind of `newItem` and the kind of property `target` are compatible.

4.3 Adaptation of the Approach to Ecore

This section presents an adaptation of the design approach, described in Section 4.1 and based on the generic meta-metamodel to ensure flexibility and adaptability to different metamodeling technologies, to Ecore. As mentioned in Section 2.1.3.1, Ecore is a metamodeling technology provided by the Eclipse Modeling Framework (EMF) and is a well-known implementation of Meta Object Facility (MOF), which is a model driven engineering standard for modeling and metamodeling.

By using Ecore, modelers can define new metamodels, reuse existing ones and manipulate their instances. Since MOF and its implementation, Ecore, facilitate interoperability between different MDE platforms and applications, Ecore can be a good metamodeling candidate to target. Thus, providing an extension of the mutation operator design approach to Ecore can be beneficial, as it is widely used in an MDE context and, thus, provides ample potential for implementation than other modeling technologies.

Figure 4.4 presents the abstract syntax of the core modeling elements of Ecore taken from [15]. It is represented using a graphical syntax that is similar to the UML class diagram. The model structure mirrors the terminology that was given for the generic meta-metamodeling structure, (introduced in Section 4.1). For instance, entities are mapped to `EClass`, `EDataType`, properties are mapped to `name`, `abstract`, and associations are mapped to `eStructuralFeatures`, `eSuperTypes` etc.

In the following subsections, the restrictions of the proposed design approach against the generic meta-metamodel are investigated over Ecore and its modeling concepts. Further, an adaptation of the abstract operators (AMOs) to Ecore is proposed, followed by a discussion on other modeling concepts of Ecore that could be used to impose more restrictions.

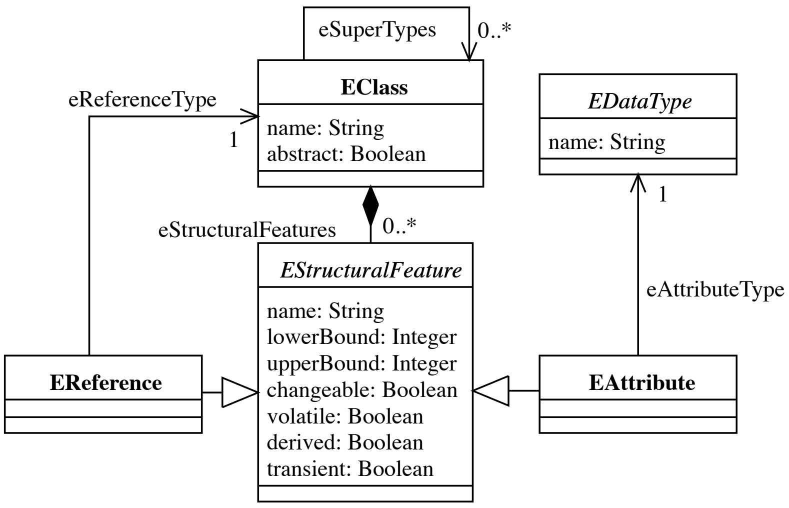

FIGURE 4.4: A representation of core elements of Ecore taken from [15]

4.3.1 Conformance restriction

It has already been mentioned that the conformance restrictions of the proposed design approach of mutation operators (introduced in Sect. 4.1.1) depend on two concepts: compatibility and multiplicity. In terms of compatibility, Ecore and its underlining technology, provide the concept of *types*, which are fully implemented in Java classifiers. These correspond to entities in the generic meta-metamodeling structure. With types, the compatibility of instances is maintained automatically by Ecore and its underlying Java Language.

In addition, the compatibility of types is checked over the multiple inheritance of types, allowing an instance of an entity to be compatible with more than one kind (which was not the case for the generic meta-metamodel, discussed in Sect. 4.1.1). Thus, the original approach is extended to support the multiple inheritance of entities. In addition, Ecore provides *EDataTypes*, which are affiliated to Java primitive or object types and are used for modeling the properties (excluding end associations) of entities.

In terms of the multiplicity of properties, the lower and upper bounds constraints are also provided by Ecore and they can be used to prevent the production of invalid mutants, similar to the multiplicity restriction indicated for the generic meta-metamodel (Sect. 4.1.1).

4.3.2 Further restrictions for Ecore

Ecore metamodeling technology offers further modeling concepts that characterize properties. These modeling concepts (along with other concepts) include *changeable*, *derived* and *transient*, which can be explicitly indicated in a metamodel [15].

A changeable property value is modifiable and can be changed to a new value. By contrast, static (i.e. not changeable) properties are not modifiable. Hence, their values cannot be mutated. Therefore, the design approach can be extended to avoid the generation of mutation operators for static properties, as they produce invalid mutants, which violates the conformance constraint of the metamodel expressed using the Ecore metamodeling framework.

The values of derived properties are determined or computed from other properties. Thus, designing mutations for such properties is unnecessary. Accordingly, the extension of the approach to Ecore can include additional restrictions disallowing the mutation design for derived properties. As a consequence, this may reduce the number of mutation operators and, as a result, the overall computational cost of performing mutation analysis.

Lastly, the value of a transient property is excluded from the serialization of the containing entity when persisting the containing model. In other words, if there is a model that contains transient properties or association values, such values are not going to be part of the containing model (model resource) and are simply ignored. Derived properties are good examples of properties that are sometimes indicated as transient. Thus, the approach can be extended further to include a restriction by which the design of operators for transient properties should not be allowed. In this case, the number of mutation operators and the cost of mutation analysis can be reduced.

4.3.3 Adaptation of AMOs to Ecore

The original set of AMOs presented in 4.2.1 and 4.2.2 was designed based on conformance and equality restrictions. AMOs can further be extended to include modeling concepts provided by the metamodeling technology. In the following sections, an adaptation of AMOs to Ecore is presented which covers additional modeling concepts specific to Ecore, namely multiple inheritance and Ecore EDataTypes, both of which can be disclosed at the metamodel level.

4.3.3.1 Inheritance

Ecore offers inheritance and multiple inheritance concepts between entities, according to which an entity may have one or multiple super-kinds. As such, the compatibility conditions of AMOs can be extended from a direct equality check between two kinds using the equal character (=) to a further check to super-kinds. Multiple kinds investigation are preformed by Ecore and its underlying technology so that all sub-kinds of a relationship property are collected from the metamodel and evaluated against a mutating value. Hence, the compatibility of AMOs preconditions can be re-expressed using '$\sqsubseteq$' (instead of '='), transforming the immediate super-kind relation as follows:

$$[subkind] \sqsubseteq [super\text{-}kind]$$

4.3.3.2 Ecore EDatatype

Another modeling concept provided by Ecore is a set of predefined data-types that can be associated to the properties of entities. These predefined data-types are referred to as EDatatypes, and they can be affiliated to Java primitive or Java object types. In order to modify the values of EDatatype properties, the set of mutation operators of single properties **`AMO-single`** can be used to instantiate a set of specific concrete operators for Java object types, such as *java.lang.String*, *java.lang.Integer* etc.

For primitive types, however, operators of **`AMO-single`** can be extended to include conditions that would take the default values of primitive types into account. With Ecore, the properties of primitive types can have default values, which are determined by the underlying technology of Ecore and Java. Those values are never unset or deleted. Therefore, the addition operator **`AMO-single-ADD`** (which requires a no-value state for the target property) and **`AMO-single-DEL`** (which requires a value to be contained by the target property) do not apply for properties of primitive types. As a consequence, only the

`AMO-single-REP` operator is applicable, since the value of the mutated property can be modified by replacement with another value. Table 4.1 presents all available EDatatypes in Ecore, their corresponding Java types (whether they are primitive or object types), and their default values according to the Java Language Specification in [105]. The table also shows the applicability of mutation for each EDatatype.

Table 4.1: A list of EDatatypes provided by Ecore

Ecore datatype	Affiliated Java type (primitive or object)	Default value	Allowed mutation action
EBoolean	boolean	false	replacement
EByte	byte	0	replacement
EShort	short	0	replacement
EInt	int	0	replacement
ELong	long	0	replacement
EFloat	float	0.0f	replacement
EDouble	double	0.0d	replacement
EChar	char	'\u0000'	replacement
EString	java.lang.String	null	addition, deletion, replacement
EBooleanObject	java.lang.Boolean	null	addition, deletion, replacement
EByteObject	java.lang.Byte	null	addition, deletion, replacement
EShortObject	java.lang.Short	null	addition, deletion, replacement
EIntegerObject	java.lang.Integer	null	addition, deletion, replacement
ELongObject	java.lang.Long	null	addition, deletion, replacement
EFloatObject	java.lang.Float	null	addition, deletion, replacement
EDoubleObject	java.lang.Double	null	addition, deletion, replacement
ECharacterObject	java.lang.Character	null	addition, deletion, replacement
EDate	java.util.Date	null	addition, deletion, replacement
EBigInteger	java.math.BigInteger	null	addition, deletion, replacement
EBigDecimal	java.math.BigDecimal	null	addition, deletion, replacement
EJavaClass	java.lang.Class	null	addition, deletion, replacement

4.4 Instantiation of AMOs

Section 4.2 presented a set of metamodel-agnostic mutation operators (AMOs) that are integrated with constraints to reduce the risk of generating invalid and equivalent mutations. The AMOs can be instantiated to generate specific mutation operators for a given metamodel. The application and instantiation of the abstract operators, however, rely heavily on the user, who may wish to instantiate a selective set of operators against certain modeling concepts and properties defined in the target metamodel. This section gives examples of how the AMOs are used and instantiated to generate *concrete mutation operators (CMOs)* based on an example of a metamodel.

The metamodel used for these examples is given in Figure 4.5. It is fragment of a metamodel of a language called MiniLang and expressed in a graphical syntax similar to the UML class diagram. The examples generate four CMOs, two of which are valid and the remaining two invalid. The properties targeted in the examples are property `condition` of entity `FOLogicExp` with a replacement mutation, property `statements` of entity `IfStat` with addition and deletion mutations, and property `method` of entity `FOLLogicExp` with a replacement mutation. The AMOs that are going to be instantiated, which are determined by the multiplicity of the mentioned properties, are **`AMO-single-REP`**, **`AMO-multiple-ADD`**, **`AMO-multiple-DEL`** and **`AMO-single-REP`** respectively.

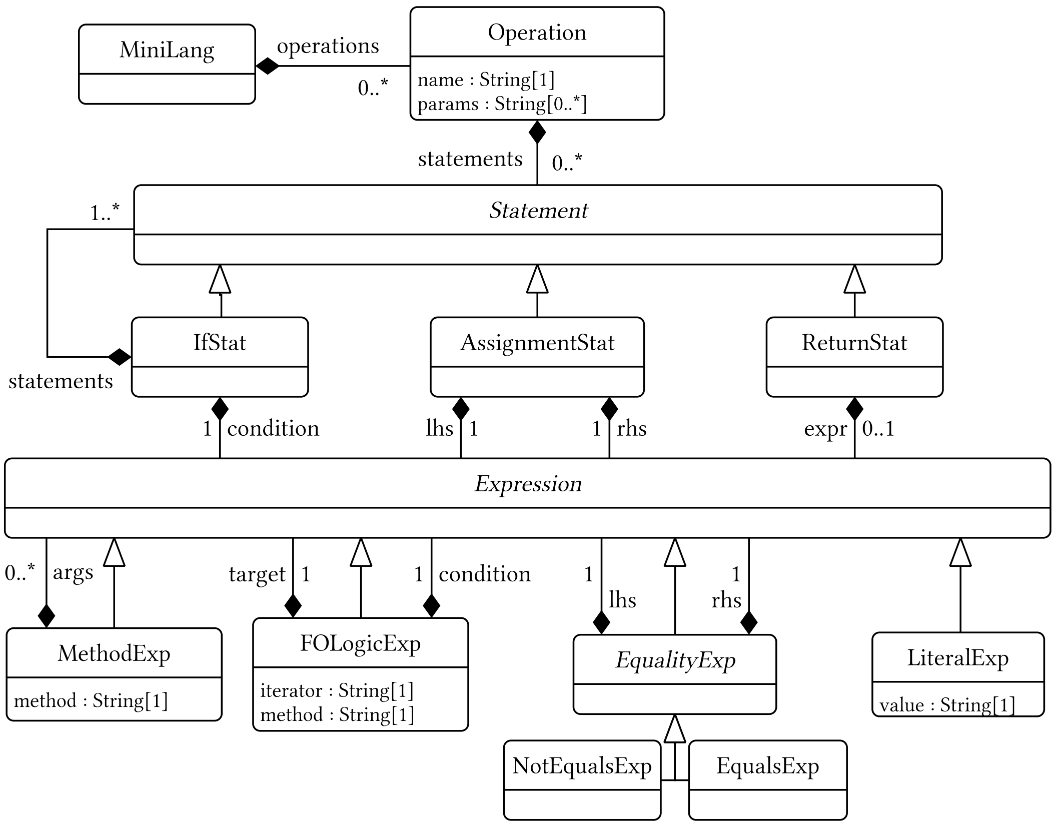

Figure 4.5: The abstract syntax of the MiniLang metamodel

4.4.1 Example: a replacement CMO for property `condition`

This example instantiates the abstract operator **`AMO-single-REP`** against the property `condition` of the modeling entity `FOLogicExp`. The objective is to generate an operator that produces a valid mutant by mutating the instance model (depicted in Figure 4.6

that models the piece of code in Listing 4.1) of the metamodel in Figure 4.5. The mutation changes the Boolean value of the property `condition` (highlighted in red) of a first-order logic operation ***`folExp1`*** from a Boolean equal comparison operator to a not equal comparison operator, in other words, changing from an instance of `EqualsExp` to an instance of `NotEqualsExp` (highlighted in green) that is a new created instance of `NotEqualExp`. The values of `lhs` and `rhs` of the original value of (***`equalExp1`***) are copied over (or assigned) to the new instance ***`notEqualExp1`*** (which is mutated). This mutation imitates the misuse of an equality operator of Boolean property `condition` of a first-order logic.

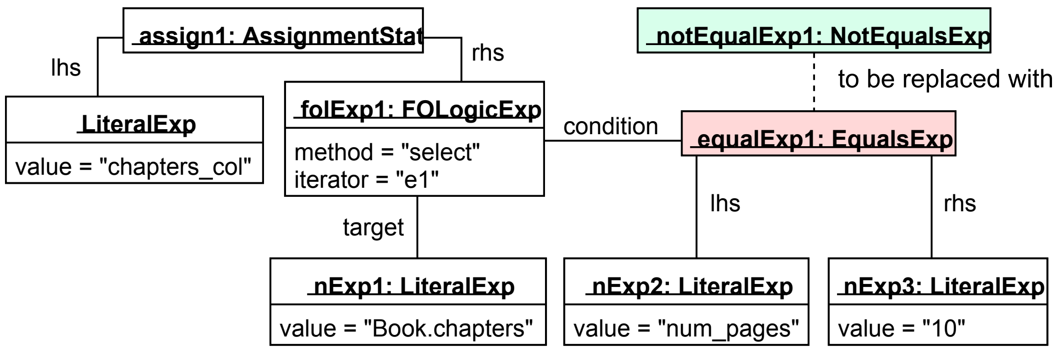

FIGURE 4.6: A fragment of an instance model of the MiniLang metamodel 4.5 used for the instantiation example 4.4.1

```
...
chapters_col = Book.chapters.select(e1 |  e1.num_pages == 10);
...
```

LISTING 4.1: An example of code that is modeling in Fig. 4.6

- **`CMO-single-REP(Property condition,`** ***`notEqualExp1`*** **`)`**: replaces the value of `condition` with a new instance ***`notEqualExp1`*** of the entity `NotEqualsExp`.
  - $(condition.getValue() \neq \varnothing) \wedge (notEqualExp1 \neq \varnothing)$: this constraint is true as both the value of the property `condition`, which equals to ***`equalExp1`***, and the new value ***`notEqualExp1`*** are valid (or defined).
  - $[notEqualExp1] \sqsubseteq [condition]$: the returned kind of instance ***`notEqualExp1`*** is `EqualsExp`, a sub-kind of entity `Expression`, which is the same kind as that of the end association property `condition`. Therefore, this condition is satisfied.
  - $condition.getValue() \neq notEqualExp1$: this constraint is valid since the instances ***`equalExp1`*** and ***`notEqualExp1`*** have different values.

4.4.2 Example: an addition CMO for property `statements`

This example demonstrates the instantiation of the abstract operator **`AMO-multiple-ADD`** against the property `statements` of an instance of `IfStat`. This generates a CMO that produces a valid mutant by adding a return statement, assuming that the if statement is contained or placed within an operation definition. Figure 4.7 illustrates the mutation that occurs when the return statement ***retStatement*** instance (highlighted in green) is added to `statements`. The purpose of this mutation is to ensure that statements following the mutated if statement block are important. In other words, this particular mutation verifies whether statements that follow the mutated condition block (the if block) are executed.

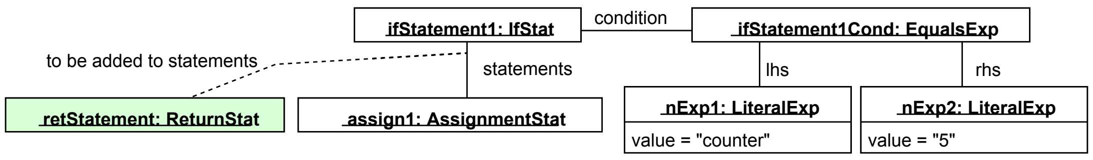

FIGURE 4.7: A fragment of an instance model of the MiniLang metamodel 4.5 used for the instantiation example 4.4.2

- **`CMO-multiple-ADD(Property statements, retStatement)`**: adds the instance of ***retStatement*** to the collection of statements of property `statements`.
  - $(statements.getValue() \neq \emptyset) \wedge (retStatement \neq \emptyset)$: this constraint is true since both values of `statements` and ***retStatement*** are valid and defined.
  - $[retStatement] \sqsubseteq [statements]$: the constraint is valid because the instance ***retStatement*** is of the same kind (`Statement`), as the property `statements`.
  - $retStatement \notin statements.getValue()$: instance ***retStatement*** is not contained by the value of property `statements` and, therefore, the condition is valid.
  - $|statements.getValue()| + 1 \leq statements.upperBound()$: the condition is fulfilled since, following the addition, the total size of property `statement` is two, which is less than the upper bound limit of the unbounded natural (indicated by *).

4.4.3 Example: a deletion CMO for property `statements`

This example generates a deletion CMO by instantiating **`AMO-multiple-DEL`** for the property `statements` of entity `IfStat`. The target model for the mutation is represented in Figure 4.8, so that the instance ***assign1*** (highlighted in red) contained by the property `statements` of instance ***ifStatement1*** is deleted. Since only one instance ***assign1*** is available, the preconditions of the generated operator will prevent the deletion of the only instance ***assign1***. As the lower bound of `statements` is one, deleting the last element will produce an invalid mutant. The generated concrete operator is given as follows:

- **`CMO-multiple-DEL(Property statements,assign1)`**: deletes the instance ***assign1*** from the collection of values of the property `statements`.

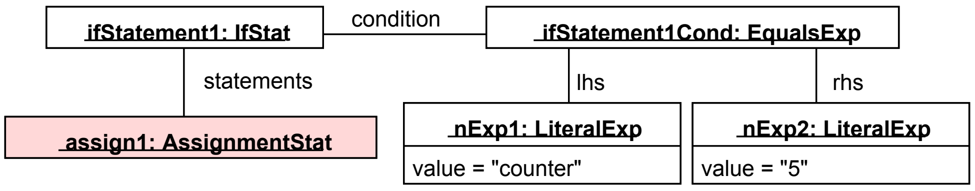

FIGURE 4.8: A fragment of an instance model of the MiniLang metamodel 4.5 used for the instantiation example 4.4.3

- $(statements.getValue() \neq \varnothing) \wedge (assign1 \neq \varnothing)$: this condition is true since both the values of the target property of instance ***ifStatement1*** and ***assign1*** are defined and valid.
- $assign1 \in statements.getValue()$: the condition is satisfied because the instance ***assign1*** is contained by the value of property `statements`.
- $|statements.getValue()| - 1 \geq statements.lowerBound()$: this condition is **violated** because, following deletion, the total size of the value of the property `statements` would equal to zero, which is less than the allowed lower bound of one, as indicated in the metamodel 4.5.

4.4.4 Example: a replacement CMO for property `method`

In this example, a replacement CMO for property `method` of entity `FOLLogicExp` is instantiated to generate a CMO operator that allows the replacement of the method name with another name. The target model to be mutated is represented in Figure 4.9. In order to demonstrate the benefits of using the preconditions of the operator generator, the example attempts to replace the current method name ("select") with the same name as follows:

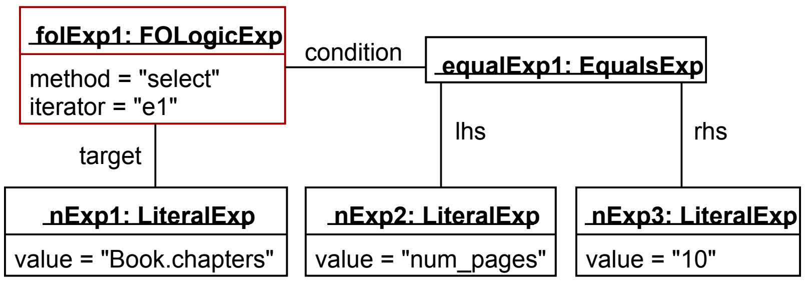

FIGURE 4.9: A fragment of an instance model of MiniLang metamodel 4.5 used for the instantiation example 4.4.4

- **`CMO-single-REP(Property method, "select")`**: replaces the value of property `method` with the string "select".
  - $(method.getValue() \neq \varnothing) \wedge (\text{"}select\text{"} \neq \varnothing)$: this constraint is true as the both values of the property `method` and the value "select" are defined values and are not unset.

- $[\text{"}select\text{"}] \sqsubseteq [method]$: the kind of property `method` is `EString`, which is affiliated to `java.lang.String` that is same kind of "select" value. Therefore, this condition is satisfied.
- $method.getValue() \neq \text{"}select\text{"}$: this condition is **violated** because both values are equal to "select". Therefore, this mutation is not allowed to prevent the production of an equivalent mutation.

4.5 Chapter Summary

This chapter has presented a mutation operator design approach tailored to MDE languages. The approach is based on a generic metamodeling model (or meta-metamodel) of entities with properties (including relations between entities). In addressing the objectives of this thesis, the mutation design approach has imposed restrictions onto the process of generating mutation operators by including conformance, equality and definition constraints. The purpose is to tackle a few of the limitations of mutation design in an MDE context (as discussed in Section 3.1). The chapter has also introduced a set of abstract mutation operators (AMOs) derived from the design approach and its restrictions.

The AMOs are accompanied by constraints aimed at producing low quantities of useless mutants (i.e. invalid and equivalent mutants). Test developers, who wish to use AMOs, can instantiate a selective set of AMOs operators upon certain modeling concepts (properties of entities) defined in a given metamodel, in order to mutate models that conform to that metamodel. The instantiation will generate a set of concrete mutation operators that are customized for the metamodel and can be used to mutate models (and programs) that conform to such metamodel.

The chapter has also put forward an adaptation of the approach and the AMOs to the Ecore metamodeling language, which is a widely used practical implementation of MOF in model driven engineering. The adaptation has addressed specific modeling concepts of Ecore that were, in turn, used to impose more restrictions on top of what the original design approach offers. Further, a number of instantiation examples of AMOs has been presented over an Ecore model of a simple metamodel of a language. In the examples, abstract operators were instantiated to produce concrete ones for selected modeling properties, with preconditions validated against example models that conformed to the simple metamodel.

Since the implementation of CMOs relies on the underlying modeling technology alongside the precondition, the resulting concrete operators from AMOs are not made (by themselves) executable to support flexibility to a user selective language. The following chapter will introduce a model mutation language that provides abstraction layers to facilitate the implementation of CMOs and also the checking of their constraints.

Chapter 5

Epsilon Mutator (EMU)

Chapter 4 presented an approach for mutation operator design for MDE languages. It also presented a set of metamodel-agnostic mutation operators (AMOs). The proposed approach along with the AMOs address the objectives put forward by this thesis, whereby AMOs are restricted and controlled by a set of preconditions of conformance, equality and definition constraints. Users of the approach may adapt it to a desired modeling technology, implement AMOs, and then perform validation against the preconditions, whose aim is to reduce the likelihood of producing useless mutants. In order to facilitate the instantiation of AMOs in a convenient manner, this chapter introduces the **E**psilon **MU**tator (EMU): a language and associated tooling for the mutation approach and AMOs users; and amenable to model mutation.

EMU is an extension of the Epsilon Pattern Language (EPL) [18] (reviewed in Sect. 2.1.3.2). There is a number of reasons for specifically choosing EPL. The first is that it belongs to a family of languages developed on top of the Epsilon Object Language (EOL) provided by the Epsilon Platform, which supports various types of models. By extending EPL, EMU (which currently only supports EMF) may be extended to support different types of models. Section 5.1 gives an overview of such support.

Another reason for choosing EPL is that its output, and consequently that of EMU, can be further manipulated using other Epsilon languages. In fact, Epsilon languages, which are tailored to different model management tasks, such as model transformation, model validation etc., have different dialects based on a common expression language (EOL). Thus, users of EMU are more likely to be familiar with other Epsilon languages. Hence, developing a transformation or validation program or any model management task using the Epsilon languages to target outputs of EMU is easier than using languages from outside the Epsilon platform.

Further, EPL has two distinct features: the first is pattern matching (which exists in most model management languages) and the second is a set of actions performed against the matching results of the pattern matching process. If the set of actions is restricted to one action to allow one single mutation at a time, EPL is a good fit for mutation application purposes. As such, EMU overrides minor EPL execution semantics and (along checking and validating internally the precondition of AMOs as well as and applying CMOs) allows one action to be performed that produces only one mutation per application for a selected modeling concept (property) defined in a given metamodel.

Using EMU and the pattern matching mechanism, test developers can first navigate through a target model of a metamodel and mutate a specific value of a property. Then, based on the property and a target mutation action, a corresponding AMO is triggered over all values and occurrences of the property in the model and a single mutation is produced for each occurrence. Finally, the preconditions of the triggered operator are validated explicitly by the engine, in order to obstruct the production of useless mutants. In addition, the language of EMU allows test developers to specify further constraints and conditions on top of AMOs' preconditions for flexibility purposes and for added control over the operators' application. Also, since EMU is developed atop of Epsilon platform, users of the language can further validate mutants of EMU programs with further constraints (e.g., constraints integrated into metamodels) using OCL, EVL or EOL.

It is important to mention that EMU is not a mutation testing framework that one can use to build or assess test set or to produce mutation metrics such mutation score. Instead, it is a prototype language and tooling support that can be used to mutate models in MDE context and produce valid models for mutation testing purposes.

The remainder of this chapter presents EMU in the following order. Firstly, the Mutant Integration Layer that enables model mutation is presented in Section 5.1. Next, the abstract and concrete syntaxes of EMU are presented in Section 5.2 and Section 5.3 respectively. Finally, the execution semantics of EMU is presented in Section 5.4 with explanation and examples.

5.1 Mutant Integration Layer

As mentioned previously, AMOs impose a number of preconditions in order to produce low levels of useless mutants. The preconditions rely on three constraints: conformance, equality and definition. In order to allow the implementation of these preconditions for particular modeling technologies, EMU provides the Mutant Integration Layer. This extends the Epsilon Model Connectivity Layer (as illustrated in Fig. 2.6) and its core abstraction component *IModel* [18], providing further abstraction components (related to model mutation and the implementation of the preconditions) over modeling technologies such as EMF, XML, ArgoUML etc. that allow EMU programs to manipulate models of these modeling technologies. The abstraction components in question are *IProperty* and *IMutant*. Their UML class diagram is depicted in Figure 5.1.

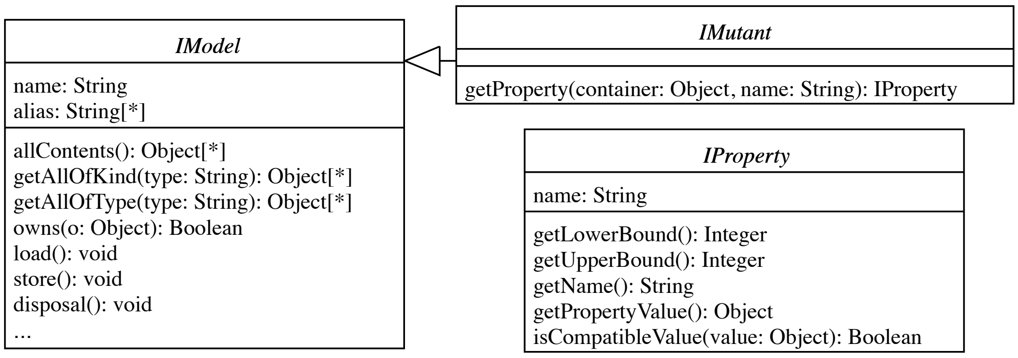

FIGURE 5.1: Epsilon *IModel* abstraction and Mutant Integration Layer

Currently, a mutation driver implementation of the above abstraction components for EMF/Ecore based models is already provided, including support for precondition and constraint parameters of AMOs (such as multiplicity, property and value kinds etc.).

5.2 Abstract Syntax

Figure 5.2 shows a subset of EPL abstract syntax that is only used and overridden by EMU. An EMU module can have a number of mutation applications in terms of *patterns*, each of which is associated with a number of binding roles (pattern matching roles) that query the target models and obtain corresponding instances of the pattern matching result. Querying the model is facilitated by an EOL expression in terms of a *domain* (to specify the scope of instances of the target model) and *guard* (to impose more filtering conditions and constraints) over the result of each query. The mutation must be introduced using an EOL *do* block that is re-executed against every model instance that exists in the binding roles. For example, if a model contains five *greater than* comparison operators that a test developer wants to mutate, then the binding roles should obtain five places in the model at which the mutation expressions in the *do* block are repeated five times.

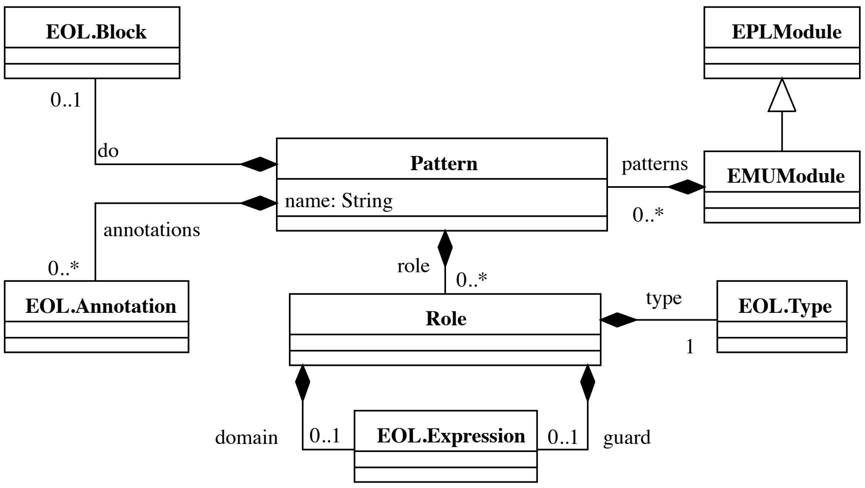

FIGURE 5.2: The abstract syntax of EMU

5.3 Concrete Syntax

The concrete syntax of EMU is given in Listing 5.1. To begin with, two mandatory annotations (starting with the symbol @, as in lines 2 and 3) are required for each mutation operator. The annotation @*action* determines the target mutation action whether addition, deletion or replacement. The annotation @*property* distinguishes the target property name (whether an entity attribute or an association property) that will be modified.

Since the *do* block is an EOL block that may contain a number of EOL statements, EMU treats the block as a black-box, and hence, the ability to know the user desire mutation action and target property currently (when EMU extends EPL) is impossible. Therefore, the annotations are made compulsory to guide any mutation.

In addition, since a mutation (pattern) can have multiple binding roles (used to query and filter the target model instances), the optional annotation @*role* can be supplied to determine the target role that combines a set of instances in which the property name, which is indicated by the annotation @*property*, exists. In other words, if there are multiple binding roles, each containing a distinct set of instances, that own a property with a name equal to the mutated property name given by @*property*, then the optional annotation @*role* is used to distinguish the target binding role in order to avoid any ambiguous references to properties that have the same name as the target mutated property.

```
1  EMUModule:
2          '@action' ('add'|'delete'|'replace')
3          '@property' ID
4          ('@role' ID)?
5          'pattern' ID
```

```
6           ROLE (',' ROLE)* ('{' DO '}')
7   ROLE:
8           ID ':' EOL.Type (DOMAIN|GUARD)*
9   DOMAIN:
10          ('in'|'from') ':' EOL.Expression
11  GUARD:
12          'guard' ':' EOL.Expression
13  DO:
14          'do' '{' EOL.Block '}'
```

Listing 5.1: EMU concrete syntax

Upon specifying the mutation action and the target mutated property, a number of binding roles can be provided. Binding roles are concepts that accumulate a set of model instances obtained from querying and filtering the target model using EOL expressions in terms of the *DOMAIN* and *GUARD* language concepts. The execution engine then iterates through the obtained instances using the defined variable of the binding role in line 8 (ID ':' EOL.Type), executing the EOL *do* block at every instance.

