An Extensible Static Analysis Framework for Automated Analysis, Validation and Performance Improvement of Model Management Programs

Ran Wei

Doctor of Philosophy

University of York
Computer Science

January 2016

Abstract

Model Driven Engineering (MDE) is a state-of-the-art software engineering approach, which adopts *model*s as first class artefacts. In MDE, modelling tools and task-specific model management languages are used to reason about the system under development and to (automatically) produce software artefacts such as working code and documentation.

Existing tools which provide state-of-the-art model management languages exhibit the lack of support for automatic static analysis for error detection (especially when models defined in various modelling technologies are involved within a multi-step MDE development process) and for performance optimisation (especially when very large models are involved in model management operations). This thesis investigates the hypothesis that static analysis of model management programs in the context of MDE can help with the detection of potential runtime errors and can be also used to achieve automated performance optimisation of such programs. To assess the validity of this hypothesis, a static analysis framework for the Epsilon family of model management languages is designed and implemented. The static analysis framework is evaluated in terms of its support for analysis of task-specific model management programs involving models defined in different modelling technologies, and its ability to improve the performance of model management programs operating on large models.

For my Parents Yingbin and Yongtian.

And in loving memory of my uncle, Yingji.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

I am most grateful to my supervisor, Dr. Dimitrios Kolovos for his invaluable guidance, support and encouragement throughout my research studies, altogether with his sincere friendship. I am also enormously grateful to Prof. Richard Paige, Dr. Louis Rose and Dr. Fiona Polack for their support and countless rewarding discussions.

I would like to thank my colleagues and friends Dr. Konstantinos Barmpis, Dr. Antonio Garcia-Dominguez, Dr. James Williams, Adolfo Sanchez-Barbudo Herrera, Athanasios Zolotas, Septavera Sharvia, and Dr. Frank Burton for their support and friendship.

I would also like to give credits to myself for my persisting endavour to carry out my "ideal" experiments, which were not included in this thesis – I should have finished 3 months earlier. #research.

I would also like to expression my warmest gratitude to my parents, Yingbin and Yongtian for the love and support they have provided to me throughout my research studies.

I would also thank my true friends based in the UK and China for their great deal of support during my studies.

At the end, I would like to thank my wife Yuan Tian (we got married 09/09/2016), who provided her warmest support and love whilst I was struggling with completing this thesis. I would have not done so well if not for her encouragements.

Author Declaration

Except where stated, all of the work contained in this thesis represents the original contribution of the author. This work has not been submitted for an award at this or any other institution. Parts of the work presented in this thesis have been previously published by the author:

- Ran Wei, Dimitrios S. Kolovos, Antonio Garcia-Dominguez, Konstantinos Barmpis, and Richard F. Paige. **Towards Partial Loading of XMI Models**. In *Proceedings of ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MoDELS 2016 (to appear).

- Antonio Garcia-Dominguez, Dimitrios S. Kolovos, Konstantinos Barmpis, Ran Wei and Richard F. Paige. **Stress-Testing Centralised Model Stores**. In *Proceedings of the 11th European Conference on Modelling Foundations and Applications*, ECMFA 2016, pages 48–63, Springer, 2016.

- Ran Wei and Dimitrios S. Kolovos. **An Efficient Computation Strategy for allInstances()**. In *Proceedings of the 3rd Workshop on Scalability in Model Driven Engineering*, BigMDE 2015, pages 48–57, CEUR-WS, 2015.

- Seyyed Shah, Ran Wei, Dimitrios S. Kolovos, Konstantinos Barmpis, Louis Rose and Richard F. Paige. **A Framework to Benchmark NoSQL Data Stores for Large-Scale Model Persistence.** In *Proceedings of ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems*, MoDELS 2014, pages 586–601, Springer, 2014.

- Ran Wei and Dimitrios S. Kolovos. **Automated Analysis, Validation and Suboptimal Code Detection in Model Management Programs.** In *Pro-*

ceedings of the 2nd Workshop on Scalability in Model Driven Engineering, BigMDE 2014, pages 32–41, CEUR-WS, 2014.

- Dimitrios S. Kolovos, Ran Wei and Konstantinos Barmpis. **An approach for Efficient Querying of Large Relational Datasets with OCL-based Languages.** In *Proceedings of the 2nd Extreme Modelling Workshop*, XM 2013, pages 48–57, Citeseer, 2013.

1. Introduction

The complexity of software grows as technologies advance. Today's software engineers build complex systems with interoperating components and sophisticated graphical user interfaces, due to the fact that computers are ubiquitous and software systems are adopted to process data in more fields and contexts. To address software complexity, a variety of software engineering methodologies and technologies, which aim to improve the quality of software and productivity of software developers, have been proposed and adopted over the years. Although methodologies and technologies may focus on different aspects and may vary significantly from each other, they share a common vision: to raise the level of abstraction at which software is designed and implemented.

This trend is evident from the shift of software engineering technologies. For example, software engineers have moved from assembly languages programming to procedure oriented programming (e.g. Fortran, C), object oriented programming (e.g. C++, Java, .NET) and aspect oriented programming (e.g. AspectJ). Designing and implementing software at a raised level of abstraction allows software engineers to manage software complexity by focusing on the more important aspects of the software system, in order to improve its quality and the productivity of the software engineers.

Model Driven Engineering (MDE) is a contemporary software engineering approach which enables engineers to abstract away technological details (such as programming languages) and focus on the problem domain of the system (specific domains such as bank account management systems, patient record management systems, etc.). To this extent, *models* are used to capture the relevant details of the problem domain. A MDE-based software development process is *driven* by performing a series of task-specific model management operations on the *models* to automatically produce software artefacts such as working code and documentation.

This chapter presents a brief overview of the current state of practice of Model Driven Engineering and highlights the problems that motivated the work of this thesis. The research hypothesis and methodology are also outlined and a summary of the results and the main contributions of the thesis is provided. Finally, this chapter provides an overview of the organisation of this thesis and a summary of the remaining chapters.

1.1. Motivation

This section presents a summary of the current state of practice of MDE and highlights the problems that motivated the work presented in this thesis.

1.1.1. Model Driven Engineering: State of Practice

The practice of MDE involves two important aspects, *modelling* and *model management*. *Modelling* is essentially a process of applying abstraction to describe a system within a problem domain. Models can be characterised as structured or unstructured. A structured model is a type of model which conforms to a set of syntactic well-formedness constraints which is known as the model's *modelling language* or *metamodel*. An unstructured model is a type of model which does not conform to any rules (or *metamodel*s). In MDE, developers have the freedom to define their own *modelling language*s which capture relevant concepts within their problem domains. Using such *modelling language*s, developers are able to describe their systems by producing models that conform to them. The current state of practice of MDE involves managing models defined in different technologies. Existing modelling technologies include the Eclipse Modelling Framework (EMF) [6], Meta Data Repository (MDR) [10], etc. Some tools also support models defined using plain XML, CSV, etc. [9]. Currently, *modelling language*s can also be defined using graphical notations (e.g. EMF) and textual notations (e.g. Emfatic [11], Xtext[12] etc.)

In order to deliver its promised benefits, MDE also relies on mechanisms that are able to automate a range of task-specific *model management* operations. For example, to realise the benefits of increased productivity in software construction, mechanisms that can automatically generate (programming language) source code from models are desirable. To enable interchange of models between different modelling languages/technologies, mechanisms that can automatically transform models defined in one modelling technology to models defined in other technologies are desirable. In the current state of practice of MDE, existing task-specific model management operations include model validation, text-to-model transformation, model-to-model transformation, model-to-text transformation, model comparison, model merging, etc. There is also a broad range of task-specific model management languages (backed by different tools/platforms) within the context of MDE which enable such model management operations (discussed in Chapter 2).

MDE provides well documented benefits over traditional approaches to software engineering. Case studies suggest that adopting MDE can improve the productivity by reducing the amount of time required to develop a system by means of automated model transformations. Adopting MDE also reduces the number of errors discovered throughout development [13]. In [14, 15], MDE has been shown to increase productivity by as much as a factor of 10. The use of MDE also introduces benefits in maintaining software systems. For example, to re-deploy an existing system onto another platform (possibly in another programming language), model transformations can be devised to automatically generate source code in the target programming language.

Whilst MDE brings many benefits, it faces a number of challenges. Firstly, as the languages and tools in MDE are relatively new (mature tools have been around for about 5 years), few tools support automatic analysis and validation of model management programs to detect potential runtime errors at compile time. On the other hand, as MDE has been increasingly applied to larger and more complex systems [16, 17] over the last decade, several studies have observed that existing tools are being stressed to their limits in terms of their capacity to *efficiently* support tasks such as management and persistence of large models with more than a few million model elements [18]. There are also many other challenges to MDE, such as learning curve and adoption [19]. This thesis focuses on the first two challenges identified.

1.1.2. Static Analysis in MDE

Software defects have been around as long as software. This is due to the fact that software is implemented by humans, and it is inevitable that human actions often produce incorrect results [20]. Fixing defects gets more expensive the later the defects are identified in the development process. Software companies typically spend more than 80% of their development budget on quality control [21]. Thus, it is desirable to identify potential defects in the software as early in the development process as possible.

A number of defect detection methodologies are used to identify potential defects in the software. Defect detection can be achieved through dynamic testing and static analysis. Dynamic testing and static analysis are often used together to ensure the quality of software. Apart from defect detection, static analysis can also be used to fulfil other purposes. For example, static analysis techniques are often used in compilers to optimise the performance of programs. For instance, data flow analysis [22] is often carried out by compilers to determine *very busy expression*s so that such expressions are evaluated only once at runtime to avoid heavy computations (discussed in Chapter 3).

In the context of MDE, despite the raised level of abstraction, model management programs are still likely to contain defects. Whilst dynamically testing model management programs has received much attention and a number of approaches have been proposed and implemented [23, 24], there is limited literature on static analysis in the context of MDE. Although a number of static analysis tools have been implemented for OCL, ATL, IncQuery, etc. [25, 26, 7, 8], the functionality of such tools is mostly limited to providing auto-completion facilities and defect detection facilities. Chapter 4 reviews, compares and highlights the limitations of existing static analysis tools in MDE. Throughout the review, the following limitations of existing static analysis tools in MDE have been identified:

- **Lack of support for analysis of cross-technology model management**. There is no support for static analysis of model management programs that involve models captured using different technologies (e.g. a transformation that produces an EMF model from an XML document or a set of constraints that validates a UML model against a spreadsheet-based model).

- **Little reuse between static analysers of different languages**. Although model management languages share a lot of functionalities (e.g. support for model navigation) and even syntax (e.g. many model management languages reuse subsets of OCL), they are typically implemented from scratch. As a result, static analysis facilities that could benefit a wide range of model management tasks (e.g. partial model loading – discussed in Chapter 9) need to be implemented a number of times on top of similar technical architectures.

- **Lack of automated performance analysis and optimisation through static analysis**. Existing static analysis tools for MDE are mostly limited to error detection and auto-completion. However, as discussed in Chapter 3, static analysis can also be used to optimise performance by identifying performance hotspots (e.g. very-busy-expression detection). In the context of MDE, there is no static analysis tool that provides the support for automated performance analysis and optimisation.

1.1.3. Scalability in MDE

Scalability has been recognised as a major challenge to the wider adoption of MDE [27]. As MDE is increasingly applied to larger and more complex systems, the current generation of modelling and model management technologies are being stressed to their limits in terms of their capacity to accommodate collaborative development, efficient management and persistence of models (in XMI[1] format) larger than a few hundreds of megabytes in size. In [18], the authors identify a number of scalability challenges that MDE needs to address towards its wider adoption, such as providing scalable domain specific languages, scalable query and transformation engines, scalable collaborative modelling and scalable model persistence. Therefore, it is imperative for MDE to achieve scalability across the MDE technical space so that it can remain relevant and continue to deliver its widely recognised benefits. A number of existing technologies/approaches have been proposed to target scalability, including building incremental query and transformation engines [28, 8], providing performance optimisation for transformations such as lazy loading [29]

[1] XML Metadata Interchange

and storing models using state-of-the-art database technologies as opposed to storing models in text-based technologies [30, 31, 32].

1.2. Hypothesis and Objectives

With respect to the current situation discussed above, the context of the research hypothesis is as follows:

A Model Driven Engineering process involves many different model management tasks such as validation, model-to-model transformation, model-to-text transformation, comparison and merging. Model management programs that automate such tasks inevitably contain defects. Currently, there is a number of static analysis tools built for independent MDE languages/tools, but their capabilities are limited in terms of their ability to handle models defined in diverse modelling languages. In addition, independent model management languages and tools may lead to consistency, reuse and interoperability problems. Thus, there is a need for a static analysis framework that can target a broad range of model management languages with consistent syntax, which are able to simultaneously manage models defined in diverse modelling languages/technologies. To this end, the Epsilon platform [9] has been selected as the research platform for the work presented in this thesis, due to its extensible support for a broad range of model management languages, and a variety of modelling technologies.

From a static analysis point of view, existing MDE static analysis tools lack the support for automated performance analysis and optimisation. Apart from defect detection, static analysis can also be used to optimise performance of programs. In the context of this thesis, the static analysis for model management programs may also be used to address the scalability challenges for MDE.

In this context, the hypothesis of this thesis is stated as follows:

Reusable static analysis facilities can be used to identify errors in different types of model management programs (e.g. model transformations, validation constraints) that operate on multiple models defined using diverse modelling technologies, and to enhance the performance of programs operating on large models.

The objectives of the thesis are:

- To build a static analysis framework for the Epsilon platform, atop which reusable static analysis tools can be developed;
- To build a facility which supports the analysis of programs that manage models defined in diverse modelling technologies;
- To use the framework to develop static analysis tools for the Epsilon model management languages demonstrating its reusability and extensibility;
- To use the static analysis framework to develop facilities for analysis and automated optimisation of the performance of programs operating on large models.

Although this research positions Epsilon as its research platform, the outcomes of this research are not bound to Epsilon. Since EOL re-uses a large part of OCL's (Object Constraint Language) syntax, the static analysis techniques presented in this thesis can be applied to any language with an OCL-like syntax without extensive changes. On the other hand, the means to address scalability through static analysis can be used as approaches to solve similar problems for other model management languages/tools.

1.3. Research Methodology

A typical software engineering process involving analysis, design, implementation and testing iterations has been followed to evaluate the research hypothesis.

1.3.1. Iterative Analysis

In the analysis phase, an in-depth analysis of the Epsilon Model Connectivity (EMC, Section 6.1) and Epsilon Object Language (EOL, Section 6.4.1) was performed to study how the static analysis framework can be implemented to achieve the same extensibility as EMC and EOL in order to construct the infrastructure of the static analysis framework.

After the infrastructure of the static analysis framework was constructed, analysis was performed to discover which static analysis technique was best suited for the purpose of this research, in order to construct the static analysis framework.

After the static analysis framework was implemented, analysis was performed on the Epsilon Validation Language (EVL, Section 8.1) and the Epsilon Transformation Language (ETL, Section 8.2) in order to implement static analysers for these two languages.

Once the static analysis framework was constructed, analysis was performed to discover how the static analysis framework could be extended to implement facilities that provide automated performance analysis and optimisation to address the scalability challenges from various aspects.

1.3.2. Iterative Design and Implementation

Following the first analysis iteration, an extensible model connectivity layer fulfilling the purpose of static analysis was designed and implemented. Altogether, a *metamodel* of EOL was designed and implemented, together with a facility that transforms EOL programs into EOL models that conform to the EOL metamodel.

Following the second analysis iteration, a static analysis facility which performs analysis on EOL programs was designed and implemented.

Following the third analysis iteration, the static analysis framework was extended to add the modules in order to support the analysis of programs written in EVL and ETL.

Following the fourth analysis iteration, automated performance analysis and optimisation facilities were designed and implemented which address the scalability challenges in MDE from different aspects.

1.3.3. Iterative Testing and Evaluation

Throughout the design and implementation phases, several case studies have been used to assess the quality and usefulness of the proposed approach and the correctness of the implementation. Significant feedback has been provided by academic peers who have reviewed publications on several aspects of the framework. Errors and design defects were identified throughout the testing and were considered in the next analysis, design and implementation iterations.

1.4. Research Results

As a result of this work, an extensible static analysis framework (named the Epsilon static analysis framework), which provides support for analysing model management programs written in Epsilon languages interacting with models defined in diverse modelling technologies, has been constructed. The Epsilon static analysis framework comprises two components. To access metamodels defined in different modelling technologies, an extensible model connectivity layer, the Epsilon Static Analysis Model Connectivity layer (ESAMC) was designed and implemented. To analyse programs written in Epsilon languages, a core static analyser for the Epsilon Object Language (EOL) was designed and implemented. These two infrastructural components enable the development of modelling technology specific drivers and of static analysers for other Epsilon languages. A schema-less XML driver was created atop ESAMC so that programs involving schema-less XML models can be analysed by the Epsilon static analysis framework. Static analysers for the Epsilon Validation Language (EVL) and the Epsilon Transformation Language (ETL) were also developed atop the core EOL static analyser so that EVL and ETL programs can also be analysed.

Apart from static analysis facilities for potential runtime error detection for programs written in EOL, EVL and ETL, this thesis has also contributed three automated performance analysis and optimisation facilities. A sub-optimal performance pattern detection facility has been constructed, which aims to detect sub-optimal source code patterns that can lead to potential performance degradation. A set of more efficient computation strategies for accessing model elements (the call to *allInstances()* and operations of the same nature) which exploit the results of static analysis, have been implemented, which demonstrates significant performance improvement. A facility has been constructed which is able to partially load XMI[2]-based models by exploiting the results of static analysis, which demonstrates significant improvements in loading time and resource consumption.

[2]XML Metadata Interchange

The hypothesis has been validated by demonstrating that programs which simultaneously manage models defined in different modelling technologies can be statically analysed to identify potential runtime errors, and that the results of static analysis can be exploited to achieve automated performance optimisation of programs that manage very large models.

1.5. Summary of Contributions

The contributions of the work presented in this thesis to the Epsilon platform, and to model management in general are listed as follows:

1.5.1. Contributions to Epsilon

To investigate and assess the validity of the hypothesis of this thesis, a static analysis framework for languages the Epsilon platform was developed. This contributed the following facilities to Epsilon:

- Ecore-based EOL, EVL and ETL metamodels, which formalise the respective languages' abstract syntaxes;
- AST2EOL, AST2EVL and AST2ETL transformations, which transform ANTLR-based homogeneous abstract syntax trees into instances of EOL, EVL and ETL metamodels;
- Epsilon Static Analysis Model Connectivity layer (ESAMC), an enhanced version of Epsilon Model Connectivity layer (EMC) that provides interfaces for accessing metamodels defined in different modelling technologies in a uniform way;
- EOL, EVL and ETL static analysers, which formalise the scoping rules for variable resolution, and the type resolution semantics of the respective languages.

1.5.2. Contributions to Model Management

In terms of contributions that are not bound to Epsilon, this thesis demonstrated that:

- Meaningful static analysis of programs that involve models defined in diverse modelling technologies is feasible and practical.

- The results of static analysis can be used to reason about and to automatically optimise the performance of model manamgement programs operating on large models. More specifically by leveraging the results of static analysis:

  - Sub-optimal performance pattterns can be identified;

  - Efficient computation and caching strategies can be defined for computationally expensive operations such as collecting all instances of a type in a model (e.g. the *allInstances()* operation);

  - Partial loading of XMI models can be achieved.

1.6. Thesis Structure

In Chapter 2, a detailed review of Model Driven Engineering is performed. Section 2.1 discusses the terminologies and principles of MDE. The concept of *model* and *modelling language* are discussed with examples. Different types of *model management* operations are also discussed with their corresponding literature and tools. Section 2.2 discusses MDE tools, and focuses on EMF and Epsilon, which are relevant to the research of this work.

In Chapter 3, a detailed review of static program analysis is performed. Section 3.1 discusses the origin of software defects and why they are inevitable; Section 3.2 discusses different means of defect detection; Section 3.3 discusses the characteristics of static analysis; and Section 3.4 discusses the techniques that are adopted by contemporary static analysis facilities.