Using a domain language concept, a scope of model instances can be specified by *in* (for static query) or *from* (for dynamic query) keywords, as originally facilitated by EPL [18], followed by the query using an EOL expression (in line 10). The difference between the static and dynamic model queries of a pattern is that in the first instance the result of evaluating the query (EOL expressions) is computed once for all executions of the *do* block and the results are not accessible by other query expressions of other binding roles of the pattern. On the contrary, the dynamic query result is re-computed whenever the containing binding role's values are iterated and executed. Its values are accessible by other binding roles of the pattern.

In using the *GUARD* language concept, the user can impose further filtering and constraints over the resulting values of the role query using the keyword *guard*, as in line 12. This feature allows test developers to specify additional constraints over the AMOs' preconditions in order to increase flexibility and control over model queries.

Finally, the language concept *DO* allows users to specify new values to be used with mutation operators. Since AMOs are parameterized to allow their instantiation to different metamodels for flexibility purposes, the EOL *do* block can be used to supply a desired value to match the application of the concrete mutation operator. The supplied value is then re-used with all occurrences of the target modeling property in the target model.

5.4 Execution Semantics with Examples

The purpose of EMU is to facilitate the implementation of CMOs of a particular metamodel. This section gives examples of using EMU to instantiate and implement the CMOs illustrated in the previous instantiation examples given in Section 4.4. Further, the execution semantics of EMU are explained. The examples were based on metamodel Figure 4.5, which was presented in Sect. 4.4, to instantiate concrete operators for properties `condition` of entity `FOLogicExp`, `statements` of entity `IfStat`, and `method` of entity `FOLLogicExp`.

5.4.1 CMO implementation of Example 4.4.1

In the example in Sect 4.4.1, the Listing 5.2 gives an EMU program that implements the replacement CMO of property `condition` of entity `FOLogicExp`, which involved the replacement of an equal comparison operator with a not-equal comparison operator of the Boolean property `condition`.

The reason of this example is to demonstrate how a common mutation in programming languages in which an equality operator is modified with replacement action by using EMU. This mutation has preserved the left-hand-side and the right-hand-side of the instance (*`equalExp1`* by assigning them to the new instance *`notEqualExp1`*.

To begin with, the EOL expression in line 1 indicates that a replacement action is to be triggered against the modeling property `condition` of entity `FOLogicExp` by the expression in line 2 (that gives the property name) and the expression in line 4 (that gives the modeling concept or entity that contains the property). Based on the mutation action and property name, the EMU triggers the abstract operator **`AMO-single-REP`** and its preconditions that will be validated by the EMU engine.

The domain of `FOLogicExp` instances are filtered based on the type of `condition`. Since the operator replaces conditions of instances of `EqualsExp` to `NotEqualsExp`, the expression in line 6 filters the instances to be of type `EqualsExp` by calling first order logic operation *select* on all instances, and checking their types using the *isTypeOf()* method, passing the modeling entity `EqualsExp` as an argument. The instances are statically obtained from the contained model at once, using the keyword *in* at the beginning of line 6. In the final step, the *do* block is re-executed on all obtained instances in the target model.

```
1  @action replace
2  @property condition
3  pattern condition_equal2notequal
4  instance:FOLogicExp
5    in:FOLogicExp.all.select(e|
6     e.condition.isTypeOf(EqualsExp)){
7    do {
8     //create new Boolean object
9     var n_equal = new NotEqualExp();
10
11    //copy over lhs from old instance
12    n_equal.lhs=instance.condition.lhs;
13
14    //copy over rhs from old instance
15    n_equal.rhs=instance.condition.rhs;
16
17    //assign new condition
18    //to this FOLogicExp
19    instance.condition = n_equal;
20   }
21 }
```

Listing 5.2: A CMO implementation for Example 4.4.1

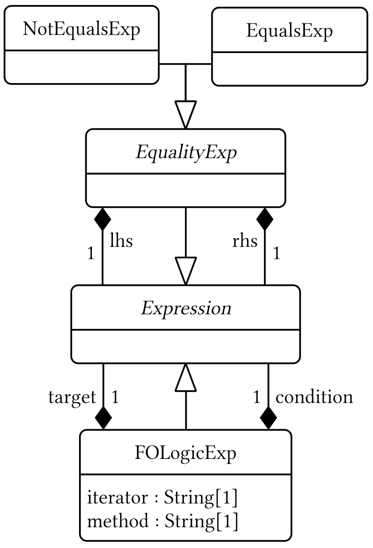

Figure 5.3: A fragment of the metamodel in Fig. 4.5

5.4.2 CMO implementation of Example 4.4.2

The CMO example in Sect. 4.4.2 was about adding a return statement (an instance of entity `ReturnStat`) to a collection of statements given by the property `statements` of type `IfStat`.

The EMU implementation of such example is given in Listing 5.3. In the implementation, the addition mutation action is to be triggered (in line 1) against the property `statements` (in line 2). Since the property is multi-valued (its upper bound is above one and is not limited), the addition mutation involves adding a return statement to a collection of statements.

In this example, the binding role (in line 4) does not have any domain (*in* or *from*) keywords and, therefore, by default all instances of entity `IfStat` are fetched by the engine with no domain filtration, contrary to what was presented in line 6 of Listing 5.2.

5.4.3 CMO implementation of Example 4.4.3

The demonstrated example in Sect. 4.4.3 involved the generation of a CMO that deleted a single statement represented by the multi-valued property `statements` of entity `IfStat`. The purpose of that example was to demonstrate the benefit of the AMOs preconditions in which they were used to prevent the production of invalid mutant when deleting the only single statement in that required and multi-valued property using an example model.

```
@action add
@property statements
pattern add_return_statement_to_If
instance:IfStat {
  do {
   // create new ReturnStat object
   var ret_stat = new ReturnStat();

   // add the new object to the collection
   // of statements
   instance.statements.add(ret_stat);
  }
}
```

LISTING 5.3: A CMO implementation for Example 4.4.2

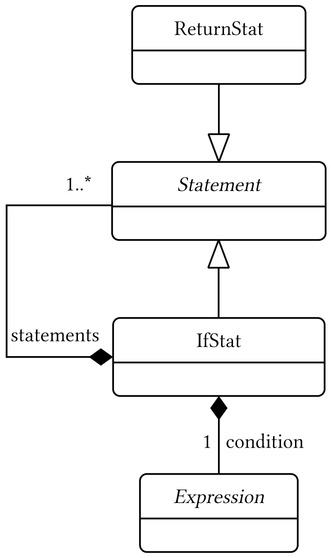

FIGURE 5.4: A fragment of the metamodel in Fig. 4.5

The EMU implementation (Listing 5.4) of that CMO example, however, is more general and not specific to a situation when there is only one statement. The implementation works on any value represented by the property `statements` and on any model that conforms to the MiniLang metamodel. The checking of the preconditions of the concrete operator for the property `statements` is left to the EMU engine, which is going to prevent any violation of the preconditions including the deletion of the last statement from a list of statements.

The implementation has two binding roles. The first holds instances of `IfStat` (in line 5). The role is not domain specific and, therefore, by default all instances of the kind `IfStat` are collected once from the input model. The second binding role holds instances of the kind `Statement` that are collected dynamically (using the keyword *from*) from instances obtained from the first binding role in line 5. This means that for every single instance of `IfStat` that exists in the variable *instance*, the second binding role retrieves (in every iteration) the value of property `statements`, which is a collection of instances of the kind `Statement`, and stores that value into the variable *stat*, as in line 7. The instructions of the *do* block are re-executed on every instance that exists in variable *stat*. As a consequence, in every execution, a single statement given by *stat* is deleted from the collection of property `statements`.

Note that, since the implementation has two binding roles, the annotation *@role* is necessary to indicate which binding role has the target property. In this example, the target property is `statements`, which is contained in the variable *instance* of kind `IfStat` (as in line 5).

```
@action delete
@property statements
@role instance
pattern delete_one_statement
instance:IfStat,
stat:Statement
 from:instance.statements {
  do {
   // remove a statement from collection
   // of statements
   instance.statements.remove(stat);
  }
}
```

Listing 5.4: A CMO implementation for Example 4.4.3

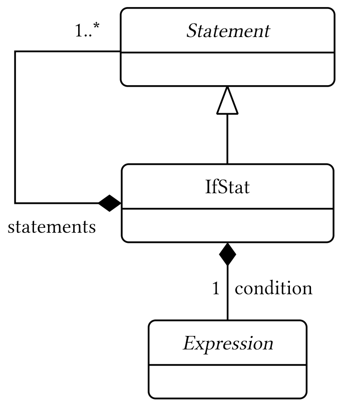

Figure 5.5: A fragment of the metamodel in Fig. 4.5

5.4.4 CMO implementation of Example 4.4.4

For this EMU concrete example, the implementation of the CMO in example 4.4.4, which targeted the property `method` of the entity `FOLogicExp`, is presented in Listing 5.5. Although the mutation example was too specific to replace the "select" method name with a same name, that is "select", and deliberately produce an equivalent mutant to demonstrate the benefit of using the preconditions of that concrete operator, this implementation considers all values of the property `method` and puts a constraint (using the keyword *guard* as in line 5) to prevent the production of useless mutants. The constraint eliminates all `FOLogicExp` instances that have "select" as a method name.

```
 1  @action replace
 2  @property method
 3  pattern replace_name_2_select
 4  instance:FOLogicExp
 5   guard: instance.method <> "select"
 6   do {
 7     // replace method name to select
 8     instance.method = "select";
 9   }
10  }
```

Listing 5.5: A CMO implementation of Example 4.4.4

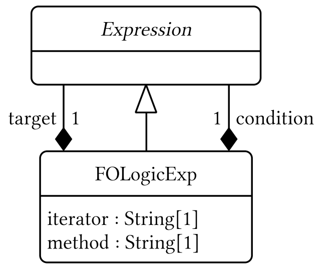

Figure 5.6: A fragment of the metamodel in Fig. 4.5

5.5 Chapter Summary

The chapter has presented EMU, a dedicated model mutation language and associated tooling that facilitates the implementation of concrete mutation operators (CMOs). The mutation engine of the EMU checks preconditions associated with mutation operators in order to reduce the risk of producing invalid or equivalent mutants. Furthermore, EMU allows users to query and filter model instances that are modified using EOL expressions. EMU facilitates the imposition of additional constraints and conditions on model instances on top of AMOs' preconditions, for flexibility and efficiency. The chapter also provided a few concrete examples of implementing mutation operators, explaining briefly the syntax and execution semantics of EMU over a simple metamodel example.

Chapter 6

Evaluation: Empirical Mutation Analysis

This chapter describes in detail and presents the results of an experimental study in which AMOs, the core contribution of this thesis presented in Chapter 4, were evaluated in order to determine whether they fulfilled the objectives put forward by this thesis. These are as follows: producing a set of metamodel-agnostic operators, useful mutants and low proportions of useless mutants. For the preparation and carrying out of the experiment, the experimental process for framing questions and linking them to hypothesis, the approaches to obtaining results, and the elements used in the experiments have followed the guidelines for good practice and principles described in [106].

The experiment was conducted against programs of two candidate model management languages: the Atlas Transformation Language (ATL) and the Epsilon Object Language (EOL), which are well known, mature languages in the MDE community. The former is used for model transformation and the latter for model query and model update. The selection of these two candidate languages was based on a number of reasons. To begin with, ATL is a hybrid language (supporting declarative and imperative language instructions), while EOL is a purely imperative language. As two different language paradigms, proving that AMOs can work for both provides initial evidence that the proposed operators (AMOs) can be indeed instantiated over different types of language. Also, since each language defines a wide range of distinct properties, there is an opportunity to investigate whether AMOs generate useful mutants by mutating such properties, which further provides evidence of the applicability and feasibility of AMOs.

In addition, ATL and EOL have metamodels (i.e. abstract syntaxes) expressed in EMF/Ecore. Using the available metamodels, the AMOs can be instantiated over modeling concepts defined in these metamodels, which can then be used to mutate EMF-based models that conform to the metamodels. The ATL metamodel is available as a resource within the Eclipse ATL modeling platform, whereas the EOL metamodel is made available by [107]. Since EMU provides an integration layer (as mentioned in Section 5.1) for EMF-based models, test developers can use the available resources of EMU or extend them to mutate models of ATL and EOL.

Furthermore, as candidate languages, ATL and EOL receive tooling support that facilitates the transformation of the textual syntaxes of ATL and EOL programs to EMF models (target model types that are possible to be mutated using EMU and its provided mutant integration layer, and vice versa. Such transformation tooling support (mainly text2model and model2text) is provided by the Eclipse ATL modeling platform[1] and by [107] for the ATL and EOL programs respectively.

Producing new metamodels and tooling support for text2model and model2text transformation from scratch for a given language is achievable. This thesis, however, is bounded by time constraints that preclude the production of a new metamodel and corresponding tool support. Therefore, using the available resources and tools (as available for ATL and EOL) to evaluate the proof of concept of AMOs and EMU is sufficient for this thesis.

The resources of the experiment are made available online. The ATL resources can be found at the repository `https://github.com/Fhma/MT_ATL/releases/tag/rv1` and the EOL ones can be found at `https://github.com/Fhma/MT_EOL/releases/tag/rv1`.

The remainder of this chapter is organized as follows. Section 6.1 presents the experiment questions, which address the objectives introduced in Section 3.3. Section 6.2 describes the overall experimental approach, and Section 6.3 examines the ATL and EOL candidate programs that were used to carry out the experiment. Section 6.5 provides an overview of the used mutation operators, with which models of the ATL and EOL programs were mutated. In Section 6.6, the results of the experiment are presented with a discussion and analysis, including several observations and findings revealed during the experiment.

6.1 Experiment Questions

In order to assess the main contribution of the thesis, namely the AMOs, and consequently evaluate its hypothesis, the described experiment was conducted in order to answer questions related to the core features of the AMOs, namely their status as metamodel agnostic and the usefulness of the mutation operators.

[1] `https://www.eclipse.org/atl/`

In Section 4.4, AMOs have been used to generate several concrete operators using an example of a metamodel. In this experiment, however, AMOs were used to generate CMOs for the ATL and EOL programming languages by examining their metamodels. The generated operators are evaluated for their worth and ability to produce different types of mutants such as live, killed etc.

The experiment was also conducted to evaluate whether CMOs produce useful mutants or not. Useful mutants (as defined in research objective 2a in Section 3.3) are mutants that help test developers to improve their test suite. One type of useful mutants is live mutants, denoting those not killed (i.e. when they produce exactly the same output as the original program) by any test case but which are detectable if the test developer includes stronger test cases. Another useful type of mutant is the non-trivially killed mutants, which are those detected by at least one test case but not all test cases. Such mutants are potentially good because they may lead to live mutants if evaluated against low-quality test inputs.

Finally, the experiment aims to evaluate whether AMOs preconditions prevent any useless mutants. Useless mutants (as already defined in research objective 2b) are those mutants that do not help test developers to improve or enhance a given test suite by any means. Such mutants are invalid, equivalent or trivially killed: invalid mutants break conformance to a target language; equivalent mutants are not detectable by any test case and are impossible to kill; trivially killed mutants are detected by all test cases and are easy to kill.

Accordingly, the experiment questions formulated to evaluate and validate the hypothesis of the thesis are as follows.

- **Q1:** can AMOs be instantiated over the ATL and EOL modeling concepts and do they generate concrete mutation operators for the used languages?
- **Q2:** do preconditions of AMOs prevent any invalid or equivalent mutants?
- **Q3:** do AMOs produce useful mutants (live, non-trivially killed mutants)?

6.2 Experimental Approach

The experimental process was divided into two stages: stage one and stage two. Stage one focused on producing mutants by executing mutation operators (discussed in Section 6.5) against a set of non-trivial candidate programs of ATL and EOL (discussed in Section 6.3). Stage two focused on executing valid mutants generated from stage one against a set of non-trivial test models.

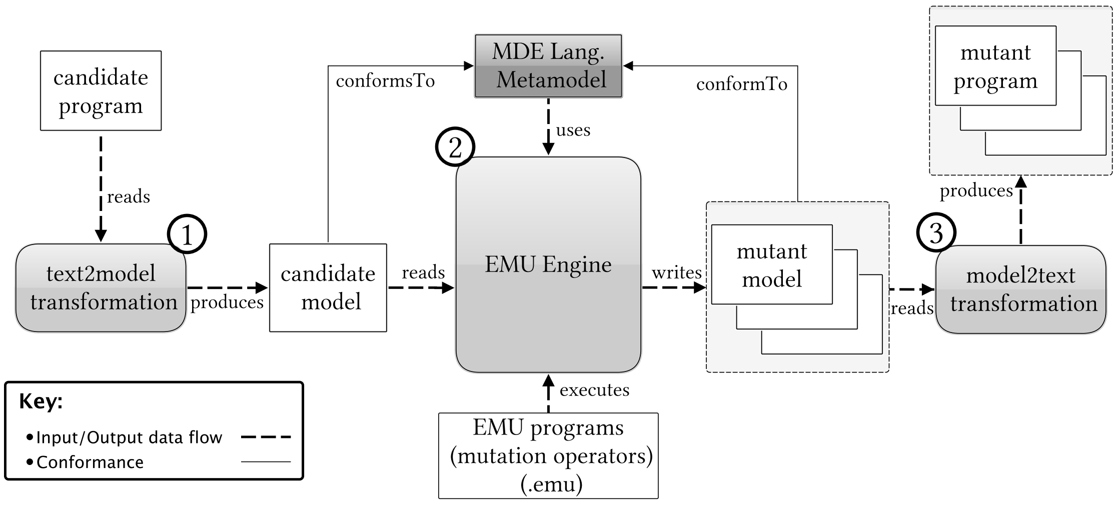

Figure 6.1: Stage one – mutants production process

In stage one, which is illustrated in Figure 6.1, a candidate program, which conformed to either ATL or EOL, was firstly parsed and transformed into an EMF model by a text2model transformation (as indicated by ① in Fig. 6.1). Next, the EMU engine (as shown in ② in Figure 6.1) read the candidate model with the help of an MDE metamodel and executed a set of mutation operators (written manually in EMU) against the candidate model. The execution had produced automatically a large number of mutant models that must conform to the same metamodel of the candidate model. Finally, the EMF mutant models were transformed back to text (i.e. textual syntax) using model2text transformation (as shown by ③ in Figure 6.1). The output artifacts (mutants) were either valid when mutants conformed to the used MDE metamodel or invalid when the conformance constraint was violated. Since invalid mutants were considered useless, they were not loaded onto stage two of the experiment; they were simply marked as invalid, since they would fail at any attempt towards execution.

In stage two of the experiment (as illustrated in Figure 6.2), which requires test developers to execute manually valid mutants of stage one on predefined test models, the target MDE language engine (as indicated by ①), read the candidate program and valid mutant and then executed them. As a consequence, two outputs were produced: one from executing the original program and the other from executing the mutant program. The outputs were then compared (as shown by ②) to determine killed and not-killed mutants. If the outputs were not the same (or not equal), then a mutant was marked as killed. If the outputs were equal, then the mutant was marked as not-killed. If there was no output for a mutant program, for example the MDE engine failed to load or execute the mutant against a test input, then the mutant program was marked as invalid.

The comparison process (as indicated by ② in Figure 6.2) received expected outputs, which were obtained from executing original programs on test models (as will be explained in Sect. 6.4), and actual outputs from the MDE language execution engine. Based on the type of files, the comparison process either used EMFCompare (as was discussed in Section 2.3.1.3) to compare the outputs in the case of models or files' content comparison in the case of textual artifacts.

The not-killed mutants obtained from stage two were either live or equivalent mutants. In order to distinguish between them, further analysis was required by investigating manually the code of each not-killed mutant to determine whether it was detectable by first locating the mutated part of the program and then back-tracking to find inputs and paths with which the execution can reach the mutated part. If detectable, then a new test model was to be built to kill the mutant. If it had been killed in the new test model, then the mutant was marked as live. If not, then the mutant was marked as equivalent. Techniques such as detecting equivalents using compiler optimization, time analysis, memory use, test coverage or any other side channels were not used because ATL and EOL lack the tooling support for such techniques and, therefore, these required supporting implementation and integration, which lies beyond the scope of this thesis.

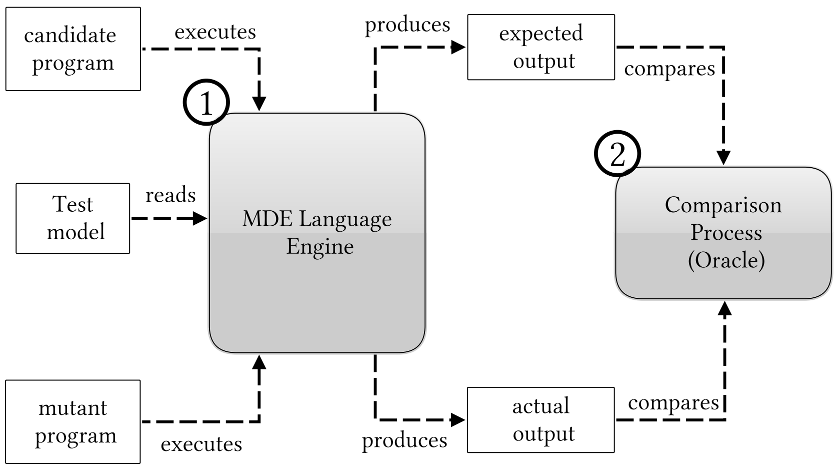

Figure 6.2: Stage two – mutants execution process

6.3 Candidate Programs

The described mutation testing experiment was performed on representative and non-trivial ATL and EOL programs that were collected from various resources to address research objective 3. In the case of ATL, programs were collected from a publicly available online repository of ATL programs at `www.eclipse.org/atl/atlTransformations/`. This public repository contains a set of various transformation programs used intensively within the MDE community (in research work such as [108, 109, 110, 101]). In addition, a number of ATL programs has been obtained from the GitHub online repository at `https://github.com` using the search functionality provided. In the case of EOL, a set of EOL programs and examples from the Epsilon platform source code repository, available at `www.eclipse.org/epsilon/download/#sourcecode`, as well as from the GitHub repository have also been collected.

Table 6.1: ATL metamodel coverage and model instances of candidate programs

Candidate program	Entities (out of 25)	Model instances
Book2Publication	11	23
Make2Ant	7	25
Table2TabularHTML	16	103
TabularHTML2XML	13	102
Table2SVGPieChart	19	254
Total coverage of ATL entities	20	
Total model instances	533	

In order to answer the experiment questions in 6.1, a selective process (presented in Algorithm 1) of ten candidate programs (five programs for each language) from a list of the collected programs was conducted based on (1) the language model coverage (i.e. the metamodel), and (2) the total number of language concepts. The intention was to instantiate as many AMOs as possible over diverse modeling concepts defined in the ATL or EOL metamodels and to investigate whether AMOs yield useful mutants, which provides evidence toward the feasibility of this thesis method and AMOs. Tables 6.1 and 6.2 gives the candidate programs including the metamodel coverage to each metamodel as well as the number of model instances.

Table 6.2: EOL metamodel coverage and model instances of candidate programs

Candidate program	Entities (out of 86)	Model instances
ShortestPath	29	147
Formatting	36	838
EcoreHelper	38	1248
ECoreUtil	34	1591
ECore2GMF	37	1370
Total coverage of EOL entities	47	
Total model instances	5194	

Regarding the execution of the selected candidate programs, the ATL programs were executed using the ATL transformation engine version 3.6.0, which was released in May 2015, whereas the EOL programs were executed using the Epsilon platform version 1.3.0, which was released in February 2016.

Algorithm 1: Candidate programs selection

```
let $P$ be a list of files that exist in $folder_{input}$ and sorted in descending order based on metamodel
  coverage and total model instances (in case of ties)
let $R$ be an empty set for storing selected programs
let $coverage_{best}$ be the coverage of the first program available in list P (i.e. $P_0$)
while there is a slot for one more program do
    if the end of $P$ is reached then
        traverse again list $P$ and add programs as they appear to set $R$
        continue
    obtain the coverage of current program $P_i$
    if current coverage of program $P_i$ is greater than $coverage_{best}$ then
        update the best coverage and add current program $P_i$ to set $R$
return $R$

function GetCoverage($p$)
    let $E$ be a set of all entities defined in a metamodel
    let $E_{exsit}$ be an empty set for all entities of this program $p$
    foreach $instances \in p$ do
        get all entities of instances and add them to the set $E_{exsit}$
    return $|E_{exsit}|/|E|$
end function
```

6.4 Test Models

The test models that were used to test the candidate and mutated programs in stage two (① in Figure. 6.2) were generated semi-automatically using the Epsilon Model Generator (EMG) [83]. EMG accepts a metamodel, since the test models are themselves models, and a generation code, which queries the input metamodel and creates model instances of entities and properties. The overall process of generating test models is illustrated in Figure 6.3. The metamodels and the generation script that were used to construct the test models are made available in Appendix B.

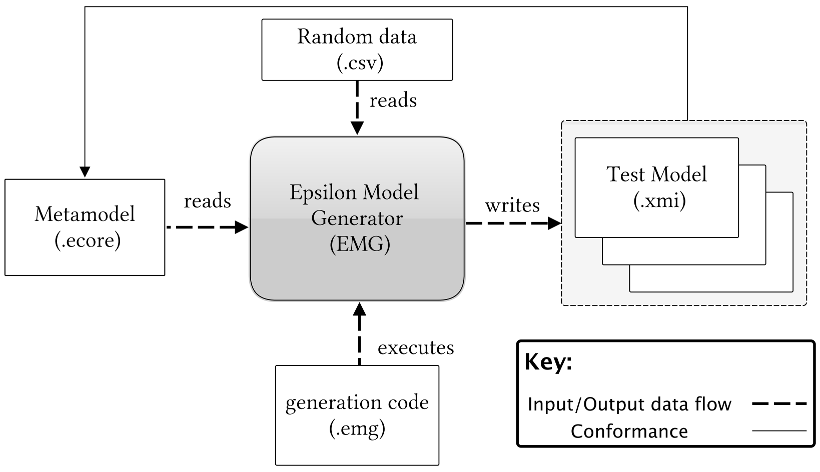

Figure 6.3: Test models generation process

The generation process that was followed in this experiment produces good models by imposing a number of criteria to reduce the likelihood of threats to the validity of the evaluation. The first criterion was that any generated test model must conform to the used metamodels because the conformance of the models is an essential property in MDE. Also, non-conforming test models are likely to be rejected by the underlying modeling platform (Ecore and its underlying implementation). The conformance of test models is performed using EOL statements and expressions.

The second criterion was that the proportion of entities in test models should be consistent with the proportion of entities likely to occur in real test inputs. For example, a real test model instance of a *Table* metamodel with *rows* and *cells* would likely consist of more *cells* than *rows*, and more *rows* than *tables* instances. This was achieved by giving any instantiated entity in the metamodel a (manually set) weight value; the probability of a generated instance being of a given entity is then proportional to the entity's weight.

The third criterion was that the size of the input models should be consistent with the diversity of realistic test inputs. The size of a model was specified as a configuration parameter for the model generator that was initially set to a minimum number of required instances of entities for a valid model and then multiplied by 2^x, where x was selected randomly from the set $\{1, 2, 3, 4, 5\}$. The required entities were given priority to be constructed first, as they were required by other entities and properties. The properties of each entity were initialized with random values obtained from the currently generated model, for example, in the case of obtaining values for end association properties, and from a list of typed values given to the model generator as an input, such as in the case of obtaining values for the properties of primitive types.

The last criterion was that any generated testing model must successfully execute against the candidate programs because outputs of their execution would be used as oracles when compared with the actual outputs of the mutants of candidate programs. If the output of a candidate program was identical to the output of its mutant, then the mutant was marked as not killed. If, however, the candidate program produced an output model that was different than its mutant, then the mutant was marked as killed. The verdict of whether a mutant was detected (killed) or not detected (not killed) was determined by a comparison process of expected outputs (obtained from executing test models against original programs) and actual outputs (obtained from executing test models against mutants) as indicated by ② in Figure 6.2.

For each input metamodel of candidate programs (that is metamodels of test models), a total of 20 test models were generated semi-automatically by considering the metamodels of test models and the criteria above; and the process in Figure 6.3. The coverage of ATL and EOL candidate programs were not considered because of tooling support limitation.

6.5 Concrete Mutation Operators

Stage one of this experiment focused on generating all possible mutants of the ATL and EOL candidate programs by applying a set of concrete mutation operators (CMO) based on ATL and EOL Ecore-based metamodels. In the following subsections, a description of these CMOs is given. First, the process with which the CMOs were generated is presented in Section 6.5.1. Next, a list of overlap mutation operators between ATL and EOL is given in Section 6.5.2, followed by a list of specific mutation operators for ATL and EOL in Section 6.5.3 and Section 6.5.4 respectively.

Some of the presented mutation operators have cross-references to their CMO implementations, which are made available in Appendix A. A complete implementation of CMOs can be found online, as mentioned in the introduction to this chapter.

6.5.1 Systematic mutation operator definition process

The process used for generating CMOs was systematic by traversing all entities and properties constructed in the ATL and EOL metamodels and manually applying addition, deletion and replacement mutation actions upon the properties of entities. This systematic process of generating CMOs is presented in Algorithm 2.

Although that this systematic process was used as part of this evaluation, the process is not considered as part of the method of the thesis for defining CMOs for a given language. This is because it is entirely up to test developers to decide whether to use the systematic process or not based on their preferences. For example, a test develop may want to investigate the quality of her/his test suite with respect to a selective set of properties (or language concepts) from MDE language, and hence, instantiate AMOs only upon the selected properties. Another test developer may want to use the process above to instantiate AMOs upon all properties defined in a metamodel and generate a large set of concrete mutation operators, which is a process that is reasonable for new languages according to Ammann and Offutt [12]. For this reason, the AMOs are made independent from the process that is used to instantiate CMOs on language properties for the thesis but was used here in the experiment.

The systematic approach took into account a general point to ensure that any produced mutation operator yielded, where possible, valid mutants by avoiding the construction of

Algorithm 2: Systematic process for generating concrete mutations for a metamodel

```
let $E$ be a set of all entities defined in a metamodel
foreach entity exists in $E$ do
    let $MO$ be an empty set for collecting all mutation operators of this entity
    inherit all mutation operators of super-entities of entity and add them to $MO$
    foreach property defined in entity do
        define mutation operators and add them to $MO$ according to the following
        if single-valued properties then
            /* Instantiate AMO-Single and define */
            operator that adds a compatible value to property
            operator that replaces a current value of property with a new compatible one
            operator that deletes the value of property
        else
            /* Instantiate AMO-Multiple and define */
            operator that adds a simple and not complex value
            operator that replaces a compatible sibling. For instance, if entities $E_a$ and $E_b$ both
              extend entity $E_c$, then an instance value of $E_a$ is replaced with a new instance value
              of $E_b$
            operator that deletes a selected value from a collection of values
```

complex addition and replacement operators. This means that when a mutation requires a number of elements to be constructed, then it is considered a complex mutation in this thesis and was avoided because of a number of issues.

One issue is that mutation values for association properties can require instances of entities, which may themselves require properties and associations that must be initialized first with some values. In general, mutation literal values are easier to obtain for primitive type properties than for associations. Values for associations, however, become complicated when the creation of their values depends further on new values and so on.

Another issue is what level of element creation with values should be enough for an "appropriate" addition or replacement mutation for associations. Such issue requires intensive investigation and lies outside the scope of this thesis, which considers simple addition and replacement operators, mirrored from mutation operators of languages found in the relevant literature (as discussed in Section 2.2.3). For example, *continue*, *break* etc. statements are added to looping blocks and the value of a modeling concept is replaced with another from the same category as the original without complex change in the model.

The final issue related to complex addition and replacement mutations is that allowing to have mutation values of multiple elements for a single mutation may produce vulnerable mutants that are easy to be detected and killed. According to the competent programmer hypothesis of mutation analysis [64], which states that an experienced programmer is more likely to produce a program that is correct or almost correct, errors made by programmers are small and, consequently, the norm for a mutation is to introduce a single mutation. Having a mutation, however, that introduces multiple mutation values for properties drives the mutation analysis away from its norm and makes the hypothesis no longer applicable.

Before concluding this subsection, it is worth mentioning two issues. The first is related to a large set of mutation operators. For new metamodels, such as EOL and ATL (although there have been attempts to design mutation operators for many ATL language concepts mentioned in Section 2.3.2), the aforementioned systematic process generated a large set of mutation operators. For new languages, this process is reasonable [12], as the process ensures that no modeling feature is overlooked or missed during the process of operator generation. This is an issue that could have been encountered using other operator design approaches, such as reproducing mutations from similar task-specific MDE languages.