In Chapter 4, a number of existing static analysis tools in the context of MDE are reviewed. Section 4.1 discusses the need for static analysis tools in the context of MDE; Section 4.2 presents the review strategy of the static analysis tools and outlines what attributes the review is focused on; Section 4.3 reviews the static analysis tools for Dresden OCL and Eclipse OCL; Section 4.4 reviews the static analysis tools for Eclipse ATL; Section 4.5 and 4.6 reviews the static analysis tools for Acceleo and Xpand; Section 4.7

reviews the static analysis tool for EMF Inc-Query. Section 4.9 presents the findings from the review of the static analysis tools.

Chapter 5 summarises the findings of the review performed in Chapter 2, 3 and 4 and identifies the shortcomings of contemporary static analysis tools in MDE. More specifically, Section 5.1 performs the research analysis and identifies the shortcomings of existing static analysis tools in MDE. Then, the analysis identifies the research platform on which the research of this thesis is carried out. In Section 5.2, the research hypothesis is stated and a set of objectives for validating the proposed hypothesis are outlined. In Section 5.3, the research scope is outlined, and in Section 5.4, the research methodology is discussed which evaluates the validity of the hypothesis.

Chapter 6 discusses the first development iteration which constructed the infrastructure of the static analysis framework. Section 6.1 discusses the structure of the Epsilon platform and identifies its main components. Section 6.2 discusses the existing Epsilon Model Connectivity layer (EMC) and its functionality. Section 6.3 identifies a few shortcomings of EMC with regard to static analysis and proposes the Epsilon Static Analysis Model Connectivity (ESAMC) layer, which is designed specifically for the static analysis framework. Section 6.4 discusses the infrastructure of the Epsilon static analysis framework, which includes the discussion of the created EOL metamodel, and the EOL program to EOL model transformation.

Chapter 7 discusses the EOL static analyser. Section 7.1 identifies the techniques used by the EOL static analyser and discusses its design. Section 7.2 discusses a utility facility, named the EOL visitor, which is created by a model-to-text transformation which is able to automatically construct visitor facilities based on Ecore models. Section 7.3 and Section 7.4 discusses the variable resolution and the type resolution processes of the static analysis.

In Chapter 8, the EOL static analyser is extended to create the EVL static analyser and the ETL static analyser. Section 8.1 and Section 8.2 discuss the EVL and ETL static analysers, including the EVL and ETL metamodels, the EVL and ETL visitor frameworks, the EVL and ETL variable resolution and type resolution facilities. Section 8.2.5 discusses the transformation rule dependency calculation for ETL transformation rules and its potential application.

Chapter 9 discusses the evaluation of the Epsilon static analysis framework. The extensibility of the static analysis framework is illustrated by the construction of the EVL and ETL static analysers. The extensibility of the Epsilon Static Analysis Model Connectivity (ESAMC) is evaluated by the construction of a driver that adds support for managing schema-less XML documents for the static analysis framework in Section 9.1. Examples are also provided to demonstrate how plain XML models and EMF models can be managed within a single ETL transformation. The evaluation then progresses to the EOL, EVL and ETL static analysers in Section 9.2, where existing model management programs are analysed using these static analysers and the identified defects are reported.

Chapter 10 presents the applications of static analysis which aim to address scalability problems in MDE. In Section 10.1, a sub-optimal perform pattern detection approach is discussed which, by using pattern matching techniques, is able to detect potential performance degradation patterns by analysing EOL programs together with the models they interact with. Section 10.2 discusses a facility integrated into the Epsilon execution engine, which provides more efficient computation strategies to compute calls to *allInstances()* (and calls to allOfKind(), allOfType() and all()), so that the execution of Epsilon programs can be optimised at runtime. Benchmarks involving running programs against very large models are reported in Section 10.2.7. Section 10.3 discusses a facility named SmartSAX which integrates static analysis with an enhanced version of the EMF SAX (Simple API for XML) parser which realises partial loading of EMF models. Such work involves an automated effective metamodel extraction facility presented in Section 10.3.4. SmartSAX is then used to run EOL programs on very large models, which benchmarks the resource consumption of the partial loading algorithm. The benchmarks are reported in Section 10.3.7.

Chapter 11 concludes by summarising the findings of this thesis and providing directions to further work in the field of static analysis of model management programs.

2. Background: Model Driven Engineering

Model Driven Engineering (MDE) is a contemporary software development approach. In an MDE process, models are first class artefacts. Models are used to capture relevant details of a system under development and are used, by different model management operations in an automated manner, to reason about the system and to generate software development artefacts such as partial (or complete) implementation of the system or documentations. This chapter presents a detailed review of MDE. Section 2.1 introduces the terminology and fundamental principles used in MDE, the development methodologies and guidance for MDE, and model management operations and corresponding languages which support various model management operations. Section 2.2 discusses existing MDE technologies and platforms. Section 2.3 discusses the challenges to MDE. Finally, Section 2.4 summarises this chapter.

2.1. Terminologies and Principles of Model Driven Engineering

Compared to traditional software engineering approaches, in an MDE-based software engineering process, engineers construct and manipulate similar artefacts (such as code and documentation). However, in addition to traditional approaches, MDE approaches involve working with different types of artefacts, such as *metamodels*, *models* and *model management programs*. This section describes the terminologies, principles, artefacts and activities involved in MDE.

2.1.1. Models

To talk about models, it is necessary to talk about abstraction. Psychologically, the human mind subconsciously and continuously re-establishes reality by applying cognitive processes that alter the subjective perception of it. Among the various cognitive processes applied, abstraction is one of the most prominent ones [33]. In principle, abstraction is used to:

- generalise specific features of real world objects (generalisation);
- classify objects into coherent clusters (classification); and
- aggregate objects into more complex ones (aggregation).

Generalisation, classification and aggregation represent natural behaviours that the human mind is natively able to perform in everyday life. Abstraction is also widely applied in science and technology, where it is often referred to as *modelling*. People can informally define a *model* that is a simplified version of reality. *Model*s fulfil different purposes, such as to provide views of a phenomenon from different angles, or to reach an agreement on a topic, etc. Therefore, by definition, a *model* is a subset of reality that describes it as abstractly as needed.

*Model*s have been and still are of great importance in many scientific contexts. For example, the *uniform motion model* in physics is something that does not exist in the real world, but is very useful in understanding the theory and for delivering the theory in teachings. It also acts as the basis for subsequent and more complex theories.

*Model*s are also created for various purposes. *Model*s can be *descriptive* of the reality of a system or a context. *Model*s can also be *prescriptive*, used to determine the scope and details of a problem, or to define how a system should be implemented.

On a philosophical level, it stands true that "everything is a model" [34], since it is the way of the human mind to perceive and process things by "modelling" them. This explains the fact that models have become crucial also in technical fields such as mechanics, civil engineering, computer science and computer engineering. In the context of production processes, modelling enables engineers to investigate, verify, document and discuss the properties of products before they are produced. In many cases, models are even used for directly automating the production of goods.

MDE, which adopts *model*s as first class artefacts, has been shown to increase efficiency and effectiveness in software development, as demonstrated by various quantitative and qualitative studies [35]. In a software engineering process which adopts the MDE approach, according to [33], *model*s can be used in various ways:

- models as sketches: models are used for communication purposes. Only partial views of the system are specified;
- models as blueprints: models are used to provide a complete and detailed specification of the system; and
- models as programs: models, instead of code, are used to develop the system.

Thus, in MDE, a *model* is an abstract representation of a system of a problem domain, which is created by software engineers to capture only the relevant details of such system.

2.1.2. Modelling Languages

*Model*s can be generally characterised as structured and unstructured, depending on whether they conform to rigorously specified rules. Structured models have rigorously defined rules to which they must comply (e.g. notations that the models must use/not use). On the other hand, unstructured *model*s are artefacts that do not conform to any rules. Thus, the users of unstructured models are free to express their views without notational/semantic restrictions. Whiteboard drawings and low fidelity prototypes are examples of unstructured models.

In MDE, structured, rather than unstructured, models are used [36]. A structured model is defined by a set of syntactic and semantic well-formedness constraints. In the context of MDE, these rules are encoded in a *modelling language*. Often, a *modelling language* is specified as a model, hence *modelling languages* are also referred to as *metamodels*.

Between a *model* and a *metamodel*, there exists a relationship known as *conformance*. A *model* is said to *conform* to a *metamodel* when every concept used in the *model* is specified in the *metamodel* [34]. *Conformance* can be described by a set of constraints between *models* and *metamodels* [37]. For example, a *conformance* constraint might state that for an *attribute a* of a *Type T*, *a* should be single-valued. When all the constraints are satisfied by the *model*, the *model* is said to *conform* to its *metamodel*.

A metamodel typically encompasses three types of constraints:

- **The abstract syntax** is the set of concepts defined by the metamodel. Examples of such concepts are packages, classes, attributes, etc. The abstract syntax is a set of abstract concepts. Therefore, its representation can be of any form. For example, a program compiler may use abstract syntax trees to represent the instances of the abstract syntax of a programming language, whereas the instances of the abstract syntax of the programming language can also be represented as program source code.
- **The concrete syntax** provides a notation to represent the *abstract syntax* of the metamodel. For example, the concepts of the metamodel, such as classes and references, can be represented as a collection of boxes connected by lines. Concrete syntax may be optimised for consumption by machines (e.g. stored in files using formats such as XML Metadata Interchange (XMI) [38]) or by humans (e.g. graphical syntax of Unified Modelling Language [1]).
- **The semantics** provides the meaning of the concepts with respect to the *problem domain*. The semantics of a metamodel may be specified rigorously, by defining constraints in a language, such as the Object Constraint Language (OCL) [39], or in a semi-formal manner by employing natural languages.

Abstract syntax, concrete syntax and semantics are used together to specify metamodels (modelling languages) [40].

2.1.3. Meta Object Facility: A metamodelling language

The Object Management Group (OMG)[1] has standardised a language for specifying metamodels, the Meta-Object Facility (MOF). MOF originated in the Unified Modelling Language [1]. MOF enables developers to define the abstract syntax of modelling languages. MOF is complemented by the Object Constraint Language (OCL) [41], a formal language that can be used to define model constraints in terms of predicate logic.

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

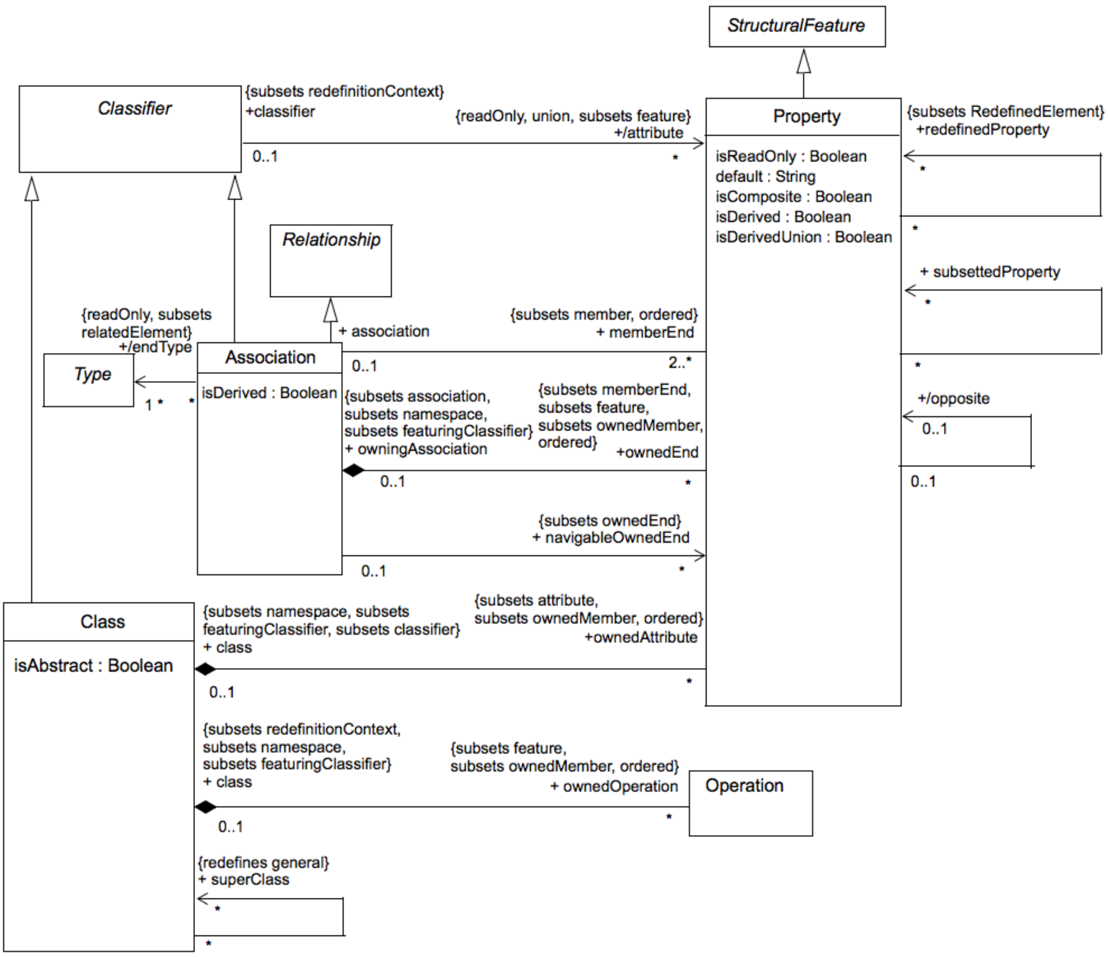


Figure 2.1.: A fragment of the UML metamodel defined in MOF, from [1].

For MOF, models are serialised/represented with another OMG standard, the XML Metadata Interchange (XMI, [38]), which is a dialect of XML to support the storage, loading and exchange of models.

MOF is a modelling language for defining modelling languages. It is sometimes also referred to as a *metamodelling language* or *metametamodel* because the language concepts introduced in MOF also define MOF itself. Figure 2.1 shows part of the UML metamodel defined in MOF, which uses the concrete syntax similar to that of UML class diagrams.

The purpose of the MOF standard is to enhance consistency in the way in which modelling languages are specified. Without a standardised metamodelling language, modelling tools can have diverse modelling languages, which makes interoperability challenging. With a common metamodelling language in place, tools can create modelling languages with the metamodelling language and exchange such modelling languages with no compatibility issues. Thus, a standardised metamodelling language promotes modelling tool interoperability.

2.1.4. An Example

To illustrate the terms *model*, *metamodel*, *metametamodel* , and *conformance*, an example adopted from [2] is provided.

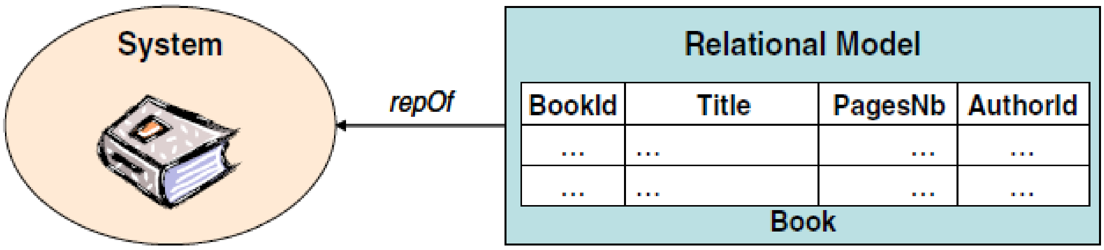


Figure 2.2.: A relational model represents the books in a library from [2]

In Figure 2.2, a *model* named "Relational Model" (right part of the figure) is presented, which is a possible representation of a collection of books in a library (left part of the figure) stored in a relational database.

The *conformance* relationship of the *Relational Model* and its *metamodel*, called *Relational Metamodel*, is described in Figure 2.3. The *Relational Metamodel* defines concepts such as *Table*, *Column* and *Type*, instances of which are used to define the *Relational Model*.

Figure 2.4 illustrates the *conformance* relationship (or meta-relationship [2]) between the instances of the concepts used in the *Relational Model* and the concepts defined in the *Relational Metamodel*. In Figure 2.4, in the *Relational Model*, *Book* is an instance of *Table* defined in the *Relational Metamodel*, whereas *BookId*, *Title*, *PagesNb* and *AuthorId* in the *Relational Model* are instances of *Column* in the *Relational Metamodel*, and finally *String* and *Int* in the *Relational Model* are instances of *Type* in the *Relational Metamodel*.

Figure 2.5 illustrates the *conformance* relationship between the instances of the con-

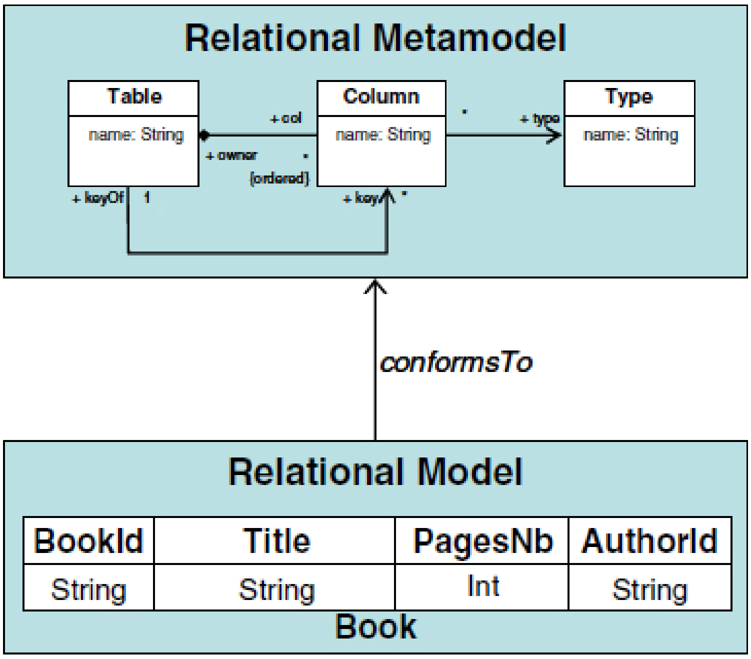


Figure 2.3.: The “conformance” relation between a model and its metamodel from [2]

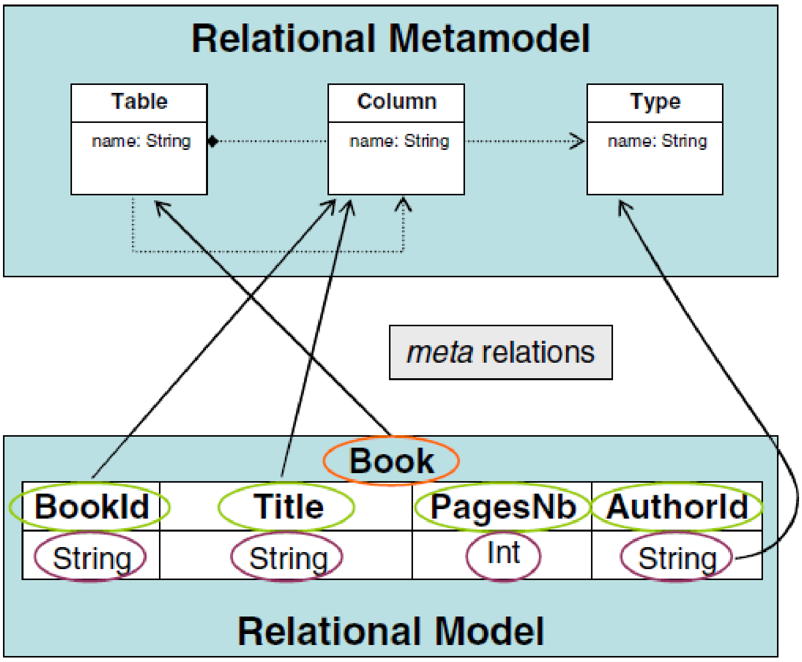


Figure 2.4.: The “meta” relation between model and metamodel elements [2]

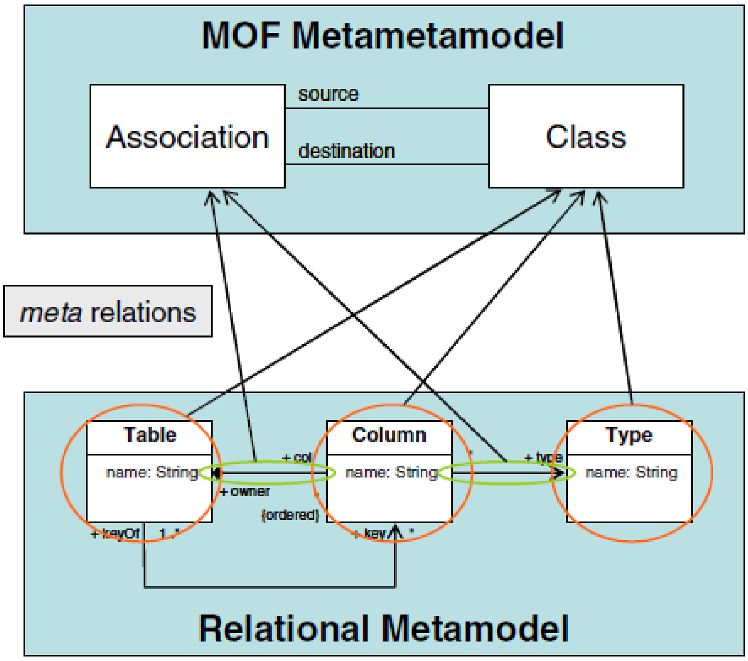


Figure 2.5.: The “meta” relation between metamodel and metametamodel elements [2]

cepts used in the *Relational Metamodel* and the concepts defined in MOF. In MOF, an entity named *Class* is defined which is used to define entities in a *modelling language*. In this example, *Table*, *Column* and *Type* in the *Relational Metamodel* are instances of *Class*. The *Associations* in the *Relational Metamodel*, such as *owner*, *col* etc. are instances of *Association* defined in MOF. As a *metamodelling language*, the concepts defined in MOF also define MOF itself. For example, in Figure 2.5, concept *Class* in MOF is used to define both *Class* and *Association*, whereas the concept *Association* is used to define the association between *Association* and *Class* (the *source* and *destination* associations).