The second issue is related to the challenging process of implementing CMOs while inspecting the source of programs or metamodels. The proposed mutation operator design approach, AMOs and EMU, do not directly support the inspection of source code and metamodels. However, the Epsilon platform and Eclipse plugin development, which EMU relies on, offer a number of plugins that makes models (models of programs) and metamodels query an easier task compared with looking directly at source code of programs and metamodels. This is because that models and metamodels are structured according to modeling concepts that are easier to navigate and query.

6.5.2 ATL and EOL overlap mutation operators

Since ATL and EOL have a small number of language concepts that are similar to general programming language concepts, many CMOs overlap with the mutation operators of those general programming languages mentioned in Section 2.2.3. The ATL and EOL overlap mutation operators are related to:

1. Binary comparison replacement: this mutation has been implemented for many general-purpose and domain-specific programming languages. The objective of such operator is to change binary comparison operators, such as $>$, $\geq$ etc., to different ones from the same category. For ATL and EOL, the Boolean modeling concepts with which this mutation replacement can be applied were found in the following:
   - `ATL::IfStat.condition` (available at A.1)
   - `ATL::InPattern.filter`
   - `EOL::FOLMethodCallExpression.conditions`
   - `EOL::ExpressionOrStatementBlock.condition`
   - `EOL::IfStatement.condition`

- `EOL::WhileStatement.condition`

2. Binary logical replacement: changing the logic of a Boolean expression is a common mutation operator for a number of languages. The operator objective is to replace logical operators, such as AND, OR, XOR etc., with others. The ATL and EOL modeling concepts with which this mutation operator can be applied were the same ones listed for the binary comparison replacement operator in Enumeration 1.
3. Variable name replacement: another common mutation operator found in the literature is the replacement of a variable's name, which is normally a String type, with another name. The purpose is to mimic errors of using wrong variable names by language users. For ATL and EOL, the following modeling concepts allowed the variable name replacement operator.
   - `ATL-OCL::VariableDeclaration.varName`
   - `EOL::VariableDeclarationExpression.name`
4. Variable type replacement: this mutation has also been implemented for a number of programming languages. The purpose is to imitate the error of misusing variable types that are used to represent the value. For ATL and EOL, the modeling language concepts against which the variable type (for either primitive model) replacement can be applied were:
   - `ATL-OCL::VariableDeclaration.type` (available at A.2)
   - `ATL-OCL::CollectionType.elementType`
   - `ATL-OCL::OclType.name`
   - `EOL::Expression.resolvedType` (available at A.10)
   - `EOL::NewExpression.typeName`
5. Operation/function return type replacement: another common mutation operator that is related to operations/functions with returning value is the wrong specification of the type that is actually returned by an operation. ATL and EOL both support operations using the following language concepts with which this operator can be implemented.
   - `ATL-OCL::Operation.returnType`
   - `EOL::OperationDefinition.returnType`
6. Operation/function name replacement: inaccurate call of an operation name is a typical mutation operator for a number of programming languages. This mutation introduces such an error by replacing the name of a method with another name. For the ATL and EOL languages, the following concepts allow this operator to be applied:
   - `ATL-OCL::Operation.name`
   - `EOL::MethodCallExpression.method`
   - `EOL::FOLMethodCallExpression.method`
7. Operation/function parameters/arguments deletion: the action of this operator is to

delete one of the parameters defined in an operation or delete one of the arguments passed to an operation. The purpose is to mimic the error of mismatching the number of parameters of an operation with the caller arguments. The modeling language concepts of ATL and EOL with which this mutation operator can be implemented were found in:

- `ATL-OCL::Operation.parameters` (available at A.3)
- `ATL::CalledRule.parameters`
- `EOL::OperationDefinition.parameters` (available at A.9)
- `EOL::MethodCallExpression.arguments`

8. Statement deletion: this mutation operator has been found in a number of mutation analysis studies. Its purpose is to ensure that the deleted statement has a direct effect on the outputs of a target program. For ATL and EOL, the following modeling concepts were used to implement this mutation operator:
   - `ATL::ActionBlock.statements`
   - `ATL::ForStat.statements`
   - `ATL::IfStat.thenStatements`
   - `ATL::IfStat.elseStatements`
   - `EOL::Block.statements`

6.5.3 ATL concrete mutation operators

As mentioned previously, ATL is a model transformation language with which output models are created based on elements defined in input models. This is achieved by including input pattern matching elements against the input models to produce output pattern elements for output models. This section covers a list of CMOs for model transformation modeling concepts of ATL, and those modeling concepts of OCL that are directly referred to from within the ATL metamodel (as illustrated in Figures 6.4 and 6.5

In addition, since there are some mutation operators already defined for ATL in [100, 101, 102, 103], if a mutation operator exists for a particular language concept of ATL that is also covered in this experiment and can be instantiated with AMOs, a reference to the existing mutation operator in those research work is given. Furthermore, some of the generic mutation operators by Mottu et al. [96] for model transformation is considered while defining CMOs for ATL, although such operators require investigation to find modeling concepts to which they can be applied. However, there were mutation operators exist for model transformation and ATL in the literature that were not considered here as this experiment follows a systematic approach (mentioned in Sect. 6.5.1) to define simple mutations to address critical points related to competent programmer hypothesis and to avoid generating weak mutation operators. Furthermore, the list of mutation operators for ATL, which is about to be given below, only related to ATL concepts; and OCL concepts that are directly referenced to from ATL metamodel.

1. `Module.elements`: the modification for this property included changing module elements (for example, matching rule, lazy matching rule, called rule and helper)

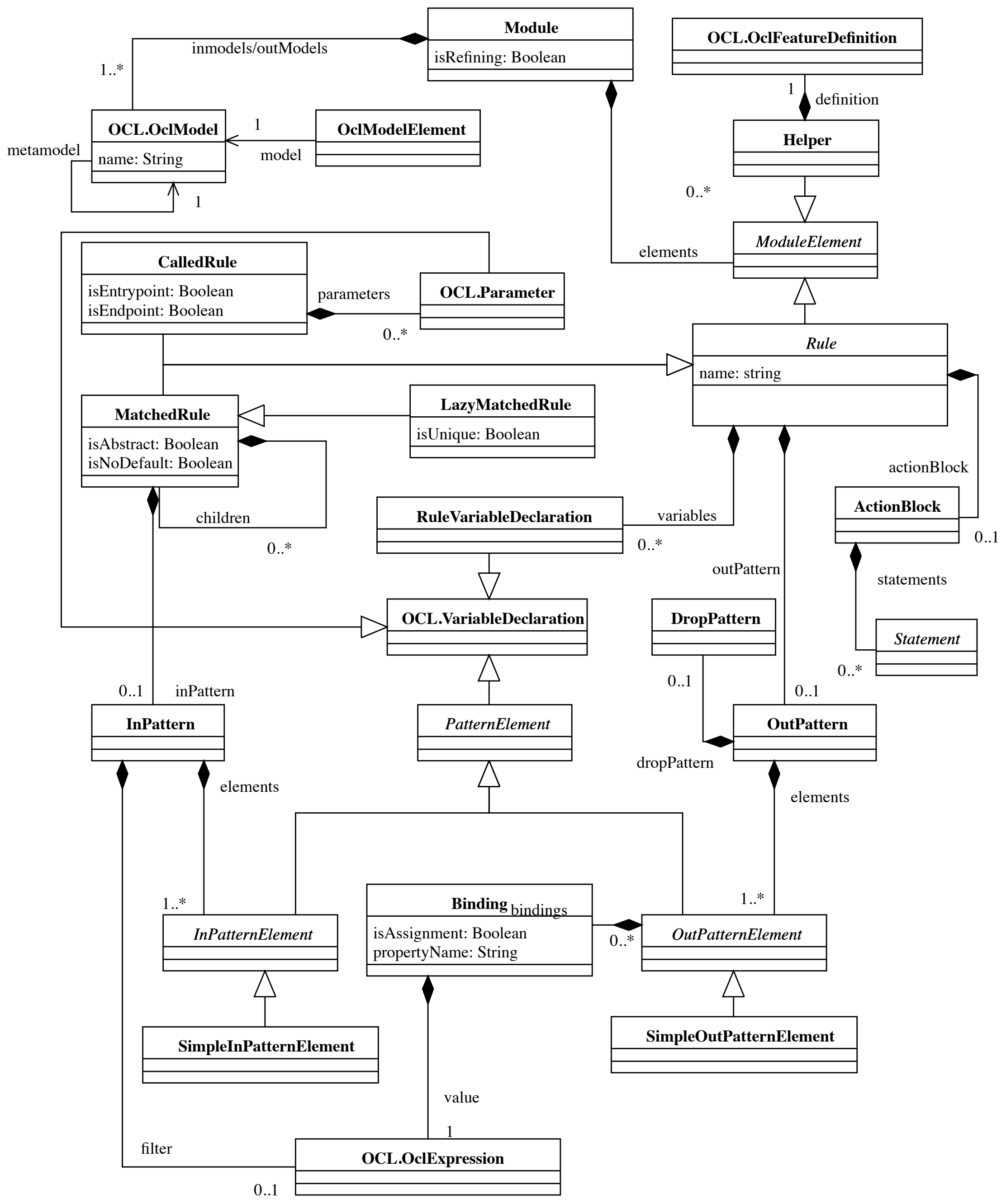

Figure 6.4: ATL Metamodel – ATL module concept

through replacement and deletion mutation actions. The addition action was not examined, as it required complex mutations such as specifying a rule with input/output patterns. For the replacement action, the replacement of a matching rule, which is a standard rule that is executed once for every match in the input model, with a lazy matching rule, which is executed upon request from within other rules, such as matching rules or called rules, and vice versa was implemented. This replacement operator has also been mentioned in [100]. This was to mimic the error of misusing the keyword *lazy* at the front of a rule declaration. For the deletion action, a simple deletion of elements of the ATL module had taken place to ensure that an ATL module element had an effect on the outputs of a target program. This was an operator also proposed in [102] for deleting rules and helpers and in [103] for deleting rules. The CMOs of this property are as follows:

- `CMO-multiple-REP(Module elements)`

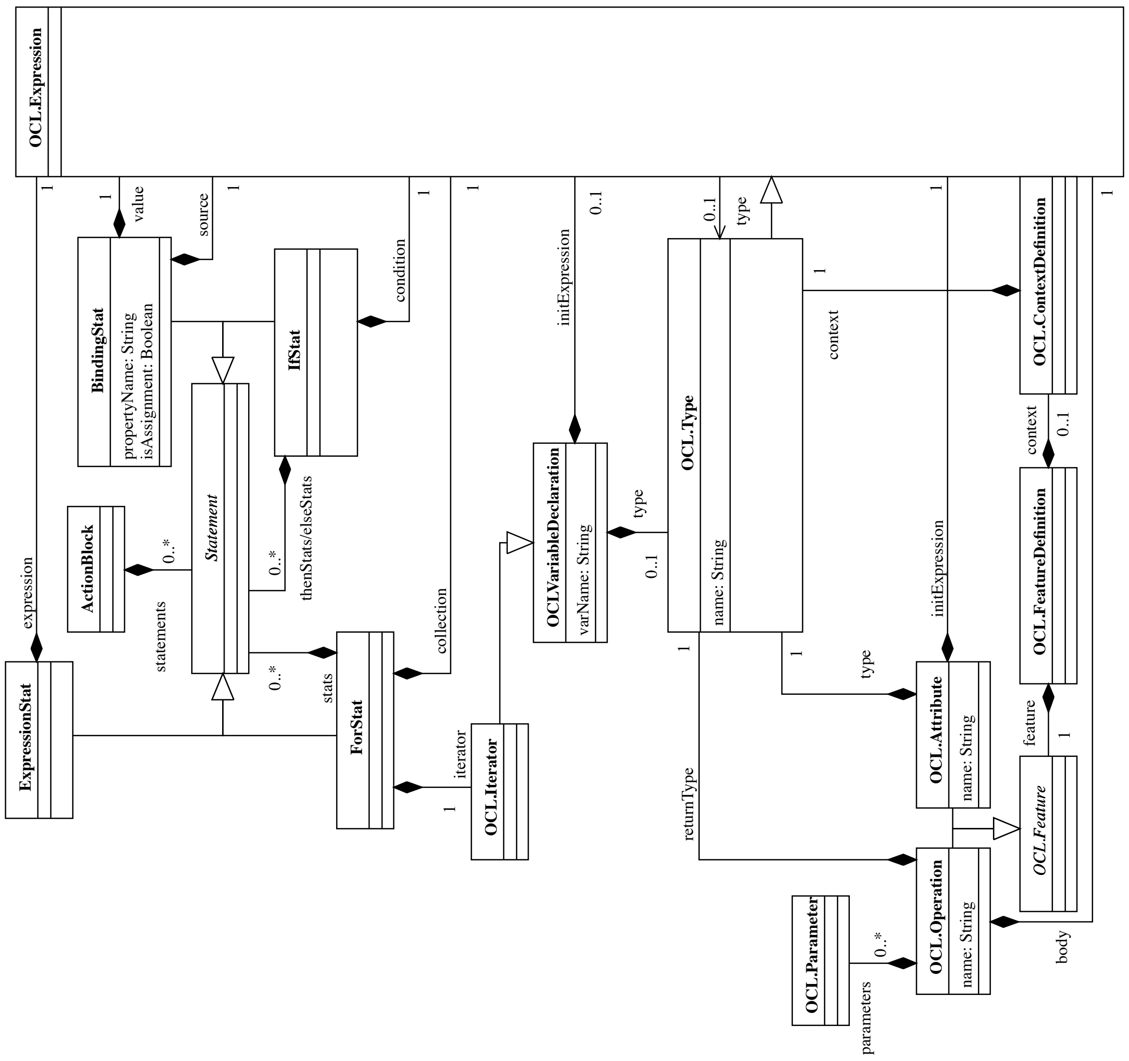

FIGURE 6.5: ATL Metamodel – ATL–OCL related concepts

- **`CMO-multiple-DEL(Module elements)`**

2. `Module.inmodels/outmodels`: the mutations designed for these modeling concepts manipulated the input and output models of a transformation through the deletion action. The purpose was to imitate the error of forgetting input or output models of the ATL module. The addition and replacement operators were not considered. The former would have no impact on the mutated programs if the added new model had not been referred to from the mutated program. The latter would simply have no siblings available to be replaced with (that is, another type of model). The CMOs of these deletion operators are:
   - **`CMO-multiple-DEL(Module inModels)`**
   - **`CMO-multiple-DEL(Module outModels)`**
3. `Module.refining`: the mutation for this property was to change the execution behavior of the ATL engine from creation mode to refining mode, in which input models were updated in-place. The addition and deletion actions were not considered because this modeling concept was a Boolean primitive type and always had a default value equal to false. Hence, it did not accept any new values and its value was never unset. Therefore, it only accepted replacement action. The CMO of this property is **`CMO-single-REP(Module isRefining)`**.
4. `Rule.actionBlock`: this property value, which contains declarative instructions to the ATL engine, was modified by the deletion mutation. The purpose of this operator was to determine whether a block had any effect on the output models. The addition and replacement actions were not applied, as they required complex mutations such as patterns for input and output matching and mapping elements. The CMO of this property is **`CMO-single-DEL(Rule actionBlock)`**.
5. `Rule.name`: this concept was modified by a replacement with another name. The CMO is **`CMO-single-REP(Rule name)`**.
6. `Rule.outPattern`: this modeling concept was modified by a deletion action, as it is allowed by the ATL metamodel. The purpose was to mimic the error of not specifying the output pattern to generate output models. The addition and replacement actions were not considered as they required complex mutations to be constructed, such as model variables, types and their usage to produce actual output models. The CMO for this property is **`CMO-single-DEL(Rule outPattern)`**.
7. `Rule.variables`: the value of this modeling concept, which was a list of declared variables of a called rule, was modified by a deletion action in order to mimic the error of failing to include variables in the called rule block of statements. The addition and replacement actions were not considered, as they required complex mutation values for variable names, types and usage in binding statements to make a difference to the output models. The CMO of this language concept is **`CMO-multiple-DEL(Rule variables)`**.

8. `OutPattern.elements`: an out-pattern may have a number of elements that are variable declarations used to create model instances in the output models. The modification for this language concept was made by successive deletion actions against elements from a list of elements of out-patterns; such an action was also found in [96, 101, 102]. The addition and replacement actions were not implemented, as they required complex mutations for model variables, types and their usage to produce actual output elements in output models, although such mutations were proposed in [96, 101, 102]. The CMO of this concept is **`CMO-multiple-DEL(OutPattern elements)`**.
9. `OutPatternElement.bindings`: an out-pattern element may have a number of binding statements that append information onto output models. The mutation for this concept, which was also proposed in [96, 100, 101, 102, 103], deleted one binding statement at a time in order to ensure that bindings had an actual effect on output models. The addition and replacement actions were not considered for this property, as they required complex operator definitions and obtained values for concepts such as names of targeted output model elements and values for each output element. Addition operators alone were proposed in [96, 100, 101, 102]. The deletion CMO of this modeling property is **`CMO-multiple-DEL(OutPatternElement bindings)`**.
10. `CalledRule.endpoint/entrypoint`: a called rule must have either entrypoint or endpoint keywords. If entrypoint is used at the front of a rule, the engine executes the rule before any other rules, whereas if endpoint is the rule it is executed at the very end of the transformation execution. The replacement modifications of these flag values manipulated the execution behavior and mimicked the misuse of the keywords. The addition and deletion mutation actions were not considered, as the target modeling properties were Boolean literals and had default values. The CMOs of these properties are:
    - **`CMO-single-REP(CalledRule isEntrypoint)`**
    - **`CMO-single-REP(CalledRule isEndpoint)`**
11. `MatchedRule.children`: rules inheritance is a feature provided by ATL. This modeling property can be changed by replacement and deletion. The addition action was not considered, as it required complex mutation values for a new matching rule along with input/output patterns and elements. For the replacement action, a child rule was replaced with another rule. This action was to mimic the error of extending a wrong super (father) rule from a child rule perspective. For deletion, a simple delete of a relation for the super-rule was applied, which was also proposed in [102].
    - **`CMO-multiple-REP(MatchedRule children)`**
    - **`CMO-multiple-DEL(MatchedRule children)`**

12. `MatchedRule.inPattern`: a matching rule may have an input pattern matching block that contains a number of elements for navigating input models. The deletion action deleted an input pattern in its entirety. This was to mimic the error of forgetting the input pattern by the language user. The addition and replacement operators, were not applied as they required complex mutation operators. The CMO for this property is **`CMO-single-DEL(MatchedRule inPattern)`**.
13. `InPattern.elements`: the modification for this property was designed to delete elements of an input pattern, which are variable declarations used to navigate input models. This action imitated the error of forgetting an element that contributes to output models, which was also a mutation proposed in [101, 102]. While addition mutation operators have been proposed for this modeling concept in [101, 102], the process for defining mutation operators in this experiment did not consider addition and replacement actions because they required complex mutation values for variable names, types and their usage into pattern matching against input models. The deletion CMO for this property is **`CMO-multiple-DEL(InPattern elements)`**.
14. `InPattern.filter`: an input pattern may have a Boolean expression (mainly binary OCL operator expressions, for example, arithmetic operators, comparison operators and logical operators) that filters the instances obtained from an input model. This modeling property was modified by a deletion action that removed filters from input patterns in order to mimic the error of forgetting to include filters by ATL language users. The mutation was also proposed in [100, 101, 102]. The replacement CMO has been previously given in Enumerations 1 and 2. Although the addition action mutation has been previously proposed in [100, 101], the addition operator was not considered here as it required complex mutations, such as names of variables, literal values and binary operators to go between them. The deletion CMO is **`CMO-single-DEL(InPattern filter)`**.
15. `MatchedRule.isAbstract`: the modification for this property was a replacement of the Boolean status of a matching rule from true to false. This was to emulate the error of missing the keyword *abstract* at the front of a matching rule. Since this property concept is a Boolean primitive type and always has a default value, the addition and deletion actions were not applied. The CMO of this property is **`CMO-single-REP(MatchedRule isAbstract)`**.
16. `LazyMatchedRule.unique`: a unique lazy rule is different from a lazy rule, where outputs generated by a unique rule for a given match of input instances are never overridden for that particular match. If a unique lazy rule is used again for the match, the same created output elements are used every time. This property was changed by a replacement action against its Boolean value from true (meaning unique lazy rule) to false (meaning standard lazy rule) and vice versa. The CMO for this property is **`CMO-single-REP(LazyMatchedRule isUnique)`**. The addition and

deletion operators were not generated because this property was a Boolean property and, as such, always set. Therefore, addition and deletion were not possible.

17. `Attribute.name`: this property is used for representing helper attributes in ATL. It was modified by a replacement action to mimic the error of calling attributes (just like helper functions but with no parameters) with wrong names. This mutation was also proposed in [102]. The addition and deletion mutations are not considered because this property is compulsory (lower and upper bounds equal to one) and it must have a value. The replacement CMO is **`CMO-single-REP(Attribute name)`**.
18. `Attribute.type`: the property was modified by a replacement of its value (which is an ATL type) with another type defined in the ATL metamodel. As this property was characterised as mandatory in the ATL metamodel, it must have a value when it is first constructed and only replacement mutation is allowed. Attempts at performing addition or deletion mutations would have simply produced invalid mutations and were, therefore, avoided. The CMO of this property is **`CMO-single-REP(Attribute type)`**.
19. `Binding.propertyName`: this language concept was modified by a replacement of its value with another name, which mapped to a property defined in the output metamodel in matching rules. This operator was used against bindings of matching rules. The same action of replacement mutation for the property was also proposed in [101, 102]. Since this property is mandatory, it always had a value. Thus, addition and deletion actions were not permitted. The replacement CMO is **`CMO-single-REP(Binding propertyName)`**.
20. `BindingStat.propertyName`: this property was modified by replacement of its name, just like the one above but for binding statements of called rules. This mutation was also proposed in [96, 101, 102]. The addition and deletion actions were not applied since this concept was characterized as mandatory in the ATL metamodel. The replacement CMO is **`CMO-single-REP(BindingStat propertyName)`**.
21. `BindingStat.source`: this modeling concept was modified by a replacement of its value, which is a source name, with another source. The same mutation was also proposed in [103]. This emulated the navigation to a target property from the wrong source of property. The CMO of this property is **`CMO-single-REP(BindingStat source)`**.
22. `OclFeatureDefinition.context`: the change of this language concept, which is applied to helper functions and attributes, involved replacement and deletion actions. The addition action was not considered, as it required complex mutations. For replacement, the mutation replaced a context (basically a type) of a model element or entity with another. The same action for the mutation was also proposed in [102]. The replacement value was obtained from input and/or output metamodels, including ATL primitive types and collections. The objective was to mimic the

error of using incorrect types for context definitions. For deletion, a simple deletion of the context was applied. The CMOs of replacement and deletion operators, which were also proposed in [102], are:

- `CMO-single-REP(OclFeatureDefinition context)`
- `CMO-single-DEL(OclFeatureDefinition context)`

23. `OclFeatureDefinition.feature`: this modeling concept was modified by replacement, in which an OCL operation was changed into an OCL attribute (attributes are just like operations but without parameters) and vice versa. This was to imitate the misuse of an appropriate feature definition of helper functions or attributes. The addition and deletion actions were not applied, as the property feature of a feature definition entity is characterized as a compulsory property in the ATL metamodel. Hence, its value only accepted replacement. The CMO of this property is `CMO-single-DEL(OclFeatureDefinition feature)`.
24. `OclModel.name`: this property was modified by replacement of a model name with another. This was to mimic the misuse of incorrect model name. The replacement CMO of this property is `CMO-single-REP(OclModel name)`.
25. `VariableDeclaration.initExpression`: this property was changed only by a deletion of its value that is an initial expression of an OCL feature definition. The objective was to determine whether the initial value affected the output model. The addition and replacement actions require literal expressions and, therefore, were not performed. The replacement operator is `CMO-single-DEL(VariableDeclaration initExpression)`.
26. `VariableDeclaration.type`: this language concept was modified by all three mutation actions (addition, deletion and replacement), all of which modified the associated type of variables. The objective was to mimic the wrong declaration of variables by the user of the ATL language. For addition, a type from input/output metamodels was assigned to a variable declaration. For the deletion mutation, a simple deletion of the type was performed, which is a mutation also mentioned in [102]. Both addition and deletion CMOs are given below. The replacement operator was already discussed in Enumeration 4.
    - `CMO-single-ADD(VariableDeclaration type)`
    - `CMO-single-DEL(VariableDeclaration type)`

6.5.4 EOL concrete mutation operators

EOL is a pure imperative language that is mainly used to query, create and modify models. The query functionality is facilitated by a set of methods similar to the query methods of OCL, such as *select*, *collect* etc., whereas the modification part is enabled by a set of controlling and looping statements, such as if,for,$while$ etc. For this experiment, the metamodel of EOL used to generate concrete mutation operators is oragnized into Figures 6.6, 6.7, 6.8 and 6.9.

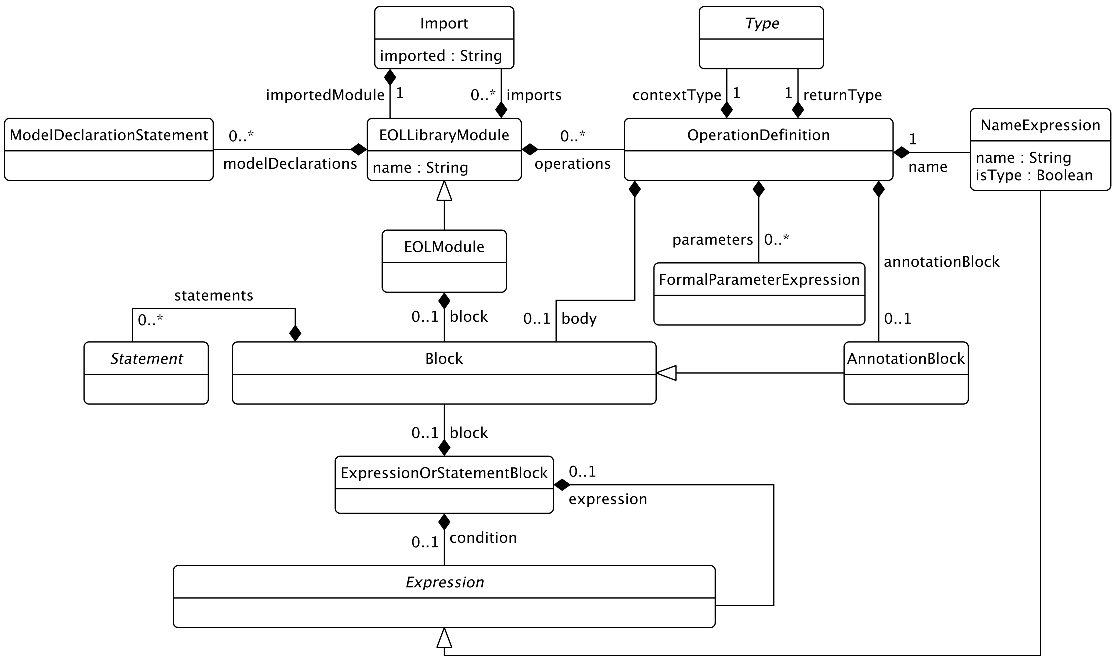

FIGURE 6.6: EOL Metamodel – EOL module

1. `AnnotationStatement.name`: executable EOL annotations, which are used for operations/functions, are like variables with a name and expression. The names of these annotations were modified by a replacement action with other names to emulate the error of using wrong names within the operation block. Since this language concept is mandatory, it must have a value and, hence, the addition and deletion actions were not applied. The CMO is **`CMO-single-REP(AnnotationStatement name)`**.
2. `AssignmentStatement.rhs`: this property was modified by a replacement action when the right-hand-side of an assignment statement was a Boolean. The binary operators (for example, $>$, $\geq$ etc.) and logical operators (such as *and*, *xor* etc.) were replaced with others. Since this property was made mandatory in the used EOL metamodel, the addition and deletion actions were not considered, as they would produce invalid mutants. The CMO of this property is **`CMO-single-REP(AssignmentStatement rhs)`**.
3. `Block.statements`: this modeling concept had been mutated by three modifications: addition, deletion and replacement. The deletion mutation is already given in Enumeration 8. The addition mutation added a simple statement that did not require complex mutation to a block of statements – adding a looping control expression, such as *continue*, *break*, *breakAll* to a looping structure such as *while*

and *for*. This was to mimic the error of misusing controlling statements, including changing their behavior. The replacement mutation involved replacing the controlling structure of *while* and *if* with each other, replacing *continue*, *break* and *breakAll* with one another, and replacing return statements with simple statements (that is, removing a *return* keyword from an expression and leaving the expression unaffected). These actions were used to investigate the behavior of a block of statements and their actual effect on output models. All the mentioned operators for this property were mirrored from operators of traditional programming languages.

- **`CMO-multiple-ADD(Block statements)`**
- **`CMO-multiple-REP(Block statements)`**

4. `CollectionExpression.contents`: this concept serves to model elements of a declared collection variable. The value of the property was modified by deletion, in which one element was removed from the elements. Since the contents are literal expressions, the addition and replacement mutation actions that required literal values were not considered. The deletion CMO is **`CMO-multiple-DEL( CollectionExpression contents)`**.
5. `EOLLibraryModule.imports`: this property value was modified by sequentially deleting import statements from a list of statements. Since this language concept was literal, the addition and replacement actions were not applied because they would involve an infinite space of values. The deletion CMO of this property is **`CMO-multiple-DEL(EOLLibraryModule imports)`**.
6. `EOLLibraryModule.modelDeclarations`: the mutation for this language property was to delete the model declarations of an EOL module one at a time. The addition and replacement actions were not considered, as they require literal string values for model names from an infinite space of values. The CMO is **`CMO-multiple-DEL( EOLLibraryModule modelDeclarations)`**.
7. `EOLLibraryModule.operations`: the mutation for this property was to delete one operation from a collection of operations defined in an EOL module. The addition and replacement of operations were not considered, as they required complex mutation values for the operation block, for example, logic expression and statements. The deletion CMO is **`CMO-multiple-DEL(EOLLibraryModule operations)`**.
8. `EOLModule.block`: this property was modified by deleting a block of an EOL module. The objective of this mutation was to determine whether a block had a direct effect on output artifacts. The addition and replacement actions were not defined, as they required complex mutation values for a complete block that had an impact on output artifacts. The deletion CMO is **`CMO-single-DEL(EOLModule block)`**.