2.1.5. Metamodelling Architectures and Domain Specific Modelling

Layers of abstraction for the *real system*, the *model*, the *metamodel* and the *metametamodel* in the example are a reflection of the structure of a typical *metamodelling framework*. A metamodelling framework typically provides a three-layered (M1-M3) hierarchical architecture, as seen in Figure 2.6. The M3 layer contains the core *metamodelling language*, which is used to define *modelling languages*. The M2 layer contains the *mod-*

elling languages defined using the *metamodelling language* in the M3 layer, whereas the M1 layer contains the *model*s devised from the *modelling language*s defined in the M2 layer. The *model*s in the M1 layer represent the real systems in the M0 layer.

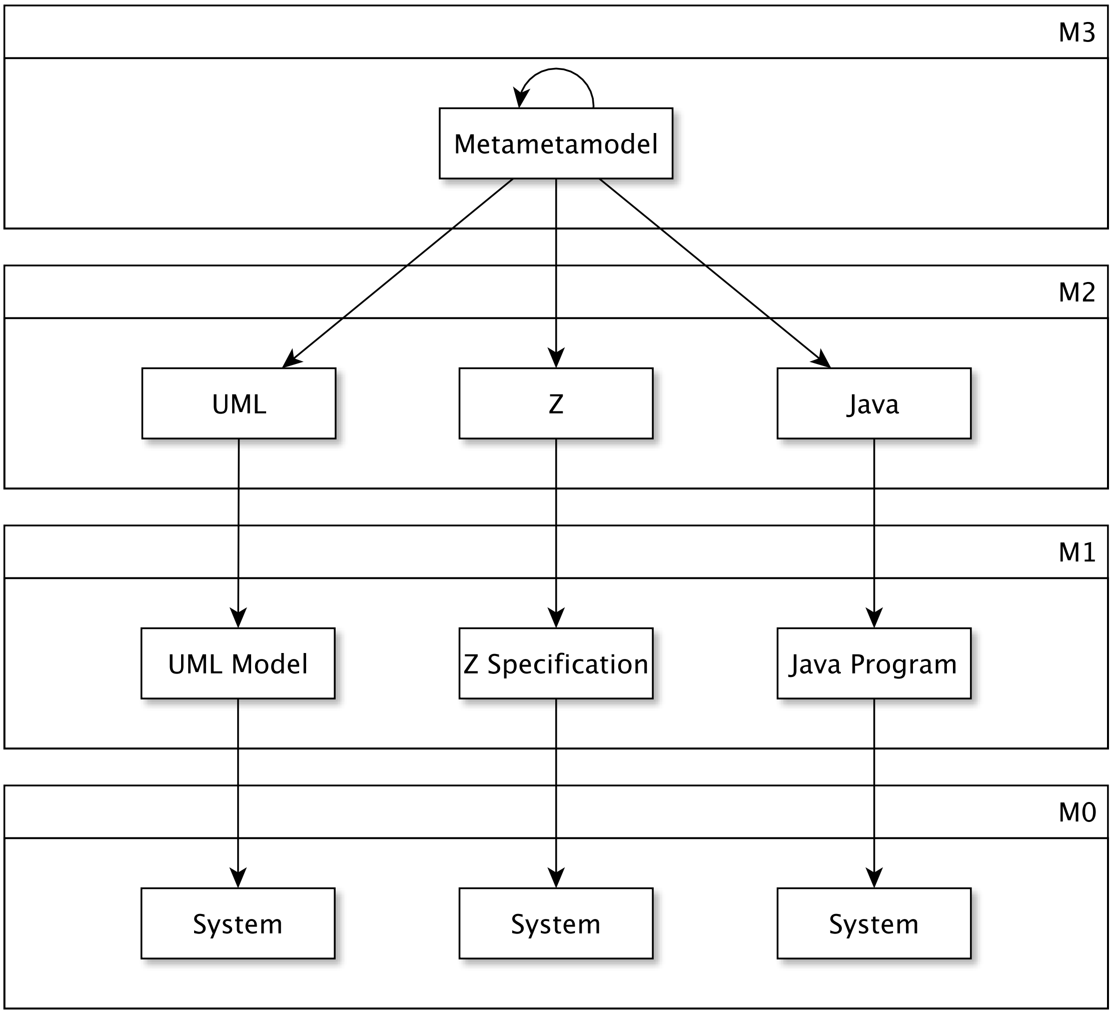


Figure 2.6.: The four layer metamodelling architecture.

There are many choices for solutions when addressing a set of related problems. [42] states that for a set of related problems, a specific approach tailored to target only the problems is likely to provide better outcomes than a generic approach designed to target all problems. The set of problems in this context is referred to as the *domain* or the *problem domain*. A *Domain Specific Language* (DSL) is a language that is designed specifically for describing a certain technical or business domain. Examples of DSLs include: HTML markup language for web page development, MatLab for mathematics, SQL for database management, etc. If a DSL is aimed at modelling, it may also be referred to as *Domain Specific Modelling Language* (DSML). Examples of DSMLs include OMG's Business Process Model Notation (BPMN) [43] to model business processes, Open Group's ArchiMate [44] for enterprise architecture modelling, OMG's MARTE [45] for modelling and analysing real time and embedded systems, etc.

The use of Domain Specific Languages to represent the various facets of a system is referred to as *Domain Specific Modelling*, and is identified as one of the fundamental aspects of MDE [46]. Several metamodelling frameworks have been proposed to allow domain specific modelling, including the OMG Meta Object Facility (MOF) [47], the Eclipse Modelling Framework (EMF) [6], the Microsoft Domain Specific Languages Tools [48], the Generic Modelling Environment (GME) [49], etc.

2.1.6. Model Management Operations and Tools

Domain Specific Modelling enables software engineers to define modelling languages and create models that conform to them. In MDE, models are processed/manipulated to produce software development artefacts. In [33], the authors suggest that the core concepts of MDE are *models* and *transformation*s. However, research shows that in an MDE-based software development process, other operations, such as model validation, model comparison, model merging, etc. are of equal importance and are also frequently performed throughout MDE development processes [36]. Collectively, such operations, as suggested by [36], are referred to as *model management* operations. This section presents an overview of frequently performed *model management* operations and existing tools that support such operations.

Model-to-model Transformation

Model-to-model transformations are considered to be of great importance to MDE [50]. According to [51], model transformations can be categorised as text-to-model transformations, model-to-model transformations and model-to-text transformations. In general, a model-to-model transformation operation has the form depicted in Figure 2.7. In a model-to-model transformation, the input and the output of a transformation are termed its *source* and *target* respectively. To transform the source *Ma* to a target *Mb* (where *Ma* conforms to its metamodel *MMa* and *Mb* conforms to its metamodel *MMb*, both metamodels *MMa* and *MMb* conform to metametamodel *MMM*), the transformation is done by executing the set of transformation rules *Mt*, which conforms to its metamodel *MMt* (in this case, a model transformation language).

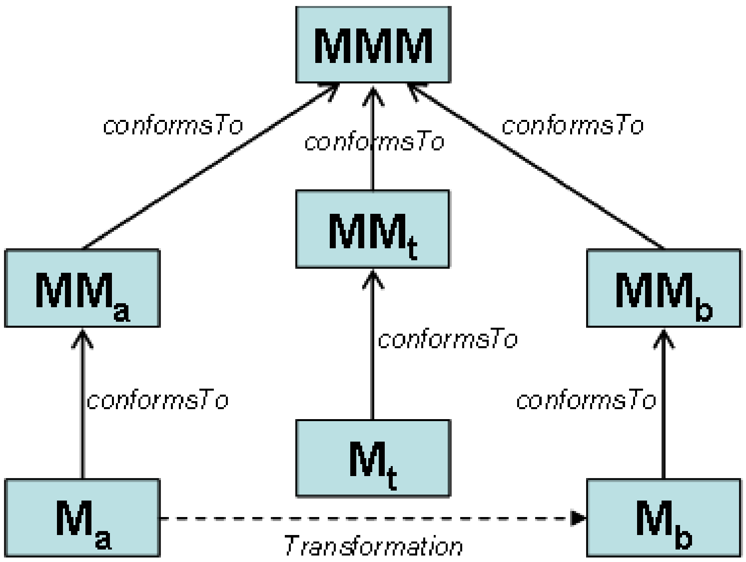


Figure 2.7.: The form of a model transformation [3]

Model transformations can generally be characterised by the number of source models and the number of target models involved in a model transformation. In a model transformation, when multiple source models are transformed into one target model, the transformation is normally referred to as *model merging* [52]; when one source model is transformed into one target model, it is a general model transformation; when one source model is transformed into multiple target models, it is normally due to the need to represent the same information in different representations; when any number of source model(s) are used but the transformation produces only Strings, it is normally referred to as a or *model query*.

With regard to the metamodels of the source and target models, two types of model transformations are generally recognised: *Exogenous Transformations* and *Endogenous Transformations* [53]. *Exogenous Transformations* refers to model transformations where the source model(s) and the target model(s) conform to different metamodels. *Endogenous Transformations* refers to model transformations where only the input models are modified.

Exogenous Transformation is also called *mapping transformation* or *translation transformation*. As its name suggests, a translation transformation is an algorithm that defines how a number of source models are mapped to a set of target models [36]. Translation transformations are predominantly rule-based. There are generally three model transformation styles:

- *Imperative Transformation* is the transformation style where the mappings between elements of the source model and the target model are directly and explicitly specified in an executable language (such as XTend [54] and Kermeta [55]). Imperative transformation style allows complex transformation rules to be created. However, imperative transformation languages have limitations, such as the overhead to implement scheduling and traceability, the difficulty of reusing transformation rules [36], etc.

- *Declarative Transformation* is the transformation style where the mappings between elements of the source model and the target model are specified using declarative constraints. In [56], a transformation framework is proposed, which uses OCL to declare the relations between source and target elements. Such relations are then translated into executable Java code which implements the transformation. A similar approach is adopted in [57], except OCL expressions are translated into XSLT to implement transformations.

  Declarative model transformations are typically carried out based on the principles of graph transformations. Examples of using the principle for declarative model transformations as graph transformations are QVT-R [58] and the *Triple Graph Grammar* [4]. A *Triple Graph Grammar* consists of the source metamodel, the target metamodel and the correspondence model (which conforms to the correspondence metamodel) that links the elements between the source metamodel and

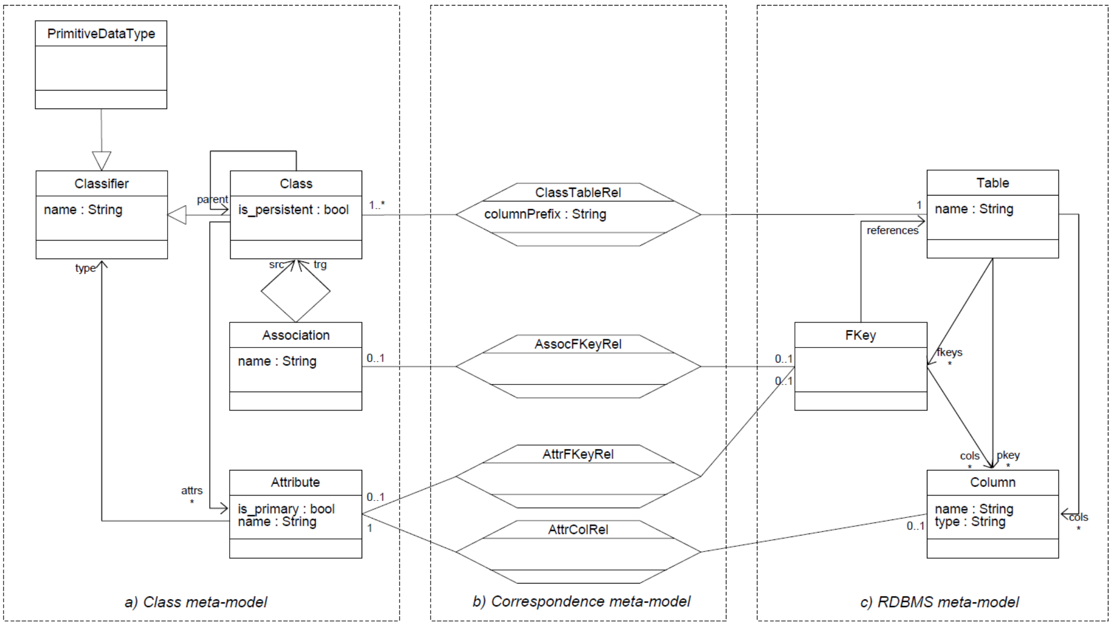


Figure 2.8.: An example of a triple graph grammar [4]

the target metamodel. A transformation engine that supports triple graph grammar transformations performs pattern matching of the source model and produces the target model using the transformation correspondence metamodel. An example of the approach is illustrated in Figure 2.8, which transforms *Class* models into *RDBMS* (Relational Database Management System) models. In Figure 2.8a, the *Class* metamodel is provided. The *Class* metamodel defines entities such as *Class*, *Association*, *Attribute*, etc. In Figure 2.8c, the *RDBMS* metamodel is provided. The *RDBMS* metamodel defines entities such as *Table*, *FKey* (Foreign Key), and *Column*. In Figure 2.8b a correspondence metamodel, which defines correspondences between objects, is presented. The entity *ClassTableRel* states that, on the one hand, each *Class* corresponds to one *Table*. On the other hand, each *Table* corresponds to at least one *Class*. This relationship can be observed from the multiplicity at the end of the association. The class *AttrColRel* links one *Attribute* with at most one *Column*, whereas each *Column* is linked with exactly one *Attribute*. The class *AttrFKeyRel* associates an *Attribute* with its *is_primary* field being true with a *Fkey*. Similarly, the class *AttrColRel* associates an *Attribute* with its *is_primary* being false with a column. The triple graph transformation engine takes this triple graph grammar and performs the transformation from Class to RDBMS according to the link metamodel provided in Figure 2.8b.

Declarative Transformation languages provide developers with a higher level of abstraction than imperative transformation languages, which makes transformations easier to specify, but demonstrates limitations to specify complex model transformations [59].

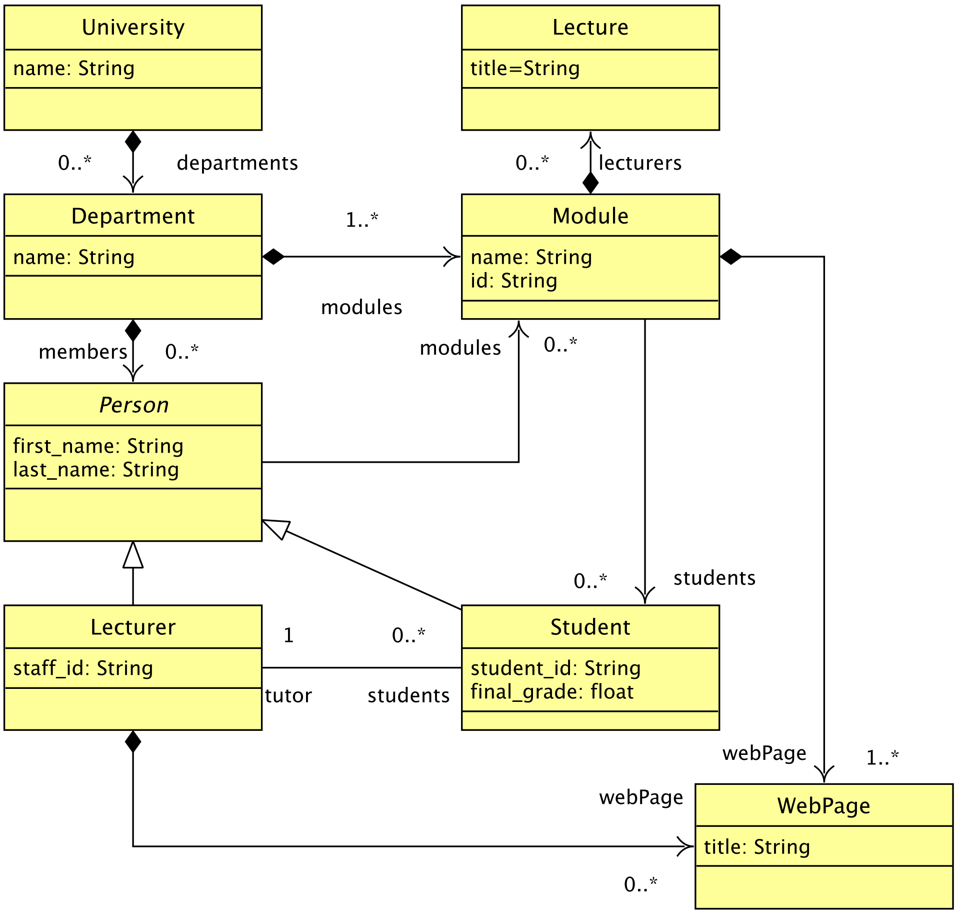


Figure 2.9.: A simple university metamodel

- *Hybrid Transformation* is the transformation style that enhances declarative transformation approaches with imperative features, so that complex transformation rules can be specified while preserving the desirable features of declarative transformation languages. Therefore, the hybrid style can be used to solve most of the needs for model transformations. Examples of hybrid transformation style languages are the Atlas Transformation Language (ATL) [60] and the Epsilon Transformation Language (ETL) [61].

An example of declarative M2M transformation written in the Epsilon Transformation Language (ETL) is provided in Listing 2.1. The source of the transformation is a university model conforming to the metamodel shown in Figure 2.9. The target of the transformation is a social network model conforming to the metamodel shown in Figure 2.10.

The first rule (line 1-7), named Student2Person, transforms *Student*s into *Person*s: the body of the rule specifies that the *first_name* and *last_name* should be copied over and the *Person*(s) a *Student knows* are derived from the student's tutor in the *University metamodel*.

```
1  rule Student2Person
2  transform s: Student
3  to p: Person {
4    p.first_name = s.first_name;
5    p.last_name = s.last_name;
6    p.knows = s.tutor.equivalent();
7  }
8  rule Lecturer2Person
9  transform l: Lecturer
10 to p: Person {
11   p.first_name = l.first_name;
12   p.last_name = l.last_name;
13   p.knows = l.students.equivalents();
14 }
```

Listing 2.1: An example of model-to-model transformation written in Epsilon Transformation Language [61].

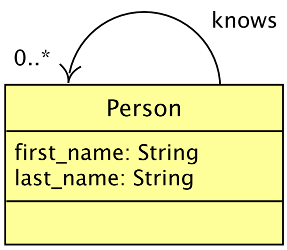


Figure 2.10.: A simple social network metamodel

The second rule (line 8-14), named Lecturer2Person, transforms the *Lecturer*s into *Person*s. Like rule 1, the names are copied over and the *Person*(s) a *Lecturer* *knows* are derived from the *Lecturer*'s supervised *Student*s.