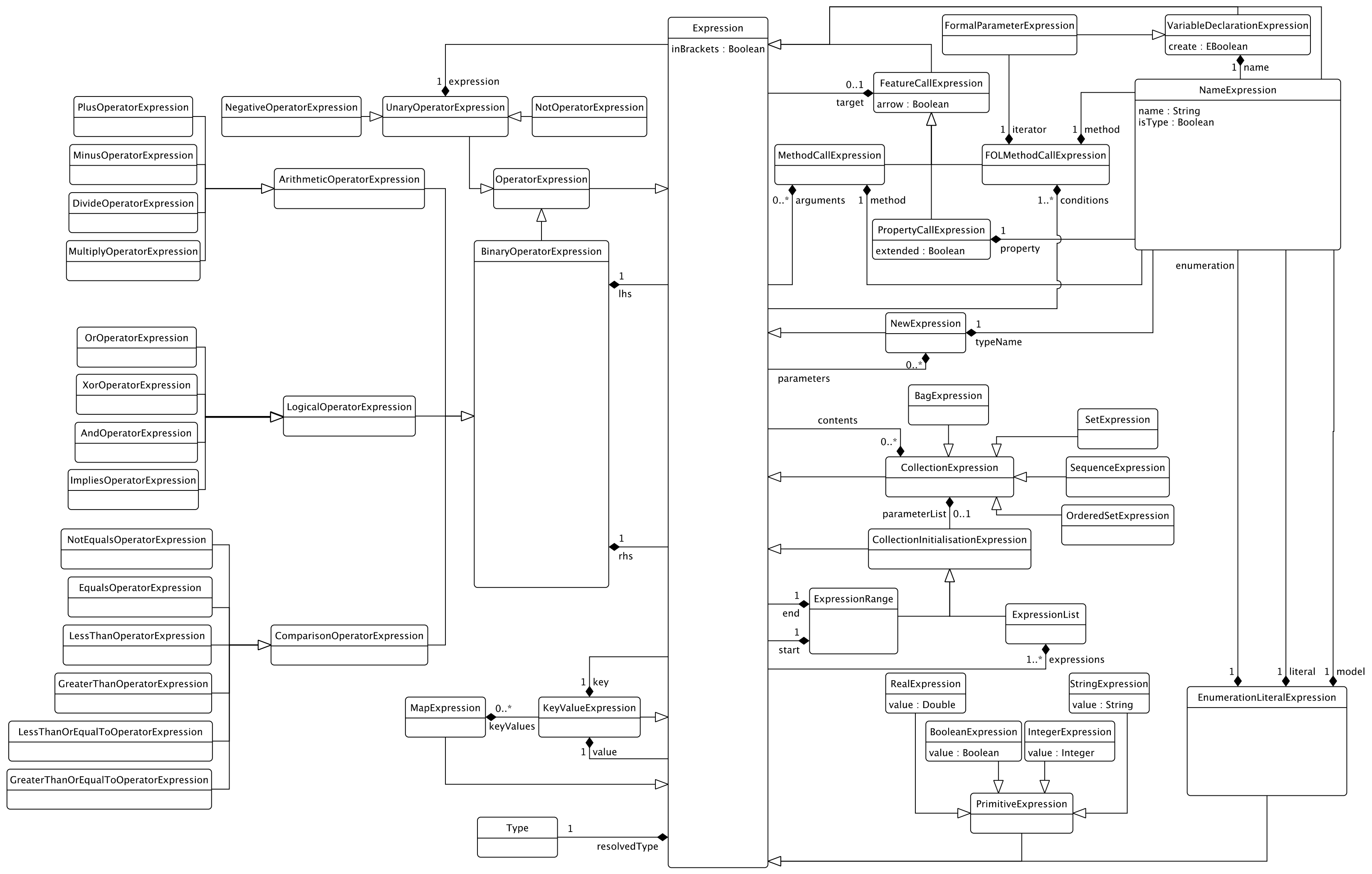

Figure 6.7: EOL Metamodel – EOL expressions

9. `Expression.inBrackets`: the mutation for this property removed the brackets from operator expressions: binary operator, logic operator and arithmetic operator expressions. This was to imitate the error of misplacing brackets between Boolean expressions. Since this concept was of a Boolean primitive type and had a default value, the addition and replacement actions were not considered. The replacement CMO is **`CMO-single-REP(Expression inBrackets)`**.
10. `ExpressionOrStatementBlock.block`: the mutation of this property was to delete a statement block. The addition operator and replacement operator for this property were not considered, as they required complex mutation values. The deletion CMO is **`CMO-single-DEL(ExpressionOrStatementBlock block)`**.
11. `ExpressionOrStatementBlock.condition`: this property was modified by deleting the condition of an expression. The replacement operator has been given previously in Enumerations 1 and 2. The addition action was not applied, as it required complex mutation, for example, values to interconnect the binary operands and operators. The deletion CMO is **`CMO-single-DEL(ExpressionOrStatementBlock condition)`**.
12. `ExpressionOrStatementBlock.expression`: the mutation for this property deleted a single expression (like statement deletion mutation). The addition and replacement operators were not generated, as they required complex values to be constructed. The deletion CMO is **`CMO-single-DEL(ExpressionOrStatementBlock expression)`**.
13. `ExpressionRange.start/end`: these properties are used to model the start range and end range of integer values given to integer variables when these are declared. The replacement mutation for these properties modified start/end integer numbers with another number. Since these expressions are made compulsory in the EOL metamodel, they always had values. Hence, the addition and deletion actions were not designed. The replacement CMOs are:
    - **`CMO-single-REP(ExpressionRange start)`**
    - **`CMO-single-REP(ExpressionRange end)`**
14. `FeatureCallExpression.target`: the mutation for this property removed the target of a feature call expression, such as methods, functions and properties. Targets can be variables or nested methods and functions. This mutation was to emulate the error of forgetting to specify an accurate target for calling a feature (whether method/function or property) call expression. The addition and replacement actions were not considered, as they require literal values and complex construction of, for example, nested methods and functions. The deletion CMO is **`CMO-single-DEL(FeatureCallExpression target)`**.

15. `FOLMethodCallExpression.conditions`: the language concept holds Boolean logical expressions, such as binary comparison expressions etc. The replacement mutation has already been presented in Enumeration 1. The deletion mutation deleted Boolean conditions, one at a time, from a collection of conditions its CMO is **`CMO-multiple-DEL(FOLMethodCallExpression conditions)`**. The addition mutation was not considered, as it required complex constructions like a binary operator and its left-/right-hand-side operands.
16. **`IfStatement.elseBody`**: the only available mutation for this property was a deletion mutation in which a block of an else-body of an if-statement was deleted. The objective was to determine whether the block had any effect on output models. The addition and replacement actions were not designed, as they required complex mutations for a block of statements. The deletion CMO for this property is **`CMO-single-DEL(IfStatement elseBody)`**.
17. **`IfStatement.elseIfBodies`**: this property also had only the deletion action performed in which every block in a collection of statement blocks was deleted. The addition and replacement actions were not considered, as they required complex constructions of values. The deletion CMO is **`CMO-multiple-DEL(IfStatement elseIfBodies)`**.
18. **`MapExpression.keyValues`**: the only mutation defined for this property was a deletion mutation operator that deleted, one at a time, a key-value pair from a collection of pairs associated to a map variable declaration. Since the key-value pairs are literal values, the addition and replacement actions were not considered because literal values need to be obtained from an infinite space of values. The deletion CMO of this concept is **`CMO-multiple-DEL(MapExpression keyValues)`**.
19. **`MethodCallExpression.arguments`**: the arguments of a method were changed using three mutation actions. The deletion action has been discussed previously in Enumeration 7. The addition mutation added an extra argument, whereas the replacement swapped two arguments with each other. All actions were defined to imitate calling methods with the wrong number of arguments or a different sequence of arguments. The addition and replacement CMOs are:
    - **`CMO-multiple-ADD(MethodCallExpression arguments)`**
    - **`CMO-multiple-REP(MethodCallExpression arguments)`**
20. `ModelElementType.modelName`: the name of a model element of variable declaration was changed by addition, deletion and replacement mutations. For the addition and replacement actions, a model name obtained from input and output metamodels was used for the variables' declaration of model elements. The objective of addition and replacement was to mimic the misuse of a model element name. For the deletion action, a simple deletion of a model name was performed to introduce the ambiguous reference to a variable to input and output model entities. All CMOs

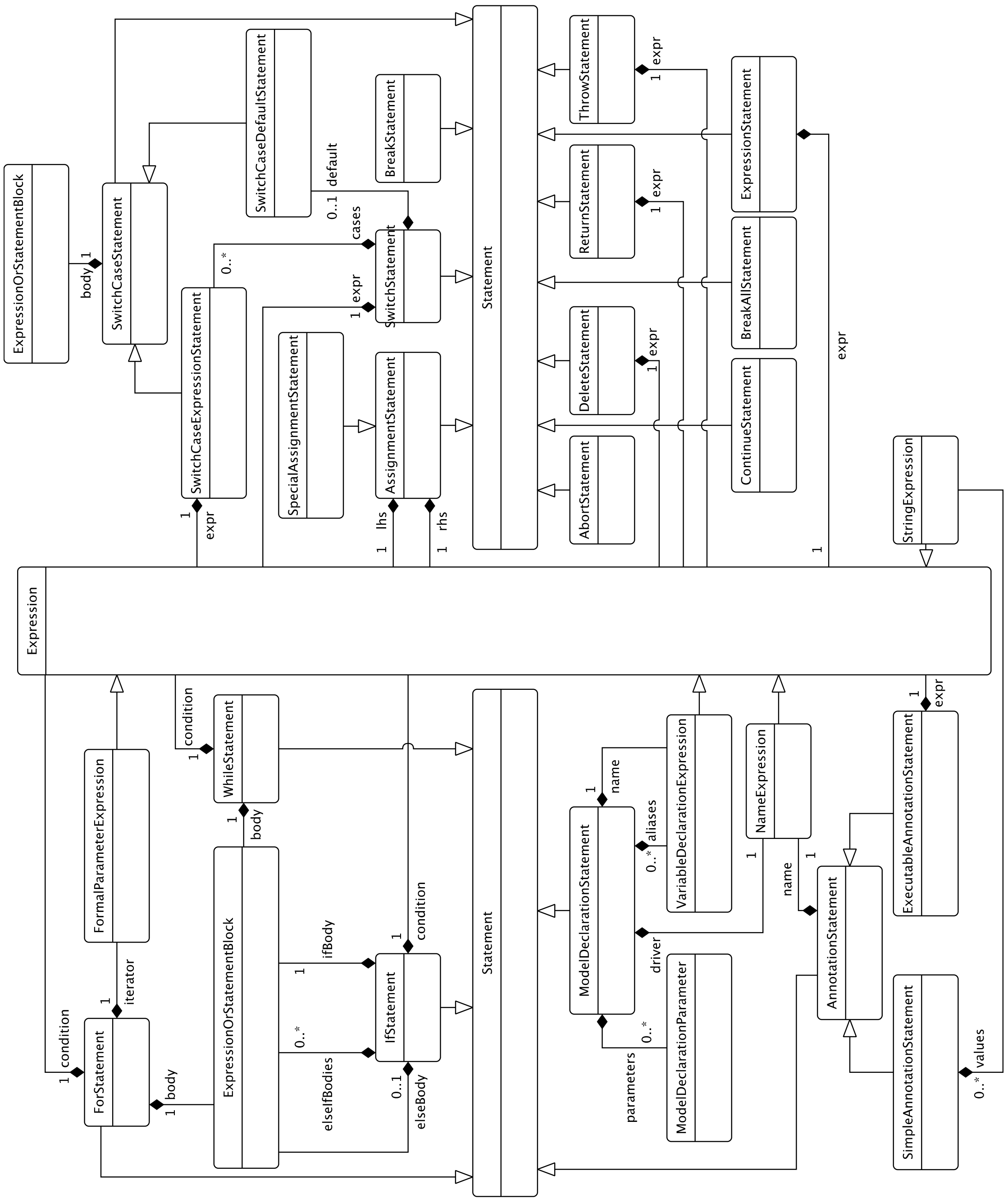

Figure 6.8: EOL Metamodel – EOL statements

for this property are:

- `CMO-single-ADD(ModelElementType modelName)`
- `CMO-single-DEL(ModelElementType modelName)`
- `CMO-single-REP(ModelElementType modelName)`

21. `OperationDefinition.annotationBlock`: the only mutation defined for this property was a deletion in which an annotation block of an operation, defined before the operation definition, was removed. An annotation block may contain statements that are used in the operation main block and its statements, and the objective was to determine whether annotation had any impact on the output

models. The addition and replacement operators were not considered as they require complex construction and literals. The deletion CMO for this property is **`CMO-single-DEL(OperationDefinition annotationBlock)`**.

22. `OperationDefinition.body`: the mutation for this property involved only the deletion of a block. The addition and replacement actions were not considered, as they require complex mutation values for statements and expressions that affected output artifacts. The deletion CMO is **`CMO-single-DEL(OperationDefinition body)`**.
23. `OperationDefinition.contextType`: only a replacement mutation was defined for this property in which the context type of an operation was changed with another type obtained from the EOL metamodel types (presented in Figure 6.9), including primitive types. Since this language concept is made compulsory, it always has a value and, therefore, the addition and deletion actions were not permitted. The replacement CMO of this property is **`CMO-single-REP(OperationDefinition contextType)`**.
24. `OperationDefinition.parameters`: the language concept was modified by addition, in which an extra parameter was added to a list of parameters of an operation declaration. This was to emulate the error of mismatching the number of parameters from the caller point of view. The deletion operator has been discussed previously in Enumeration 7. The replacement mutation was not considered, as there were no siblings with which to be replaced. The addition CMO for this property is **`CMO-multiple-ADD(OperationDefinition parameters)`**.
25. `PropertyCallExpression.extended`: a model element in EOL can have extended properties that are not supported in the metamodel to which it conforms. This is concretely identified by the tilde character (∼) at the front of the property. The only mutation for this property was replacement, in which the state of extended property was put to false when it was true. The replacement CMO for this property is **`CMO-multiple-REP(PropertyCallExpression extended)`**.
26. `ReturnStatement.expression`: the only mutation for this property was a replacement, in which a return statement of operator expressions, such as arithmetic, comparison and logical, was replaced by another operator from among their operands. The replacement CMO is **`CMO-single-REP(ReturnStatement expression)`**.
27. **`SimpleAnnotationStatement.values`**: this modeling concept was only modified by a deletion mutation, in which a value from a list of values was removed one at a time. The addition and replacement mutation actions were not applied, as they required literal values from an infinite space of values. The deletion CMO of this language concept is **`CMO-multiple-DEL(SimpleAnnotationStatement values)`**.

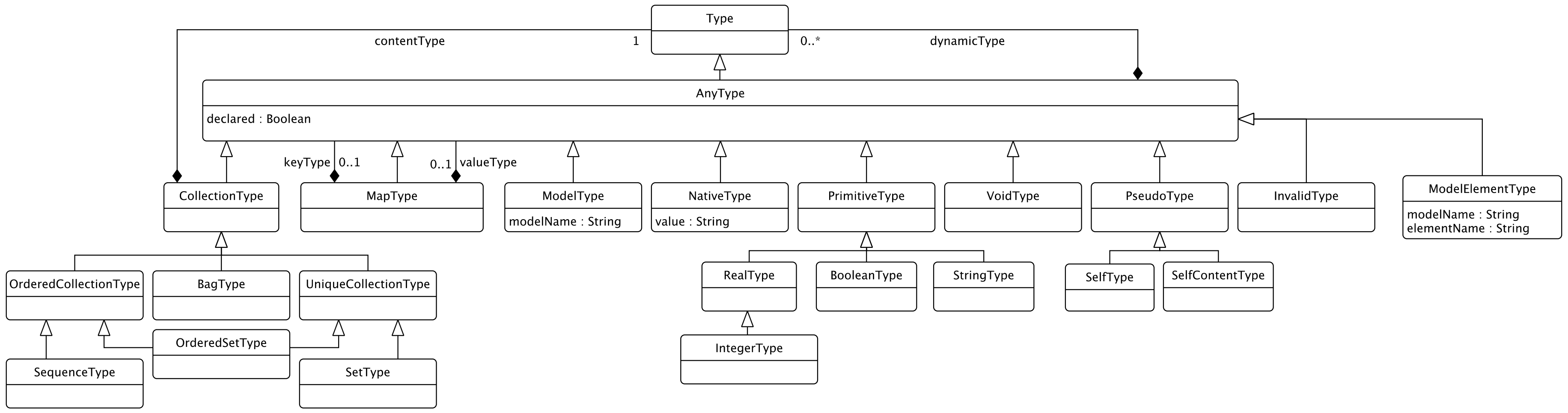

Figure 6.9: EOL Metamodel – EOL types

28. `SwitchCaseExpressionStatement.cases`: this property models a collection of switch cases. The only available mutation was deletion, in which a case from a list of cases was removed. The addition and replacement mutation actions were not applied: the former required complex constructions and the latter lacked siblings with which to be replaced. The deletion CMO for this property is **`CMO-multiple-DEL( SwitchCaseExpressionStatement cases)`**.
29. `SwitchStatement.default]`: the only mutation for this property, which models a default case of a switch statement, was a deletion mutation in which a default case was removed. The addition and replacement mutation actions were not considered for this property, as they required complex construction. The CMO for this property is **`CMO-single-DEL(SwitchStatement default)`**.
30. `VariableDeclarationExpression.create`: the only mutation designed for this property was a replacement, in which the create state (using *new* keyword) at the variable declaration expression was changed from true to false. The addition and replacement actions were not applied, as the property is Boolean and it always has a value. Thus, adding a value or deleting an existing one were not permitted. The replacement CMO for this property is **`CMO-single-REP(VariableDeclaration-Expression create)`**.

6.6 Results

This section presents the experimental results, which were obtained, first of all, from generating mutants of ATL and EOL from candidate programs by executing the list of mutation operators given in Section 6.5, and secondly by executing the generated mutants against the test models described in Section 6.4. This section is divided into three subsections. The first presents and discusses the results of performing mutation analysis against ATL programs and the second does the same for EOL programs. Finally, an overall evaluation of the research hypothesis is provided in the third section.

6.6.1 Empirical results of ATL programs

Prior to embarking on a detailed discussion of the results, this section attempts to provide answers to the experiment questions presented previously in Section 6.1 by examining the outcomes of stage one and stage two of the experiment. Following that, a detailed analysis of the results is presented during which several elements of the experiment, such as the used metamodels, the preconditions for the AMOs, and the types of mutants (including invalid, not killed and killed) are investigated in an effort to discern how these elements impacted on the overall results. The complete results of the ATL mutation analysis are made available in Appendix C.1.

The results of stage one of the experiment, concerned with applying mutation operators on selected ATL programs, are illustrated in Table 6.3. It is shown that a total of 287 mutants were prevented by the AMOs and their preconditions and marked as invalid, as they did not conform to the ATL metamodel. This yields a positive answer to question (**Q2**), which asks whether the preconditions of AMOs obstruct **any** invalid mutations from being generated.

Table 6.3: Valid and invalid mutants of mutating ATL programs (stage one)

Candidate Program	Valid	Invalid
Book2Publication	66	15
Make2Ant	204	20
Table2TabularHTML	496	50
TabularHTML2XML	495	64
Table2SVGPieChart	2234	138
Total	3495	287

The results of stage two of the experiment, which involved executing the valid mutants obtained from stage one against a set of test models, are given in Table 6.4. It is shown that different types of mutant were obtained (for example, live, killed etc.). This implies that the AMOs in fact helped to generate different types of mutations. In particular, this thesis (as mentioned earlier) considers live and killed mutants as useful and otherwise not worthless (namely not invalid or equivalent mutants). From a test developer's perspective, live mutations are useful as they are hard to detect and require more challenging test inputs forcing their detection. In addition, killed mutations are also important from a test developer's point of view, as they reveal the quality of test inputs, which is an essential objective of conducting mutation analysis. Thus, experiment question (**Q3**) investigating whether AMOs help to generate different types of useful mutants, such as live and non-trivially killed mutants, is also positively answered.

Table 6.4: Overall results of executing ATL mutants (stage two)

	Killed	Live	Equivalent	Invalid
Total	3050	80	213	128
%	87.26	2.29	6.09	4.34

Regarding the first question (**Q1**), it can be concluded that the results obtained from stages one and two suggest that AMOs can indeed be instantiated over ATL and OCL modeling concepts. The instantiation generated mutation operators that were able to produce various types of useful mutations. Therefore, experiment question **Q1** is answered positively. Hence, all research questions **Q1**, **Q2**, **Q3** were answered in the affirmative.

6.6.1.1 Invalid mutations

Generally, invalid mutants can be caused by the improper implementation of mutation operators. In this ATL mutation analysis experiment, the list of mutation operators in Table 6.5 generated only invalid mutations that were intercepted by the EMU engine during the execution of stage one. The intercepted mutants had violated the AMOs preconditions, which were implemented in the EMU engine. Such invalid mutations indicate that the listed operators were improperly designed and need to be enhanced.

Table 6.5: Mutation operators that only generated invalid mutants during stage one

No	Mutation Operator
1	`CMO-M-DEL(InPattern elements)`
2	`CMO-M-DEL(Module inModels)`
3	`CMO-M-DEL(Module outModels)`
4	`CMO-S-DEL(Iterator type)`
5	`CMO-S-DEL(MatchedRule actionBlock)`
6	`CMO-S-REP(BindingStat propertyName)`
7	`CMO-S-REP(BooleanType name)`
8	`CMO-S-REP(IntegerType name)`
9	`CMO-S-REP(MapType name)`
10	`CMO-S-REP(OclAnyType name)`
11	`CMO-S-REP(RealType name)`
12	`CMO-S-REP(SequenceType name)`
13	`CMO-S-REP(StringType name)`

For instance, operators 1–5 in Table 6.5 generated a set of invalid mutants that violated the lower multiplicity bound of the element. The first three generated invalid mutants when the operators attempted to delete the last element in the multi-valued properties `elements`, `inModels` and `outModels`, which all have a lower multiplicity limit equal to one. Thus, the operators can be re-implemented and enhanced, so that they can be applied against properties with values greater than one by including filter expressions to their CMO implementations. The remaining two operators also generated invalid mutants when they attempted to delete values of optional single-valued properties `type` and `actionBlock` at a time when there were no values to delete. In order to overcome this, the operators can be enhanced so that filter expressions are included into their CMO implementations making them only applicable to optional properties that actually contain values.

Another example is operators 7–13, which generated a set of invalid mutants when they attempted to modify the name of the OCL type by replacement. In fact, the operators inherited a super mutation operator **`CMO-Single-REP(OclType name)`** that was initially implemented to replace the value of property `name` of a model element type such as Book, Library, with another name. For OCL built-in types such as primitive types (String, Integer etc.) and collection types (Sequence, Set etc.), however, which extend the entity `OclType` as illustrated in the ATL metamodel in Figure 6.5, the property `name` is not applicable since there is no value for the property. Hence, their values cannot be replaced, which led to invalid mutants. As such, a re-implementation of the super mutation operator (that is, **`CMO-Single-REP(OclType name)`**) is required so that the primitive types and collection types of OCL are excluded from the mutation process, adding a filter that only fetches types defined in the input/output model.

In addition, invalid mutations can also be caused by an improperly constructed MDE metamodel, which in this case is the ATL metamodel. For instance, the mutation operator **`CMO-S-REP(BindingStat propertyName)`**, which is number 6 in Table 6.5, was designed to replace the property name of a binding statement with another name. In the used ATL metamodel, the property `propertyName` of the entity`BindingStat` of output pattern elements was never initialized with a value in any candidate programs and, hence, the property never had a value. In fact, the property was made compulsory in the used ATL metamodel, whereas it has no corresponding concept in the ATL syntax grammar. In other words, it is impossible to find a valid ATL program that has a value for the property name of a binding statement, as the ATL grammar does not allow that. As a consequence, the ATL metamodel should be revised in a way that the property `propertyName` of the entity `BindingStat` is removed from the metamodel.

Table 6.6: Mutation operators that only generate invalid mutants during stage two

No	Mutation Operator
1	**`CMO-M-DEL(Operation parameters)`**
2	**`CMO-M-DEL(Rule variables)`**
3	**`CMO-M-DEL(CalledRule parameters)`**
4	**`CMO-S-DEL(Rule inPattern)`**
5	**`CMO-S-DEL(Rule outPattern)`**
6	**`CMO-S-DEL(VariableDeclaration initExpression)`**
7	**`CMO-S-DEL(VariableDeclaration type)`**

There are also many invalid mutants that were not prevented by EMU but were rather detected by the ATL transformation engine when they were executed against the test models. Such mutants were generated from the set of operators given in Table 6.6, which accumulated 128 mutants out of 3,495. Unlike the first set of operators presented in Table 6.5 that mostly require re-implementation, the second set of operators are worthless from a test developer's point of view, as they do not contribute to the test quality assessment. Hence, they can be avoided in future mutation analyses of ATL programs.

6.6.1.2 Live mutations

In mutation analysis, live mutants are not detected (or killed) by any test input and they always produce the same outputs as the original program. They are, however, detectable mutants in the sense that a test developer, who is conducting mutation testing, is challenged to provide more test inputs that will force such mutants to give different outputs than the original program. The proposed AMOs have helped in generating a few concrete mutation operators that were able to produce a total of 80 live mutants out of the 3,495 from the used ATL programs. These concrete operators are given in Table 6.7.

Table 6.7: Mutation operators that contributed to live mutants

No	Mutation Operator	Gen.	Killed	Equiv.	Live	Invalid
1	`CMO-S-DEL(InPattern filter)`	4	3	-	1	-
2	`CMO-S-REP(InPattern filter)`	14	12	-	2	-
3	`CMO-S-REP(Parameter type)`	15	6	-	9	-
4	`CMO-S-REP(Operation returnType)`	77	43	-	34	-
5	`CMO-S-REP(InPatternElement type)`	296	253	9	34	-

The live mutants produced by operators 1 and 2 in Table 6.7 were found by adding new test models on top of the ones used for stage two (as part of the experimental approach discussed in Section 6.2). The operators modified by replacement the Boolean comparison expressions or logical expressions of the filters of input patterns in transformation rules. As such, it is argued in this thesis that the operators can be considered useful, and that they should be used for any future ATL programs mutation analysis experiments.

Furthermore, the conducted experiment of ATL programs revealed another type of live mutants that contains faults that should have been detected by the ATL transformation engine. More specifically, after replacing an OCL type (whether a model type or an OCL built-in type) with another type using mutation operators 3–5 in Table 6.7, the ATL transformation engine ignored the injections of incorrect types and continued the mutants' execution against all test models with no run-time error; an opposite action of what the ATL engine was expected to trigger. The live mutants that should have been detected were those generated by operator `CMO-S-REP(Parameter type)` when it targeted called-rule parameters' types, operator `CMO-S-REP(Operation returnType)` when it targeted return types of operations, and operator **`CMO-S-REP(InPatternElement type)`** when it targeted input pattern types of lazy rules.

It is possible to conclude that AMOs can be instantiated in the ATL metamodel and generate operators that, when applied, produce not only stubborn mutants, which require more test cases, but also live mutants that reveal odd execution behaviors of the ATL engine, which can be resolved in future releases of the ATL transformation engine.

6.6.1.3 Killed mutations

In theory, killed mutants indicate that a test suite is good enough to detect common programmer errors. As mentioned in Section 2.2.3, there are two types of killed mutants: trivially killed and non-trivially killed (or simply just killed) mutants. A trivial mutant is a mutant that is easy to detect and is killable by all test inputs. On the contrary, a non-trivial mutant is a mutant that is killed by at least one single test input but not all test inputs. From a test developer's point of view, a non-trivial mutant is potentially more useful than a trivial one because a mutation operator that contributes to non-trivial mutants may provide mutants of plausible programmer errors.

Table 6.8 presented a list of mutation operators that contributed to killed mutants; and most of which were trivial ones. For example, some modifications of listed operators involved replacement or deletion of blocks that usually contained multiple statements or constructs (like the case of operators 2, 3, 5, 6, 14, 16, 18, 22, 25 and 26). Such actions had been found impacting largely on original programs which produced, as a result, vulnerable mutants that were detected easily by test models. The operators are sorted in descending order based on the number of non-trivial mutations. The reason for such sorting is to list mutation operators based on their usefulness to ATL test developers, who wish to conduct mutation analysis and reuse these operators.

6.6.1.4 Equivalent mutations

Equivalent mutants are mutants that produce the same outputs as the original program when executed against test inputs. Unlike live mutations, discussed in Section 6.6.1.2, that can be detected by finding a good set of inputs, equivalent mutants can never be killed, as no test input can be included to force their detection. In this empirical mutation analysis of ATL programs, a total of 213 mutants out of 3,495 were classified as equivalent. There are only two main mutation operators that have contributed to equivalent mutants: operator **`CMO-S-REP(MatchedRule isAbstract)`** and operator **`CMO-S-REP(MatchedRule name)`**. The first was designed to introduce a change to the

Table 6.8: Mutation operators that contributed to killed mutants

No.	Mutation Operator	Non-trivial	Trivial	Equiv.
1	`CMO-S-REP(RuleVariableDeclaration varName)`	78	11	51
2	`CMO-S-DEL(LazyMatchedRule actionBlock)`	1	1	-
3	`CMO-M-DEL(InPattern elements)`	2	3	-
4	`CMO-S-REP(CalledRule name)`	1	2	1
5	`CMO-M-DEL(ActionBlock statements)`	3	7	-
6	`CMO-M-DEL(OutPattern elements)`	2	5	-
7	`CMO-S-REP(IfStat condition)`	2	5	-
8	`CMO-S-REP(LazyMatchedRule name)`	2	9	-
9	`CMO-S-REP(Attribute name)`	2	11	2
10	`CMO-S-REP(OutPatternElement type)`	136	800	8
11	`CMO-S-REP(OclModelElement name)`	14	88	11
12	`CMO-S-REP(OclModel name)`	6	42	12
13	`CMO-M-DEL(SimpleOutPatternElement bindings)`	14	112	4
14	`CMO-M-REP(Module elements)`	2	19	-
15	`CMO-S-REP(Binding propertyName)`	29	303	-
16	`CMO-M-DEL(Module elements)`	5	62	5
17	`CMO-S-REP(OclFeatureDefinition feature)`	2	31	4
18	`CMO-S-DEL(MatchedRule outPattern)`	1	16	-
19	`CMO-S-REP(Attribute type)`	92	-	34
20	`CMO-S-REP(SequenceType elementType)`	68	-	30
21	`CMO-S-REP(RuleVariableDeclaration type)`	21	-	-
22	`CMO-S-DEL(MatchedRule actionBlock)`	-	2	-
23	`CMO-S-REP(CalledRule isEntrypoint)`	-	2	-
24	`CMO-M-DEL(ForStat statements)`	-	4	-
25	`CMO-M-DEL(IfStat thenStatements)`	-	4	-
26	`CMO-S-DEL(CalledRule actionBlock)`	-	4	-
27	`CMO-S-REP(BindingStat source)`	-	5	-
28	`CMO-M-DEL(MatchedRule children)`	-	7	3
29	`CMO-S-DEL(OclFeatureDefinition context)`	-	20	2
30	`CMO-S-REP(Operation name)`	-	21	2
31	`CMO-M-REP(MatchedRule children)`	-	70	3
32	`CMO-S-REP(OclContextDefinition context)`	-	584	10

Boolean values of abstract properties of matching rules. The mutants have no effect on output models when the matching rules themselves have no input and output pattern elements. The second was designed to change matching rules names and also has no effect on output models. This is because the transformation engine executes all rules in a particular order regardless of their names.

6.6.1.5 Comparison of evaluation results with Guerra et al. [103]

As mentioned previously in Section 2.3.2, Guerra et al. [103] have recently examined a set of ATL mutation operators defined in [100, 101, 102, 103] over six ATL programs. Although they did not discard equivalent mutants, they distinguished four out of 55 mutation operators which produced hard-to-kill mutants, while the rest delivered easy-to-kill ones. Regarding hard operators, the following gives a list of mutations that contributed to hard-to-kill mutants [103], where each mutation is discussed with respect to the ATL operators used for this experimental study.

- Helper Deletion: this operator deletes a helper function element from a transformation module. Mutation operator number 16 in Table 6.8, which deletes module elements including helpers, has produced only killed mutants, none of which were live, which corresponds to the terminology of Guerra et al. [103] of hard-to-kill mutants. This can be caused by a number of reasons. One reason can be that the initial test models used in the first run of this experiment (for example, when executing mutants for the first time) were so good that they detected mutants generated from the mutation operator **`CMO-M-DEL(Module elements)`**. Another reason can be that the mutated operations or called rules by the Helper Deletion operator of Guerra et al. [103] were triggered by some of the testing inputs but not all (for instance, the mutated places were never reached).
- ParameterDeletionMutator: this operator deletes a parameter from a list of parameters of an operation or a called rule. In this experiment, operators number 1 and 3 in Table 6.6, which correspond to this operator by Guerra et al., have contributed only to invalid mutants that could not be loaded to the ATL engine. This result differs from that reported by Guerra et al. This difference indicates that the mutation operators, which correspond to the operator ParameterDeletionMutator, require re-implementation.
- ParameterModificationMutator: this operator changes the type of a parameter of an operation or a called rule with another type, for example, model types, ATL primitive types or ATL collection types. The operator that corresponds to this operator in this experimental implementation is **`CMO-S-REP(Parameter type)`** in Table 6.7. The operator has produced 15 mutants out of which nine were considered live when the mutated parameters were related to called rules only. Section 6.6.1.2 gives more details about live mutants.
- AddNavigationAfterOptionalFeature (ANAOF): there is no corresponding operator that performs the same action as this operator since the current experiment has only considered OCL concepts directly referred to from within ATL. The operator AddNavigationAfterOptionalFeature of Guerra et al., which was applied to an OCL concept that is not directly referred to from within ATL, lies beyond the scope of this experiment.

6.6.2 Empirical results of EOL programs

This section presents the results of the mutation analysis of EOL programs using the same approach followed for ATL programs. First, the experiment questions, presented in Section 6.1, are discussed. This is followed by a detailed examination of the results of the experiment including an analysis of the types of mutants. The complete results of EOL mutation analysis are made available in Appendix C.2.

In regard to **Q2** that investigates whether the AMOs preconditions intercept any invalid mutants from being generated, Table 6.9 indicates that more than a third of mutants (around 23 percent) were intercepted by EMU, while implementing the preconditions of AMOs. Reducing the number of invalid mutations is one of the objectives of this thesis. This is because invalid mutations are worthless from a test developer's point of view, as they do not challenge a given test set and also negatively prolong the mutation analysis execution.

Table 6.9: Valid and invalid mutants of mutating EOL programs (stage one)

Candidate Program	Valid	Invalid
ShortestPath	172	55
Formatting	800	207
EcoreHelper	1358	307
ECoreUtil	1553	258
ECore2GMF	1553	428
Total	5436	1255

In order to address **Q3**, which questions whether AMOs generate useful mutants (for example, live and killed mutants), the obtained results of stage two of the experiment in mutating EOL programs, as outlined in Table 6.10, shows that AMOs are indeed able to generate useful mutants.

Table 6.10: Overall results of executing EOL mutants (stage two)

	Gen.	Trivial	Killed	Live	Equiv.	Invalid
Total	5436	2294	787	1126	507	722
%		42.20	14.47	20.71	9.32	13.28

From both sets of results (stage one and stage two) of mutating EOL programs, it can be concluded that AMOs can indeed be instantiated over EOL modeling concepts defined in the used metamodel. This answers **Q1**, as the instantiation of AMOs generated operators that were able to produce different types of mutants. Hence, all research questions presented in Section 6.1 (**Q1**, **Q2**, **Q3**) have been answered in the affirmative.

In the following sub-sections, a detailed discussion of the results of stages one and two of mutating EOL programs takes place in which several elements of the experiment, such as the used metamodels, AMOs, test inputs etc. are analyzed.

6.6.2.1 Invalid mutations

An invalid mutant can be invalid during the error injection process (i.e. when applying a mutation operator to introduce a syntactic change) or invalid when interrupted during execution by the language engine. Sometimes, semantically invalid mutants can be valid syntactically for the language to which they conform. In the experiment of mutating EOL candidate programs, a total of 1,255 out of 6,691 mutants (as illustrated in Table 6.9) were classified as syntactically invalid during the mutation injection process (stage one). As such, they were prevented from being generated by AMOs and their preconditions. Also, a total of 722 out of 5,436 (as shown in Table 6.10) syntactically valid mutants were semantically invalid and detected (failed to load) by the EOL engine during the process of executing the mutants (stage two). Both sets of results show that the AMOs and their preconditions intercepted more than half of the total of invalid mutants, which is a positive result for AMOs and their preconditions.

Table 6.11: Mutation operators that contributed to invalid mutants in stage one

No.	Mutation Operator	Valid	Invalid
1	`CMO-S-REP(GreaterThanOperatorExpression inBrackets)`	-	1
2	`CMO-S-REP(LessThanOperatorExpression inBrackets)`	-	1
3	`CMO-S-REP(LessThanOrEqualToOperatorExpression inBrackets)`	-	1
4	`CMO-S-REP(MinusOperatorExpression inBrackets)`	-	1
5	`CMO-S-REP(OrOperatorExpression inBrackets)`	4	6
6	`CMO-S-REP(NotEqualsOperatorExpression inBrackets)`	-	5
7	`CMO-S-REP(AndOperatorExpression inBrackets)`	1	15
8	`CMO-S-REP(NegativeOperatorExpression inBrackets)`	-	16
9	`CMO-S-REP(NotOperatorExpression inBrackets)`	2	18
10	`CMO-S-REP(EqualsOperatorExpression inBrackets)`	1	43
11	`CMO-S-REP(PlusOperatorExpression inBrackets)`	1	101
12	`CMO-M-DEL(FOLMethodCallExpression conditions)`	-	49
13	`CMO-S-ADD(ModelElementType modelName)`	-	84
14	`CMO-S-DEL(ExpressionOrStatementBlock condition)`	17	193
15	`CMO-S-DEL(ExpressionOrStatementBlock expression)`	-	210
16	`CMO-S-REP(PropertyCallExpression extended)`	45	389

Regarding invalid mutants during the mutation injection process (stage one), Table 6.11 presents a set of concrete operators that mostly produces invalid mutants, which need to be re-implemented or enhanced. In particular, all EMU implementations of the presented concrete mutation operators need to include filters that collect model instances with which the number of invalid mutations is reduced. For instance, the concrete mutation operators 1–11 try to set (by replacement) the Boolean values of the property `inBrackets` to negative values (that is, false). The property is used to model the situation in which operator expressions, for example binary logical expressions, binary comparison expressions and binary arithmetic expressions, are enclosed in parentheses. The invalid mutations that were intercepted by EMU and AMOs preconditions, were encountered when the prior values of the property `inBrackets` were already negative. When the operator tried to change the values to false as well, invalid mutants were produced. Thus, the number of invalid mutants can be reduced by re-implementing the operators, so that filters are added to restrict the application to instances that have the property `inBrackets` values set to true.

Likewise, invalid mutants produced by operator **`CMO-M-DEL(FOLMethodCallExpression conditions)`** can be reduced by including a filter. The large number of invalid mutants was obtained during attempts to delete the last item of the multi-valued property `conditions` (a first-order logic expression can have one or multiple conditions). In order to reduce the number of invalid mutations, a filter that only fetches the model instances of entity `FOLMethodCallExpression` that have more than one condition can be included, so that the mutation deletion action is not performed against the only condition existing in the collection of conditions. This enhancement would reduce the large number of invalid mutants generated by the mutation operator **`CMO-M-DEL(FOLMethodCallExpression conditions)`**.

The aforementioned mutation operators contributed the most to invalid mutations, and were intercepted and exposed by the EMU engine and AMOs preconditions. Such exposure gives credibility to the preconditions of AMOs, as they can indicate the failed implementation of CMOs. As such, the user of AMOs, who may be interested in constructing a mutation operator to test an important feature in the metamodel and language, will be notified of any false implementation when a high number of invalid mutants is detected by the EMU and the preconditions of an operator.

Table 6.12: Mutation operators that contributed to invalid mutants in stage two

No.	Mutation Operator
1	**`CMO-S-DEL(ExpressionOrStatementBlock condition))`**
2	**`CMO-S-DEL(OperationDefinition body)`**
3	**`CMO-S-DEL(PropertyCallExpression target)`**

Regarding invalid mutations during the execution of the EOL mutants (stage two), Table 6.12 presents a set of concrete operators that contributed only to invalid mutations. The first mutation operator (**`CMO-S-DEL(ExpressionOrStatementBlock condition)`**) interestingly contributed only to invalid mutants in both stages one and two. This indicates that the operator may not need a re-implementation but rather that the EOL metamodel needs some adjustments related to the property `condition` in order to lower the number of invalid mutants generated by that property. In fact, the modeling entity `ExpressionOrStatementBlock`) needs reconstruction in a way that the high number of invalid mutants from its properties `condition` and `expression` (and their concrete mutation operators 14 and 15 in Table 6.11) is reduced.

Modeling entity `ExpressionOrStatementBlock` is mainly used to model two situations: modeling a block of code (for example, a body of controlling statements such as if statements, *while* statements etc.) and nested if statements $else-if$ (as given in Listing 6.1 of the EOL metamodel fragment lines 11, 12 and 13), where the keyword *val* indicates an end association between two entities by value. In the first case, the optional (lower multiplicity boundary is 0) property `block` in line 5 is used, whereas the optional properties `condition` (line 6) and `expression` (line 7) are not used. In the second, (i.e. the nested if statement), the `block` and `condition` are used, while the property `expression` is not. This means that the property `condition` in the second situation is seen as compulsory, and deleting its value definitely generates invalid mutations according to the EOL language specifications (as proven also by the number of invalid mutations of mutation operator **`CMO-S-DEL(ExpressionOrStatementBlock condition)`** in Tables 6.11 and 6.12). Thus, the modeling entity `ExpressionOrStatementBlock` and its usage for nested $else - if$ blocks in line 12 must be re-defined, so that the generation of invalid mutants is reduced.

```
1  class Block {
2   val Statement[*] statements;
3  }
4  class ExpressionOrStatementBlock {
5   val Block[0..1] block;
6   val Expression[0..1] condition;
7   val Expression[0..1] expression;
8  }
9  class IfStatement extends Statement {
10  val Expression[1] condition;
11  val ExpressionOrStatementBlock[1] ifBody;
12  val ExpressionOrStatementBlock[*] elseIfBodies;
13  val ExpressionOrStatementBlock[0..1] elseBody;
14 }
```

Listing 6.1: Original EOL metamodel fragment expressed using EMFatic

In Listing 6.2, a few adjustments have been suggested to fix the original EOL metamodel fragment given in Listing 6.1 and re-define the property `elseIfBodies`. The modeling entity `ElseIfStatement` in line 1 is a replacement of the entity `ExpressionOrStatement Block`, where the property `condition` is made compulsory with lower limit multiplicity set to one rather than optional. This triggered a high number of invalid mutations with the original EOL metamodel, which makes the property's value accept a replacement operator only, while the ability to have addition or deletion mutation operators, which previously contributed the most to invalid mutations, is not allowed.

```
1  class ElseIfStatement extends Statement {
 val Expression[1] condition;
 val Block[1] block;
}
class IfStatement extends Statement {
 val Expression[1] condition;
 val Block[1] ifBody;
 val ElseIfStatement[*] elseIfBodies;
 val Block[0..1] elseBody;
}
```

Listing 6.2: Improved EOL metamodel fragment of Listing 6.1

The second mutation operator **`CMO-S-DEL(OperationDefinition body)`** in Table 6.12, which contributed only to invalid mutations, is an operator that was designed to delete the body block of an operation or a function. Although the body of an operation is compulsory, according to the EOL language specifications, and must be always present, the EOL metamodel that was used for the experiment (illustrated in Figure 6.6) was constructed in such a way that the body of an operation is specified as optional (that is, the lower bound of the limit was set to zero) and, consequently, the deletion mutation action became possible. Thus, when the operator was executed, it generated only invalid mutants, which suggests a design fault in the EOL metamodel when a property that should have been placed as compulsory is otherwise put as optional by setting the lower bound of the target property to zero. Hence, the used metamodel should be fixed so that the property `body` of entity `OperationDefinition` has the lower bound of one.

6.6.2.2 Live mutations

In this experiment of mutating EOL programs, the used AMOs were able to generate a set of mutation operators that resulted in a total of 1,126 live mutants, out of 5,436 mutants (as given in Table 6.10), which were later killed by adding more challenging test cases. The set of mutations that contributed to the mentioned live mutants are given in Table 6.13 and they are sorted based on the percentage of live mutants produced against the overall generated mutations. Test developers who wish to apply mutation testing to EOL programs may use this list of mutation operators as a start for quality mutation testing.

There are a few interesting findings in many live mutants of EOL programs. The first finding is that many live mutants were a result of unexpected behavior of the EOL engine when it violates the EOL specification that restricts the usage of binary comparison operators (for example, $>$, $\geq$, $<$ and $\leq$) to only numerical literals. Instead, the EOL engine had overlooked the usage of comparison operators with unsupported literals, such as Strings, Booleans and Objects, when the engine was expected to yield

Table 6.13: Mutation operators that contributed to live mutants

No	Mutation Operator	Gen.	Killed	Live	Equiv.	Invalid
1	`CMO-S-REP(AndOperatorExpression inBrackets)`	1	-	1	-	-
2	`CMO-S-REP(OrOperatorExpression inBrackets)`	4	1	2	1	-
3	`CMO-S-REP(ReturnStatement expression)`	62	25	28	9	-
4	`CMO-M-ADD(Block statements)`	145	66	62	17	-
5	`CMO-S-DEL(IfStatement elseBody)`	41	13	15	13	-
6	`CMO-S-REP(ExpressionOrStatementBlock condition)`	85	53	31	1	-
7	`CMO-S-REP(PropertyCallExpression extended)`	45	29	16	-	-
8	`CMO-S-DEL(ExpressionOrStatementBlock block)`	209	92	68	32	17
9	`CMO-M-REP(Block statements)`	160	96	51	13	-
10	`CMO-M-REP(MethodCallExpression arguments)`	102	68	32	2	-
11	`CMO-M-DEL(MethodCallExpression arguments)`	170	117	50	3	-
12	`CMO-S-REP(VariableDeclarationExpression create)`	14	9	4	1	-
13	`CMO-M-DEL(Block statements)`	825	496	235	94	-
14	`CMO-M-REP(FOLMethodCallExpression conditions)`	96	60	26	10	-
15	`CMO-S-REP(IfStatement condition)`	230	138	62	30	-
16	`CMO-S-DEL(MethodCallExpression target)`	529	381	123	25	-
17	`CMO-S-REP(MethodCallExpression method)`	119	75	26	18	-
18	`CMO-M-ADD(MethodCallExpression arguments)`	581	423	125	33	-
19	`CMO-S-REP(ModelElementType modelName)`	205	93	32	16	64
20	`CMO-S-REP(VariableDeclarationExpression name)`	178	140	27	11	-
21	`CMO-S-REP(AssignmentStatement rhs)`	27	18	4	5	-
22	`CMO-S-REP(FormalParameterExpression name)`	160	126	22	12	-
23	`CMO-S-DEL(ModelElementType modelName)`	205	54	28	58	65
24	`CMO-S-REP(OperationDefinition returnType)`	97	45	12	18	22
25	`CMO-M-ADD(OperationDefinition parameters)`	103	76	10	17	-
26	`CMO-M-DEL(EOLModule operations)`	103	76	10	17	-
27	`CMO-S-REP(FOLMethodCallExpression method)`	212	169	19	24	-
28	`CMO-S-DEL(FOLMethodCallExpression target)`	49	42	3	4	-
29	`CMO-M-DEL(IfStatement elseIfBodies)`	17	16	1	-	-
30	`CMO-M-DEL(OperationDefinition parameters)`	71	65	1	5	-

an unaccepted usage of operators. The mutation operators that produced those live mutants are numbers 3 (22 mutants), 6 (28 mutants), 14 (18 mutants) and 15 (27 mutants) in Table 6.13.

Another unexpected behavior of the EOL engine was spotted in a few cases when mutating returned statements of operations by replacement and addition. With replacement in particular, on a few occasions, the EOL engine executed normally some operations with no return statements, although the declaration of such operations indicated a return nature. Mutation operator number 3 in Table 6.13 replaced a return statement by removing the keyword *return*, leaving the expression associated with it in place. Some of the produced mutants of this operator (14 mutants in total) were executed normally and returned a value even though there were no return keywords specified.

In very rare occasions, an addition of a return statement into an operation block that is not intended to do any return values is another unexpected behavior of the EOL engine. Mutation operator number 4 in Table 6.13 did such an addition mutation, where a return statement was added to a block (including operation blocks). When the mutation operator was applied, it produced a couple of mutants that were not detected by the EOL engine when executed against test models, as should be the case.

6.6.2.3 Killed mutations

Killed mutations are those mutations detected by original test cases, without the need to add more challenging tests. The proposed AMOs were able to generate non-trivial mutants and also trivial mutants. Those mutants amounted to 3,081 out of 5,436 of the total mutants. Table 6.14 presents a set of mutation operators that contributed to killed mutants and which are sorted according to their percentage of the total of non-trivially killed mutants.

Table 6.14: Mutation operators that contributed to killed mutants

No	Mutation Operator	Killed	Trivial	Non-trivial
1	`CMO-S-REP(OrOperatorExpression inBrackets)`	1	-	1
2	`CMO-M-DEL(IfStatement elseIfBodies)`	16	4	12
3	`CMO-S-REP(ExpressionOrStatementBlock condition)`	53	14	39
4	`CMO-S-REP(OperationDefinition contextType)`	5	2	3
5	`CMO-M-ADD(Block statements)`	66	32	34
6	`CMO-S-REP(ReturnStatement expression)`	25	13	12
7	`CMO-S-REP(AssignmentStatement rhs)`	18	10	8
8	`CMO-S-DEL(ExpressionOrStatementBlock block)`	92	56	36
9	`CMO-S-DEL(ModelElementType modelName)`	54	33	21
10	`CMO-S-DEL(IfStatement elseBody)`	13	8	5
11	`CMO-S-REP(PropertyCallExpression extended)`	29	18	11
12	`CMO-M-REP(FOLMethodCallExpression conditions)`	60	38	22
13	`CMO-M-DEL(Block statements)`	496	335	161
14	`CMO-S-REP(ModelElementType modelName)`	93	69	24
15	`CMO-M-REP(Block statements)`	96	72	24
16	`CMO-M-ADD(MethodCallExpression arguments)`	423	324	99
17	`CMO-S-REP(VariableDeclarationExpression name)`	140	108	32
18	`CMO-S-DEL(MethodCallExpression target)`	381	297	84
19	`CMO-S-REP(IfStatement condition)`	138	109	29
20	`CMO-S-REP(FOLMethodCallExpression method)`	169	135	34
21	`CMO-S-REP(MethodCallExpression method)`	75	60	15
22	`CMO-S-REP(VariableDeclarationExpression resolvedType)`	10	8	2
23	`CMO-S-REP(OperationDefinition returnType)`	45	36	9
24	`CMO-S-REP(FormalParameterExpression name)`	126	101	25
25	`CMO-S-DEL(FOLMethodCallExpression target)`	42	34	8
26	`CMO-M-DEL(MethodCallExpression arguments)`	117	99	18
27	`CMO-M-ADD(OperationDefinition parameters)`	76	70	6
28	`CMO-M-DEL(EOLModule operations)`	76	70	6
29	`CMO-M-REP(MethodCallExpression arguments)`	68	64	4
30	`CMO-M-DEL(OperationDefinition parameters)`	65	62	3
31	`CMO-S-REP(VariableDeclarationExpression create)`	9	9	-
32	`CMO-S-DEL(EOLModule block)`	2	2	-
33	`CMO-S-REP(PlusOperatorExpression inBrackets)`	1	1	-
34	`CMO-S-REP(WhileStatement condition)`	1	1	-

6.6.2.4 Equivalent mutations

In this experiment, a total of 507 mutants out of 5,436 generated valid mutants has been classified as equivalent mutants, which were not killed even when more test inputs were added. Table 6.15 gives a list of mutation operators that only contributed to equivalent mutants and which can be avoided for future mutation analysis of EOL programs.

Table 6.15: Mutation operators that contributed to equivalent mutants

No	Mutation Operator
1	`CMO-S-REP(NotOperatorExpression inBrackets)`
2	`CMO-S-REP(EqualsOperatorExpression inBrackets)`
3	`CMO-S-REP(FormalParameterExpression resolvedType)`

6.6.3 Research Hypothesis Evaluation

This chapter evaluates the research hypothesis put forward by this thesis, which was introduced in Section 3.2. The results obtained from the experiment suggest that the proposed AMOs, which consist of preconditions and rules, have contributed to (1) the generation of few invalid and equivalent mutants, and (2) the generation of plausible and worthy mutants, namely live and non-trivially killed mutants. Furthermore, the proposed AMOs are metamodel-agnostic and they have been used to generate concrete mutation operators for the candidate languages ATL and EOL. Therefore, initial evidence that AMOs can be used against a given model of a language has been presented. The results provide sufficient confidence that the research hypothesis can be validated and that the research objectives in Section 3.3 have been fulfilled. They are as follows.

- Propose and design a set of metamodel-agnostic mutation operators: this objective has been satisfied with AMOs used against different metamodels during the experiment. In particular, AMOs have been used for the best known model management languages: ATL, which is used for model transformation, and EOL which is used for model query and model manipulation. In both cases, metamodels of those languages have been used to generate concrete mutation operators as presented in Section 6.5.
- The proposed mutation operators should help to produce plausible mutants: the conducted experiment of ATL and EOL programs has shown that AMOs are indeed capable of producing plausible mutants, such as live and non-trivially killed mutants. For live mutants, Section 6.6.1.2 has indicated that many can be produced from ATL programs and Section 6.6.2.2 has shown that many more can be produced from EOL programs. In both cases, those mutants are evidence that AMOs are able to generate concrete operators that produce live mutants.

- The proposed mutation operators should produce few invalid mutants: the abstract mutation operators (AMOs) are integrated with preconditions that are "copied over" to concrete ones when instantiated. Preconditions are checked by the EMU engine, which executes the concrete mutation operators. The results show that a number of invalid mutations had been prevented before their execution and only a small percentage of invalid mutants had remained undetected.
- The usefulness of the proposed mutation operators should be evaluated using an empirical mutation analysis over relevant programs. This was achieved using the selection process presented in Section 6.3.

Chapter 7

Conclusion

Mutation testing is a powerful testing technique if the operator set that is used for testing is well designed [12]. For decades, it has been used to test programs written in traditional programming languages such as C, Java and SQL. Limited investigation into mutation testing in model driven engineering, however, is unfortunate, as it is a particularly good candidate for mutation testing. One of the key principles of MDE – that everything is a model – offers the opportunity to derive mutation operators systematically, as MDE languages are usually expressed as models built on common metamodeling languages, such as MOF and Ecore.

The thesis has explored an efficient and effective mutation design approach for the rapidly emerging MDE languages. In order to conduct this research, a field review on related concepts and principles of MDE and mutation testing, as well as current approaches for integrating mutation testing into model driven engineering and their key challenges and limitations has been provided in Chapter 2. Chapter 3 presented an analysis of the problem this thesis tries to address. The problem analysis identified a number of critical limitations of previous attempts at mutation operator design for MDE languages. Based on those, the thesis' proposed solution for mutation design was discussed in Chapter 4. Chapter 5 presented the Epsilon Mutator, which is the implementation of the solution for mutating program models of MDE languages. Finally, the thesis put forward an intensive empirical mutation analysis to evaluate the proposed mutation design solution using Epsilon Mutator against programs written in the ATL and EOL languages.

The remainder of this chapter is divided into two main sections. Section 7.1 gives a summary of the thesis contributions and Section 7.2 outlines opportunities for future work.

7.1 Thesis Contributions

The main contributions of this thesis are the creation of a novel systematic design approach for mutation operators for MDE languages, the AMOs, and its evaluation with focus on feasibility. The following sub-sections present each of these in some detail.

7.1.1 Systematic Mutation Design

The objective of the thesis was to derive an mutation operator design for the rapidly emerging model driven engineering languages. This has been achieved by considering the metamodeling potential of MDE, in which metamodels (models of languages) are built on common metamodeling languages such as MOF, and the conformance constraints defined in those metamodels. In order to make this simple and applicable to different modeling technologies, the thesis defined a generic meta-metamodel to derive manually and systematically a set of abstract mutation operators for model driven engineering languages, considering addition, deletion and replacement mutation actions. An example of using the design approach and adapting it to Ecore was performed.

7.1.2 AMOs

The proposed mutation operator design for MDE languages included the definition of a set of abstract metamodel-agnostic mutation operators to mutate models of MDE programs. In an effort to make those operators as effective as possible, conformance, equality and definition constraints were integrated into the operators as preconditions. For example, constraints related to conformance were enforced with respect to the compatibility and multiplicity of entities and properties defined in a metamodel. In terms of compatibility, only values of compatible types were allowed to be used for producing mutants. In terms of multiplicity, the lower and upper bounds of values that a property of an entity can represent were used to control mutant production in a way that the limit bounds were not violated when the property was mutated with values (mainly when executing addition and deletion operators). An extension of the abstract operators was demonstrated for the Ecore metamodeling architecture, in which further constraints related to inheritance, multiple inheritance and data types were given.

The operators were organized into two main categories according to type and multiplicity of properties: single-valued properties and multi-valued properties. Each category contained operators for the addition, deletion and replacement of data, and each operator had a set of suitable preconditions. Examples of instantiating the AMOs over a simple metamodel that conformed to Ecore was demonstrated to show that preconditions are checked and verified before the operator application. In order to make the abstract operators more applicable and instantiable to different metamodeling technologies, AMOs were integrated into the EMU mutation injector, which is a prototype of a language dedicated to model mutation, that validates preconditions internally and prevents the generation of useless (invalid and equivalent) mutants.

7.1.3 Evaluation

In order to evaluate the proposed solution, an thorough empirical mutation analysis was conducted over several non-trivial ATL and EOL candidate programs. The AMOs were instantiated systematically over ATL and EOL metamodels to create concrete mutation operators by going through all modeling concepts in the metamodels and applying each abstract mutation operator on each concept defined in the metamodel. The implementation of CMOs was specified as: EMU programs mutating all models of candidate programs to produce mutants that were then executed against plausible testing models.

The results of the evaluation showed that the proposed mutation design approach and the AMOs were beneficial. First, the AMOs, which are the result of applying the mutation design approach, were instantiated over ATL and EOL metamodels to create concrete mutation operators. Many ATL concrete mutation operators targeted ATL concepts that are already defined for ATL in the literature.

In addition, the designed concrete mutation operators were found to be effective mutations that fulfilled this thesis' objectives in terms of useful and useless mutants. For the latter, the number of invalid mutants that were prevented from being generated was 287 out of 3,782 (around 7.6 percent) for ATL mutants, and 1,255 out of 6,691 (around 18.7 percent) for EOL mutants. Both, thus, reported low numbers of invalid mutants. Targeting low numbers of invalid mutants has a positive impact on overall mutation analysis since these are omitted from the execution process of mutation testing.

In regard to useful mutants, such as live and non-trivial mutants, the results of the experiment of ATL mutants showed that 80 were found to be live and 483 to be non-trivially killed, out of a total of 3,495. For EOL, 1,126 were found to be live and 787 were found to be non-trivially killed, out of 5,436. Both results indicate that AMOs are able to generate concrete operators that can be useful in producing live and non-trivially killed mutants.

7.2 Future Work

This doctoral research was bounded by time constraints that did not allow the investigation of some promising directions of inquiry. These future research opportunities are explored in the following sections.

7.2.1 Further experiments and evaluations

An area of the thesis that can be extended is evaluation. Although the evaluation was conducted thoroughly by adopting a quality process, there is still a number of elements that can be improved upon. One of these elements is candidate programs. Since the experiment included concrete mutation operators (provided in Section 6.5.4) for ATL and EOL, there is always an opportunity to re-run these operators against more candidate programs of ATL and EOL with real test models. This extension can help in more evaluation over the used concrete operators and ultimately AMOs and their application.

The empirical mutation analysis conducted offers two areas that are open for future improvement. One is related to invalid mutants. Operators that produce invalid mutants may too often require re-implementation. Therefore, there is another opportunity to re-define those mutations, and repeat the experiment mutation analysis to evaluate those mutation operators. In addition, the experiment on ATL programs conducted here covered OCL language concepts that are directly referred to from within ATL language concepts. Hence, a complete coverage of OCL is another available opportunity for future work.

A further opportunity from an experimental perspective is performing mutation testing to mutate programs of other MDE languages. Potentially good candidates for this, although the list is not exhaustive, are the Epsilon Validation Language (EVL) and the Epsilon Transformation Language (ETL), as they have already constructed metamodels [107]. This is made easier by using abstract mutation operators and EMU, which can be used to implement any concrete mutation operators generated by AMOs.

A final experiment-related future opportunity is to perform mutation testing to models of programs of different modeling technologies. Examples of such technologies include HTML, XSLT, SysML, Simulink and others. Test developers who want to mutate models of these technologies need to extend the Mutant Integration Layer (mentioned in Section 5.1) and the Epsilon Model Connectivity Layer [18] to run EMU, which implements AMOs and is currently intractable with models of those modeling technologies.

7.2.2 Additional constraints

This thesis has used mutation operator implementations generated from abstract syntaxes. In fact, the mutation design solution and AMOs, along with their constraints (presented in Section 4.1), are based on abstract syntax alone; this may be viewed as a limitation. Since metamodels can also be integrated with constraints expressed in validation languages, such as OCL and EVL, there is an opportunity to include validation expressions of metamodels in the process of producing mutants. This can be achieved just prior to applying the mutation operator of a modeling concept, and by checking whether the intended mutant does not break the validation expressions.

7.2.3 Independent EMU syntax

The objective of the current implementation of EMU is a proof of concept prototype language dedicated to model mutation. As such, EMU is built on the Epsilon Pattern Language with minor modifications to its execution engine. EMU uses EPL syntax and some of its language concepts for model mutation purposes. Consequently, it lacks independent syntax (abstract and concrete). Thus, there is the opportunity to develop a full implementation of EMU in the future and to integrate it into the Epsilon Platform, which consists of a family of domain specific languages for maintaining models of different metamodeling technologies.

Appendix A

Mutation Operators

A.1 ATL Mutation Operators

```
operation isComparisonOperator(op:String):Boolean{
  if(op=">" or op=">=" or op="<" or op="<=" or op="=" or op="<>")
   return true;
  return false;
}
operation isLogicalOperator(op:String):Boolean{
  if(op="and" or op="or" or op="xor")
   return true;
  return false;
}
//Impl1:change the comparison condition to gt
@action replace
@property condition
pattern change_cond_operatorcall_to_gt
instance:IfStat in:IfStat.all.select(e|e.condition.isDefined() and e.condition.isTypeOf(
    OperatorCallExp))
guard:isComparisonOperator(instance.condition.operationName) and instance.condition.
    operationName<>">"{
do { var call_exp = OperatorCallExp.createInstance();
  call_exp.source = instance.condition.source;
  call_exp.arguments.addAll(instance.condition.arguments);
  call_exp.operationName = ">";
  instance.condition = call_exp;
}}
//Impl2:change the comparison condition to greater than or equal
@action replace
@property condition
pattern change_cond_of_operatorcallexp_to_gte
instance:IfStat in:IfStat.all.select(e|e.condition.isDefined() and e.condition.isTypeOf(
    OperatorCallExp))
guard:isComparisonOperator(instance.condition.operationName) and instance.condition.
    operationName<>">="{
do { var call_exp = OperatorCallExp.createInstance();
  call_exp.source = instance.condition.source;
  call_exp.arguments.addAll(instance.condition.arguments);
  call_exp.operationName = ">=";
  instance.condition = call_exp;
}}
//Impl3:change the comparison condition less than
@action replace
@property condition
pattern change_cond_of_operatorcallexp_to_lt
instance:IfStat in:IfStat.all.select(e|e.condition.isDefined() and e.condition.isTypeOf(
    OperatorCallExp))
guard:isComparisonOperator(instance.condition.operationName) and instance.condition.
    operationName<>"<"{
```

```
do { var call_exp = OperatorCallExp.createInstance();
  call_exp.source = instance.condition.source;
  call_exp.arguments.addAll(instance.condition.arguments);
  call_exp.operationName = "<";
  instance.condition = call_exp;
}}
//Impl4:change the comparison condition less than or equal
@action replace
@property condition
pattern change_cond_of_operatorcallexp_to_lte
instance:IfStat in:IfStat.all.select(e|e.condition.isDefined() and e.condition.isTypeOf(
     OperatorCallExp))
guard:isComparisonOperator(instance.condition.operationName) and instance.condition.
     operationName<>"<="{
do { var call_exp = OperatorCallExp.createInstance();
  call_exp.source = instance.condition.source;
  call_exp.arguments.addAll(instance.condition.arguments);
  call_exp.operationName = "<=";
  instance.condition = call_exp;
}}
//Impl5:change the comparison condition to equal
@action replace
@property condition
pattern change_cond_of_operatorcallexp_to_equal
instance:IfStat in:IfStat.all.select(e|e.condition.isDefined() and e.condition.isTypeOf(
     OperatorCallExp))
guard:isComparisonOperator(instance.condition.operationName) and instance.condition.
     operationName<>"="{
do { var call_exp = OperatorCallExp.createInstance();
  call_exp.source = instance.condition.source;
  call_exp.arguments.addAll(instance.condition.arguments);
  call_exp.operationName = "=";
  instance.condition = call_exp;
}}
//Impl6:change the comparison condition to not equal
@action replace
@property condition
pattern change_cond_of_operatorcallexp_to_not_equal
instance:IfStat in:IfStat.all.select(e|e.condition.isDefined() and e.condition.isTypeOf(
     OperatorCallExp))
guard:isComparisonOperator(instance.condition.operationName) and instance.condition.
     operationName<>"<>"{
do { var call_exp = OperatorCallExp.createInstance();
  call_exp.source = instance.condition.source;
  call_exp.arguments.addAll(instance.condition.arguments);
  call_exp.operationName = "<>";
  instance.condition = call_exp;
}}
//Impl7:replace the logical expression to not logical
@action replace
@property condition
pattern negative_state_of_operatorcallexp_using_not
instance:IfStat in:IfStat.all.select(e|e.condition.isDefined())
guard:not (instance.condition.isTypeOf(OperatorCallExp) and instance.condition.operationName =
      "not"){
do { var call_exp = OperatorCallExp.createInstance();
  call_exp.source = instance.condition;
  call_exp.operationName = "not";
  instance.condition = call_exp;
}}
//Impl8:remove not logical expression
@action replace
@property condition
pattern remove_not_from_operatorCallExp
instance:IfStat in:IfStat.all.select(e|e.condition.isDefined() and e.condition.isTypeOf(
     OperatorCallExp))
guard:instance.condition.operationName = "not"{
do { instance.condition = instance.condition.source;
}}
//Impl9:replace the logical expression to "and"
@action replace
@property condition
pattern change_operationname_of_operatorcallexp_to_and
instance:IfStat in:IfStat.all.select(e|e.condition.isDefined() and e.condition.isTypeOf(
     OperatorCallExp))
```

```
guard:isLogicalOperator(instance.condition.operationName) and instance.condition.operationName
     <>"and"{
do { var call_exp = OperatorCallExp.createInstance();
  call_exp.source = instance.condition.source;
  call_exp.arguments.addAll(instance.condition.arguments);
  call_exp.operationName = "and";
  instance.condition = call_exp;
}}
//Impl10:replace the logical expression to "or" logical
@action replace
@property condition
pattern change_operationname_of_operatorcallexp_to_or
instance:IfStat in:IfStat.all.select(e|e.condition.isDefined() and e.condition.isTypeOf(
     OperatorCallExp))
guard:isLogicalOperator(instance.condition.operationName) and instance.condition.operationName
     <>"or"{
do { var call_exp = OperatorCallExp.createInstance();
  call_exp.source = instance.condition.source;
  call_exp.arguments.addAll(instance.condition.arguments);
  call_exp.operationName = "or";
  instance.condition = call_exp;
}}
//Impl11:replace the logical expression to "xor" logical
@action replace
@property condition
pattern change_operationname_of_operatorcallexp_to_xor
instance:IfStat in:IfStat.all.select(e|e.condition.isDefined() and e.condition.isTypeOf(
     OperatorCallExp))
guard:isLogicalOperator(instance.condition.operationName) and instance.condition.operationName
     <>"xor"{
do { var call_exp = OperatorCallExp.createInstance();
  call_exp.source = instance.condition.source;
  call_exp.arguments.addAll(instance.condition.arguments);
  call_exp.operationName = "xor";
  instance.condition = call_exp;
}}
```