At runtime, the transformation rules will be scheduled *implicitly* by the execution engine, and invoked for each *Lecturer* and *Student*. On line 6, the built-in operation *equivalent()* is used to produce a *Lecturer* by invoking the corresponding transformation rule (rule *Lecturer2Person* in this example). The call to *equivalent()* is an example of explicit rule scheduling, in which the developer defines when a rule is invoked.

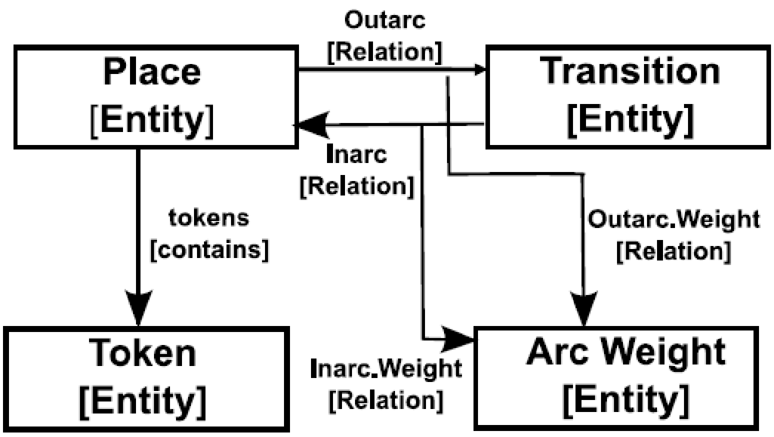


Figure 2.11.: The metamodel of a PetriNet, extracted from [5]

Endogenous Transformation is also called update transformation, or rephrasing transformation [53], and is used to perform modifications of existing models. Rephrasing transformations can be further classified as update transformations in the small and in the large. Rephrasing transformations in the large applies to sets of model elements for which model transformation rules are executed in a batch manner. On the other hand, update transformation in the small may require user intervention since the model elements which require update are normally specified by the user. The Epsilon Wizard Language [62] is an example of endogenous transformation language.

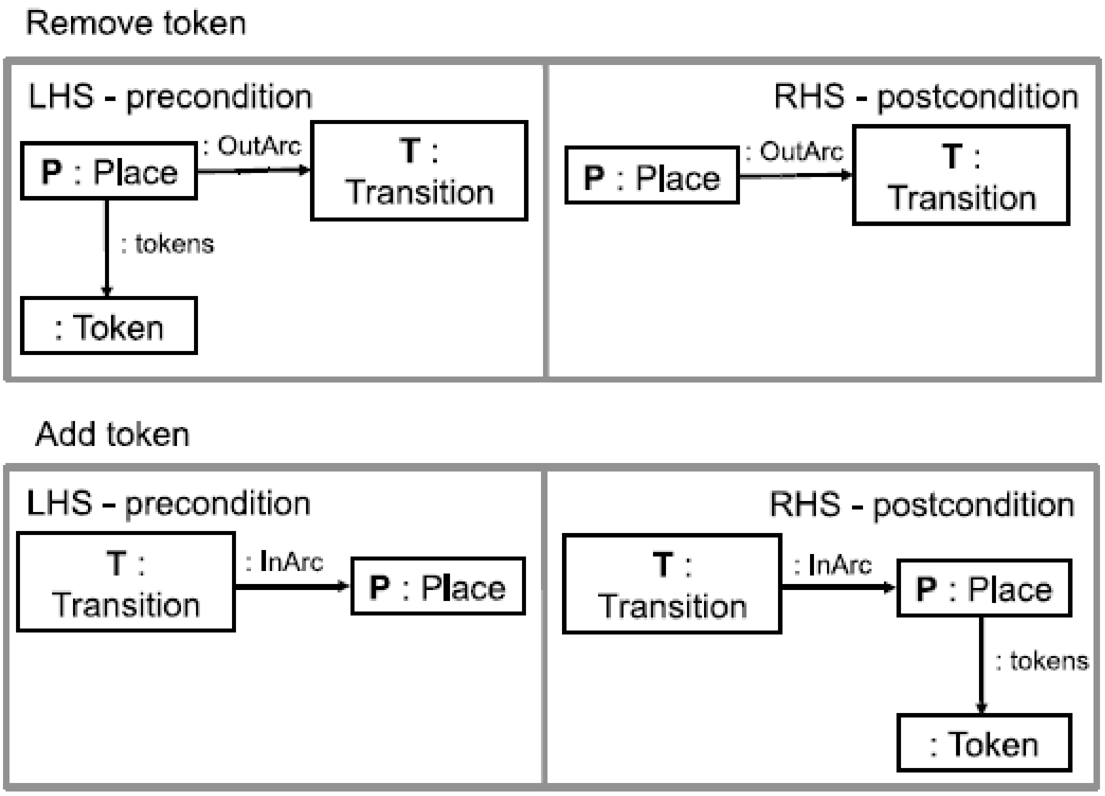


Figure 2.12.: A graph transformations example: removing and adding token [5].

Since models can be considered as graphs. A number of tools that perform endogenous transformations ared based on the principles of graph transformations. VIATRA2 [5] is a platform that supports graph transformation principles to perform model transformations. In Figure 2.11, a metamodel of a PetriNet [5] is provided. The *PetriNet* metamodel defines entities *Place*, *Transition*, *Token* and *ArcWeight*. *Place*s and *Transition*s are connected with *Arc*s; a *Place* can contain a number of *Token*s, etc. In Figure 2.12, graph transformations rules are defined. The upper part of the figure removes a token from a place. The *LHS* precondition is the pattern that needs to be transformed: a

Place P with a *Token* that connects to a *Transition* T by an *OutArc*. The *RHS* post condition specifies the transformation, which is to remove the *Token* from P. The transformation engine performs pattern matching with the *LHS* and then removes the Token on each pattern matched. The same principle applies to the example at the lower part of Figure 2.12, where a *LHS* precondition defines the pattern: a *Transition* T connected to a *Place* P with an *InArc* from T to P. A *RHS* postcondition defines the transformation: a *Token* is added to the *Place* P for each of the patterns matched.

There are also other transformation languages available to perform endogenous transformations, such as Graph Rewrite Generator (GrGen.NET) [63] for graph modelling, pattern matching and rewriting, Attributed Graph Grammar (AGG) [64] for attributed graph transformation, etc.

Model-to-text Transformation

Model-to-text (M2T) transformations are used to produce text files, such as source code, documentation, as well as model serialisation to text files (for example, saving models in XMI files). OMG first recognised the lack of a standardisation for M2T transformation with its M2T Language Request for Proposals. A set of languages were developed including Acceleo [65], Xpand [66], and the Epsilon Generation Language [67].

In M2T languages, *template*s are commonly used to enable repeatability. A *template* defines *static* sections which, during M2T transformations, are outputted verbatim, and *dynamic* sections, which contain expressions and statements that are executed to produce text from the contents of the target models involved in M2T transformations.

An example of an M2T transformation, written in the Epsilon Generation Language (EGL), is provided in Listing 2.2. In this example, the source of the transformation is a model that conforms to the social network metamodel in Figure 2.10, where the target of the transformation is plain text. In the template, the assumption is that the variable *person* is an instance of *Person* in the metamodel. In EGL, there are two different types of *dynamic* sections. When a *dynamic* section is contained within [% and %], the section is a normal section and contains statements, such as control flows like *if* statements, etc. When a *dynamic* section is contained within [%= and %], the section is known as a *dynamic output* section. The value which the contained expression evaluates to is output to the text. In the example, the sections on lines 2 and 4 are normal *dynamic* sections, whereas the sections on lines 1 and 3 are *dynamic output* sections.

```
1 Name: [%=person.first_name%] [%=person.last_name%]
2 Knows: [% for(p in person.knows) {%]
3   [%=p.first_name%] [%=p.last_name%]
4 [% } %]
```

Listing 2.2: An example of model-to-text transformation written in Epsilon Generation Language [67].

Text-to-Model Transformation

Text-to-Model (T2M) transformation is used to transform text into models. T2M transformation is typically implemented as a parser that generates models from text inputs. Existing parser generators, such as ANTLR [68], can be used to produce structured artefacts (such as abstract syntax trees) from text inputs. T2M tools typically reuse parser generators and process the artefacts obtained to produce models that can be managed with model management tools.

Xtext [12], EMFText [69], Rascal [70] and Spoofax [71] are contemporary tools that support Text-to-Model transformations. Given a grammar, T2M tools in general are capable of generating a metamodel and a parser that transforms text inputs into a model that conforms to the metamodel.

Listing 2.3 shows an exemplar DSL grammar defined in Xtext. In line 2, a *Domainmodel* is defined, which in line 6 defines that it should have a number of *Type*s. A *Type* is either a *DataType* or an *Entity*, as it is defined in line 7. *DataType* is defined in line 11, whereas *Entity* is defined in line 14. An *Entity* may have *superType*s (of type *Entity*), as it is defined in line 15. An *Entity* may contain a number of *Feature*s, as the grammar suggests in line 16. *Feature* is defined in line 20, and it has a *type* which is of type *Type*.

Given the grammar provided in Listing 2.3, Xtext is able to generate an Ecore metamodel from the grammar, a parser which translates text inputs into models, a static analysis tool which is used for code validation and code completion, and an Eclipse plug-in project which integrates all the functions, including refactoring tools, etc. [12].

```

Domainmodel :
(elements += Type)*
;

Type:
DataType | Entity
;

DataType:
'datatype' name = ID
;

Entity:
'entity' name = ID ('extends' superType = [Entity])? '{'
  (features += Feature)*
'}'
;

Feature:
(many ?= 'many')? name = ID ':' type = [Type]
;
```

Listing 2.3: An exemplar DSL grammar defined in Xtext [12]

Model Validation

Development of large systems always faces the risk of inconsistency. Inconsistency issues can arise throughout an MDE-based software development process. Model validation provides a mechanism to assess the integrity of the models that drive an MDE process. In general, inconsistency can appear in two different forms [72]: *incompleteness* and *contradiction. Incompleteness* arises from missing information. For example, creating an object without populating its compulsory properties is an instance of *incompleteness. Contradiction* arises from incompatible information in models. For example, an instance of a class A in a model has a reference to an instance of another class B, where in the metamodel these classes A and B do not relate in any way. In [73], the authors further classify inconsistency into *internal* and *external* inconsistencies:

Internal Inconsistency includes *Metamodel Inconsistency* and *Domain Inconsistency. Metamodel Inconsistency* arises when a model fails to conform to its metamodel. An example of metamodel inconsistency is the existence of a UML class that inherits itself. *Domain Inconsistency* arises when a model fails to comply semantically with the domain rather than its metamodel. For domain inconsistency, consider the *University* metamodel mentioned in Figure 2.9. The *finalGrade* of a *Student* is of type *float.* Whilst not breaking the rules defined in the metamodel, a negative value for the *finalGrade* is not valid in the domain, as the minimum a *Student* can get is 0.0.

External Inconsistency arises among models that are used to describe a system. According to [74], [75] and [76], two types of external consistency are identified: *horizontal* and *vertical. Horizontal Inconsistency* arises when multiple models are used to capture overlapping aspects of a system. Models that are developed by the same engineer can also contain inconsistencies. While there are various modelling languages that can be used to model a system, the variety also introduces potential inconsistencies among models depicted in different modelling languages. *Vertical Inconsistency* arises from incrementally refining models, where adding more details may inadvertently change the semantics of the models.

Due to the potential consistency issues described above, the need for model validation is obvious. There are a number of languages in the MDE field that can be used to validate models for consistency. The Object Constraint Language (OCL) [41] is an OMG-standardised language for specifying constraints on MOF and UML models. OCL is a side-effect free language, suitable for expressing metamodel consistency rules. OCL has been used extensively for internal consistency checking. The Epsilon Validation

Language (EVL) [73], built in the Epsilon[2] platform, is a validation language with similar concepts as OCL constraints. In addition, EVL provides more capabilities, such as dependent constraints, inter-model consistency checking, customisable error messages and fixes to repair inconsistencies [73].

Model Comparison

Model comparison, according to [77], is "the process of establishing correspondences of interest between elements that belong to different models, that are potentially expressed using different modelling languages and/or technologies". Model comparison is a necessary operation to perform before merging/integrating two or more models into a single model. There are a number of proposals for model comparison. Xlinkit [74], SiDiff [78], EMF Compare [79] and the Epsilon Comparison Language [74] are examples of contemporary model comparison tools.

Model Merging

As discussed in the model transformation section, Model Merging is a special case of model transformation, where a number of input models are transformed (integrated, merged) into a single output model. The reason model merging is listed as a task specific model management operation is because, unlike general model transformations, it requires comparison between models to be carried out before it can be performed, so that the output model of model merging does not include redundant information. In addition, model merging is commonly performed on models that conform to different metamodels, which increases the complication of model merging. Successful and extensively used model merging techniques are Atlas Model Weaving (AMW) [80], based on ATL, which can be used to merge both heterogeneous models[3] and homogeneous models[4]. Additionally, AMW can also be used to merge metamodels and homogeneous models. The Epsilon Merging language [52] also serves the purpose of model merging and is typically used in conjunction with the Epsilon Comparison Language [52] for

[2]Extensible Platform for Specification of Integrated Languages for mOdel maNagement
[3]models that conform to different metamodels
[4]models that conform to the same metamodel

model comparison.

2.1.7. Summary

This section introduced the core terminologies and principles of MDE. Models provide an abstract view of a real world system by only capturing aspects of interest. Metamodels provide the syntactical and semantical well-formedness constraints to construct models. The metamodelling architecture forms the basis of modelling in the context of MDE. Throughout an MDE based software development process, various model management tasks are performed to eventually produce software development artefacts, such as working code or documentation.

2.2. MDE Technologies

Well established and mature platforms comprising tools and languages to support common activities in MDE are available. This section discusses two MDE technologies that are used in the remainder of this thesis.

Section 2.2.1 provides an overview of the Eclipse Modelling Framework (EMF) [6], which provides a pragmatic implementation of the MOF standard and acts as the baseline for many MDE tools and languages. Section 2.2.2 provides an overview of Epsilon, an extensible platform, which provides a wide range of model management languages and supports a wide range of modelling technologies.

2.2.1. Eclipse Modelling Framework (EMF)

Based on Eclipse, the Eclipse Modelling Framework (EMF) project [6] provides a metamodelling language, Ecore, which (partially) implements the MOF 2.0 standard [47]. EMF is the most widely used contemporary MDE modelling framework and is supported by a large number of automated model management languages and tools, such as ATL, VIATRA, Epsilon, Eclipse OCL and Acceleo.

An overview of Ecore is provided in Figure 2.13. In EMF, metamodel elements are organised in *EPackages* and are represented by *EClassifier*s. An *EClassifier* can be either of primitive data type, represented by *EDataType*, or a complex type, represented by *EClass*. An *EClass* can contain attributes represented by *EAttributes* and define relationships to other *EClass*(es) represented by *EReferences*.

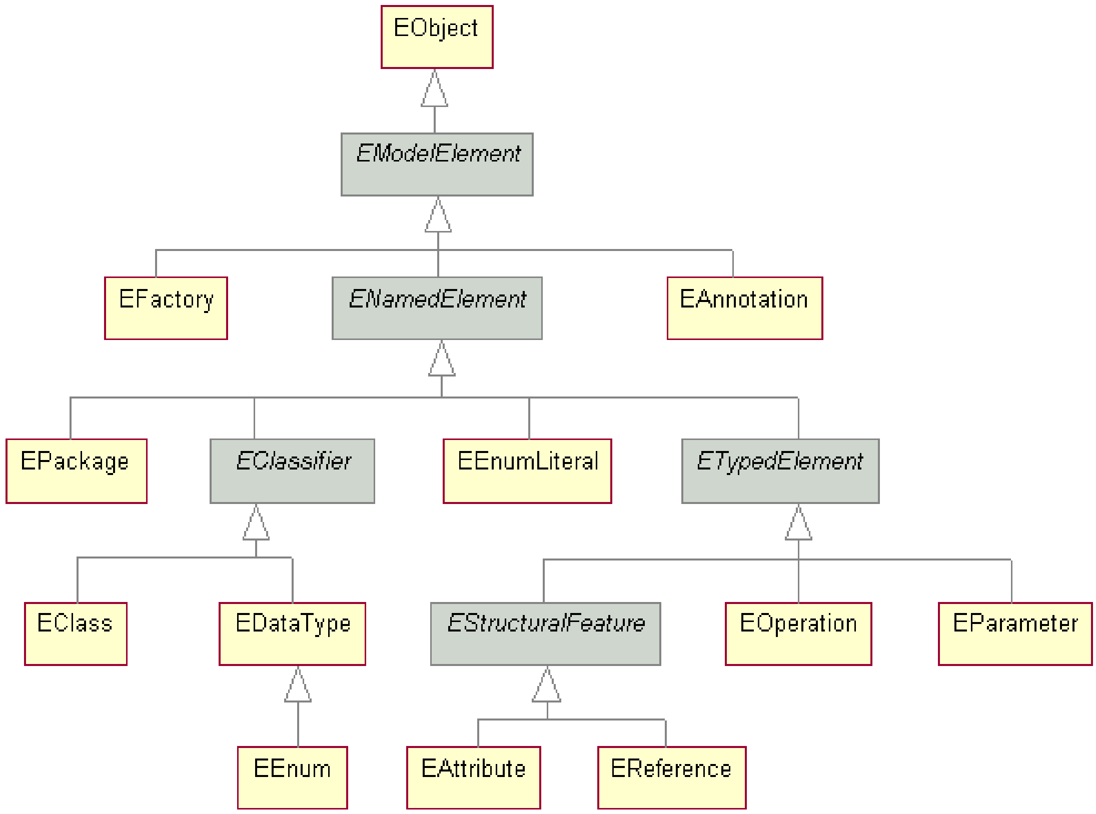


Figure 2.13.: The Ecore metamodelling language, from[6]

EMF provides a code generation facility, which is able to generate metamodel-specific editors given a metamodel defined in Ecore. Based on Eclipse, EMF model editors comprise a comprehensive set of views for viewing and manipulating models. The generated metamodel-specific editors support the loading, storing and exchanging of models in XML Metadata Interchange (XMI) format [38], which is a dialect of XML optimised for model interchange. In addition, EMF also supports pluggable model persistence formats. Existing tools, such as Neo4EMF [81] and MongoEMF [32], which are backed by NoSQL databases [82], offer more scalable alternatives than XMI for model persistence.

Apart from constructing metamodels using Ecore, EMF provides facilities to extract models from XML Schema, Java annotated interface source file, and from XMI documents generated by other modelling tools, such as Rational Rose [6].

The code generator for EMF also provides a facility to generate a set of *Interface* and *Implementation* Java classes for each type defined in a given Ecore model. These classes can be edited to include behaviour for their *operation*s defined in the model. Such behaviour is also able to persist throughout code regenerations from metamodels.

A large number of MDE tools are based on EMF, such as Eclipse OCL [26], Eclipse ATL [60], Eclipse Epsilon [36], Xtext [12], GMF [83], etc. EMF has become the *de facto* standard for building MDE tools [81]. EMF, acting as a common base for MDE tools, enables MDE operations such as reverse engineering [84], model transformation [36] and code generation [67].

2.2.2. The Epsilon platform

The *Extensible Platform for Specification of Integrated Languages for mOdel maNagement* (Epsilon) [36] is an Eclipse based platform that supports MDE. The architecture of Epsilon is shown in Figure 2.14. Epsilon essentially contains two main components, the Epsilon Model Connectivity layer (EMC) and the Epsilon family of languages.

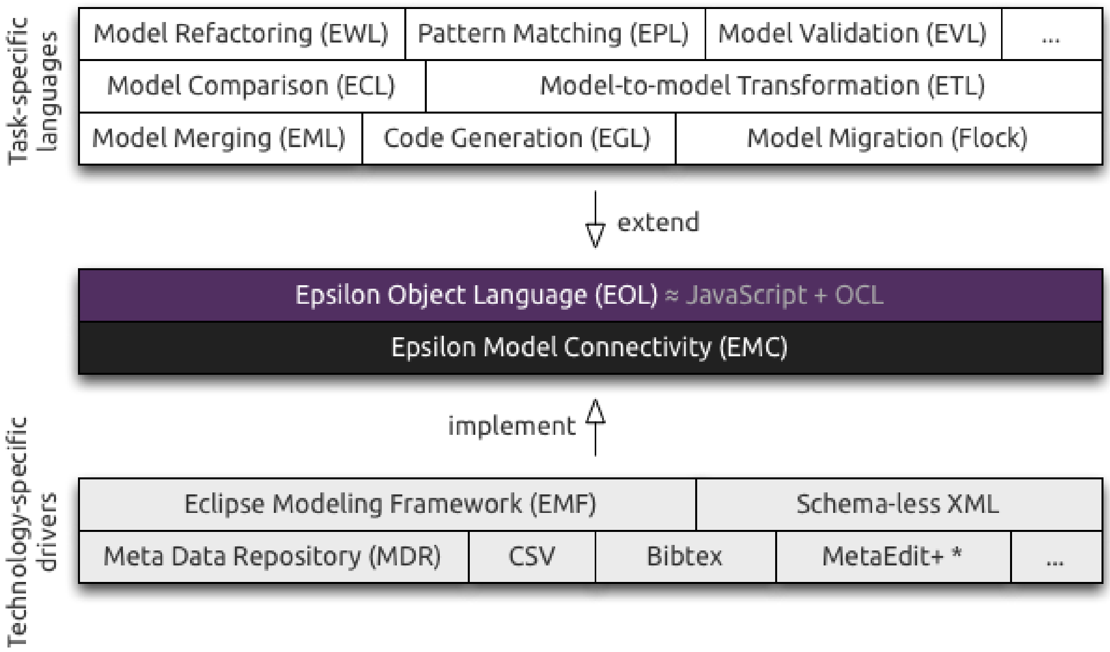


Figure 2.14.: The architecture of the Epsilon platform.

Epsilon is modelling technology agnostic [36]. Whilst many model management languages and tools are bound to a particular subset of modelling technologies (for example, Eclipse OCL is able to work with Ecore models and UML2 models only), Epsilon is able to access and manage models defined in various modelling technologies. Currently, Epsilon supports models defined using EMF, schema-less XML, Meta Data Repository (MDR), CSV, MetaEdit+, etc., which are backed by technology-specific drivers [36]. Epsilon is extensible in the sense that further technology-specific drivers can be developed to support models defined in other modelling technologies.

Epsilon promotes reuse when building task-specific model management languages. The core language of Epsilon is the Epsilon Object Language (EOL), which provides functionality that is similar to OCL, but with additional language features such as imperative statements, access to multiple models (backed by EMC), model updating (backed by EMC), error reporting and user feedback, etc. Atop EOL, task-specific languages can be created by reusing and extending EOL, which promotes consistency of syntax among the languages. Currently, a number of task-specific languages have been created atop EOL:

- *Epsilon Generation Language* (EGL) to perform mode-to-text transformations;
- *Epsilon Wizard Language* (EWL) to perform update model-to-model transformations;
- *Epsilon Comparison Language* (ECL) to perform model comparison;
- *Epsilon Merging Language* (EML) to perform model merging;
- *Epsilon Transformation Language* (ETL) to perform model-to-model transformations;
- *Epsilon Validation Language* (EVL) to perform model validation;
- *Epsilon Flock* to perform model migration;
- *Epsilon Pattern Language* (EPL) to perform pattern-based querying.

Apart from these languages, Epsilon also supports the aggregation of model management operations to form complex workflows [36].

Epsilon is well positioned as a platform for the research of this thesis due to its broad support of modelling technologies and task-specific model management languages, the extensible EMC layer for further modelling technology-specific driver development, and the extensible core language EOL for further task-specific model management language development.

2.3. Challenges to MDE

MDE delivers a number of benefits to software engineers. In MDE, some manual steps of traditional software engineering can be automated due to the use of domain specific modelling and model management operations. In [14, 15], MDE has been shown to increase productivity by as much as a factor of 10. In addition, tool interoperability enables a variety of tools to work on different aspects of the model without any compatibility issues.

Whilst the benefits of MDE are well recognised, there are some challenges to it. Concerns are raised for the human factor in MDE, such as learnability and acceptance of MDE by software engineers. In this section, only the challenges that are relevant to this thesis are discussed. These are correctness, maintainability and scalability.

2.3.1. Correctness of Model Management Operations

Software constantly contains errors. This is due to the fact that software are implemented by humans, and human actions often cause incorrect results. Fixing defects gets more expensive the later in the development process where they are identified. Software companies typically spend more than 80% of their development budget on quality control [21]. Thus, it is necessary to identify potential defects in the software as early in the development process as possible.

In the context of MDE, despite the raised level of abstraction, model management programs can still contain defects. Thus, there is a need to ensure the correctness of model management programs. For example, to exclude a given rule in a model transformation named $R1$, it is necessary to know if $R1$ is invoked by any other rules in the whole transformation [85] for the transformation rule to behave correctly.

The tools reviewed in Chapter 4 provide various degrees of support for analysing different model management programs. However, such tools only support either the analysis of programs that manage models defined by only a subset of contemporary modelling technologies, or the analysis of programs that are only written in a single (or a subset) of task-specific modelling languages. Such limitations are discussed in Chapter 5.

2.3.2. Scalability

As MDE is increasingly applied in larger and more complex systems, the current generation of modelling languages and model management tools are being stressed to their limits in terms of their capacity to accommodate various activities, such as collaborative development, efficient management and persistence of models, when dealing with models larger than a few hundred megabytes in size [18].

For MDE to remain relevant in software engineering, MDE languages and tools must confront the challenges associated with scalability, so that they can be used in larger scale complex models and systems. In [18], the authors suggest that to achieve scalability, MDE typically needs to:

- enable the construction of large models and domain specific languages in a systematic manner;
- enable large teams of modellers to collaboratively construct and refine large models;
- enhance the current generation of model querying and transformation tools so that they can accommodate large models (with millions of model elements) in an efficient manner;
- provide an infrastructure for efficient storage, indexing and retrieval of such models.

Scalability issues of MDE tools is a key concern for their applications and has been referred to as the “holy grail” of MDE [86].

Currently, there are a number of tools which strive to address some of these scalability issues, such as Neo4EMF [81], Morsa [31], Hawk [87], etc. which are backed by contemporary NoSQL databases in order to enable persistence and collaborative development of models containing several millions of elements.

2.4. Chapter Summary

In this chapter, a background review of MDE was provided. The main concepts within MDE were discussed, including *model*s, *modelling language*s and *metamodelling architecture*s. A series of model management operations were also identified and discussed, including model-to-model transformation, model-to-text transformation, text-to-model transformation, model validation, model comparison and model merging. The Eclipse Modelling Framework (EMF) and the Epsilon platform, which are closely related to this thesis, were discussed. Finally, a number of challenges for the current practice of MDE, that are directly related to this thesis, were identified and discussed. This thesis tries to address the challenges by means of static analysis. The next chapter will present a review on static analysis fundamentals.

3. Background: Static Program Analysis Fundamentals

Static program analysis is a fully automatic technique for reasoning about the behaviour of a program without actually executing it [88]. Static analysis is recognised as a complementary measure to dynamic testing. Static analysis is able to infer type incompatibility, detect potential runtime errors and optimise the performance of programs. Modern compilers typically provide static analysis to perform code optimisation to improve performance [89].

This chapter provides a field review on static source code analysis. It starts with a description of the importance of defect detection and ways to detect defects in software. The characteristics of static analysis are discussed and a number of static analysis techniques adopted by contemporary static analysis tools are presented.

3.1. Software Defects

Software defects have existed as long as software. According to IEEE [20], a *software defect* can be defined as "an imperfection or deficiency in a software system where that software system does not meet its requirements or specifications and needs to be either repaired or replaced". *Defects* originate from *errors*, where an *error* is "a human action that produces an incorrect result" [20]. This is one of the reasons that software often contains defects, as humans inevitably make mistakes [90].

Some defects are identified easily whilst others are either found late in the development process, or after the release of the software. Defects can cause several types of problems, including logical/functional problems (incorrect outputs computed by the software system), runtime errors such as unexpected crashing, resource leaks, degraded performance, etc. Poor software quality caused by defects is a serious problem that developers and users of software face today. On average, software contains 10 to 20 defects per thousand lines of code after compilation and testing [91]. Large software contains millions of lines of code and therefore contains potentially thousands or tens of thousands of defects upon release. Fixing defects gets more expensive the later in the development process these are identified . Software companies typically spend more than 80% of their development budget on quality control [21]. A research carried out by NIST in [92] illustrated that software defects cost the U.S. economy $59.5 billion annually as of year 2002.

3.2. Defect Detection

To address the defects that inevitably exist in software systems, a broad range of techniques for automatic or semi-automatic detection and prevention of defects has been proposed and developed. Whilst there are various categorisations of software testing, one clear distinguishing feature of defect detection is whether the detection requires executing the software (termed *Dynamic Testing*) or not (termed *Static Testing* or *Static Analysis*) [93].

3.2.1. Dynamic Testing

Dynamic Testing is a defect detection approach that involves executing the program being tested a number of times, and analysing the information collected from the executions. Dynamic testing provides accurate defect reports, as the information collected is based on the actual execution of the program. Additionally, dynamic test cases are straightforward to implement based on software specifications [91]. On the other hand, dynamic testing can be time- and resource-consuming because it requires the instrumentation of the program(s) being tested. Exhaustive testing for all possible code paths is practically difficult. Dynamic testing can be carried out at different stages of software development in different levels. In particular, *Unit Testing*, *Integration Testing*, *System Testing* and *Acceptance Testing* are recognised testing methods to capture defects at different stages of software development [93].

3.2.2. Static Testing

Static Testing, or *Static Code Analysis*, is a testing approach that analyses the source code of the software without executing it, which can be done manually (code inspection) or automatically (using automated static analysis tools). Static analysis sometimes incurs negligible resource consumption compared to dynamic testing, as it does not require the execution of the software. Sophisticated automated static analysis tools make a trade-off between performance and accuracy. Static analysis typically considers all possible execution paths and does not require test suites. Importantly, static analysis can detect possible defects while the source code of the program is being developed and can generate immediate reports on potential defects. The downside of static analysis is that it is not straightforward: unlike designing dynamic test cases, static analysis often requires more complex implementation [21]. Static analysis often generates more false alarms on defects, forming a very high noise level which makes the defect alarms hard to analyse.

Static analysis can be used to protect against specific types of software runtime errors which, when detected at early stages, can prevent more significant defects from occurring in the future. A non-exhaustive list of runtime problems that static analysis can detect is as follows:

- Incomplete code, such as uninitialised variables, functions with unspecified return values, incomplete control flow statements (e.g. missing cases in switch statements);
- Improper resource management, for example memory leaks. This issue is of great importance for programming languages with no garbage collection mechanisms;
- Illegal operations: division by zero, improper values for functions, overflows and underflows, index array out of bounds, etc.;
- Dead code: code sections that cannot be reached. This kind of defect may only be inappropriate coding style, but it may also indicate the risk of potential errors.

Whilst static analysis can reduce the resources spent on dynamic testing or even detect defects that cannot be identified by dynamic testing, it is not a replacement for dynamic testing [93]. Static analysis can be used to check that the program executions do not unexpectedly terminate or crash, but it does not guarantee correct execution. Thus, static analysis and dynamic testing are always carried out as complements of each other in a development process [94]. This thesis focuses on static analysis techniques and their application in MDE.

3.3. Static Analysis Characteristics

Static analysis tackles a problem which is known to be undecidable (according to Rice's Theorem) [95]. In other words, it is not possible to design a static analysis tool which proves any non-trivial property on any program both accurately and automatically. As a consequence, static analysis is inherently imprecise [95]. Typically, static analysis tools infer that a *property* (e.g. an error) may hold for a given program. When analysing a program P for a certain error E using a static analysis tool SA, the outcome of the analysis falls in one of the following four categories:

- **(a)** P holds E, SA infers that E *may* exist; in some cases, SA is able to infer that E *definitely* exists;
- **(b)** P holds E, but SA infers that E *does not* exist;
- **(c)** P does not hold E, and SA infers that E *does not* exist;
- **(d)** P does not hold E, but SA infers that E *may* exist;

With respect to static analysis, case (b) is often referred to as a *false negative*, while case (d) is referred to as a *false positive* [96]. *False negative* typically refers to a situation where a static analysis fails to report defects that actually exist in a software system. *False positive* typically refers to a situation where a static analysis reports defects which do not really exist.

False negatives and *false positives* are used to measure the precision of static analysis tools. A static analysis tool is said to be *sound* if all reported defects are actual defects (i.e. no false positives) but it does not guarantee that all errors can be detected (i.e. there may be false negatives) [97]. A static analysis tool is said to be *complete* if all defects in a program are detected and reported (i.e. no false negatives), but there may be false positives.

The precision of a static analysis tool determines how frequently *false positive* reports are produced. The more precise the analysis, the more likely it is to generate fewer *false positives*. Precision of a static analysis tool usually correlates with the time taken to perform the analysis [95]. The more precise the analysis, the more time it is likely to take to perform it. The trade-off between precision and analysis time is one of the design concerns of a static analysis tool. If a static analysis examines a program in a short time, it is likely to generate many *false positive*s which increases the noise level. In contrast, a precise static analysis tool, which reports significantly fewer *false positive*s, is not likely to finish its analysis in a timely manner.

There are some approaches to reduce false positives whilst preserving the short time of analysis, which normally adopt techniques that filter out unlikely defect reports from the false positives. For example, CodePeer [98] classifies defects into high, medium and low risk levels, with low risk level defect warnings only presented to the users on demand. However, without care, such techniques will result in removal of actual defects causing *false negatives*.

There is also a number of techniques that can be used to make the best out of the trade-off between precision and analysis time [99]. A *flow sensitive* analysis focuses on the *control flow graph* (See Section 3.4.2) of the program, while a *flow insensitive* analysis does not. Flow sensitive analysis is usually more precise than flow insensitive analysis. For example, for a block of code, flow sensitive analysis is able to infer that a certain value x is defined in a particular line, while flow insensitive analysis is only able to infer that x may be defined throughout the block. A *path sensitive* analysis considers only available paths through the program. It takes into consideration the values of variables and boolean expressions in if/switch conditions and loops to reason the execution branches.

3.4. Static Analysis Techniques

In this section, techniques used by most static analysis tools are discussed. Such techniques include data flow analysis (Section 3.4.2) and abstract interpretation (Section 3.4.3). A number of other techniques used in contemporary static analysis tools are also discussed in Section 3.4.4.

3.4.1. Lattice Theory

Static analysis techniques discussed in this section largely rely on the lattice theory. This section provides an overview of the lattice theory and some terminologies used therein [100].

Definition 3.1. Partial order set. A *partial order set* (sometimes referred to as *poset*) is a set U and a binary relation $\sqsubseteq$, on U, such that:

- $\forall x \in U : x \sqsubseteq x$ (reflexivity)
- $\forall x, y, z \in U : (x \sqsubseteq y \wedge y \sqsubseteq z) \Rightarrow x \sqsubseteq z$ (transitivity)
- $\forall x, y \in U : (x \sqsubseteq y \wedge y \sqsubseteq x) \Rightarrow x = y$ (antisymmetry)

Example 3.2. Consider a finite set $U : \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ with the usual ordering $\leqslant$. Such ordering is reflexive because $1 \leqslant 1$, is transitive because $(1 \leqslant 2 \wedge 2 \leqslant 3) \Rightarrow 1 \leqslant 3$, is antisymmetric because $(1 \leqslant 1 \wedge 1 \geqslant 1) \Rightarrow 1 = 1$. Thus, $(U, \sqsubseteq)$ is a partial order.

Definition 3.3. Least upper bound. Assume that $(U, \sqsubseteq)$ is a partial order and assume that $A \subseteq U$. If there exists an element $z \in U$ such that:

- $\forall x \in A : x \sqsubseteq z$
- $\forall y \in U : (\forall x \in A : x \sqsubseteq y) \Rightarrow z \sqsubseteq y$

then z is called the *least upper bound* of A. The *least upper bound* of A is denoted as $\sqcup A$. The $\sqcup$ can also be used to denote a least upper bound of two elements in a set. Suppose $a, b \in A$, then the least upper bound of a and b can be denoted as $a \sqcup b$.

Definition 3.4. Greatest lower bound. Assume that $(U, \sqsubseteq)$ is a partial order and that $A \subseteq U$. If there exists an element $z \in U$ such that:

- $\forall x \in A : z \sqsubseteq x$
- $\forall y \in U : (\forall x \in A : y \sqsubseteq x) \Rightarrow y \sqsubseteq z$

then z is called the *greatest lower bound* of A. The *greatest lower bound* of A is denoted as $\sqcap A$. The $\sqcap$ can also be used to denote the greatest lower bound of two elements in a set. Suppose $a, b \in A$, then the greatest lower bound of a and b can be denoted as $a \sqcap b$.

Definition 3.5. Lattice. If $(U, \sqsubseteq)$ is a partial order where $U \neq \emptyset$, and for all $x, y \in U$, $x \sqcup y$ and $x \sqcap y$ exist, then the system $(U, \sqsubseteq)$ is called a *lattice*.

If $\sqcup A$ and $\sqcap A$ exist for arbitrary subsets A of U, then the system $(U, \sqsubseteq)$ is called a *complete* lattice.

Two elements of a complete lattice $(U, \sqsubseteq)$ are of particular interest: the element $\top = \sqcup U$ (top) and $\bot = \sqcap U$ (bottom). And $\forall x \in U : \bot \sqsubseteq x \sqsubseteq \top$.

Definition 3.6. Fixed point. If U is a set and $f : U \to U$ is a function, then $u \in U$ is called a *fixed point* of f if $f(u) = u$. A *fixed point* $u \in U$ is called the *minimal fixed point* if for all other fixed points $v \in U$ of f, $v \not\sqsubseteq u$. If a function f has exactly one minimal fixed point, then this fixed point is called the *least fixed point* of f.

Tarski's Theorem [101] proved that for an increasing function f on a complete lattice, f *must* have a least fixed point, which can be computed.

Lattice theory is the foundation of a number of important static analysis techniques, such as data flow analysis and abstract interpretation. Such techniques are reviewed in the following sections.

3.4.2. Data Flow Analysis

Data flow analysis is a process to (statically) collect run-time information about data in programs. Data flow analysis is a form of flow sensitive analysis. The semantics of the operators are not used during data flow analysis [102]. Data flow analysis is based on the *control flow graph*(s) of programs, *lattice theory* and *least fixed point algorithm*(s) [102].

To perform data flow analysis on a program, a *control flow graph* (CFG) of that program needs to be obtained first. A control flow graph consists of *node*s or **basic block**s. A *basic block* contains *statement*(s) or similar concepts of the programming language in which the program under question is written. A *basic block* is an abstract concept such that:

- control enters a *basic block* only at its beginning;
- control exits a *basic block* only at its end (under normal execution); and
- control cannot halt or jump out of *a basic block* except at its end

Such condition implies that when control flow enters a *basic block*, the *statement*(s) contained in the *basic block* are all executed.

Control Flow Graph

A *control flow graph* is an abstract representation of a program. Each node in the graph is represented by a *basic block*. Directed edges are used to connect the *basic block*s, which represent control flow branches. An *entry block* is a *basic block* that has no incoming edges. An *exit block* is a *basic block* that has no outgoing edges.

Control Flow Path

A *control flow path* is a path in the control flow graph that starts at an entry block and ends at an exit block [22]. There may be more than one possible control flow paths in a given control flow graph of a program. Often, the number of possible control flow paths is infinite because of the unpredictability of the bounds of loops.

Example

To better understand data flow analysis, an example program from [22] is provided in Listing 3.1. The program is written in an imaginary programming language called TIP (stands for Tiny Imperative Language). The control flow graph for the program in Listing 3.1 is shown in Figure 3.1.

```
x = 2;
y = 4;
x = 1;
if(y>x)
  z = y;
else
  z = y*y;
x = z;
```

Listing 3.1: A program created in an ad-hoc language from [22].