Listing A.1: `CMO-single-REP(IfStat condition)`

```
//Impl1:repalce the type of RuleVariableDeclaration
//Impl1.1:replace the type of primitive type to StringType
@action replace
@property type
pattern change_ruleVariableDeclaration_type2StringType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isTypeOf(
     RuleVariableDeclaration))
guard:instance.type.isKindOf(Primitive) and not instance.type.isTypeOf(StringType){
do { instance.type = StringType.createInstance();
}}
//Impl1.2:replace the type of primitive to BooleanType
@action replace
@property type
pattern change_ruleVariableDeclaration_type2BooleanType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isTypeOf(
     RuleVariableDeclaration))
guard:instance.type.isKindOf(Primitive) and not instance.type.isTypeOf(BooleanType){
do { instance.type = BooleanType.createInstance();
}}
//Impl1.3:replace the type of primitive to IntegerType
@action replace
@property type
pattern change_ruleVariableDeclaration_type2IntegerType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isTypeOf(
     RuleVariableDeclaration))
guard:instance.type.isKindOf(Primitive) and not instance.type.isTypeOf(IntegerType){
do { instance.type = IntegerType.createInstance();
}}
//Impl1.4:replace the type of primitive to RealType
@action replace
@property type
pattern change_ruleVariableDeclaration_type2RealType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isTypeOf(
     RuleVariableDeclaration))
guard:instance.type.isKindOf(Primitive) and not instance.type.isTypeOf(RealType){
```

```
do { instance.type = RealType.createInstance();
}}
//Impl1.5:replace the type of collection to BagType
@action replace
@property type
pattern change_ruleVariableDeclaration_type2BagType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isTypeOf(
     RuleVariableDeclaration))
guard:instance.type.isKindOf(CollectionType) and not instance.type.isTypeOf(BagType){
do { instance.type = BagType.createInstance();
}}
//Impl1.6:replace the type of collection to OrderedSetType
@action replace
@property type
pattern change_ruleVariableDeclaration_type2OrderedSetType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isTypeOf(
     RuleVariableDeclaration))
guard:instance.type.isKindOf(CollectionType) and not instance.type.isTypeOf(OrderedSetType){
do { instance.type = OrderedSetType.createInstance();
}}
//Impl1.7:replace the type of collection to SequenceType
@action replace
@property type
pattern change_ruleVariableDeclaration_type2SequenceType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isTypeOf(
     RuleVariableDeclaration))
guard:instance.type.isKindOf(CollectionType) and not instance.type.isTypeOf(SequenceType){
do { instance.type = SequenceType.createInstance();
}}
//Impl1.8:replace the type of collection to SetType
@action replace
@property type
pattern change_ruleVariableDeclaration_type2SetType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isTypeOf(
     RuleVariableDeclaration))
guard:instance.type.isKindOf(CollectionType) and not instance.type.isTypeOf(SetType){
do { instance.type = SetType.createInstance();
}}
//Impl1.9:replace the type to different OclModelElement:
// change name
@action replace
@property type
@role instance
pattern change_ruleVariableDeclaration_type_change_name
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isTypeOf(
     RuleVariableDeclaration))
guard:instance.type.isTypeOf(OclModelElement),
otherType:OclModelElement in:OclModelElement.all.select(e|e<>instance.type)
guard:instance.type.name<>otherType.name and instance.type.`model`.name = otherType.`model`.
     name{
do { var new_type = OclModelElement.createInstance();
  new_type.`model` = instance.type.`model`;
  new_type.name = otherType.name;
  instance.type = new_type;
}}
//Impl1.10:replace the type to different OclModelElement:
// change model
@action replace
@property type
@role instance
pattern change_ruleVariableDeclaration_type_change_model
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isTypeOf(
     RuleVariableDeclaration))
guard:instance.type.isTypeOf(OclModelElement),
otherType:OclModelElement in:OclModelElement.all.select(e|e<>instance.type)
guard:instance.type.name = otherType.name and instance.type.`model`.name<>otherType.`model`.
     name{
do { var new_type = OclModelElement.createInstance();
  new_type.`model` = otherType.`model`;
  new_type.name = instance.type.name;
  instance.type = new_type;
}}
//Impl2:repalce the type of PatternElement
//Impl2.1:replace the type of primitive type to StringType
@action replace
```

```
@property type
pattern change_PatternElement_type2StringType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(PatternElement))
guard:instance.type.isKindOf(Primitive) and not instance.type.isTypeOf(StringType){
do { instance.type = StringType.createInstance();
}}
//Impl2.2:replace the type of primitive to BooleanType
@action replace
@property type
pattern change_PatternElement_type2BooleanType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(PatternElement))
guard:instance.type.isKindOf(Primitive) and not instance.type.isTypeOf(BooleanType){
do { instance.type = BooleanType.createInstance();
}}
//Impl2.3:replace the type of primitive to IntegerType
@action replace
@property type
pattern change_PatternElement_type2IntegerType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(PatternElement))
guard:instance.type.isKindOf(Primitive) and not instance.type.isTypeOf(IntegerType){
do { instance.type = IntegerType.createInstance();
}}
//Impl2.4:replace the type of primitive to RealType
@action replace
@property type
pattern change_PatternElement_type2RealType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(PatternElement))
guard:instance.type.isKindOf(Primitive) and not instance.type.isTypeOf(RealType){
do { instance.type = RealType.createInstance();
}}
//Impl2.5:replace the type of collection to BagType
@action replace
@property type
pattern change_PatternElement_type2BagType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(PatternElement))
guard:instance.type.isKindOf(CollectionType) and not instance.type.isTypeOf(BagType){
do { instance.type = BagType.createInstance();
}}
//Impl2.6:replace the type of collection to OrderedSetType
@action replace
@property type
pattern change_PatternElement_type2OrderedSetType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(PatternElement))
guard:instance.type.isKindOf(CollectionType) and not instance.type.isTypeOf(OrderedSetType){
do { instance.type = OrderedSetType.createInstance();
}}
//Impl2.7:replace the type of collection to SequenceType
@action replace
@property type
pattern change_PatternElement_type2SequenceType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(PatternElement))
guard:instance.type.isKindOf(CollectionType) and not instance.type.isTypeOf(SequenceType){
do { instance.type = SequenceType.createInstance();
}}
//Impl2.8:replace the type of collection to SetType
@action replace
@property type
pattern change_PatternElement_type2SetType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(PatternElement))
guard:instance.type.isKindOf(CollectionType) and not instance.type.isTypeOf(SetType){
do { instance.type = SetType.createInstance();
}}
//Impl2.9:replace the type to different OclModelElement:
// change name
@action replace
@property type
@role instance
pattern change_PatternElement_type_change_name
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(PatternElement))
guard:instance.type.isTypeOf(OclModelElement),
otherType:OclModelElement in:OclModelElement.all.select(e|e<>instance.type)
guard:instance.type.name<>otherType.name and instance.type.`model`.name = otherType.`model`.
    name{
do { var new_type = OclModelElement.createInstance();
  new_type.`model` = instance.type.`model`;
  new_type.name = otherType.name;
  instance.type = new_type;
}}
//Impl2.10: replace the type to different OclModelElement:
// change model
@action replace
@property type
@role instance
pattern change_PatternElement_type_change_model
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(PatternElement))
guard:instance.type.isTypeOf(OclModelElement),
otherType:OclModelElement in:OclModelElement.all.select(e|e<>instance.type)
guard:instance.type.name = otherType.name and instance.type.`model`.name<>otherType.`model`.
    name{
do { var new_type = OclModelElement.createInstance();
  new_type.`model` = otherType.`model`;
  new_type.name = instance.type.name;
  instance.type = new_type;
}}
//Impl3: repalce the type of Parameter
//Impl3.1: replace the type of primitive type to StringType
@action replace
@property type
pattern change_Parameter_type2StringType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Parameter))
guard:instance.type.isKindOf(Primitive) and not instance.type.isTypeOf(StringType){
do { instance.type = StringType.createInstance();
}}
//Impl3.2: replace the type of primitive to BooleanType
@action replace
@property type
pattern change_Parameter_type2BooleanType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Parameter))
guard:instance.type.isKindOf(Primitive) and not instance.type.isTypeOf(BooleanType){
do { instance.type = BooleanType.createInstance();
}}
//Impl3.3: replace the type of primitive to IntegerType
@action replace
@property type
pattern change_Parameter_type2IntegerType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Parameter))
guard:instance.type.isKindOf(Primitive) and not instance.type.isTypeOf(IntegerType){
do { instance.type = IntegerType.createInstance();
}}
//Impl3.4: replace the type of primitive to RealType
@action replace
@property type
pattern change_Parameter_type2RealType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Parameter))
guard:instance.type.isKindOf(Primitive)
and not instance.type.isTypeOf(RealType){
do { instance.type = RealType.createInstance();
}}
//Impl3.5: replace the type of collection to BagType
@action replace
@property type
pattern change_Parameter_type2BagType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Parameter))
guard:instance.type.isKindOf(CollectionType) and not instance.type.isTypeOf(BagType){
do { instance.type = BagType.createInstance();
}}
//Impl3.6: replace the type of collection to OrderedSetType
@action replace
@property type
pattern change_Parameter_type2OrderedSetType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Parameter))
guard:instance.type.isKindOf(CollectionType) and not instance.type.isTypeOf(OrderedSetType){
do { instance.type = OrderedSetType.createInstance();
}}
//Impl3.7: replace the type of collection to SequenceType
@action replace
@property type
pattern change_Parameter_type2SequenceType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Parameter))
guard:instance.type.isKindOf(CollectionType) and not instance.type.isTypeOf(SequenceType){
```

```
do { instance.type = SequenceType.createInstance();
}}
//Impl3.8:replace the type of collection to SetType
@action replace
@property type
pattern change_Parameter_type2SetType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Parameter))
guard:instance.type.isKindOf(CollectionType) and not instance.type.isTypeOf(SetType){
do { instance.type = SetType.createInstance();
}}
//Impl3.9:replace the type to different OclModelElement:
// change name
@action replace
@property type
@role instance
pattern change_Parameter_type_change_name
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Parameter))
guard:instance.type.isTypeOf(OclModelElement),
otherType:OclModelElement in:OclModelElement.all.select(e|e<>instance.type)
guard:instance.type.name<>otherType.name and instance.type.`model`.name = otherType.`model`.
    name{
do { var new_type = OclModelElement.createInstance();
  new_type.`model` = instance.type.`model`;
  new_type.name = otherType.name;
  instance.type = new_type;
}}
//Impl3.10:replace the type to different OclModelElement:
// change model
@action replace
@property type
@role instance
pattern change_Parameter_type_change_model
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Parameter))
guard:instance.type.isTypeOf(OclModelElement),
otherType:OclModelElement in:OclModelElement.all.select(e|e<>instance.type)
guard:instance.type.name = otherType.name and instance.type.`model`.name<>otherType.`model`.
    name{
do { var new_type = OclModelElement.createInstance();
  new_type.`model` = otherType.`model`;
  new_type.name = instance.type.name;
  instance.type = new_type;
}}
//Impl4:repalce the type of Iterator
//Impl4.1:replace the type of primitive type to StringType
@action replace
@property type
pattern change_Iterator_type2StringType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Iterator))
guard:instance.type.isKindOf(Primitive) and not instance.type.isTypeOf(StringType){
do { instance.type = StringType.createInstance();
}}
//Impl4.2:replace the type of primitive to BooleanType
@action replace
@property type
pattern change_Iterator_type2BooleanType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Iterator))
guard:instance.type.isKindOf(Primitive) and not instance.type.isTypeOf(BooleanType){
do { instance.type = BooleanType.createInstance();
}}
//Impl4.3:replace the type of primitive to IntegerType
@action replace
@property type
pattern change_Iterator_type2IntegerType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Iterator))
guard:instance.type.isKindOf(Primitive) and not instance.type.isTypeOf(IntegerType){
do { instance.type = IntegerType.createInstance();
}}
//Impl4.4:replace the type of primitive to RealType
@action replace
@property type
pattern change_Iterator_type2RealType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Iterator))
guard:instance.type.isKindOf(Primitive) and not instance.type.isTypeOf(RealType){
do { instance.type = RealType.createInstance();
}}
```

```
//Impl4.5:replace the type of collection to BagType
@action replace
@property type
pattern change_Iterator_type2BagType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Iterator))
guard:instance.type.isKindOf(CollectionType) and not instance.type.isTypeOf(BagType){
do { instance.type = BagType.createInstance();
}}
//Impl4.6:replace the type of collection to OrderedSetType
@action replace
@property type
pattern change_Iterator_type2OrderedSetType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Iterator))
guard:instance.type.isKindOf(CollectionType) and not instance.type.isTypeOf(OrderedSetType){
do { instance.type = OrderedSetType.createInstance();
}}
//Impl4.7:replace the type of collection to SequenceType
@action replace
@property type
pattern change_Iterator_type2SequenceType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Iterator))
guard:instance.type.isKindOf(CollectionType) and not instance.type.isTypeOf(SequenceType){
do { instance.type = SequenceType.createInstance();
}}
//Impl4.8:replace the type of collection to SetType
@action replace
@property type
pattern change_Iterator_type2SetType
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Iterator))
guard:instance.type.isKindOf(CollectionType) and not instance.type.isTypeOf(SetType){
do { instance.type = SetType.createInstance();
}}
//Impl4.9:replace the type to different OclModelElement:change name
@action replace
@property type
@role instance
pattern change_Iterator_type_change_name
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Iterator))
guard:instance.type.isTypeOf(OclModelElement),
otherType:OclModelElement in:OclModelElement.all.select(e|e<>instance.type)
guard:instance.type.name<>otherType.name and instance.type.`model`.name = otherType.`model`.
    name{
do { var new_type = OclModelElement.createInstance();
  new_type.`model` = instance.type.`model`;
  new_type.name = otherType.name;
  instance.type = new_type;
}}
//Impl4.10:replace the type to different OclModelElement:change model
@action replace
@property type
@role instance
pattern change_Iterator_type_change_model
instance:VariableDeclaration in:VariableDeclaration.all.select(e|e.isKindOf(Iterator))
guard:instance.type.isTypeOf(OclModelElement),
otherType:OclModelElement in:OclModelElement.all.select(e|e<>instance.type)
guard:instance.type.name = otherType.name and instance.type.`model`.name<>otherType.`model`.
    name{
do { var new_type = OclModelElement.createInstance();
  new_type.`model` = otherType.`model`;
  new_type.name = instance.type.name;
  instance.type = new_type;
}}
```

Listing A.2: `CMO-Single-REP(VariableDeclaration type)`

```
//Impl1.1:change paramenters of OclOperation:remove one
// paramenter at a time
@action delete
@property parameters
@role instance
pattern operation_remove_one_parameter
instance:Operation in:Operation.all.select(e|e.parameters.size()>=1),
param:Parameter from:instance.parameters{
do { instance.parameters.remove(param);
}}
```

Listing A.3: **CMO-multiple-DEL(Operation parameters)**

```
// Impl 1: replace a MathcedRule with unique LazyMatchedRule
@action replace
@property elements
@role instance
pattern matchedrule2uiquelazyrule_replacement
instance:Module,
rule:MatchedRule from:instance.elements.select(e|e.isTypeOf(MatchedRule))
guard:(rule.children.isUndefined() or rule.children.size()=0) and (rule.superRule.isUndefined
    () or rule.superRule.size()=0){
do { var lazy = LazyMatchedRule.createInstance();
  lazy.isUnique = true;
  lazy.inPattern = rule.inPattern;
  lazy.outPattern = rule.outPattern;
  lazy.actionBlock = rule.actionBlock;
  lazy.variables = rule.variables;
  lazy.name = rule.name;
  instance.elements.remove(rule);
  instance.elements.add(lazy);
}}
// Impl 2: replace MatchedRule with not unique LazyMatchedRule
@action replace
@property elements
@role instance
pattern matchedrule2lazyrule_replacement
instance:Module,
rule:MatchedRule from:instance.elements.select(e|e.isTypeOf(MatchedRule))
guard:(rule.children.isUndefined() or rule.children.size()=0) and (rule.superRule.isUndefined
    () or rule.superRule.size()=0){
do { var lazy = LazyMatchedRule.createInstance();
  lazy.isUnique = false;
  lazy.inPattern = rule.inPattern;
  lazy.outPattern = rule.outPattern;
  lazy.actionBlock = rule.actionBlock;
  lazy.variables = rule.variables;
  lazy.name = rule.name;
  instance.elements.remove(rule);
  instance.elements.add(lazy);
}}
// Impl 3: replace LazyMatchedRule with MatchedRule
@action replace
@property elements
@role instance
pattern lazyrule2matchedrule_replacement
instance:Module,
rule:LazyMatchedRule from:instance.elements.select(e|e.isTypeOf(LazyMatchedRule))
guard:(rule.children.isUndefined() or rule.children.size=0) and (rule.superRule.isUndefined()
    or rule.superRule.size=0){
do { var new_rule = MatchedRule.createInstance();
  new_rule.inPattern = rule.inPattern;
  new_rule.outPattern = rule.outPattern;
  new_rule.actionBlock = rule.actionBlock;
  new_rule.variables = rule.variables;
  new_rule.name = rule.name;
  instance.elements.remove(rule);
  instance.elements.add(new_rule);
}}
```

Listing A.4: **CMO-multiple-REP(Module elements)**

```
@action replace
@property children
@role instance
pattern replace_one_child_at_a_time_with_different_matchedrule
instance:MatchedRule,
child:MatchedRule from: instance.children,
new_child:MatchedRule in:MatchedRule.all.select(e| (e <> instance) or (instance.children.
    isDefined() and instance.children.includes(e))){
do { instance.children.remove(child);
  instance.children.add(new_child);
}}
```

Listing A.5: **CMO-multiple-REP(MatchedRule children)**

```
// Impl 1: replacing with new string
@action replace
@property propertyName
pattern propertyName_changed_to_new
instance:Binding {
do { instance.propertyName = instance.propertyName + "_";
}}
// Impl 2: replacing with another name copied over from  an existing one
@action replace
@property propertyName
@role instance
pattern propertyName_copied_over_from_another_binding
instance:Binding,
otherBinding:Binding from: instance.outPatternElement.bindings.select(e|e <> instance){
do { instance.propertyName = otherBinding.propertyName;
}}
```

Listing A.6: **CMO-single-REP(Binding propertyName)**

```
// Impl. 1: change a source name with another one
@action replace
@property source
pattern replace_source_from_another_one
instance:BindingStat in:BindingStat.all.select(e|e.source.isDefined() and e.source.isTypeOf(
     NavigationOrAttributeCallExp)
and e.source.source.isDefined() and e.source.source.isTypeOf(NavigationOrAttributeCallExp)){
do { var sources = BindingStat.all.select(e| e.source.isDefined() and e.source.isTypeOf(
     NavigationOrAttributeCallExp)
  and e.source.source.isDefined() and e.source.source.isTypeOf(NavigationOrAttributeCallExp)
  and e.source.name <> instance.source.name).collect(e|e.source.name).asSet();
  var new_source = NavigationOrAttributeCallExp.createInstance();
  new_source.name = sources.random();
  new_source.source = instance.source.source;
  instance.source = new_source;
}}
// Impl. 2: change a source to another one obtained from called source
@action replace
@property source
pattern replace_source_of_specific_father_source
instance:BindingStat in:BindingStat.all.select(e|e.source.isDefined() and e.source.isTypeOf(
     NavigationOrAttributeCallExp)
and e.source.source.isDefined() and e.source.source.isTypeOf(NavigationOrAttributeCallExp)){
do { var sources = BindingStat.all.select(e| e.source.isDefined() and e.source.isTypeOf(
     NavigationOrAttributeCallExp)
  and e.source.source.isDefined() and e.source.source.isTypeOf(NavigationOrAttributeCallExp)
  and e.source.source.name <> instance.source.source.name).collect(e|e.source.source.name).
     asSet();
  var new_source = NavigationOrAttributeCallExp.createInstance();
  new_source.name = sources.random();
  new_source.source = instance.source.source;
  instance.source = new_source;
}}
// Impl 3: change a source to another one by omitting a middle NavigationOrAttributeCallExp
@action replace
@property source
pattern replace_omitting_middle_source
instance:BindingStat in:BindingStat.all.select(e|e.source.isDefined() and e.source.isTypeOf(
     NavigationOrAttributeCallExp)
and e.source.source.isDefined() and e.source.source.isTypeOf(NavigationOrAttributeCallExp)){
do { var new_source = NavigationOrAttributeCallExp.createInstance();
  new_source.name = instance.source.name;
  new_source.source = instance.source.source.source;
  instance.source = new_source;
}}
```

Listing A.7: **CMO-single-REP(BindingStat source)**

```
// Impl 1: replace an operation with a new attribute
@action replace
@property feature
pattern replace_oclOperation_with_oclAttribute
instance:OclFeatureDefinition in:OclFeatureDefinition.all.select(e|e.feature.isTypeOf(
    Operation))
guard:instance.feature.parameters.size = 0 {
do { var attribute = Attribute.createInstance();
  attribute.name = instance.feature.name;
  attribute.initExpression = instance.feature.body;
  attribute.type = instance.feature.returnType;
  instance.feature = attribute;
}}
// Impl 2: replace an attribute with an operation
@action replace
@property feature
pattern replace_oclAttribute_with_oclOperation
instance:OclFeatureDefinition in:OclFeatureDefinition.all.select(e|e.feature.isTypeOf(
    Attribute)){
do { var operation_ = Operation.createInstance();
  operation_.name = instance.feature.name;
  operation_.body = instance.feature.initExpression;
  operation_.returnType = instance.feature.type;
  instance.feature = operation_;
}}
```

Listing A.8: `CMO-single-REP(OclFeatureDefinition feature)`

A.2 EOL Mutation Operators

```
@action delete
@property parameters
@role instance
pattern delete_one_parameter
instance:OperationDefinition,
param:FormalParameterExpression from:instance.parameters{
do { instance.parameters.remove(param);
}}
```

Listing A.9: **CMO-multiple-DEL(OperationDefinition parameters)**

```
operation isSelfOrResult(instance:Any):Boolean {
if(instance.eContainer.isTypeOf(OperationDefinition)
and (instance.name.name = "self" or instance.name.name = "_result")){
return true;
}
return false;
}
// 1.0: Primitive types: RealType from/to IntegerType
// 1.1: RealType from/to IntegerType
// 1.1.1: RealType to IntegerType
@action replace
@property resolvedType
pattern replace_RealType_IntegerType_1
instance:Expression
in:Expression.all.select(e|e.resolvedType.isDefined())
guard:instance.resolvedType.isTypeOf(RealType) and instance.isKindOf(
     VariableDeclarationExpression) and not isSelfOrResult(instance){
do { instance.resolvedType = IntegerType.createInstance();
}}
// 1.1.2: RealType from IntegerType
@action replace
@property resolvedType
pattern replace_RealType_IntegerType_2
instance:Expression
in:Expression.all.select(e|e.resolvedType.isDefined())
guard:instance.isKindOf(VariableDeclarationExpression) and instance.resolvedType.isTypeOf(
     IntegerType) and not isSelfOrResult(instance){
do { instance.resolvedType = RealType.createInstance();
}}
// 2.0: Collection types
// 2.1: SequenceType from/to OrderedSetType
// 2.1.1: SequenceType to OrderedSetType
@action replace
@property resolvedType
pattern replace_SequenceType_OrderedSetType_1
instance:Expression
in:Expression.all.select(e|e.resolvedType.isDefined())
guard:instance.resolvedType.isTypeOf(SequenceType) and instance.isKindOf(
     VariableDeclarationExpression) and not isSelfOrResult(instance){
do { instance.resolvedType = OrderedSetType.createInstance();
}}
// 2.1.2: SequenceType from OrderedSetType
@action replace
@property resolvedType
pattern replace_SequenceType_OrderedSetType_2
instance:Expression
in:Expression.all.select(e|e.resolvedType.isDefined())
guard:instance.resolvedType.isTypeOf(OrderedSetType) and instance.isKindOf(
     VariableDeclarationExpression) and not isSelfOrResult(instance){
do { instance.resolvedType = SequenceType.createInstance();
}}
// 2.2: SequenceType from/to BagType
// 2.2.1: SequenceType to BagType
@action replace
@property resolvedType
pattern replace_SequenceType_BagType_1
instance:Expression
in:Expression.all.select(e|e.resolvedType.isDefined())
guard:instance.resolvedType.isTypeOf(SequenceType) and instance.isKindOf(
     VariableDeclarationExpression) and not isSelfOrResult(instance){
```

```
do { instance.resolvedType = BagType.createInstance();
}}
// 2.2.2: SequenceType from BagType
@action replace
@property resolvedType
pattern replace_SequenceType_BagType_2
instance:Expression
in:Expression.all.select(e|e.resolvedType.isDefined())
guard:instance.resolvedType.isTypeOf(BagType) and instance.isKindOf(
     VariableDeclarationExpression) and not isSelfOrResult(instance){
do { instance.resolvedType = SequenceType.createInstance();
}}
// 2.3: SetType from/to  OrderedSetType
// 2.3.1: SetType to  OrderedSetType
@action replace
@property resolvedType
pattern replace_SetType_OrderedSetType_1
instance:Expression
in:Expression.all.select(e|e.resolvedType.isDefined())
guard:instance.resolvedType.isTypeOf(SetType) and instance.isKindOf(
     VariableDeclarationExpression) and not isSelfOrResult(instance){
do { instance.resolvedType = OrderedSetType.createInstance();
}}
// 2.3.2: SetType from OrderedSetType
@action replace
@property resolvedType
pattern replace_SetType_OrderedSetType_2
instance:Expression
in:Expression.all.select(e|e.resolvedType.isDefined())
guard:instance.resolvedType.isTypeOf(OrderedSetType) and instance.isKindOf(
     VariableDeclarationExpression) and not isSelfOrResult(instance){
do { instance.resolvedType = SetType.createInstance();
}}
// 2.4: SetType from/to  BagType
// 2.4.1: SetType to  BagType
@action replace
@property resolvedType
pattern replace_SetType_BagType_1
instance:Expression
in:Expression.all.select(e|e.resolvedType.isDefined())
guard:instance.resolvedType.isTypeOf(SetType) and instance.isKindOf(
     VariableDeclarationExpression) and not isSelfOrResult(instance){
do { instance.resolvedType = BagType.createInstance();
}}
// 2.4.2: SetType from BagType
@action replace
@property resolvedType
pattern replace_SetType_BagType_2
instance:Expression
in:Expression.all.select(e|e.resolvedType.isDefined())
guard:instance.resolvedType.isTypeOf(BagType) and instance.isKindOf(
     VariableDeclarationExpression) and not isSelfOrResult(instance){
do { instance.resolvedType = SetType.createInstance();
}}
// 2.5: OrderedSetType from/to  BagType
// 2.5.1: OrderedSetType to  BagType
@action replace
@property resolvedType
pattern replace_OrderedSetType_BagType_1
instance:Expression
in:Expression.all.select(e|e.resolvedType.isDefined())
guard:instance.resolvedType.isTypeOf(OrderedSetType) and instance.isKindOf(
     VariableDeclarationExpression) and not isSelfOrResult(instance){
do { instance.resolvedType = BagType.createInstance();
}}
// 2.5.2: OrderedSetType from BagType
@action replace
@property resolvedType
pattern replace_OrderedSetType_BagType_2
instance:Expression in:Expression.all.select(e|e.resolvedType.isDefined())
guard:instance.resolvedType.isTypeOf(BagType) and instance.isKindOf(
     VariableDeclarationExpression) and not isSelfOrResult(instance){
do { instance.resolvedType = OrderedSetType.createInstance();
}}
```

Listing A.10: **CMO-single-REP(Expression resolvedType)**

```
// 1.0: replacing if statement and while statement
// 1.1: replacing if statement with a new while statement
@action replace
@property statements
@role instance
pattern replace_if_with_while
instance:Block,
if_stat:IfStatement from:instance.statements guard:if_stat.elseIfBodies.size=0 and if_stat.
    elseBody=null {
do { var new_stat = WhileStatement.createInstance();
new_stat.condition = if_stat.condition;
new_stat.body=if_stat.ifBody;
instance.statements.remove(if_stat);
instance.statements.add(new_stat);
}}
// 1.2: replacing while statement with new if statement
@action replace
@property statements
@role instance
pattern replace_while_with_if
instance:Block,
while_stat:WhileStatement from:instance.statements{
do { var new_stat = IfStatement.createInstance();
new_stat.condition = while_stat.condition;
new_stat.ifBody = while_stat.body;
instance.statements.remove(while_stat);
instance.statements.add(new_stat);
}}
// 2.0: replacing expression statement with return statement
// 2.1: replacing the last expression statement of an operation
// definition with return statement
@action replace
@property statements
pattern replace_exp_stat_with_return_stat
instance:Block in:Block.all.select(e|e.container.isTypeOf(OperationDefinition))
guard:instance.statements.last().isTypeOf(ExpressionStatement){
do { var new_stat = ReturnStatement.createInstance();
var target = instance.statements.last();
new_stat.expression = target.expression;
instance.statements.remove(target);
instance.statements.add(new_stat);
}}
// 2.2 replacing return statement with expression statement
// of a operation definition
@action replace
@property statements
pattern replace_return_stat_with_exp_stat
instance:Block in:Block.all.select(e|e.container.isTypeOf(OperationDefinition))
guard:instance.statements.last().isTypeOf(ReturnStatement){
do { var target = instance.statements.last();
var new_stat = ExpressionStatement.createInstance();
new_stat.expression = target.expression;
instance.statements.remove(target);
instance.statements.add(new_stat);
}}
// 3.0: replacing delete statement with return statement
@action replace
@property statements
@role instance
pattern replace_delete_stat_with_return_stat
instance:Block,
exp:DeleteStatement from:instance.statements{
do { var new_stat = ReturnStatement.createInstance();
new_stat.expression = exp.expression;
instance.statements.remove(exp);
instance.statements.add(new_stat);
}}
// 5.0: replacing continue statement
// 5.1: replacing continue with break
@action replace
```

```
@property statements
@role instance
pattern replace_continue_with_break
instance:Block,
exp:ContinueStatement from:instance.statements{
do { var new_stat = BreakStatement.createInstance();
instance.statements.remove(exp);
instance.statements.add(new_stat);
}}
// 5.2: replacing continue with breakall
@action replace
@property statements
@role instance
pattern replace_continue_with_breakall
instance:Block,
exp:ContinueStatement from:instance.statements{
do { var new_stat = BreakAllStatement.createInstance();
instance.statements.remove(exp);
instance.statements.add(new_stat);
}}
// 6.0: replacing break statement
// 6.1: replacing break with continue
@action replace
@property statements
@role instance
pattern replace_break_with_continue
instance:Block,
exp:BreakStatement from:instance.statements{
do { var new_stat = ContinueStatement.createInstance();
instance.statements.remove(exp);
instance.statements.add(new_stat);
}}
// 6.2: replacing break with breakall
@action replace
@property statements
@role instance
pattern replace_break_with_breakall
instance:Block,
exp:BreakStatement from:instance.statements{
do { var new_stat = BreakAllStatement.createInstance();
instance.statements.remove(exp);
instance.statements.add(new_stat);
}}
// 7.0: replacing break statement
// 7.1: replacing breakall with continue
@action replace
@property statements
@role instance
pattern replace_breakall_with_continue
instance:Block,
exp:BreakAllStatement from:instance.statements{
do { var new_stat = ContinueStatement.createInstance();
instance.statements.remove(exp);
instance.statements.add(new_stat);
}}
// 7.2: replacing breakall with break
@action replace
@property statements
@role instance
pattern replace_breakall_with_break
instance:Block,
exp:BreakAllStatement from:instance.statements{
do { var new_stat = BreakStatement.createInstance();
instance.statements.remove(exp);
instance.statements.add(new_stat);
}}
```