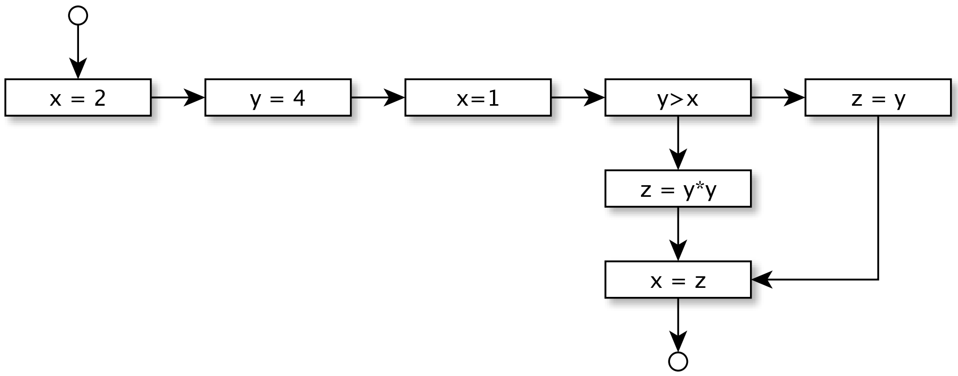


Figure 3.1.: Control flow graph of the program in Listing 3.1

There are various types of data flow analysis. *Lattice* theory is applied in data flow analysis by constructing a *lattice* for each type of data flow analysis. The constructed *lattice* depends on the type of data flow analysis and also the range of variables/values involved in the program under question.

One example of data flow analysis is the **liveness** of variables. A variable is said to be *alive* at a point in the program if its value is accessed in the remainder of the program. For the program provided in Listing 3.1, the lattice for analysing the *liveness* of the variables is:

$L = (2^{\{x,y,z\}}, \sqsubseteq)$

Where the binary relation is *alive*, with a given set $\{x, y\}$, it stands true that x and y are *alive*, so that $x \sqsubseteq \{x, y\} \wedge y \sqsubseteq \{x, y\}$.

With the *control flow graph* and the chosen *lattice*, the next step is to construct *constraints* for each *basic block*. For every node v in the control flow graph, a constraint variable $[v]$ is introduced denoting the subset of program variables that are live at the program point *before* that node. An auxiliary definition is provided: [22]

$JOIN(v) = \bigcup_{w \in succ(v)} [w]$

For the exit node the constraint is:

$[exit] = \{\}$

For conditions and output statements, the constraint is:

$[v] = JOIN(v) \cup vars(E)$

For assignments, the constraint is:

$[v] = JOIN(v) \backslash \{id\} \cup vars(E)$

For a variable declaration, the constraint is:

$[v] = JOIN(v) \backslash \{id_1, ..., id_n\}$

Finally, for all other nodes, the constraint is:

$[v] = JOIN(v)$

In the constraints, the term *id* refers to either the variable on the left hand side of the assignment operator or to variables in variable declarations, whereas $vars(E)$ refers to the variables that are not *id*s.

Thus, for the program in Listing 3.1, the constraints for the *basic block*s in the *control flow graph* are the following. In the constraints, the expression enclosed in square brackets ([]) represents the potential value in the lattice that a *basic block* may hold. For example, the $[exit]$ basic block holds no value in the lattice.

$[x = 2] = [y = 4] \setminus \{x\}$

$[y = 4] = [x = 1] \setminus \{y\}$

$[x = 1] = [y > x] \setminus \{x\}$

$[y > x] = ([z = y] \cup [z = y * y]) \cup \{x, y\}$

$[z = y] = [x = z] \setminus \{z\} \cup \{y\}$

$[z = y * y] = [x = z] \setminus \{z\} \cup \{y\}$

$[x = z] = [exit] \setminus \{x\} \cup \{z\}$

$[exit] = \{\}$

With all the *constraint*s for all *basic block*s in the *control flow graph*, the next step is to solve the *constraint*s by substituting the *basic block*s enclosed in square brackets with their range of values in the lattice. By doing so, the solved constraints are obtained:

$[entry] = \{\}$

$[x = 2] = \{\}$

$[y = 4] = \{\}$

$[x = 1] = \{y\}$

$[y > x] = \{x, y\}$

$[z = y] = \{y\}$

$[z = y * y] = \{y\}$

$[x = z] = \{z\}$

$[exit] = \{\}$

Thus, it is inferred that variable y is only *alive* before the statement $x = 1$. A smarter compiler (with the help of the static analysis) would omit line 1 in Listing 3.1 because variable x is later assigned again in line 3 in Listing 3.1.

Data flow analysis can also be applied to perform the following types of analysis:

Available Expression Analysis

Available expression analysis is able to determine which expression(s) in the program are computed in more than one place in the program. If an expression, for example, $a+b$ has previously been computed and there are no changes to a and b, then the expression $a + b$ can be substituted with the previous computation. Available expression analysis considers all the *control flow path*s of the program (the *least fixed point* algorithm is used to deal with loops). The results of the analysis can be used to optimise programs so that available expressions are replaced with their previous computations.

Very Busy Expression Analysis

A *very busy expression* is an expression that is used intensively and the value(s) of the variable(s) in the expression do not change between each usage [22]. The results of the very busy expression analysis can be used by compilers to optimise programs so that such expressions are only evaluated once.

Reaching Definition Analysis

Reaching Definition Analysis is used to determine for each *basic block*, which assignments may have been made to define the values of variables [22]. Typically, reaching definition analysis enriches the control flow graph by linking the definition of variables to their declarations. Such graph is called *def-use* graph. Reaching definition analysis is the basis of optimisations, such as dead code elimination.

Data flow analysis is a formal approach of static analysis and is mainly used in compilers to create optimised code. However, data flow analysis is not as powerful in detecting possible runtime defects. This is due to the fact that data flow analysis does not typically consider the semantics of the source code; therefore, it is not able to determine its correctness.

3.4.3. Abstract Interpretation

Abstract interpretation is a theory of semantic approximation. The essence of abstract interpretation is to create new semantics for the programming language (under question) that is an abstraction of the concrete semantics of the programming language. Hence, abstract interpretation can be defined as: *abstract since some details about the data of the program are (intentionally) forgotten, and interpretation since both a new meaning is given to the program text and the information is gathered about the program by means of an interpreter which executes the program according to this new meaning* [103]. The abstract semantics focuses on a subset of problems with regards to the programs. In [103], a *static analysis by abstract interpretation* (SAAI) framework is proposed. The framework is based on control flow analysis and the *lattice theory* - programs are interpreted as control flow graphs and a set of *constraint*s in a *lattice* L. The lattice and constraints depend on the property of the program that the static analysis targets. Unlike data flow analysis, abstract interpretation considers the semantics - operators and function calls are assigned new meanings in the abstract semantics.

For instance, instead of computing with actual integers, abstract interpretation may compute with values that describe some property of the integers. For example, using abstract interpretation, one may replace the domain of integers with the finite domain $\{\ominus, 0, \oplus, ?\}$, where $\ominus$ represents a negative integer in the interval [$-\infty$, -1], 0 represents the integer 0, $\oplus$ represents a positive integer in the interval [1, ∞] and ? represents any integer in the interval [$-\infty$, ∞]. With the domain defined, one can define an abstract interpretation of operators; for example, an addition which used to be performed to add two integers can now be redefined to add up two abstract integers. The abstract addition operation, can be defined as follows:

+	$\ominus$	0	$\oplus$	?
$\ominus$	$\ominus$	$\ominus$	?	?
0	$\ominus$	0	$\oplus$	?
$\oplus$	?	$\oplus$	$\oplus$	?
?	?	?	?	?

Such abstraction leads to loss of information but it can be inferred that if two negative integers are added, the result will be negative. Similar to addition, one can redefine abstract division operation, so that the defect of division-by-zero can be identified.

With different lattice(s), abstract interpretation is able to check for various types of runtime errors. For example, with a lattice which contains the set of possible states of a pointer in C/C++, abstract interpretation is able to check if a pointer is NULL. With a lattice which contains the set of possible type(s) in the programming language's type system, abstract interpretation is able to check if a program is *type correct*.

Abstract interpretation can be computationally expensive if the range of the lattice is not carefully defined. Because of this, abstract interpretation is considered to be challenging to apply with large programs [104].

Abstract interpretation is not only bound to *lattice theory*. In [103], the author states that any form of static analysis that interprets the semantics to an abstract semantic can be considered a form of abstract interpretation.

3.4.4. Other Techniques

Another type of approach is to build a *model* for static analysis purposes only. Programs are parsed into an instance of the *model* and properties are checked by executing validators in the model. The Lint [105] family of tools are examples that adopt such an approach. However, the Lint tool(s) exhibit a high rate of false positives. Although output can be customized to eliminate some false positives, it also increases the risk of eliminating real errors.

Another approach, called *annotation checker*, relies on the user of such tools to annotate the program they wish to check. This approach provides better checking and performance. LCLint [106] is an example of such an approach.

FindBugs [107] adopts another approach by specifying the *pattern*s of errors and storing them in a pattern base. Given a program, the search for all bug patterns is performed. This approach reduces the rate of false positives, as only potential error patterns are searched for.

The Eclipse JDT provides a static analysis facility [108], which parses Java source code into Abstract Syntax Trees (ASTs). Such ASTs are then traversed, and variable resolution and type resolution are performed to check for defined errors.

3.5. Chapter Summary

This chapter provided a background review on the importance of defect detection. Then a number of static analysis techniques were discussed. Data flow analysis and abstract interpretation are the most commonly used static analysis techniques to optimise code and check for potential runtime errors in compilers. Their limitations were also briefly discussed. Other static analysis approaches, such as the static analysis used in Lint, FindBugs and Eclipse JDT were briefly discussed.

3.6. Terminology

Software Defect: an imperfection or deficiency in a software system where that software system does not meet its requirements or specifications and needs to be either repaired or replaced.

Dynamic Software Testing: Dynamic testing is a defect detection approach that involves executing the program being tested a number of times, and analysing the information collected from the executions.

Static Software Testing: Static software testing is a testing approach that analyses the source code of the software without executing it, which is typically carried out either manually or automatically.

False negatives: In the context of static analysis, false negatives refer to the defects that exist in the source code but are not detected by a static analysis tool.

False Positives: In the context of static analysis, false positives refer to the defects reported by a static analysis tool but are not actual defects in the source code.

4. Background: Static Analysis of Model Management Programs

This chapter reviews contemporary static analysis tools in the context of MDE. Section 4.1 discusses the need for static analysis in the context of MDE. Section 4.2 identifies the aspects of the static analysis tools that the review looks into. Then, a number of static analysis tools in MDE, namely the built-in static analysis tools for Dresden OCL [109], Eclipse OCL [26], Eclipse ATL [23], Acceleo [65], Xpand [110] and EMF-IncQuery [8] are reviewed. A third-party static analyser, AnATLyzer [7], which provides higher level analysis functionalities, is also reviewed.

4.1. Static Analysis in the context of MDE

MDE has been shown to bring two positive impacts on software engineering. Firstly, metamodelling and modelling raise the level of abstraction in system design, which enables problem domain experts to design systems without the concern of low level implementation details. Secondly, the notion of automation in MDE allows the developers to automatically transform models into working code (and documentation) using a variety of model management operations (model-to-model transformations, model-to-text transformations, etc.). Automation in MDE has been shown to improve productivity [15] and the generated software (source code) is of good quality in terms of consistency and coding style.

However, model management operations are typically programmed in model management languages. As such, there is a need to ensure the correctness of programs written in such model management languages (discussed in Chapter 2). For example, the program in Listing 4.1 is a model-to-model transformation which is used to transform models that conform to the *University* metamodel to models that conform to the *SocialNetwork* metamodel, provided in Section 2.1.6 (Figure 2.10). There is an error in line 6, where the assignment statement assigns all the *Student*s that a *Lecturer* has to *p*, which is illegal because the type *Student* does not exist in the *SocialNetwork* metamodel. The program will throw a runtime error because of the assignment statement in line 6.

```
1 rule Lecturer2Person
2 transform l : University!Lecturer
3 to p : SocialNetwork!Person {
4   p.first_name = l.first_name;
5   p.last_name = l.last_name;
6   p.knows = l.students;
7 }
```

Listing 4.1: A model-to-model transformation written in Epsilon Transformation Language

For runtime errors caused by defects in the source code, in the absence of static analysis facilities, developers typically have to review the source code manually, correct it, then compile the source code and run the program again. If errors persist, developers may need to look into the metamodel to check if the operations/functions in the source code conform to the constraints in the metamodel. Such a process can introduce a long debugging curve. Thus, if no techniques of defect detection are used at the development phase of model management operations, the cost of removing defects is expensive because defects cannot be detected at early stages [93]. Moreover, if the models involved in the transformation grow larger (with hundreds or thousands of model elements inside a model) or transformation programs get more complicated, debugging using the compile-run-debug process described above can be considerably slow.

Thus, there is a need for checking the correctness of model management programs. Software defects in programs in MDE can be detected by dynamic testing and static analysis [93], as discussed in Section 3.2. This thesis focuses on static analysis of model management programs in MDE.

There are some existing static analysis tools in the context of MDE to analyse programs written in various model management programs. Dresden OCL [25] and Eclipse OCL [26] provide built-in static analysis facilities, which are used to check type safety for model validation programs written in OCL. Eclipse ATL [60] provides a static analyser to check type safety in ATL model transformations. EMF-IncQuery [8] provides a static analysis facility to check type safety in model queries. Acceleo [111] and Xpand [66] provide static analysers to check type safety for model-to-text transformations, etc. The details of these static analysers are reviewed in this chapter.

Whilst the aforementioned static analysers focus on the correctness of model management programs, a new line of work has focused on utilising static analysis to achieve additional objectives. In [7], a static analyser is implemented for ATL and produces what is called the *effective metamodel*. Through the analysis of ATL transformations, based on the *effective metamodel*, test cases are generated to exercise the transformation and discover potential defects.

4.2. Review Strategy

In the previous section, a number of contemporary static analysis tools were identified. These tools are reviewed in this chapter. The review is focused on the following set of characteristics of the tools.

4.2.1. Modelling technologies supported

As discussed in Section 2.1.5, a number of modelling technologies (such as EMF, MDR, UML2, plain XML, etc.) are being used to define modelling languages and models. In the context of MDE, models can be categorised as *closed* models and *open world* models. *Closed* models are models that have their corresponding *metamodel*s which define the concepts used in such models (such as models specified using Ecore). On the other hand, *open world* models are models that do not have corresponding *metamodel*s - they are defined and used in an ad-hoc manner. Examples of *open world* models include models defined in plain XML and CSV. Such models do not conform to metamodels but are widely used by engineers due to their popularity and simplicity [112].

In a *closed* model M which conforms to its metamodel MM, when accessing a property p of an object O of type T (T is defined MM), if T does not define property p, then the property navigation is surely illegal. On the other hand, in *open world* models, all property navigations may be legitimate due to the lack of a metamodel.

A desirable feature for model management languages or tools is the support of managing models defined in diverse modelling technologies. Such models can be either *closed* models or *open world* models. In addition, although some model management languages support multiple modelling technologies, they do not support managing models defined in different modelling technologies simultaneously within a single model management program. For example, ATL supports MDR and EMF [60], but it does not support the transformation from a model defined in EMF to a model defined in MDR.

The review assesses if diverse modelling technologies are supported, and if models defined in different modelling technologies can be used simultaneously within a single model management program.

4.2.2. Program Abstract Syntax Representation

In the context of MDE, everything is considered to be a model [34]. This stands true for model management languages. A model management language can be considered as a metamodel and programs written in the language can be considered as models that conform to the language (the metamodel). In terms of language implementation, the approach that most languages adopt is to parse the source code of a program into an abstract syntax tree (AST) using a parser and then use the AST to execute the program.

In some cases, a higher level of interpretation is performed, turning the ASTs into content-rich representations (i.e. models) of the language. The benefit of having a metamodel of a model management language is that it can loosen the coupling between the language and its underlying parser so that it makes it easier for the language implementers to substitute its underlying parsing technology. Obtaining the model of a program also allows higher order transformations [113], so that the programs can be altered with MDE technologies and/or transformed into programs that conform to other languages.

The review will assess if a language and its static analysis tools implement a high level modelling of the language.

4.2.3. Encoding/Representation of the Standard Library

A model management language, in general, has its own type system, which provides types for its developers to use. The standard library provides the operations/helpers of the types defined by the model management language. With regard to the implementation of the standard libraries, model management languages can choose from a number of approaches.

One approach is to build internal representations of the standard library programmatically. While this is a common approach for language builders, it brings some subsequent problems. By implementing the standard library programmatically in its entirety, the model management language is coupled with the underlying programming language that implements it. Thus, the re-usability of the model management language is limited, as implementing the model management language in another programming language requires the complete re-write of the standard library. Additionally, extending/altering the standard library is difficult [114]. For example, given the specification of the model management language, one may wish to create another version of the implementation of the language or one may wish to implement only a subset of the types in the language's type system. In this case, removing the types and its operations in the standard library can be tedious.

Another approach is to model the signatures of the operations/helpers of the standard library using the model management language, while the static semantics of the operations/helpers is implemented programmatically. By doing so, the coupling between the model management language and its underlying programming language is loosened to an extent, and the effort for altering/extending the standard library is minimised.

The review will assess which approach the underlying model management languages and their static analysis tools use.

4.2.4. Static Analysis Capabilities

In [115], a number of common errors which can be detected by static analysis are identified, such as:

- Reading an undefined variable
- Reading an absent property of an object
- Invoking undefined functions
- Invoking a function with an invalid number of parameters
- Type conversion and compatibility problems

Such errors are normally detected by basic static analysis mechanisms when the expressions/variables involved are properly typed. While static analysis tools provide checking against errors like the above, some static analysis tools illustrate that static analysis can be used to achieve higher level functionalities. In [7], the authors suggest that static analysis can be used to construct *effective metamodel*(s) from an ATL transformation. An *effective metamodel* [23] is a subset of the *metamodel* involved in a model management program, which indicates which instances of the type(s) defined in the *metamodel* will be used by such a program. *Effective metamodel*s give an insight of the model element coverage and are often used to construct (automatically) test cases for model management programs.

The review of the static analysis tools looks into whether the tools support functionalities (such as *effective metamodel* extraction) rather than just static analysis for checking type-related errors.

4.3. OCL Static Analysis Implementations

This section presents the reviews on the static analysis tools provided by two OCL implementations: Dresden OCL [109] and Eclipse OCL [26]. Dresden OCL and Eclipse OCL are both based on the Eclipse platform. Dresden OCL provides facilities for defining constraints in OCL and assessing the validity of these constraints against models. Eclipse

OCL, on the other hand, provides a variety of OCL-based languages for domain-specific modelling, expressing constraints, and re-writing/extending the OCL standard library. Both implementations adopt the concept of *Pivot Metamodel*. In this section, the two OCL implementations are reviewed by looking into the aforementioned aspects.

4.3.1. Dresden OCL

Dresden OCL provides a set of tools to parse and evaluate OCL constraints on models defined in various modelling technologies, such as UML, EMF and Java. Dresden OCL provides a built-in static analysis facility. In the context of this thesis, Eclipse Luna version 4.4.0 and Dresden OCL version 3.4.0 are used for the review of Dresden OCL and its static analysis facilities.

Dresden OCL Pivot Metamodel

OCL originally acted as an add-on to UML to specify constraints. However, its scope has widened in recent years to support constraints and queries over object-based modelling languages in general [41] due to the variety of approaches proposed and tools implemented in the context of MDE. Consequently, OCL faces the challenge of supporting different Domain Specific Languages. Dresden OCL addresses this challenge by introducing a *Pivot Metamodel*, which is defined in EMF's Ecore. The *Pivot Metamodel* is a metamodel that abstracts from all other metamodels [109]. The purpose of *Pivot Metamodel* is to allow arbitrary metamodels defined in different modelling languages to be converted into a common representation. Therefore, Dresden OCL is able to work with models expressed in diverse metamodelling technologies.

Modelling technologies supported

According to [109], Dresden OCL supports any modelling technology with the use of *Pivot Metamodel*. Currently, Dresden OCL supports models (and instances of the models) defined in the following technologies:

- EMF's Ecore;

- Java, Dresden OCL supports the import of Java classes as models and allows OCL constraints to be defined directly on Java types and their fields and methods;
- MDT UML, Dresden OCL supports the import of UML class diagrams modelled with the Eclipse Modelling Development Tools (Eclipse MDT); and
- XML Schemas, Dresden OCL supports the import of XML Schema Definitions (XSD) as models.