Listing A.11: `CMO-multiple-REP(Block statements)`

```
@action add
@property modelName
pattern add_model_name
instance:ModelElementType in:ModelElementType.all.select(e|e.modelName.isUndefined()){
```

```
do { var new_modeNames = ModelElementType.all.select(e|e.modelName.isDefined())->collect(c|c.
    modelName)->asOrderedSet();
instance.modelName = new_modeNames.random();
}}
```

Listing A.12: **`CMO-single-ADD(ModelElementType modelName)`**

```
operation isSelfOrResultVar(e:Any):Boolean {
if(e.container.isDefined() and e.container.isTypeOf(OperationDefinition)){
if(e.eContainer.isDefined() and e.eContainer.isTypeOf(VariableDeclarationExpression)){
if(e.eContainer.name.name = "self" or e.eContainer.name.name = "_result")
return true;
}
}
return false;
}
@action replace
@property modelName
pattern replace_model_name
instance:ModelElementType
guard:not isSelfOrResultVar(instance) and ModelElementType.all.select(e|e.modelName.isDefined
    ())->collect(c|c.modelName).size() > 1 {
do { var new_modeNames = ModelElementType.all.select(e|e.modelName.isDefined())->collect(c|c.
    modelName)->asOrderedSet();
var chosen = new_modeNames.random();
while(chosen.isDefined() and instance.modelName.isDefined() and chosen = instance.modelName)
chosen = new_modeNames.random();
instance.modelName = chosen;
}}
```

Listing A.13: **`CMO-single-REP(ModelElementType modelName)`**

```
@action replace
@property arguments
@role instance
pattern replace_arguments
instance:MethodCallExpression in:MethodCallExpression.all.select(e|e.arguments.size()>=2),
argu1:NameExpression from:instance.arguments,
argu2:NameExpression from:instance.arguments guard:argu1<>argu2 {
do { var isLastIndex = instance.arguments.indexOf(argu1) = instance.arguments.size() - 1;
// replacement
var new_argu = NameExpression.createInstance();
new_argu.name = argu1.name;
instance.arguments.remove(argu1);
instance.arguments.add(new_argu);
if(isLastIndex) {
// last index replacement so invert list
var col = instance.arguments.asSequence().invert();
instance.arguments.clear();
instance.arguments.addAll(col);
}
}}
```

Listing A.14: **`CMO-multiple-REP(MethodCallExpression arguments)`**

Appendix B

Test Models

B.1 Test Models for ATL Candidate Programs

B.1.1 Book

```
pre { var b_num:Integer = (0.4 * total).ceiling();
 var c_num:Integer = (0.6 * total).ceiling();
}
$instances b_num
@list books_list
operation Book create() {
 self.title = StringDB!Row.all.random().string;
}
$instances c_num
@list chapters_list
operation Chapter create() {
 self.title = StringDB!Row.all.random().string;
 self.nbPages = nextInteger(1,10);
 self.author = StringDB!Row.all.random().full_name;
 self.book = nextFromList("books_list");
}
```

Listing B.1: EMG generator code for Book metamodel

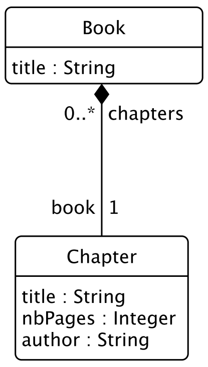

Figure B.1: Book metamodel

B.1.2 HTML

```
pre { var t_num:Integer = (0.10 * total).ceiling();
 var r_num:Integer = (0.50 * total).ceiling();
 var c_num:Integer = (0.40 * total).ceiling();
 var columns = Sequence {"full_name", "string", "id", "boolean", "year", "tiny_integer", "
     small_integer", "mid_integer", "big_ineger", "tiny_real", "small_real", "mid_real", "
     big_real", "percentage", "character", "characters_digits_2", "characters_digits_10"};
}
$instances t_num
operation TABLE create() {
 self.value = StringDB!Row.all.random().string;
 self.bgcolor = "white";
 self.background = "white";
 self.width = nextInteger(1,10).toString();
 self.cellspacing = nextInteger(0,2).toString();
 self.cellpadding = nextInteger(0,2).toString();
 self.border = nextInteger(0,5).toString();
 var max_rows:Integer = nextInteger(2,r_num);
 while(max_rows.mod(2) <> 0) max_rows = nextInteger(2,r_num);
 var counter:Integer = 1;
 var cells_max:Integer = nextInteger(1,c_num);
 while(counter <= max_rows) {
  var row = TR.createInstance();
  self.trs.add(row);
  row.value = StringDB!Row.all.random().string;
  row.bgcolor = "white";
  row.background = "white";
  row.valign = "left";
  row.align = "left";
  var counter_cells = 1;    // create cells and add them to row
  while(counter_cells <= cells_max) {
   var cell;
   if(counter = 1) { // first row = cells are headers and hence must be strings
    cell = TH.createInstance();
    cell.value = StringDB!Row.all.random().string;
   } else {
    cell = TD.createInstance();
    cell.value = nextInteger(1,100).toString();
   }
   cell.bgcolor = "white";
   cell.background = "white";
   cell.colspan=0.toString();
   cell.rowspan=0.toString();
   cell.valign = "left";
   cell.align = "left";
   cell.width = nextInteger(1,10).toString();
   row.tds.add(cell);    // add cell to row
   counter_cells = counter_cells + 1;
  }
 counter = counter + 1;
 } }
post {
 var body = new BODY;  // create a body and add all table elements to it
 body.value = StringDB!Row.all.random().string;
 body.background = "Oxfffff";
 body.bgcolor = "Ox00000";
 body.text = StringDB!Row.all.random().string;
 body.link = "www.website.com/"+StringDB!Row.all.random().first_name;
 body.alink = "www.website.com/"+StringDB!Row.all.random().first_name;
 body.vlink = "www.website.com/"+StringDB!Row.all.random().first_name;
 body.bodyElements.addAll(TABLE.all);  // add all tables to only body
 var title = new TITLE;  // create a head to be added later to html
 title.value = StringDB!Row.all.random().string;
 var head = HEAD.createInstance();
 head.value = StringDB!Row.all.random().string;
 head.headElements.add(title);
 var html = new HTML;  // add head and body to html
 html.head = head;
 html.body = body;
}
```

Listing B.2: EMG generator code for HTML metamodel

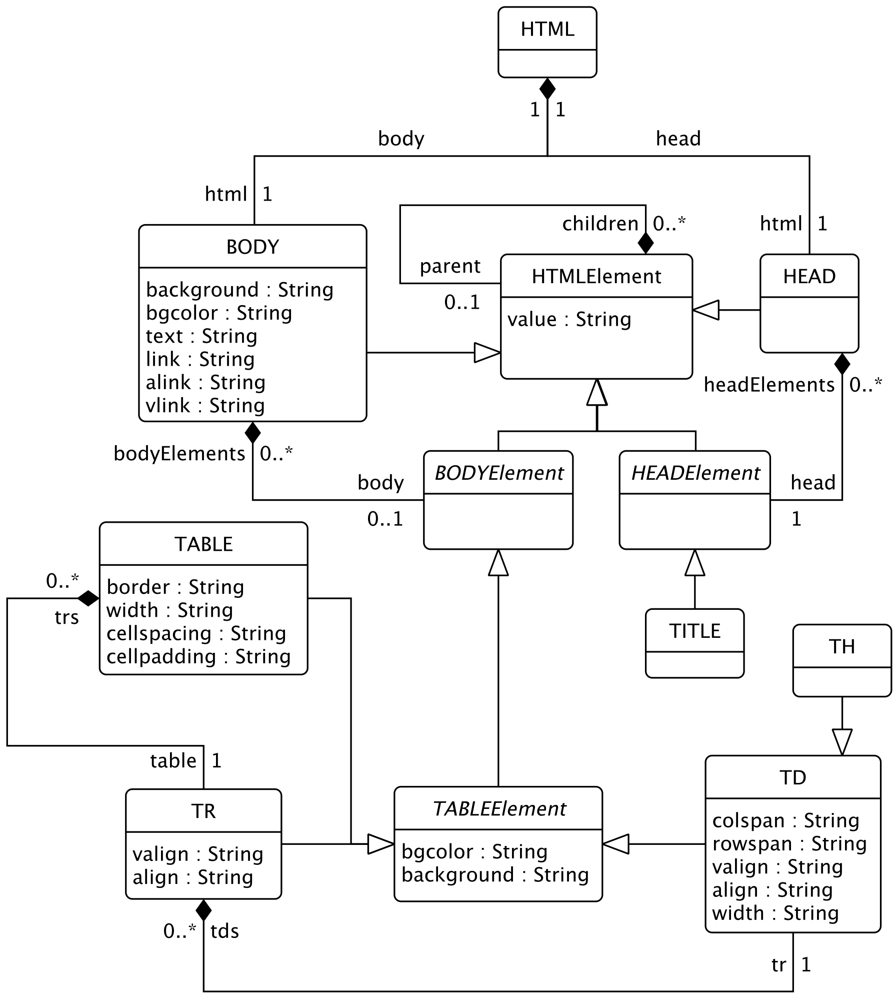

FIGURE B.2: HTML metamodel

B.1.3 Make

```
pre { var r_num:Integer = (0.2 * total).ceiling(); // rules
 var mr_num:Integer = (0.1 * total).ceiling(); // macros
 var s_numb:Integer = (0.3 * total).ceiling(); // shellLine
 var r_de_num:Integer = (0.2 * total).ceiling(); // ruleDep
 var f_de_num:Integer = (0.2 * total).ceiling(); //fileDep
}
$instances s_numb
@list shell_line_list
operation ShellLine create() {
 self.command = StringDB!Row.all.random().command;
 self.display = nextBoolean();
}
$instances r_de_num
@list rule_deps
operation RuleDep create() {
}
$instances f_de_num
@list file_deps
operation FileDep create() {
 self.date = StringDB!Row.all.random().date;
 self.name = StringDB!Row.all.random().string;
}
$instances r_num
operation Rule create() {
 self.name = StringDB!Row.all.random().string;
 var count_rule;
 if(RuleDep.all.select(e|e.belog_to_rule.isUndefined()).size()>=1)
```

```
  count_rule = nextInteger(1,RuleDep.all.select(e|e.belog_to_rule.isUndefined()).size());
 else  count_rule = nextInteger(0,RuleDep.all.select(e|e.belog_to_rule.isUndefined()).size());
 while(count_rule>=1) {
  self.dependencies.add(nextFromCollection(RuleDep.all.select(e|e.belog_to_rule.isUndefined())
    ));
  count_rule = count_rule - 1;
 }
 var count_file;
 if(FileDep.all.select(e|e.belog_to_rule.isUndefined()).size()>=1)
  count_file = nextInteger(1,FileDep.all.select(e|e.belog_to_rule.isUndefined()).size());
 else count_file = nextInteger(0,FileDep.all.select(e|e.belog_to_rule.isUndefined()).size());
 while(count_file>=1) {
  self.dependencies.add(nextFromCollection(FileDep.all.select(e|e.belog_to_rule.isUndefined())
    ));
  count_file = count_file - 1;
 }
 var count_shells;
 if(ShellLine.all.select(e|e.ruleShellLine.isUndefined()).size()>=1) {
  count_shells = nextInteger(1,ShellLine.all.select(e|e.ruleShellLine.isUndefined()).size());
  while(count_shells >= 1) {
   self.shellLines.add(nextFromCollection(ShellLine.all.select(e|e.ruleShellLine.isUndefined()
    )));
   count_shells = count_shells - 1;
  } } else {
  // No shell line available to link to this rule so, select from other shell lines of rules
    that  have more than one shell line
  var shell_line = Rule.all.select(e|e.shellLines.size() >= 2).random().shellLines.random();
  self.shellLines.add(shell_line);
 } }
$instances mr_num
operation Macro create() {
 self.name = StringDB!Row.all.random().string;
 self.value = StringDB!Row.all.random().string;
}
post {
 for(r in Rule.all) {
 for (dr in r.dependencies.select(e|e.isTypeOf(RuleDep))) {
  var other_rules = Rule.all.excluding(r);
  if(other_rules.size()==0) r.dependencies.remove(dr);
  else dr.ruledep = nextFromCollection(other_rules);
  } }
 var makeFile_var = new Makefile;
 makeFile_var.name = StringDB!Row.all.random().string;
 var com = Comment.createInstance();
 com.text = StringDB!Row.all.random().string;
 makeFile_var.comment = com;
 makeFile_var.elements.addAll(Rule.all);
 makeFile_var.elements.addAll(Macro.all);

 // checking model. Check that a make file has at least one element
 if (makeFile_var.elements.size()==0) throw "A makefile must have at least one element";
 for(r in Rule.all) {  // check that a rule must have at least a shell line
  if (r.shellLines.size()==0) throw "A rule must have at least one shell line";
  // check that a dependent rule must point to another rule
  for (dr in r.dependencies.select(e|e.isTypeOf(RuleDep))) {
   if(dr.ruledep.isUndefined()) throw "A dependent rule must point to another rule";
  } } }
```

Listing B.3: EMG generator code for Make metamodel

B.1.4 Table

```
pre { var t_num:Integer = (0.10 * total).ceiling();
 var r_num:Integer = (0.50 * total).ceiling();
 var c_num:Integer = (0.40 * total).ceiling();
 var columns = Sequence {"full_name", "string", "id", "boolean", "year", "tiny_integer", "
    small_integer", "mid_integer", "big_ineger", "tiny_real", "small_real", "mid_real", "
    big_real", "percentage", "character", "characters_digits_2", "characters_digits_10"};
}
$instances t_num
@list tables_list
operation Table create() {
```

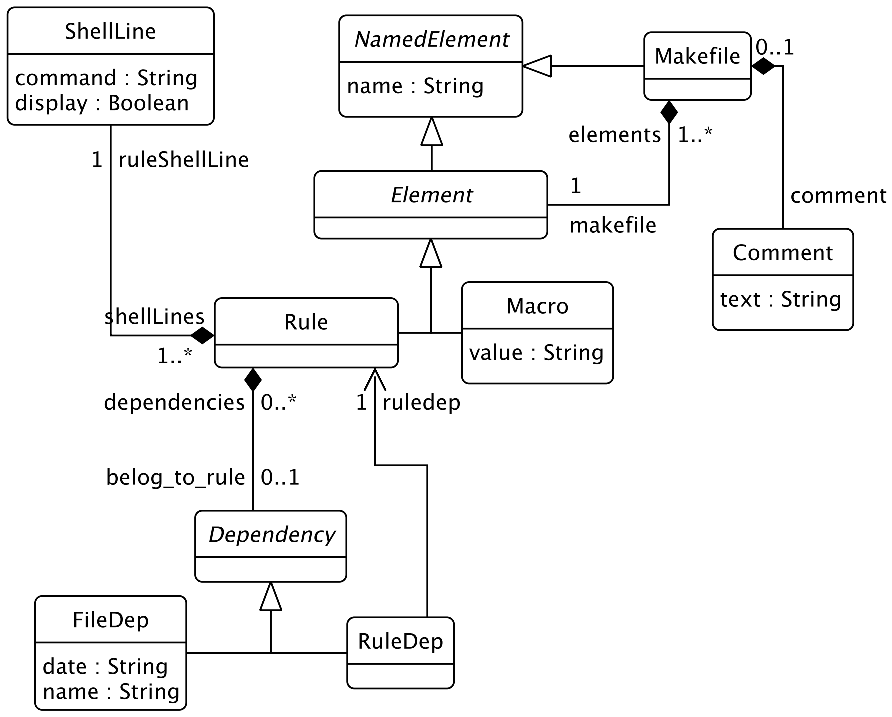

FIGURE B.3: Make metamodel

```
var max_rows:Integer = nextInteger(2,r_num);
while(max_rows.mod(2) <> 0) max_rows = nextInteger(2,r_num);
var counter:Integer = 1;
var cells_max:Integer = nextInteger(1,c_num);
while(counter <= max_rows) {
 var row = Table!Row.createInstance();
 self.rows.add(row);
 var counter_cells = 1;
 while(counter_cells <= cells_max) {
  var cell= Cell.createInstance();
  row.cells.add(cell);
  if(counter = 1) {
   cell.content = StringDB!Row.all.random().string;     // first row = cells are headers and
    hence must be strings
  } else cell.content = nextInteger(1,100).toString();
 counter_cells = counter_cells + 1;
 }
 counter = counter + 1;
} }
```

LISTING B.4: EMG generator code for Table metamodel

B.2 Test Models for EOL Candidate Programs

B.2.1 DirectedGraph

```
pre { var n_num:Integer;
 var e_num:Integer;
 if(total=2) {
  n_num = 2;
  e_num = 1;
 } else {
  n_num = (0.50 * total).ceiling(); // nodes
  e_num = (0.50 * total).ceiling(); // edges
 }
```

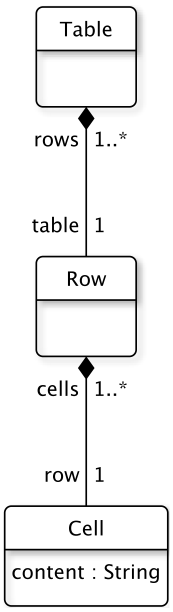

FIGURE B.4: Table metamodel

```
 var names = StringDB!Row.all.collect(e|e.first_name.toLowerCase());
}
$instances n_num
@list nodes
operation Node create() {
 var index = nextInteger(0,300);
 self.label = nextFromCollection(names);
}
$instances e_num
@list edges
operation Edge create() {
 self.weight = nextInteger(1,200);
 var n1 = nextFromList("nodes");
 self.source = n1;
 var n2 = nextFromList("nodes");
 while(n1 = n2) n2 = nextFromList("nodes");
 self.target = n2;
}

post {
 // fix the graph and ensure all nodes are connected
 var not_connected = Node.all.select(e|e.outgoing.size() = 0 and e.incoming.size() = 0).
     asSequence();
 while(Node.all.select(n|n.outgoing.size() = 0 and n.incoming.size() = 0).size()>=1) {
  for(n in Node.all.select(n|n.outgoing.size() = 0 and n.incoming.size() = 0)) {
   var choice = nextInteger(0,1);
   var edge;
   if(choice = 0) n.outgoing.add(nextFromList("edges"));
   if(choice = 1) n.incoming.add(nextFromList("edges"));
   } }
 if(Edge.all.size() < (Node.all.size() - 1)) throw "Number of edges must be >= number of (
     nodes - 1).";
 if(Node.all.select(n|n.outgoing.size() = 0 and n.incoming.size() = 0).size()>=1)
   throw "A node must be connected with at least one edge.";
 if(Edge.all.select(e|e.source.size = 0 and n.target.size = 0).size()>=1)
   throw "An edge must be connected with at least one node.";
 var g = new Graph;
 g.contents.addAll(GraphElement.all);
}
```

LISTING B.5: EMG generator code for DirectedGraph metamodel

B.2.2 Ecore test models for EuGENia

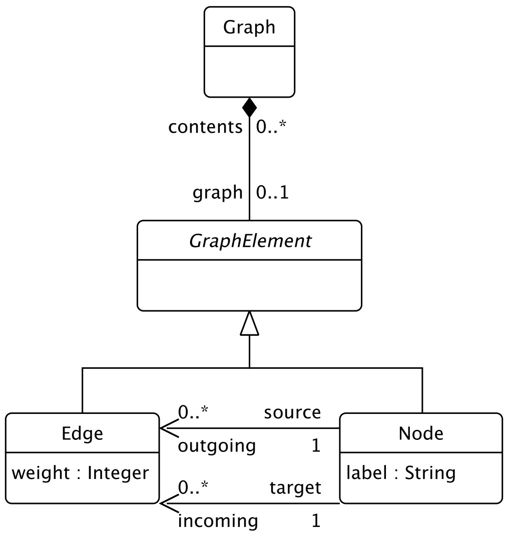

FIGURE B.5: DirectedGraph metamodel

```
pre{ var n_classes:Integer = (0.40 * total).ceiling();
 var n_attributes:Integer = (0.20 * total).ceiling();
 var n_vals:Integer = (0.20 * total).ceiling();
 var n_refs:Integer = (0.20 * total).ceiling();
 var names:Sequence = StringDB!Row.all.collect(e|e.first_name.toLowerCase()); // 299 names

 var ecore_datatypes = new Sequence;
 ecore_datatypes.clear();
 ecore_datatypes.addAll(ECore!EDataType.all.select(e|e.serializable).excludingAll(ECore!
    EDataType.all.select(e|e.name = "EJavaClass" or e.name = "EJavaObject")));

 var styles = new Sequence;
 styles.add("solid");
 styles.add("dash");
 styles.add("dot");

 var figures = new Sequence;
 figures.add("rectangle");
 figures.add("ellipse");
 figures.add("rounded");
 figures.add("svg");
 figures.add("polygon");

 var decorations = new Sequence;
 decorations.add("none");
 decorations.add("arrow");
 decorations.add("rhomb");
 decorations.add("filledrhomb");
 decorations.add("square");
 decorations.add("filledsquare");
 decorations.add("closedarrow");
 decorations.add("filledclosedarrow");

 var layouts = new Sequence;
 layouts.add("free");
 layouts.add("list");

 var selected_names = new Sequence;
}
$instances n_classes
@list classes
operation Ecore!EClass create() {
 var name = nextFromCollection(names);
 while(Ecore!EClass.all.exists(e|e.name = name))
  name = nextFromCollection(names);
 self.name = name;
 var choice:Integer = nextInteger(1,10);
 if(choice.mod(2)=0) self.abstract = true;
```

```
}

$instances n_attributes
@list attribues
operation Ecore!EAttribute create() {
 var _c = nextFromList("classes");
 _c.eStructuralFeatures.add(self);
 var name = nextFromCollection(names);
 while(_c.eStructuralFeatures.exists(e|e.name = name))
  name = nextFromCollection(names);
 self.name = name;
 self.upperBound = nextInteger(1,10);
 self.lowerBound = nextInteger(0,self.upperBound);
 self.eType = nextFromCollection(ecore_datatypes);
}

$instances n_vals
@list vals
operation Ecore!EReference create() {
 self.containment = true;
 var _c = nextFromList("classes");
 _c.eStructuralFeatures.add(self);
 var name = nextFromCollection(names);
 while(_c.eStructuralFeatures.exists(e|e.name = name))
  name = nextFromCollection(names);
 self.name = name;
 self.upperBound = nextInteger(1,10);
 self.lowerBound = nextInteger(0,self.upperBound);
 self.eType = nextFromList("classes");
}
$instances n_refs
@list refs
operation Ecore!EReference create() {
 var _c = nextFromList("classes");
 _c.eStructuralFeatures.add(self);
 var name = nextFromCollection(names);
 while(_c.eStructuralFeatures.exists(e|e.name = name))
  name = nextFromCollection(names);
 self.name = name;
 self.upperBound = nextInteger(1,10);
 self.lowerBound = nextInteger(0,self.upperBound);
 self.eType = nextFromList("classes");
}
$instances 1
operation Ecore!EPackage create() {
 self.name = "ecore";
 self.eClassifiers.addAll(Ecore!EClass.all);
 self.nsURI="http://www.eclipse.org/emf/2002/Ecore";
 self.nsPrefix="Ecore";
}
post {
 var visited = new Set;
 var n:Integer;
 // do some inheritance
 var classes = Ecore!EClass.all.select(e| not e.abstract and e.eSuperTypes.size() = 0);
 n = (nextInteger(1,classes.size)/2).ceiling();
 if(n = 0) n = 1;
 while(n >= 1) {
  var c = nextFromCollection(classes.excludingAll(visited));
  visited.add(c);
  classes.remove(c);
  var target = nextFromCollection(classes.select(e|e <> c or not e.eSuperTypes.flatten.
     includes(c)));
  if(target.eSuperTypes.includes(c)) throw "Target [" +target.name+"] is selected for source
     ["+c.name+"]";
  c.eSuperTypes.add(target);
  n = n - 1;
 }
 classes = Ecore!EClass.all.select(e| not e.abstract);
 visited.clear();
 // do some annotations to gmf
 var anno = new Ecore!EAnnotation;
 anno.source = "gmf";
 Ecore!EPackage.all.first().eAnnotations.add(anno);
 var diagram = nextFromCollection(classes);  // gmf.diagram
```

```
visited.add(diagram);
anno = new Ecore!EAnnotation;
anno.source = "gmf.diagram";
anno.details.put("diagram.extension","model");
anno.details.put("model.extension","model");
anno.details.put("onefile",""+nextBoolean());
diagram.eAnnotations.add(anno);
n = nextInteger(1,classes.excludingAll(visited).size()); // gmf.node
if(n=0) n=1;
while(n>=1) {
 var c = nextFromCollection(classes.excludingAll(visited));
 var features = c.eStructuralFeatures.select(e|e.isTypeOf(Ecore!EAttribute));
 visited.add(c);
 anno = new Ecore!EAnnotation;
 anno.source = "gmf.node";
 anno.details.put("border.color",nextInteger(0,256)+","+nextInteger(0,256)+","+nextInteger
    (0,256));
 anno.details.put("border.style",nextFromCollection(styles));
 anno.details.put("border.width",""+nextInteger(1,10));
 anno.details.put("","");
 anno.details.put("color",nextInteger(0,256)+","+nextInteger(0,256)+","+nextInteger(0,256));
 anno.details.put("figure",""+nextFromCollection(figures));
 if(not (features.size() = 0)) {
  anno.details.put("label",nextFromCollection(features).name);
  anno.details.put("label.color",nextInteger(0,256)+","+nextInteger(0,256)+","+nextInteger
    (0,256));
  anno.details.put("label.icon",""+nextBoolean());
 }
 if(anno.details.get("figure") = "svg")
  anno.details.put("svg.uri","platform:/plugin/my.plugin/"+nextFromCollection(names));
 if(anno.details.get("figure") = "polygon") {
  anno.details.put("polygon.x",""+nextInteger(1,10));
  anno.details.put("polygon.y",""+nextInteger(1,10));
 }
 c.eAnnotations.add(anno);
 n = n - 1;
}
// gmf link to EClass and non-containment EReference
visited.clear();
visited.add(diagram);
n = nextInteger(1,classes.size()-visited.size());
if(n=0) n=1;
while(n>=1) {
 var c = nextFromCollection(classes.excludingAll(visited));
 if(c.isUndefined()) break;
 var features = c.eStructuralFeatures.select(e|e.isTypeOf(Ecore!EAttribute));
 var references = c.eStructuralFeatures.select(e|e.isTypeOf(Ecore!EReference) and not e.
    containment);
 visited.add(c);
 var choice = nextInteger(1,10);
 anno = new Ecore!EAnnotation;
 anno.source = "gmf.link";
 anno.details.put("color",nextInteger(0,256)+","+nextInteger(0,256)+","+nextInteger(0,256));
 anno.details.put("style",nextFromCollection(styles));

 if(not (features.size()=0)) anno.details.put("label",nextFromCollection(features).name);
 if(choice.mod(2)=0 and references.size()>=2) {
  anno.details.put("source",nextFromCollection(references).name);
  anno.details.put("target",nextFromCollection(references).name);
 }
 anno.details.put("source.decoration",nextFromCollection(decorations));
 anno.details.put("target.decoration",nextFromCollection(decorations));
 anno.details.put("width",""+nextInteger(1,10));
 c.eAnnotations.add(anno);
 n = n - 1;
}
// gmf.compartment
var vals = Ecore!EReference.all.select(e| e.containment);
visited.clear();
n = nextInteger(1,(vals.size()-visited.size())/2);
if(n=0) n=1;
while(n>=1) {
 var f = nextFromCollection(vals.excludingAll(visited));
 visited.add(f);
 anno = new Ecore!EAnnotation;
```

```
  anno.source = "gmf.compartment";
  anno.details.put("collapsible",""+nextBoolean());
  anno.details.put("layout",nextFromCollection(layouts));
  f.eAnnotations.add(anno);
  n = n -1;
 }}
```

Listing B.6: EMG generator code for Ecore metamodel of EuGENia

B.2.3 Ecore test models for Incremental EVL

```
pre { var n_classes:Integer = (0.30 * total).ceiling();
 var n_attributes:Integer = (0.30 * total).ceiling();
 var n_vals:Integer = (0.20 * total).ceiling();
 var n_refs:Integer = (0.20 * total).ceiling();
 var names:Sequence = StringDB!Row.all.collect(e|e.first_name.toLowerCase()); // 299 names
 var ecore_datatypes = new Sequence;
 ecore_datatypes.clear();
 ecore_datatypes.addAll(ECore!EDataType.all.select(e|e.serializable).excludingAll(ECore!
     EDataType.all.select(e|e.name = "EJavaClass" or e.name = "EJavaObject")));
}
$instances n_classes
@list classes
operation Ecore!EClass create() {
 var name = nextFromCollection(names);
 while(Ecore!EClass.all.exists(e|e.name = name))
  name = nextFromCollection(names);
 self.name = name;
 var choice:Integer = nextInteger(1,10);
 if(choice.mod(2)=0) self.abstract = true;
}
$instances n_attributes
@list attribues
operation Ecore!EAttribute create() {
 var _c = nextFromList("classes");
 _c.eStructuralFeatures.add(self);
 var name = nextFromCollection(names);
 while(_c.eStructuralFeatures.exists(e|e.name = name)) name = nextFromCollection(names);
 self.name = name;
 self.upperBound = nextInteger(1,10);
 self.lowerBound = nextInteger(0,self.upperBound);
 self.eType = nextFromCollection(ecore_datatypes);
}
$instances n_vals
@list vals
operation Ecore!EReference create() {
 self.containment = true;
 var _c = nextFromList("classes");
 _c.eStructuralFeatures.add(self);
 var name = nextFromCollection(names);
 while(_c.eStructuralFeatures.exists(e|e.name = name)) name = nextFromCollection(names);
 self.name = name;
 self.upperBound = nextInteger(1,10);
 self.lowerBound = nextInteger(0,self.upperBound);
 self.eType = nextFromList("classes");
}
$instances n_refs
@list refs
operation Ecore!EReference create() {
 var _c = nextFromList("classes");
 _c.eStructuralFeatures.add(self);
 var name = nextFromCollection(names);
 while(_c.eStructuralFeatures.exists(e|e.name = name))
  name = nextFromCollection(names);
 self.name = name;
 self.upperBound = nextInteger(1,10);
 self.lowerBound = nextInteger(0,self.upperBound);
 self.eType = nextFromList("classes");
}
$instances 1
operation Ecore!EPackage create() {
 self.name = "ecore";
 self.eClassifiers.addAll(Ecore!EClass.all);
 self.nsURI="http://www.eclipse.org/emf/2002/Ecore";
 self.nsPrefix="Ecore";
}

post {
 var visited = new Sequence;
 var classes = Ecore!EClass.all;
 // do some inheritance
 var classes = Ecore!EClass.all.select(e| not e.abstract and e.eSuperTypes.size() = 0);
 var n = (nextInteger(1,classes.size)/2).ceiling();
 if(n = 0) n = 1;
 while(n >= 1) {
  var c = nextFromCollection(classes.excludingAll(visited));
  visited.add(c);
  classes.remove(c);
  var target = nextFromCollection(classes.select(e|e <> c or not e.eSuperTypes.flatten.includes(c)));
  if(target.eSuperTypes.includes(c)) throw "target [" +target.name+"] is selected for source ["+c.name+"]";
  c.eSuperTypes.add(target);
  n = n - 1;
 }
 classes = Ecore!EClass.all.select(e| not e.abstract);
 // do some annotations
 n = (nextInteger(1,classes.size)/2).ceiling();
 if(n = 0) n = 1;
 visited.clear();
 while(n >= 1) {
  var c = nextFromCollection(classes.excludingAll(visited));
  var features = c.eStructuralFeatures.select(e|e.isTypeOf(Ecore!EAttribute));
  if(features.size()=0) {
   n = n - 1;
   continue;
  }
  visited.add(c);
  var class_anno_1 = new Ecore!EAnnotation;
  class_anno_1.source = "https://eclipse.org/epsilon/incremental/OrientDbIndex";
  class_anno_1.references.add(nextFromCollection(features));
  class_anno_1.details.put("type","NOTUNIQUE_HASH_INDEX");
  c.eAnnotations.add(class_anno_1);
  n = n-1;
 }
 n = (nextInteger(1,classes.size)/2).ceiling();
 if(n = 0) n = 1;
 visited.clear();
 while(n >= 1) {
  var c = nextFromCollection(classes.excludingAll(visited));
  var features = c.eStructuralFeatures.select(e|e.isTypeOf(Ecore!EReference));
  if(features.size()=0) {
   n = n - 1;
   continue;
  }
  visited.add(c);
  var class_anno_1 = new Ecore!EAnnotation;
  class_anno_1.source = "https://eclipse.org/epsilon/incremental/equals";
  class_anno_1.references.add(nextFromCollection(features));
  c.eAnnotations.add(class_anno_1);
  n = n - 1;
 }
 var features = Ecore!EStructuralFeature.all;
 n = (nextInteger(1,features.size)/2).ceiling();
 if(n = 0) n = 1;
 visited.clear();
 while(n >= 1) {
  var f = nextFromCollection(features.excludingAll(visited));
  visited.add(f);
  var reference_anno_1 = new Ecore!EAnnotation;
  reference_anno_1.source = "https://eclipse.org/epsilon/incremental/Graph";
  reference_anno_1.details.put("edge","true");
  f.eAnnotations.add(reference_anno_1);
  n = n-1;
 }
```

```
}
```

Listing B.7: EMG generator code for Ecore metamodel of Incremental EVL

Appendix C

Mutation Analysis Results

C.1 ATL Complete Results

Mutation Operator	Gen.	Killed	Trivial	Live	Equiv.	Invalid
`CMO-S-REP(Parameter type)`	15	6	-	9	-	-
`CMO-S-REP(Operation returnType)`	77	43	-	34	-	-
`CMO-S-DEL(InPattern filter)`	4	2	1	1	-	-
`CMO-S-REP(InPattern filter)`	14	2	10	2	-	-
`CMO-S-REP(SimpleInPatternElement type)`	296	89	164	34	9	-
`CMO-S-REP(RuleVariableDeclaration type)`	21	21	-	-	-	-
`CMO-S-REP(RuleVariableDeclaration varName)`	7	7	-	-	-	-
`CMO-S-REP(Attribute type)`	126	92	-	-	34	-
`CMO-S-REP(SequenceType elementType)`	98	68	-	-	30	-
`CMO-S-REP(Iterator varName)`	36	21	-	-	15	-
`CMO-S-DEL(LazyMatchedRule actionBlock)`	2	1	1	-	-	-
`CMO-S-REP(SimpleOutPatternElement varName)`	61	30	11	-	20	-
`CMO-S-REP(SimpleInPatternElement varName)`	42	18	-	-	13	11
`CMO-S-REP(Parameter varName)`	5	2	-	-	3	-
`CMO-M-DEL(ActionBlock statements)`	10	3	7	-	-	-
`CMO-S-REP(IfStat condition)`	7	2	5	-	-	-
`CMO-S-REP(CalledRule name)`	4	1	2	-	1	-
`CMO-S-REP(LazyMatchedRule name)`	11	2	9	-	-	-
`CMO-S-REP(SimpleOutPatternElement type)`	944	136	800	-	8	-
`CMO-S-REP(Attribute name)`	15	2	11	-	2	-
`CMO-S-REP(OclModelElement name)`	113	14	88	-	11	-
`CMO-M-DEL(InPattern elements)`	18	2	3	-	-	13
`CMO-M-DEL(SimpleOutPatternElement bindings)`	130	14	112	-	4	-
`CMO-S-REP(OclModel name)`	65	6	42	-	12	5
`CMO-S-REP(Binding propertyName)`	332	29	303	-	-	-
`CMO-M-REP(Module elements)`	24	2	19	-	-	3
`CMO-M-DEL(Module elements)`	72	5	62	-	5	-
`CMO-M-DEL(OutPattern elements)`	36	2	5	-	-	29
`CMO-S-REP(OclFeatureDefinition feature)`	37	2	31	-	4	-
`CMO-S-DEL(MatchedRule outPattern)`	19	1	16	-	-	2
`CMO-M-DEL(ForStat statements)`	4	-	4	-	-	-
`CMO-M-DEL(IfStat thenStatements)`	4	-	4	-	-	-

`CMO-S-DEL(CalledRule actionBlock)`	4	-	4	-	-	-
`CMO-S-DEL(MatchedRule actionBlock)`	2	-	2	-	-	-
`CMO-S-REP(CalledRule isEntrypoint)`	2	-	2	-	-	-
`CMO-S-REP(OclContextDefinition context)`	594	-	584	-	10	-
`CMO-S-REP(Operation name)`	23	-	21	-	2	-
`CMO-S-DEL(OclFeatureDefinition context)`	22	-	20	-	2	-
`CMO-M-REP(MatchedRule children)`	82	-	70	-	3	9
`CMO-M-DEL(MatchedRule children)`	10	-	7	-	3	-
`CMO-S-REP(BindingStat source)`	9	-	5	-	-	4
`CMO-M-DEL(CalledRule parameters)`	3	-	-	-	-	3
`CMO-M-DEL(CalledRule variables)`	2	-	-	-	-	2
`CMO-M-DEL(LazyMatchedRule variables)`	5	-	-	-	-	5
`CMO-M-DEL(Operation parameters)`	2	-	-	-	-	2
`CMO-S-DEL(CalledRule outPattern)`	4	-	-	-	-	4
`CMO-S-DEL(LazyMatchedRule inPattern)`	11	-	-	-	-	11
`CMO-S-DEL(LazyMatchedRule outPattern)`	11	-	-	-	-	11
`CMO-S-DEL(MatchedRule inPattern)`	19	-	-	-	-	19
`CMO-S-DEL(Parameter type)`	5	-	-	-	-	5
`CMO-S-DEL(RuleVariableDeclaration initExpr)`	7	-	-	-	-	7
`CMO-S-DEL(RuleVariableDeclaration type)`	7	-	-	-	-	7
`CMO-S-REP(MatchedRule isAbstract)`	3	-	-	-	3	-
`CMO-S-REP(MatchedRule name)`	19	-	-	-	19	-
Total	3495	625	2425	80	213	152

Table C.1: ATL mutation operators and their produced mutants

C.2 EOL Complete Results

Mutation Operator	Gen.	Killed	Trivial	Live	Equiv.	Invalid
`CMO-S-REP(AndOperatorExpression inBrackets)`	1			1		
`CMO-S-REP(OrOperatorExpression inBrackets)`	4	1		2	1	
`CMO-S-REP(ReturnStatement expression)`	62	12	13	28	9	
`CMO-M-ADD(Block statements)`	145	34	32	62	17	
`CMO-S-DEL(IfStatement elseBody)`	41	5	8	15	13	
`CMO-S-REP(ExpressionOrStatementBlock condition)`	85	39	14	31	1	
`CMO-S-REP(PropertyCallExpression extended)`	45	11	18	16		
`CMO-S-DEL(ExpressionOrStatementBlock block)`	209	36	56	68	32	17
`CMO-M-REP(Block statements)`	160	24	72	51	13	
`CMO-M-REP(MethodCallExpression arguments)`	102	4	64	32	2	
`CMO-M-DEL(MethodCallExpression arguments)`	170	18	99	50	3	
`CMO-S-REP(VariableDeclarationExpression create)`	14		9	4	1	
`CMO-M-DEL(Block statements)`	825	161	335	235	94	
`CMO-M-REP(FOLMethodCallExpression conditions)`	96	22	38	26	10	
`CMO-S-REP(IfStatement condition)`	230	29	109	62	30	
`CMO-S-DEL(MethodCallExpression target)`	529	84	297	123	25	
`CMO-S-REP(MethodCallExpression method)`	119	15	60	26	18	
`CMO-M-ADD(MethodCallExpression arguments)`	581	99	324	125	33	
`CMO-S-REP(ModelElementType modelName)`	205	24	69	32	16	64
`CMO-S-REP(VariableDeclarationExpression name)`	178	32	108	27	11	
`CMO-S-REP(AssignmentStatement rhs)`	27	8	10	4	5	
`CMO-S-REP(FormalParameterExpression name)`	160	25	101	22	12	
`CMO-S-DEL(ModelElementType modelName)`	205	21	33	28	58	65

`CMO-S-REP(OperationDefinition returnType)`	97	9	36	12	18	22
`CMO-M-ADD(OperationDefinition parameters)`	103	6	70	10	17	
`CMO-M-DEL(EOLModule operations)`	103	6	70	10	17	
`CMO-S-REP(FOLMethodCallExpression method)`	212	34	135	19	24	
`CMO-S-DEL(FOLMethodCallExpression target)`	49	8	34	3	4	
`CMO-M-DEL(IfStatement elseIfBodies)`	17	12	4	1		
`CMO-M-DEL(OperationDefinition parameters)`	71	3	62	1	5	
`CMO-S-REP(OperationDefinition contextType)`	8	3	2		3	
`CMO-S-REP(VariableDeclarationExp resolvedType)`	20	2	8		10	
`CMO-S-DEL(EOLModule block)`	2		2			
`CMO-S-REP(PlusOperatorExpression inBrackets)`	1		1			
`CMO-S-REP(WhileStatement condition)`	1		1			
`CMO-S-REP(NotOperatorExpression inBrackets)`	2				2	
`CMO-S-REP(EqualsOperatorExpression inBrackets)`	1				1	
`CMO-S-REP(FormalParameterExpression resolvedType)`	2				2	
`CMO-S-DEL(OperationDefinition body)`	103					103
`CMO-S-DEL(PropertyCallExpression target)`	434					434
`CMO-S-DEL(ExpressionOrStatementBlock condition)`	17					17
Total	5436	787	2294	1126	507	722

Table C.2: EOL mutation operators and their produced mutants

Bibliography

[1] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan and Claypool, 2012.

[2] D. Karagiannis and H. Kühn. "Invited Paper: Metamodelling Platforms". In: *Proceedings of the Third International Conference on E-Commerce and Web Technologies*. Ed. by K. Bauknecht, A. M. Tjoa, and G. Quirchmayr. EC-WEB '02. London, UK: Springer-Verlag, 2002, pp. 182–182. URL: `http://dl.acm.org/citation.cfm?id=646162.680499`.

[3] J. Offutt, P. Ammann, and L. L. Liu. "Mutation Testing Implements Grammar-Based Testing". In: *Proceedings of the Second Workshop on Mutation Analysis*. MUTATION '06. USA: IEEE Computer Society, 2006, p. 12. ISBN: 076952897X. DOI: `10.1109/MUTATION.2006.11`. URL: `https://doi.org/10.1109/MUTATION.2006.11`.

[4] J.-M. Favre. "Foundations of Model (Driven) (Reverse) Engineering: Models - Episode I, Stories of the Fidus Papyrus and of the Solarus". In: *Post-proceedings Of Dagsthul Seminar On Model Driven Reverse Engineering*. Germany Dagsthul, 2004.

[5] D. Kolovos. "An Extensible Platform for Specification of Integrated Languages for Model Management". Doctor of Philosophy. University of York, 2008.

[6] J. Bézivin. "In Search of a Basic Principle for Model Driven Engineering". In: *Novatica/Upgrade* 5 (2004).

[7] Object Management Group. *OMG Meta Object Facility (MOF) Core Specification ver. 2.0*. `https://www.omg.org/spec/MOF/2.0`. Standard. [Online; accessed 03-March-2015]. 2006.

[8] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. "ATL: A model transformation tool". In: *Science of Computer Programming* 72.1 (2008). Special Issue on Second issue of experimental software and toolkits (EST), pp. 31 –39. ISSN: 0167-6423. DOI: `http://dx.doi.org/10.1016/j.scico.2007.08.002`. URL: `http://www.sciencedirect.com/science/article/pii/S0167642308000439`.

[9] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. http://www.omg.org/spec/QVT/1.2. Standard. [Online; accessed 06-Nov-2015]. 2015.

[10] I. Sommerville. *Software Engineering*. 9th. USA: Addison-Wesley Publishing Company, 2010. ISBN: 0137035152, 9780137035151.

[11] B. W. Boehm. "Verifying and validating software requirements and design specifications". In: *IEEE Software* 1.1 (1984), pp. 75–88.

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

[13] Object Management Group. *XML Metadata Interchange (XMI)*. http://www.omg.org/spec/XMI/. Standard. Version 2.5.1. [Online; accessed 06-Nov-2015]. 2015.

[14] H. Bruneli'ere, J. Cabot, G. Dup'e, and F. Madiot. "MoDisco: A model driven reverse engineering framework". In: *Information and Software Technology* 56.8 (2014), pp. 1012 –1032. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2014.04.007. URL: http://www.sciencedirect.com/science/article/pii/S0950584914000883.

[15] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional, 2009. ISBN: 0321331885.

[16] D. Gasevic, D. Djuric, and V. Devedzic. *Model Driven Engineering and Ontology Development*. Second. Springer, 2009.

[17] D. Kolovos, L. Rose, A. García-Domínguez, and R. Paige. *The Epsilon Book*. 2017. URL: https://www.eclipse.org/epsilon/doc/book/.

[18] D. S. Kolovos and R. F. Paige. "The Epsilon Pattern Language". In: *Proceedings of the 9th International Workshop on Modelling in Software Engineering*. MISE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 54–60. ISBN: 978-1-5386-0426-7. DOI: 10.1109/MiSE.2017..8. URL: https://doi.org/10.1109/MiSE.2017..8.

[19] Object Management Group. *Object Constraint Language (OCL)*. http://www.omg.org/spec/OCL/. Standard. Version 2.4. [Online; accessed 06-Nov-2015]. 2014.

[20] D. D. Ruscio, R. Eramo, and A. Pierantonio. "Model Transformations". In: *Formal Methods for Model-Driven Engineering (SFM 2012)*. Ed. by M. Bernardo, V. Cortellessa, and A. Pierantonio. Vol. 7320. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 91–136.

[21] P. Mohagheghi, W. Gilani, A. Stefanescu, M. A. Fernandez, B. Nordmoen, and M. Fritzsche. "Where does model-driven engineering help? Experiences from three industrial cases". In: *Software & Systems Modeling* 12.3 (2013), pp. 619–639. ISSN: 1619-1374. DOI: 10.1007/s10270-011-0219-7. URL: https://doi.org/10.1007/s10270-011-0219-7.

[22] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. "Empirical assessment of MDE in industry". In: *Software Engineering (ICSE), 2011 33rd International Conference on.* 2011, pp. 471–480.

[23] T. Stahl and M. Volter. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley and Sons Inc., 2006.

[24] T. Weigert and F. Weil. "Practical experiences in using model-driven engineering to develop trustworthy computing systems". In: *Sensor Networks, Ubiquitous, and Trustworthy Computing, 2006. IEEE International Conference on.* Vol. 1. IEEE Computer Society, 2006, pp. 208–217.

[25] P. Mohagheghi and V. Dehlen. "Where Is the Proof? - A Review of Experiences from Applying MDE in Industry". In: *Model Driven Architecture – Foundations and Applications.* Ed. by I. Schieferdecker and A. Hartman. Vol. 5095. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 432–443.

[26] P. C. Jorgensen. *Software Testing: A Craftsman's Approach.* 1st. Boca Raton, FL, USA: CRC Press, Inc., 1995.

[27] A. P. Mathur. *Foundations of Software Testing: Fundamental Algorithms and Techniques.* Pearson India, 2007.

[28] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer". In: *Computer* 11.4 (1978), pp. 34–41.

[29] A. J. Offutt and R. H. Untch. "Mutation 2000: Uniting the Orthogonal". In: *Mutation Testing for the New Century.* Ed. by W. E. Wong. Vol. 24. The Springer International Series on Advances in Database Systems. Springer US, 2001, pp. 34–44.

[30] T. A. Budd and F. G. Sayward. *Users Guide to the Pilot Mutation System.* Technical report 114. New Haven, Connecticut: Yale University, 1977.

[31] D. M. S. Andre. *Pilot mutation system (PIMS) user's manual.* Tech. rep. GIT-ICS-79/04. Atlanta, Georgia: Georgia Institute of Technology, 1979.

[32] R. A. DeMillo, D. S. Guindi, W. M. McCracken, A. J. Offutt, and K. N. King. "An extended overview of the Mothra software testing environment". In: *[1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis.* 1988, pp. 142–151. DOI: 10.1109/WST.1988.5369.

[33] K. King and J. Offutt. “A Fortran Language System for Mutation-Based Software Testing”. In: *Software: Practice and Experience* 21.7 (1991), pp. 685–718.

[34] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford. *Design of mutant operators for the C programming language*. SERC-TR-41-P. Software Engineering Research Center, Purdue University, 1989.

[35] M. E. Delamaro, J. C. Maldonado, and A. Mathur. “Proteum-A Tool for the Assessment of Test Adequacy for C Programs”. In: *in Proceedings of the Conference on Performability in Computing System (PCS ’96)*. Vol. 96. 1996, pp. 79–95.

[36] M. Ellims, D. Ince, and M. Petre. “The Csaw C Mutation Tool: Initial Results”. In: *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*. 2007, pp. 185–192.

[37] H. Shahriar and M. Zulkernine. “Mutation-Based Testing of Format String Bugs”. In: *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE*. 2008, pp. 229–238.

[38] A. J. Offutt, J. Voas, and J. Payne. *Mutation operators for Ada*. Tech. rep. ISSE-TR-96-09. Information and Software Systems Engineering, George Mason University, 1996.

[39] S. Kim, J. A. Clark, and J. A. McDermid. “The Rigorous Generation of Java Mutation Operators Using HAZOP”. In: *12th International Conference on Software and System Engineering and their Application (ICSSEA’99)*. 1999.

[40] Y.-S. Ma, Y.-R. Kwon, and J. Offutt. “Inter-class mutation operators for Java”. In: *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*. 2002, pp. 352–363.

[41] J. Offutt, Y.-S. Ma, and Y.-R. Kwon. “The Class-level Mutants of MuJava”. In: *Proceedings of the 2006 International Workshop on Automation of Software Test*. AST ’06. New York, NY, USA: ACM, 2006, pp. 78–84.

[42] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. “MuJava: A Mutation System for Java”. In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE ’06. ACM, 2006, pp. 827–830.

[43] L. Deng, J. Offutt, and N. Li. “Empirical Evaluation of the Statement Deletion Mutation Operator”. In: *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. ICST ’13. IEEE Computer Society, 2013, pp. 84–93.

[44] M. E. Delamaro, J. Offutt, and P. Ammann. “Designing Deletion Mutation Operators”. In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. 2014, pp. 11–20. DOI: 10.1109/ICST.2014.12.

[45] C. Ji, Z. Chen, B. Xu, and Z. Wang. “A New Mutation Analysis Method for Testing Java Exception Handling”. In: *Computer Software and Applications Conference, 2009. COMPSAC ’09. 33rd Annual IEEE International*. Vol. 2. 2009, pp. 556–561.

[46] A. Derezińska. “Advanced Mutation Operators Applicable in C# Programs”. In: *Software Engineering Techniques: Design for Quality*. Ed. by K. Sacha. Vol. 227. IFIP International Federation for Information Processing. Springer US, 2006, pp. 283–288.

[47] A. Derezińska and A. Szustek. “Tool-Supported Advanced Mutation Approach for Verification of C# Programs”. In: *Proceedings of the 2008 Third International Conference on Dependability of Computer Systems DepCoS-RELCOMEX*. DEPCOS-RELCOMEX ’08. IEEE Computer Society, 2008, pp. 261–268.

[48] A. Derezińska. “Classification of Advanced Mutation Operators of C# Language”. In: *Information Systems Architecture and Technology, New Developments in Web-Age Information Systems*. Oficyna Wydawnicza Politechniki Wroclawskiej, 2010, pp. 261–271.

[49] A. Derezińska and M. Rudnik. “Quality Evaluation of Object-Oriented and Standard Mutation Operators Applied to C# Programs”. In: *Objects, Models, Components, Patterns*. Springer Berlin Heidelberg, 2012, pp. 42–57.

[50] P. Delgado-Pérez, I. Medina-Bulo, J. J. Domínguez-Jiménez, A. García-Domínguez, and F. Palomo-Lozano. “Class mutation operators for C++ object-oriented systems”. In: *annals of telecommunications - annales des télécommunications* 70.3 (2015), pp. 137–148. ISSN: 1958-9395. DOI: 10.1007/s12243-014-0445-4. URL: https://doi.org/10.1007/s12243-014-0445-4.

[51] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, and J. J. Domínguez-Jiménez. “Assessment of class mutation operators for C with the MuCPP mutation system”. In: *Information and Software Technology* 81 (2017), pp. 169 –184. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2016.07.002. URL: http://www.sciencedirect.com/science/article/pii/S0950584916301161.

[52] P. Delgado-Pérez, I. Medina-Bulo, S. Segura, A. García-Domínguez, and J. José. “GiGAn: Evolutionary Mutation Testing for C++ Object-oriented Systems”. In: *Proceedings of the Symposium on Applied Computing*. SAC ’17. New York,

NY, USA: ACM, 2017, pp. 1387–1392. ISBN: 978-1-4503-4486-9. DOI: 10.1145/3019612.3019828. URL: http://doi.acm.org/10.1145/3019612.3019828.

[53] P. Delgado-Pérez, I. Medina-Bulo, and M. Núñez. "Using Evolutionary Mutation Testing to improve the quality of test suites". In: *2017 IEEE Congress on Evolutionary Computation (CEC)*. 2017, pp. 596–603. DOI: 10.1109/CEC.2017.7969365.

[54] M. Kusano and C. Wang. "CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2013, pp. 722–725.

[55] W. Chan, S. C. Cheung, and T. H. Tse. "Fault-Based Testing of Database Application Programs with Conceptual Data Model". In: *Proceedings of the Fifth International Conference on Quality Software*. QSIC '05. IEEE Computer Society, 2005, pp. 187–196.

[56] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva. "Mutating Database Queries". In: *Information and Software Technology* 49.4 (2007), pp. 398–417.

[57] J. Tuya, M. J. Suarez-Cabal, and C. de la Riva. "SQLMutation: A tool to generate mutants of SQL database queries". In: *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*. 2006, pp. 1–1. DOI: 10.1109/MUTATION.2006.13.

[58] OASIS. *WS-BPEL 2.0*. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. [Online; accessed 01-April-2016]. 2007.

[59] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo. "Mutation Operators for (WS-BPEL) 2.0". In: *ICSSEA 2008: Proceedings of the 21th International Conference on Software and Systems Engineering and their Applications*. 2008.

[60] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo. "GAmera: An Automatic Mutant Generation System for WS-BPEL Compositions". In: *2009 Seventh IEEE European Conference on Web Services*. 2009, pp. 97–106. DOI: 10.1109/ECOWS.2009.18.

[61] F. Lonetti and E. Marchetti. "X-MuT: A Tool for the Generation of XSLT Mutants". In: *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*. 2010, pp. 280–285.

[62] Y. Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678.

[63] A. Acree. "On Mutation". PhD thesis. Georgia Inst. of Technology, 1980.

[64] T. A. Budd. “Mutation Analysis of Program Test Data”. PhD thesis. Yale University, 1980.

[65] A. P. Mathur. “Performance, effectiveness, and reliability issues in software testing”. In: *Computer Software and Applications Conference, 1991. COMPSAC ’91., Proceedings of the Fifteenth Annual International.* 1991, pp. 604–605.

[66] A. P. Mathur and W. E. Wong. *An Empirical Comparison of Mutation and Data Flow Based Test Adequacy Criteria.* Tech. rep. Purdue Univ, 1993.

[67] A. J. Offutt, G. Rothermel, and C. Zapf. “An Experimental Evaluation of Selective Mutation”. In: *Proceedings of the 15th International Conference on Software Engineering.* ICSE ’93. IEEE Computer Society Press, 1993, pp. 100–107.

[68] W. E. Wong and A. P. Mathur. “Reducing the cost of mutation testing: An empirical study”. In: *Journal of Systems and Software* 31.3 (1995), pp. 185 –196.

[69] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zaof. “An experimental determination of sufficient mutant operators”. In: *ACM Trans. Softw. Eng. Methodol.* 5.2 (1996), pp. 99 –118.

[70] D. Schuler and A. Zeller. “Javalanche: Efficient Mutation Testing for Java”. In: *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering.* ESEC/FSE ’09. ACM, 2009, pp. 297–298.

[71] X. Yao, M. Harman, and Y. Jia. “A Study of Equivalent and Stubborn Mutation Operators Using Human Analysis of Equivalence”. In: *Proceedings of the 36th International Conference on Software Engineering.* ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, 919–930. ISBN: 9781450327565. DOI: 10.1145/2568225.2568265.

[72] D. Baldwin and F. Sayward. *Heuristics for Determining Equivalence of Program Mutations.* Tech. rep. GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION and COMPUTER SCIENCE, 1979.

[73] D. Schuler and A. Zeller. “Covering and Uncovering Equivalent Mutants”. In: *Software Testing, Verification and Reliability* 23.5 (2012), pp. 353–374.

[74] R. A. DeMilli and A. J. Offutt. “Constraint-based automatic test data generation”. In: *IEEE Transactions on Software Engineering* 17.9 (1991), pp. 900–910. ISSN: 0098-5589. DOI: 10.1109/32.92910.

[75] J. Wang, S.-K. Kim, and D. Carrington. “Verifying metamodel coverage of model transformations”. In: *Software Engineering Conference, 2006. Australian.* 2006.

[76] A. Andrews, R. France, S. Ghosh, and G. Craig. “Test adequacy criteria for UML design models”. In: *Software Testing, Verification and Reliability* 13.2 (2003), pp. 95–127.

[77] F. Fleurey, B. Baudry, P.-A. Muller, and Y. L. Traon. “Qualifying input test data for model transformations”. In: *Software & Systems Modeling* 8.2 (2009), pp. 185–203. ISSN: 1619-1374. DOI: 10.1007/s10270-007-0074-8. URL: https://doi.org/10.1007/s10270-007-0074-8.

[78] T. J. Ostrand and M. J.Balcer. “The Category-partition Method for Specifying and Generating Fuctional Tests”. In: *Commun. ACM* 31.6 (1988), pp. 676–686.

[79] S. Sen, B. Baudry, and J.-M. Mottu. “Automatic Model Generation Strategies for Model Transformation Testing”. In: *Theory and Practice of Model Transformations*. Ed. by R. F. Paige. Vol. 5563. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 148–164.

[80] E. Guerra and M. Soeken. “Specification-driven model transformation testing”. In: *Software & Systems Modeling* 14.2 (2015), 623—644. DOI: https://doi.org/10.1007/s10270-013-0369-x.

[81] E. Guerra, J. de Lara, D. Kolovos, and R. Paige. “A visual specification language for model-to-model transformations”. In: *Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on*. 2010, pp. 119–126.

[82] L. M. Rose and S. Poulding. “Efficient probabilistic testing of model transformations using search”. In: *Combining Modelling and Search-Based Software Engineering (CMSBSE), 2013 1st International Workshop on*. 2013, pp. 16–21.

[83] S. Popoola, D. S. Kolovos, and H. H. Rodriguez. “EMG: A Domain-Specific Transformation Language for Synthetic Model Generation”. In: *Theory and Practice of Model Transformations*. Ed. by P. Van Gorp and G. Engels. Cham: Springer International Publishing, 2016, pp. 36–51. ISBN: 978-3-319-42064-6.

[84] J.-M. Mottu, B. Baudry, and Y. L. Traon. “Model transformation testing: oracle issue”. In: *Software Testing Verification and Validation Workshop, 2008. ICSTW ’08. IEEE International Conference on*. 2008, pp. 105–112.

[85] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. L. Traon, and J.-M. Mottu. “Barriers to systematic model transformation testing”. In: *Communications of the ACM* 53.6 (2010), pp. 139–143.

[86] O. Finot, J.-M. Mottu, G. Sunyé, and C. Attiogbé. “Partial Test Oracle in Model Transformation Testing”. In: *Theory and Practice of Model Transformations*. Ed. by K. Duddy and G. Kappel. Vol. 7909. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 189–204.

[87] M. Gogolla and A. Vallecillo. “Tractable Model Transformation Testing”. In: *Modelling Foundations and Applications*. Ed. by R. B. France, J. M. Kuester, B. Bordbar, and R. F. Paige. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 221–235.

[88] E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. “Automated verification of model transformations based on visual contracts”. In: *Automated Software Engineering* 20.1 (2013), pp. 5–46.

[89] M. Kessentini, H. Sahraoui, and M. Boukadoum. “Example-based model-transformation testing”. In: *Automated Software Engineering* 18.2 (2011), pp. 199–224.

[90] N. D. Matragkas, D. S. Kolovos, R. F. Paige, and A. Zolotas. “A Traceability-Driven Approach to Model Transformation Testing”. In: *Proceedings of the Second Workshop on the Analysis of Model Transformations, (AMT 2013)*. Ed. by B. Baudry, J. Dingel, L. Lucio, and H. Vangheluwe. Vol. 1077. CEUR Workshop Proceedings. CEUR-WS.org, 2013.

[91] N. Drivalos, D. S. Kolovos, R. F. Paige, and K. J. Fernandes. “Engineering a DSL for Software Traceability”. In: *Software Language Engineering*. Ed. by D. Gašević, R. Lämmel, and E. V. Wyk. Vol. 5452. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 151–167.

[92] D. S. Kolovos, D. D. Ruscio, A. Pierantonio, and R. F. Paige. “Different models for model matching: An analysis of approaches to support model differencing”. In: *2009 ICSE Workshop on Comparison and Versioning of Software Models*. 2009, pp. 1–6. DOI: `10.1109/CVSM.2009.5071714`.

[93] J. M. Küster and M. Abd-El-Razik. “Validation of Model Transformations – First Experiences Using a White Box Approach”. In: *Models in Software Engineering*. Ed. by T. Kühne. Vol. 4364. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 193–204.

[94] M. Schmidt and T. Gloetzner. “Constructing Difference Tools for Models Using the SiDiff Framework”. In: *Companion of the 30th International Conference on Software Engineering*. ICSE Companion ’08. New York, NY, USA: ACM, 2008, pp. 947–948. ISBN: 978-1-60558-079-1. DOI: `10.1145/1370175.1370201`. URL: `http://doi.acm.org/10.1145/1370175.1370201`.

[95] O. Semeráth, R. Farkas, G. Bergmann, and D. Varró. “Diversity of graph models and graph generators in mutation testing”. In: *International Journal on Software Tools for Technology Transfer* 22.1 (2020), 57—78. DOI: `https://doi.org/10.1007/s10009-019-00530-6`.

[96] J.-M. Mottu, B. Baudry, and Y. L. Traon. “Mutation Analysis Testing for Model Transformations”. In: *Model Driven Architecture - Foundations and Applications*. Ed. by A. Rensink and J. Warmer. Vol. 4066. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 376–390.

[97] J.-M. Mottu, S. Sen, M. Tisi, and J. Cabot. “Static Analysis of Model Transformations for Effective Test Generation”. In: *Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 291–300.

[98] E. Guerra. “Specification-Driven Test Generation for Model Transformations”. In: *Theory and Practice of Model Transformations*. Ed. by Z. Hu and J. de Lara. Vol. 7307. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012.

[99] V. Aranega, J.-M. Mottu, A. Etien, T. Degueule, B. Baudry, and J.-L. Dekeyser. “Towards an automation of the mutation analysis dedicated to model transformation”. In: *Software Testing, Verification and Reliability* 25.5-7 (2015), pp. 653–683.

[100] Y. Khan and J. Hassine. “Mutation Operators for the Atlas Transformation Language”. In: *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. 2013, pp. 43–52.

[101] J. Troya, A. Bergmayr, L. Burgueño, and M. Wimmer. “Towards Systematic Mutations for and with ATL Model Transformations”. In: *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*. 2015, pp. 1–10.

[102] J. S. Cuadrado, E. Guerra, and J. de Lara. “Static Analysis of Model Transformations”. In: *IEEE Transactions on Software Engineering* 43.9 (2017), pp. 868–897. ISSN: 0098-5589. DOI: `10.1109/TSE.2016.2635137`.

[103] J. S. Cuadrado, E. Guerra, and J. de Lara. “Towards effective mutation testing for ATL”. In: *2019 ACM/IEEE 22th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Munich, German, 2019.

[104] P. Gómez-Abajo, E. Guerra, and J. de Lara. “A domain-specific language for model mutation and its application to the automated generation of exercises”. In: *Computer Languages, Systems & Structures* 49 (2017), pp. 152 –173. ISSN: 1477-8424. DOI: `https://doi.org/10.1016/j.cl.2016.11.001`. URL: `http://www.sciencedirect.com/science/article/pii/S147784241630094X`.

[105] J. Gosling. *The Java language specification*. 2nd ed. Boston; London: Addison Wesley, 2000.

[106] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln. *Experimentation in Software Engineering.* Springer Publishing Company, Incorporated, 2012. ISBN: 3642290434, 9783642290435.

[107] R. Wei. “An Extensible Static Analysis Framework for Automated Analysis, Validation and Performance Improvement of Model Management Programs”. Doctor of Philosophy. Computer Science Department, The University of York, 2016.

[108] J. Sánchez Cuadrado, E. Guerra, and J. de Lara. “Reverse Engineering of Model Transformations for Reusability”. In: *Theory and Practice of Model Transformations.* Ed. by D. Di Ruscio and D. Varró. Cham: Springer International Publishing, 2014, pp. 186–201.

[109] B. J. Oakes, J. Troya, L. Lúcio, and M. Wimmer. “Fully verifying transformation contracts for declarative ATL”. In: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS).* 2015, pp. 256–265. DOI: 10.1109/MODELS.2015.7338256.

[110] L. Burgueño, J. Troya, M. Wimmer, and A. Vallecillo. “Static Fault Localization in Model Transformations”. In: *IEEE Trans. Software Eng.* 41.5 (2015), pp. 490–506.