Other modelling technologies can be supported by adapting them to the *Pivot Metamodel* [109]. Although Dresden OCL supports diverse modelling technologies, it does not support expressing constraints to models defined in different modelling technologies within a single OCL file.

Program Abstract Syntax Representation

Dresden OCL provides the metamodel of Essential OCL [39], which is defined with the *Pivot Metamodel*. At runtime, OCL constraints are parsed into abstract syntax trees (models) which conform to the Essential OCL metamodel.

Encoding/Representation of the Standard Library

The standard library of Dresden OCL is also defined in the *Pivot Metamodel* to support different modelling technologies. Only the signatures of the standard library are defined in *Pivot Metamodel* - the semantics of the operations/helpers are implemented programmatically via the *OCL Standard Library Semantics* facility provided by Dresden OCL.

Static Analysis Capabilities

The built-in Dresden OCL static analysis facility provides basic analysis for type checking of expressions. OCL programs with injected errors mentioned in Section 4.2.4 have been created and analysed by Dresden OCL static analyser. Dresden OCL static analyser is able to detect such errors. Figure 4.1 shows a screenshot where Dresden OCL is used to write a constraint for the *University* metamodel (defined in Section 2.1.6). In line 5, an *invariant* is defined, which tries to access a property named "faculties". However, the type *University* does not define this property. Dresden OCL static analyser produces an error message accordingly.

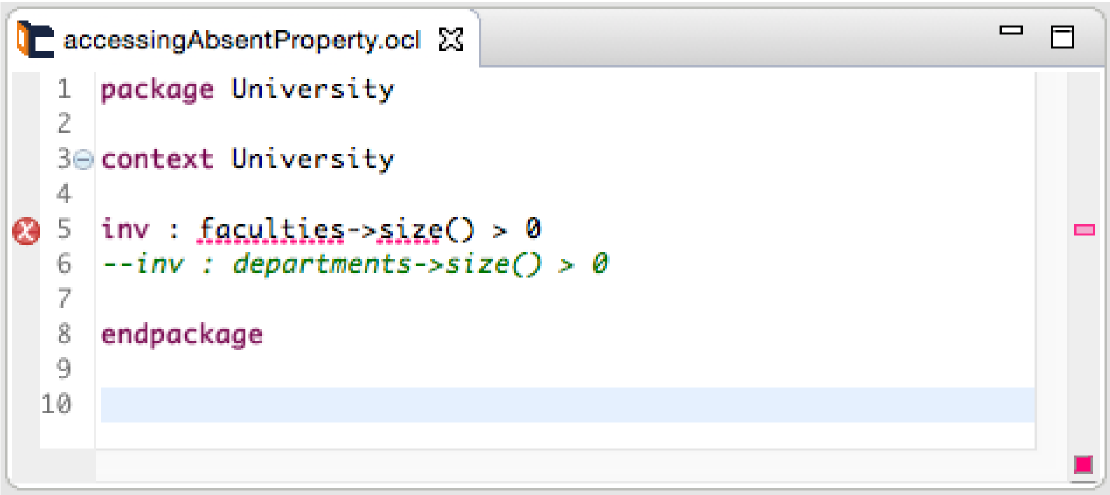


Figure 4.1.: Dresden OCL static analysis detecting metamodel-related errors

Dresden OCL does not provide any facilities based on the static analysis apart from error detection.

4.3.2. Eclipse OCL

Eclipse OCL [26] is an implementation of the OCL specification 2.4 for use with EMF-based (in particular, Ecore and UML2) metamodels. As previously mentioned, this is due to the extended scope of OCL to support expressing constraints on different modelling technologies. Eclipse OCL can also be considered as a behavioural extension of EMF [116]. Eclipse OCL has a static analysis mechanism built in it. In the context of this thesis, Eclipse Mars version 4.5.0 and Eclipse OCL version 6.0.0 are used for the review of Eclipse OCL static analysis mechanism.

The Classic Eclipse OCL metamodels and the Eclipse OCL pivot metamodel

According to [116], Eclipse OCL has two different implementations: one with the classic Eclipse OCL metamodel, and another with the Eclipse OCL pivot metamodel. The classic code base of Eclipse OCL[1] focused on providing utilities for Java programmers. It originally supported Ecore metamodels and then added support for UML. This is achieved by a shared generic metamodel, in which the differences between Ecore and UML metamodels are accommodated by template parameter lists in Java [116]. These parameter lists are rather substantial and therefore introduce cumbersome Java code for the OCL developers/consumers.

The latest Eclipse OCL[2] adopts the concept of *Pivot Metamodel*, which is similar to the Dresden OCL *Pivot Metamodel*. The *Pivot Metamodel* is derived from the UML metamodels for UML and OCL to provide a unified metamodel for UML with executable semantics. When using the *Pivot Metamodel* for Ecore or UML metamodels, an instance of the *Pivot Metamodel* is created on the fly to provide the unified merged OCL functionality for the Ecore or UML metamodel instances. The Eclipse OCL *Pivot Metamodel* is UML-aligned. It supports modelling of the OCL standard library, XMI representation of its instances, etc. [117].

Modelling technologies supported

Eclipse OCL currently supports EMF's Ecore and UML2 [118]. Additionally, with the *Pivot Metamodel* in place, it is possible to support more modelling technologies in the future.

Program Abstract Syntax Representation

All languages provided by Eclipse OCL (Essential OCL for OCL core, OCLinEcore for embedding ocl within an Ecore metamodel to add invariants for classifiers, and OCLstdlib for defining standard and custom OCL libraries) are modelled with the *Pivot Metamodel*, which is created with EMF's Ecore.

Encoding/Representation of the Standard Library

Eclipse OCL adopts the approach of modelling the signatures of the operations/helpers of the OCL standard library and implementing the execution semantics programmatically.

[1]Versions before 6.0.0, based on Eclipse Luna and before
[2]Version 6.0.0, based on Eclipse Mars release

The standard library of Eclipse OCL is modelled with the *Pivot Metamodel*. The types in the standard library align with the OCL specification version 2.5 [114]. In [109], it is identified that Eclipse OCL is not able to model the iterator operations of the standard library. However, in Eclipse OCL, a few types have been proposed and implemented [114] to help model the standard library.

OclLambda Type. Prior to OCL 2.5, Iterator operation declarations in the OCL specification variously omit the final body argument. The required type signature is defined by commentary and sometimes well-formed rules. The *OclLambda* Type was introduced in the OCL standard Library 2.5 so that iterator operators can be defined using *Lambda* expressions. The Syntax of a Lambda type is illustrated below:

```
Lambda context-type (parameter-type-list) : result-type
```

Thus, iterator operations, such as *forAll()*, can be defined:

```
iteration forAll(i : T | body: Lambda T() : Boolean) : Boolean
```

OclSelf Type. OCL standard library 2.5 introduced the *OclSelf* type to solve various corner cases in the existing OCL standard library. For example, prior to OCL standard library version 2.5, the *oclAsSet()* operation was defined as:

```
OclAny::oclAsSet() : Set<OclAny>
```

where static type information is lost since there is no way to specify that the return type of the *oclAsSet()* operation is a *Set* containing the object on which the operation is invoked. The *OclSelf* type helps with such cases. Thus, the *oclAsSet()* is redefined as:

```
OclAny::oclAsSet() : Set<OclSelf>
```

when an object calls *oclAsSet()*, its type is propagated into the return type of the operation. Thus, static type information can be maintained and propagated.

Additionally, *OclSelf* can also be used to model the *allInstances*() operation (which returns all the instances within a model of a given type):

```
static Classifier::allInstance() : Set<OclSelf>
```

Operations and Well Formedness Rules. Together with the types, the OCL standard library provides operations which are applicable to their corresponding types.

In terms of the implementation of the operations, the standard library provides a mechanism which allows an arbitrary string to be specified for use by the tooling. The string points to the location of a Java class which implements the feature of the operation. Figure 4.2, illustrates the syntax of this mechanism.

```
type Bag(T) : BagType conformsTo Collection(T) {
    annotation 'http://www.omg.org/ocl'(ClassGroup='Collection');
    /**
    The closure of applying body transitively to every distinct element of the source collection.
    **/
    iteration closure(i : T | lambda : Lambda T() : Set(T)) : Set(T)
                            => 'org.eclipse.ocl.examples.library.iterator.ClosureIteration';
    iteration collect(V)(i : T[?] | lambda : Lambda T() : V[?]) : Bag(V)
                            => 'org.eclipse.ocl.examples.library.iterator.CollectIteration';
```

Figure 4.2.: OCL Standard Library Implementable

Eclipse OCL provides a domain specific language (which is a dialect of OCL), named *OCLstdlib*, to define the OCL standard library. Developers, thus, have the freedom to extend the OCL standard library at an appropriate level of abstraction.

Static Analysis Capabilities

The static analysis facility built in to Eclipse OCL provides basic analysis for type checking of expressions. At runtime, the OCL source code is parsed into a Heterogeneous abstract syntax graph, which is essentially an instance of the Pivot metamodel. The Pivot metamodel adopts the visitor design pattern for evaluation and validation. There is a centralised validation visitor which is responsible for dispatching the validation algorithms to validate different expressions.

OCL programs with injected errors mentioned in Section 4.2.4 were created and analysed by the Eclipse OCL static analyser. Figure 4.3 shows an OCL program created to describe constraints on the *University* metamodel (defined in Section 2.1.6). In line 7, an invariant is defined which tries to access the property named "faculties". However, in the *University* metamodel, there is no such feature. An error is then issued by the Eclipse OCL static analyser.

Eclipse OCL does not provide any facilities based on the static analysis apart from error detection.

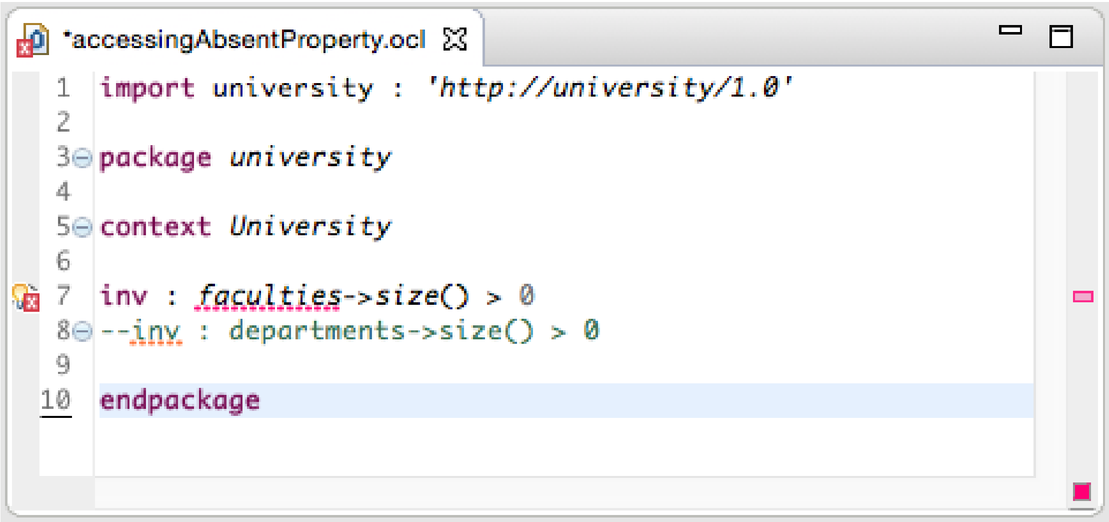


Figure 4.3.: Eclipse OCL detecting metamodel-related errors

4.4. ATL Static Analysis Implementation

This section reviews the static analysis tools for the Atlas Transformation Language (ATL) [60]. ATL is a hybrid model transformation language as an answer to the OMG MOF [47] QVT (Query/View/Transformation) RFP (Request For Proposal) [119]. ATL focuses on model-to-model transformations. Eclipse ATL provides a number of standard development tools (e.g. syntax highlighting, debugger, content assist, etc.) that aim to facilitate the development of ATL transformations.

While there is a number of static analysis tools proposed for ATL, this section reviews the static analysis tool provided by Eclipse ATL, and AnATLyzer [7], a third-party static analysis tool created with the aim of providing more accurate error reports. In the context of this thesis, Eclipse Mars version 4.5.0 and Eclipse ATL version 3.5.0 are used for the review.

4.4.1. Modelling technologies supported

ATL executes programs in an ATL Virtual Machine (currently, there are several versions of ATL Virtual Machines). Virtual Machines enables platform independence. Within an ATL virtual machine, ATL defines a generic facility which allows ATL to support diverse modelling technologies by creating drivers for them. For ATL version 3.6.0, drivers for

EMF, UML2 [118] and MDR [10] are provided.

4.4.2. Program Abstract Syntax Representation

ATL uses Textual Concrete Syntax (TCS) [120] to parse the source code of ATL programs into ATL models which conform to the ATL metamodel. The ATL metamodel is defined using EMF's Ecore. The ATL metamodel defines the language constructs of ATL and OCL, including all the types involved in OCL and ATL.

4.4.3. Encoding/Representation of the Standard Library

ATL programs run in ATL Virtual Machines. Currently, there are three versions of Virtual Machines: ATL regular VM, ATL EMFVM (EMF specific VM) and ATL EMFTVM (EMF Transformation VM) [121].

The ATL regular VM is the first VM implemented for ATL. It is built with the purpose of supporting diverse modelling technologies using the concept of model handlers [121]. ATL regular VM has the standard library implemented programmatically in it. ATL programs are compiled into "bytecode", stored in ATL's assembler files (with the ".asm" extension) [121]. The .asm files are XML-based files, which are executed by the regular VM. However, the model handlers of the ATL regular VM demonstrate significant performance issues [121].

ATL EMFVM is a redefinition of the ATL regular VM, and is specific to EMF models to address the performance issues incurred by the ATL regular VM model handlers. The standard library is implemented programmatically in EMFVM, including the operations to directly access EMF models.

ATL EMFTVM, standing for EMF Transformation Virtual Machine, is currently the most used virtual machine with advanced language features, such as multiple rule inheritance, advanced tracing, in-place transformation, etc. EMFTVM is derived from the previous two ATL VMs and "bytecode" format. However, instead of using a proprietary XML format, it stores its "bytecode" as EMF models, such that they may be manipulated by model transformations. The standard library is built programmatically in EMFTVM.

4.4.4. Static Analysis Capabilities

In terms of static analysis support, at the parsing stage, after an ATL program is parsed into an ATL model, a transformation, named ATL-WFR, is performed [121]. ATL-WFR is a model-to-model transformation written in ATL to transform an ATL model into a model that conforms to the Problem metamodel. The Problem metamodel is defined in Ecore to represent errors which are subsequently translated into markers visible in the ATL editor in Eclipse. This ATL-WFR transformation acts as a static analyser which performs very basic code analysis - mostly checking the uniqueness of transformation rules, models, variable declarations, etc.

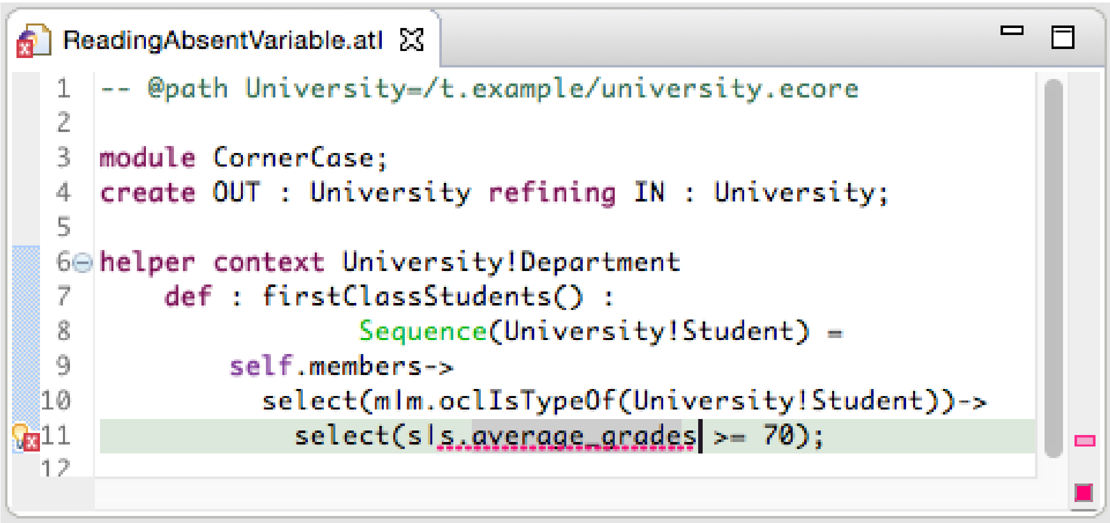


Figure 4.4.: An example of ATL static analysis

ATL programs with injected errors mentioned in Section 4.2.4 were created and analysed by Eclipse ATL static analyser. Figure 4.4 shows an ATL program which manages models that conform to the *University* metamodel (defined in Section 2.1.6). In this program, an ATL helper was created, named *firstClassStudents()*, which goes through all the *Members* of a *Department* and finds the students with an *average_grade* greater or equal to 70. In line 11, an error is injected: the name of the property is changed from *average_grade* to *average_grades*. The ATL static analyser is able to detect such error and report it in the editor.

Eclipse ATL does not provide any facilities based on the static analysis apart from error detection.

4.4.5. AnATLyzer

AnATLyzer [7] is able to discover errors in ATL transformations by combining static analysis and constraint solving. If a problem cannot be guaranteed to be an error by the static analysis facility, a *witness* model is automatically generated by AnATLyzer which is used to confirm if the problem is an error, by running the transformation with the *witness* model.

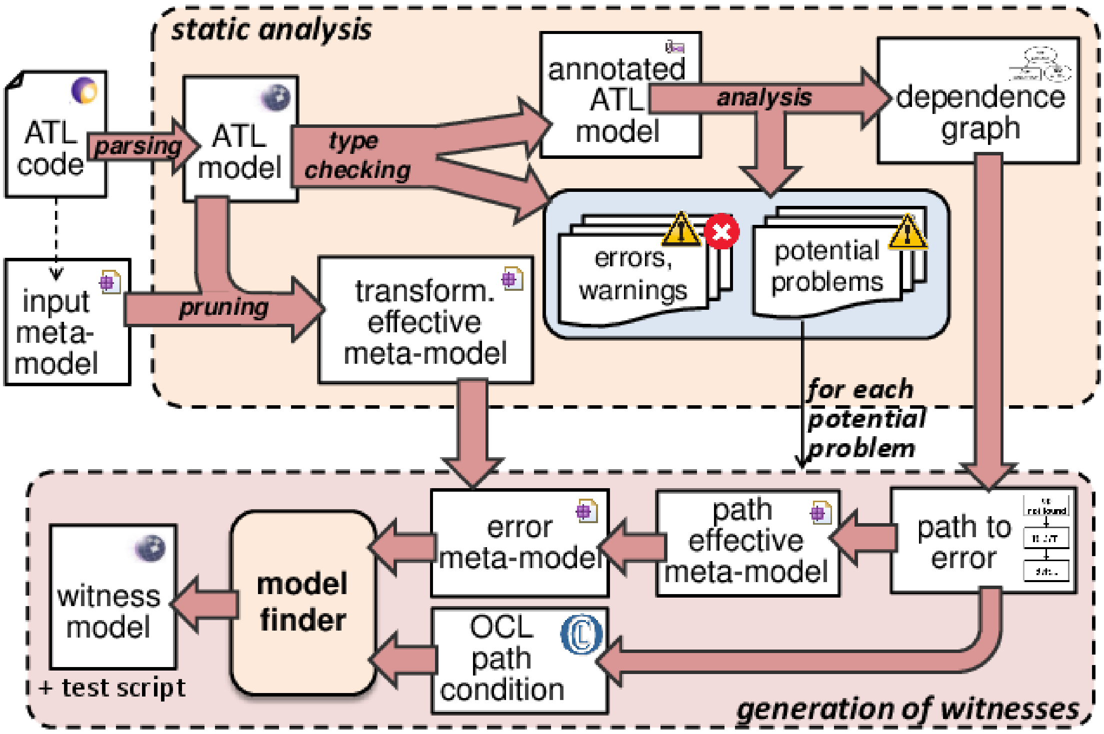


Figure 4.5.: Overview of the AnATLyzer [7]

Figure 4.5 illustrates the process performed by the AnATLyzer. The AnATLyzer performs the static analysis with the following steps:

- The ATL transformation (under question) is parsed to obtain the ATL model which conforms to the ATL metamodel.
- Type checking is performed on the ATL model in two passes. First, variable declarations, rule pattern types, helpers, etc. are annotated with their explicitly

declared types. Then, a bottom-up traversal of the ATL model is performed, propagating types, annotating each node in the ATL model, and reporting errors and warnings along the way.

Because ATL is a weakly typed language, expressions may yield different types at runtime. AnATLyzer makes use of an abstract interpretation technique, which keeps track of all possible types that an expression may have.

Because ATL does not support the *oclAsType* operation, to overcome this, AnATLyzer keeps track of calls to the operation *oclIsKindOf(targetType)* and annotates its return type with the target Type. For example, for the expression

```
expression.oclIsKindOf(University!Student)
```

the static analyser annotates the return type of *oclIsKindOf()* to type "Boolean and University!Student". AnATLyzer tracks all the calls to *oclIsKindOf()* in *select()* operations, *if* conditions and rule filters to implicitly downcast the checked expressions.

In some cases, the type errors detected statically need to be confirmed by finding a witness model. For example, the aforementioned downcasting mechanism may report a false problem. To speed up the generation of the witness model, AnATLyzer uses the *effective metamodel* of the transformation. The *effective metamodel* is calculated from the metamodel footprint obtained in the analysis phase, using a pruning algorithm similar to the one presented in [122].

Altogether, the type-checking phase annotates the nodes of the ATL model, which enables the identification of some typing errors and warnings.

- With all the types resolved in the ATL model, the AnATLyzer produces an instance of an extended version of the ATL metamodel, which contains type information of the nodes in the ATL model, and the control and data flow of the transformation, including the dependencies between transformation rules. This model, called *transformation dependence graph (TDG)*, is analysed in a second iteration to uncover further potential problems.

- Some problems detected in the analysis phase cannot be confirmed but require finding a *witness* model proving that the error can occur in practice. In order to do this, AnATLyzer first extracts the error path. This is done by extracting all possible paths that lead to the (potentially) problematic statement. In the next step, an OCL expression describing the models that make the transformation execute the problematic statement is derived.
- With the error path, AnATLyzer extracts the error path's effective metamodel and eventually extracts the error's effective metamodel. Together with the OCL path condition, the *model finder* of AnATLyzer is able to generate a *witness* model. Failing to find a *witness* model may occur in two cases: when the metamodel includes constraints preventing the existence of problematic models, or when the transformation contains expressions that prevent the error at runtime.

AnATLlyzer does not provide any facilities based on the static analysis apart from error detection.

4.5. Acceleo Static Analysis Implementation

Acceleo is an Eclipse-based code generation framework which implements the Object Management Group's (OMG) model-to-text specification [123]. It supports the generation of textual files using EMF and UML models. The Acceleo language, named MTL (Model-to-Text Language, which follows the OMG naming convention), is composed of two main types of structures: *templates* and *queries*. Acceleo adopts a subset of OCL's expressions in order to query the input models. A built-in static analysis facility is provided by Acceleo for error detection and code completion.

In the context of this thesis, Eclipse Mars version 4.5.0 and Acceleo version 3.6.1 are used for the review of the Acceleo platform and its static analysis mechanisms.

4.5.1. Modelling technologies supported

Acceleo is based on EMF, so it naturally provides support for models defined with EMF. Acceleo is also compatible with models defined with UML2. For models defined in previous UML versions, such as UML1.3 and UML1.4, Acceleo provides converters which are able to convert models defined in UML1.3 and UML1.4 to EMF models.

4.5.2. Program Abstract Syntax Representation

Since there is only very limited amount of documentation for Acceleo, this review is conducted by delving into the source code of Acceleo. The investigation of Acceleo's source code reveals that the abstract syntax of MTL is defined with EMF's Ecore. Acceleo extends Eclipse OCL to a large extent. Acceleo version 3.6.1 is built by extending Eclipse OCL version 3.5.0.

Acceleo defines two metamodels for MTL with EMF's Ecore. For parsing the source code, Acceleo defines a metamodel for MTL's concrete syntax, named MTLCST, which is used to represent a concrete syntax tree (CST) for MTL programs. The entities defined in MTLCST are limited to the language constructs of MTL at the source code level. At runtime, MTL source code is parsed into instances of MTLCST. Syntax errors are reported during this process, and markers for these errors are created in the Acceleo Editor.

The MTL metamodel extends Eclipse OCL's OCL metamodel, by importing the OCL metamodel defined in Eclipse OCL into the MTL metamodel in Ecore. The execution and static analysis works by interacting with instances of the MTL metamodel.

After the CST of a program is acquired, a model-to-model transformation is performed which converts instances of MTLCST into instances of MTL. Acceleo implements this transformation in Java, which is encapsulated in the CST2ASTConverter class. What the transformation does is to further process the elements in the program and create corresponding *OCLExpression*s.

4.5.3. Encoding/Representation of the Standard Library

Acceleo defines two built-in libraries: the standard and the non-standard library. The standard library is built conforming to Acceleo's specification, whilst the non-standard library is built for the OCL specification.

Acceleo defines the operations of the standard library and the non-standard library with EMF's Ecore. The implementation of the operations is defined programmatically. At runtime, an *AcceleoLibraryOperationVisitor* is responsible for retrieving the operations defined in the standard library and for invoking the code that implements the behaviour of the operations. The non-standard library uses the classic version of the OCL; therefore, types such as *OclSelf* (mentioned in Section 4.3.2) are not defined in it.

4.5.4. Static Analysis Capabilities

Acceleo extends Eclipse OCL to a large extent by reusing the *OCLExpressions* defined in the OCL metamodel. Therefore, the "semantic validation" mechanism (i.e. static analysis mechanism) of Eclipse OCL is used to detect errors in *OCLExpressions*.

Before the static analysis on OCL takes place, a preliminary static analysis occurs during the transformation from MTLCST to MTL, which identifies syntax errors and possible type incompatibilities. After the MTL abstract syntax graph is acquired, the OCL *ValidationVisitor* is then invoked to identify errors in *OCLExpressions*.

```
1  [comment encoding = UTF-8 /]
2  [module generate('http://university/1.0')]
3
4  [template public generateElement(anUniversity : University)]
5  [comment @main /]
6   [file (anUniversity.name.concat('.text'), false)]
7      University: [anUniversity.name/] \n
8      [for (department : Department | anUniversity.departments) separator ('\n')]
9          Department: [department.name/] \n
10         [for (member : Member | department.members) separator ('\n')]
11             [if (member.oclIsTypeOf(Lecturer))]
12                 Lecturer: [member.first_name/]
13                 Personal Web Page: [member.personalWebPage/]
14             [/if]
15         [/for]
16     [/for]
17  [/file]
18 [/template]
```

Figure 4.6.: Type inference of Acceleo for the University example (1 of 3)

With respect to static analysis capabilities, Acceleo inherits OCL's type safe policy - property accesses are bound to the property's own type only. Figure 4.6 exhibits this property. In line 12, an error is reported because the property *personalWebPage* cannot be accessed as the type of *member* in this case is interpreted as *Member*, despite the condition of the *if* statement in line 10 guarantees that the type of *member* to be

Lecturer.

```
[comment encoding = UTF-8 /]
[module generate('http://university/1.0')]

[template public generateElement(anUniversity : University)]
[comment @main /]
 [file (anUniversity.name.concat('.text'), false)]
    University: [anUniversity.name/] \n
    [for (department : Department | anUniversity.departments) separator ('\n')]
        Department: [department.name/] \n
        [for (member : OclAny | department.members) separator ('\n')]
            [if (member.oclIsTypeOf(Lecturer))]
                Lecturer: [member.first_name/]
                Personal Web Page: [member.personalWebPage/]
            [/if]
        [/for]
    [/for]
 [/file]
[/template]
```

Figure 4.7.: Type inference of Acceleo for the University example (2 of 3)

Acceleo's inference system is a very basic one. In Figure 4.8, instead of giving *member* a correct type declaration, an incorrect annotation is given, which declares the type of *member* to be *Department*. Such type declaration results in an error being reported, as the type of the collection *department.members* is statically known. Acceleo in this case only provides a warning for possible type incompatibility.

```
[comment encoding = UTF-8 /]
[module generate('http://university/1.0')]

[template public generateElement(anUniversity : University)]
[comment @main /]
 [file (anUniversity.name.concat('.text'), false)]
    University: [anUniversity.name/] \n
    [for (department : Department | anUniversity.departments) separator ('\n')]
        Department: [department.name/] \n
        [for (member : Department | department.members) separator ('\n')]
            [if (member.oclIsTypeOf(Lecturer))]
                Lecturer: [member.first_name/]
                Personal Web Page: [member.personalWebPage/]
            [/if]
        [/for]
    [/for]
 [/file]
[/template]
```

Figure 4.8.: Type inference of Acceleo for the University example (3 of 3)

Acceleo does not provide any facilities based on the static analysis apart from error detection.

4.6. Xpand Static Analysis Implementation

Xpand [110] is an MDE platform which provides textual languages that are useful in different aspects in the context of MDE, such as model validation, model-to-model transformation and model-to-text transformation [110]. The languages of the Xpand framework are based on a common programming language named Xtend. Xpand, a model-to-text transformation language and Check, a model validation language, are built atop the Xtend language [110]. In the context of this thesis, Eclipse Mars version 4.5.0 and Eclipse Xpand version 2.1.0 are used for the review of the Xpand framework.

4.6.1. Modelling technologies supported

Xpand is able to work with models defined in different modelling technologies, such as EMF Ecore models, Eclipse UML2 models, XML schemas and simple JavaBeans [66]. Xpand allows the usage of work flow (a series of model management tasks) templates which can be configured to interact with models defined in such technologies throughout the work flow (but not to manage models of different technologies within a single program). Xpand also provides an extensible interface, which allows the creation of model drivers for other modelling technologies.

4.6.2. Modelling of Xpand

The Xpand language extends the Xtend language in terms of its abstract syntax. The abstract syntax of the Xtend language is implemented using Java - there is a Java class implementation for each concept in the language abstract syntax. The Xpand language implements its abstract syntax by extending the Xtend abstract syntax.

At runtime, program source code is parsed Java-based Abstract Syntax Trees (i.e. the Java instances of the abstract syntax). Each abstract syntax implements an *analyse()* method, which deals with syntax errors and performs static analysis for type checking.

```
1 «IMPORT University»
2 «DEFINE main FOR University::University»
3     «EXPAND department FOREACH departments»
4 «ENDDEFINE»
5
6 «DEFINE department FOR Department»
7     «EXPAND member FOREACH members»
8 «ENDDEFINE»
9
10 «DEFINE member FOR Member»
11     «IF this.metaType == Lecturer»
12         «FILE first_name + ".text"»
13             Lecurer: «this.first_name»
14             PersonalWebPage: «((Lecturer)this).personalWebPage»
15         «ENDFILE»
16     «ENDIF»
17 «ENDDEFINE»
```

Figure 4.9.: Type casting in Xpand using the University example (1 of 2)

4.6.3. Encoding/Representation of the Standard Library

As previously mentioned, the Xpand language extends the Xtend language. Therefore, it also extends the standard library of the Xtend language. The standard library of the Xtend language is implemented programmatically in Java. The Xpand language implements more types atop those of the Xtend type systems, such as *Definition*, *Iterator*, etc.

4.6.4. Static Analysis Capabilities

For each abstract syntax element (implemented Java class), there is an *analyse()* method, which is used for static type inference. Xpand is a statically typed language (inherited from Xtend), and it has an advanced type inference mechanism which is able to infer the types of expressions even where type declarations are not in place. However, when dealing with inheritance, the type inference system needs the help of type casting in order to infer types correctly. The program in Figure 4.9 prints out all the *Lecturer*s of all the *Universities* in the model.

Since the type *Lecturer* is a sub-type of *Member*, type casting is needed in line 14, so that the type inference system knows that the type of *this* is *Lecturer*. It is noteworthy that the condition of the *if* statement in line 11 does not help the type inference, although the condition specifies that the *metaType* of *this* should be *Lecturer*.

On the other hand, if used inappropriately, type casting can result in runtime errors. In Figure 4.10, type casting in line 20 results in a runtime error.

```
1 «IMPORT University»
2 «DEFINE main FOR University::University»
3     «EXPAND department FOREACH departments»
4 «ENDDEFINE»
5
6 «DEFINE department FOR Department»
7     «EXPAND member FOREACH members»
8 «ENDDEFINE»
9
10 «DEFINE member FOR Member»
11     «IF this.metaType == Lecturer»
12         «FILE first_name + ".text"»
13             Lecurer: «this.first_name»
14             PersonalWebPage: «((Lecturer)this).personalWebPage»
15         «ENDFILE»
16     «ENDIF»
17     «IF this.metaType == Student»
18         «FILE first_name + ".text"»
19             Student: «this.first_name»
20             PersonalWebPage: «((Lecturer)this).personalWebPage»
21         «ENDFILE»
22     «ENDIF»
23 «ENDDEFINE»
```

Figure 4.10.: Type casting error in Xpand using the University example (2 of 2)

Xpand does not provide any facilities based on the static analysis apart from error detection.

4.7. IncQuery Static Analysis Implementation

EMF-IncQuery [8] provides a means to query EMF models in a scalable manner using declarative and re-usable specification of queries. EMF-IncQuery features a high-performance incremental query engine built on an adaptation of the RETE algorithm [124]. Based on the RETE algorithm, the evaluation times of queries are practically independent of the complexity of the query and the size of the models [8]. EMF-IncQuery provides a built-in static analysis tool for type-related error detection.

In the context of this thesis, Eclipse Mars version 4.5.0 and EMF-IncQuery version 1.0.1 are used for the review.

4.7.1. Modelling technologies supported

EMF-IncQuery supports models defined using EMF and UML2 [118].

4.7.2. Program Abstract Syntax Representation

Since there is only very limited design documentation for EMF-IncQuery, this review is conducted by inspecting the source code of EMF-IncQuery to investigate its structure. The investigation of EMF-IncQuery's source code reveals that the pattern language of EMF-IncQuery is defined with the help of Xtext [12]. Xtext is a framework which enables the creation of a full implementation of a programming language including a dedicated editor, a parser, an EMF-based metamodel of the language, a serialiser and a code formatter [12]. Xtext also provides a framework on top of which static analysis rules can be implemented.

EMF-IncQuery defines two models with Xtext for the EMF-IncQuery language: *PatternLanguage* and *EMFPatternLanguage*. *PatternLanguage* defines the language constructs of EMF-IncQuery which are used to specify patterns. *EMFPatternLanguage* acts as an add-on to the *PatternLanguage*, which adds the definition of *Import* statements used to import EMF packages in the editor.

4.7.3. Encoding/Representation of the Standard Library

EMF-IncQuery provides a number of operations such as eval() and check() to evaluate values and check for boolean conditions. Such operations are implemented programatically in Java. In addition, EMF-IncQuery uses the RETE algorithm [124] for pattern matching. The process of the query processing of IncQuery is illustrated in Figure 4.11.

At first, the query definition (source code) of a program is parsed into what is called the *PatternModel*, which is essentially an instance of the *PatternLanguage* metamodel.

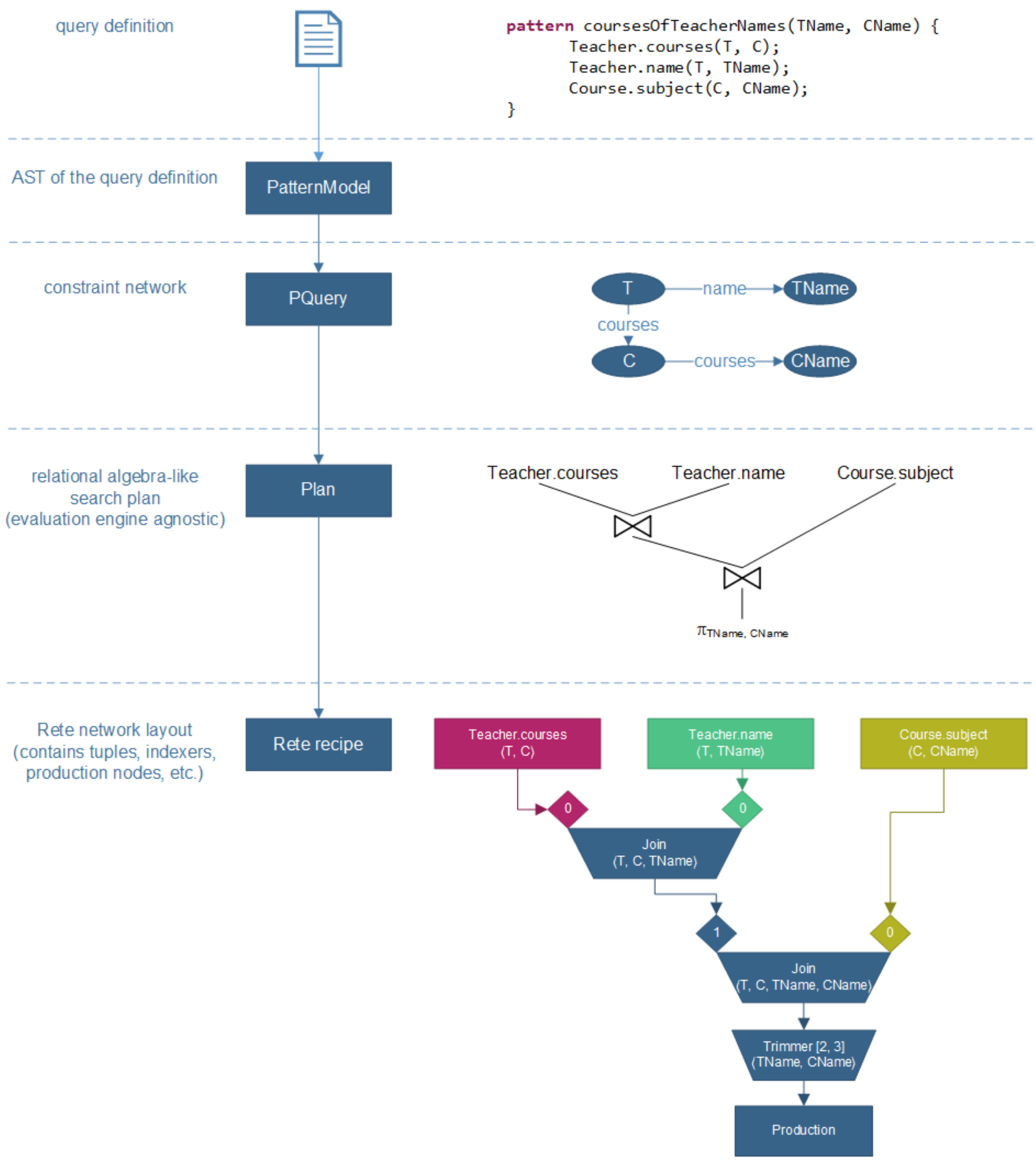


Figure 4.11.: IncQuery Query Processing [8]

The *PatternModel* is then converted into an internal representation called *PQuery* in the *pSystem*, which is essentially a constraint network. In the constraint network, queries are formed into patterns. From the constraint network, a relational algebra-like search plan is created. The search plan is then converted into a *RETE recipe*, which is an instance of the *recipes* metamodel defined with Ecore for the RETE algorithm to work on. The execution engine then uses the *Rete recipe* to produce the query results.

4.7.4. Static Analysis Capabilities

EMF-IncQuery uses a constraint-based static type-checking framework for graph patterns, which adopts a type-checking approach, called constraint satisfaction problems (CSP) for partially typed graph transformation programs [12].

In IncQuery, type checking a query is conducted as follows:

The first step of the analysis is the identification of the type system (TS) of the query, and its initialisation for the CSP solver library. The rationale behind this is that a query normally exercises a sub-set of metamodel elements; therefore, the type system used in a query consists only of the sub-set of the metamodel under question. This is similar to the concept of model pruning [122]. After the TS is collected, for each type, a unique integer set is assigned in a way that the set-subset relation between the integer sets represents the inheritance hierarchy in the type system. Informally, it is a mapping function $m : type \mapsto 2^{\mathbb{N}}$, which guarantees that $\forall T_1, T_2 \in TS : supertypeOf(T_1, T_2) \Leftrightarrow m(T_1) \subset m(T_2)$.

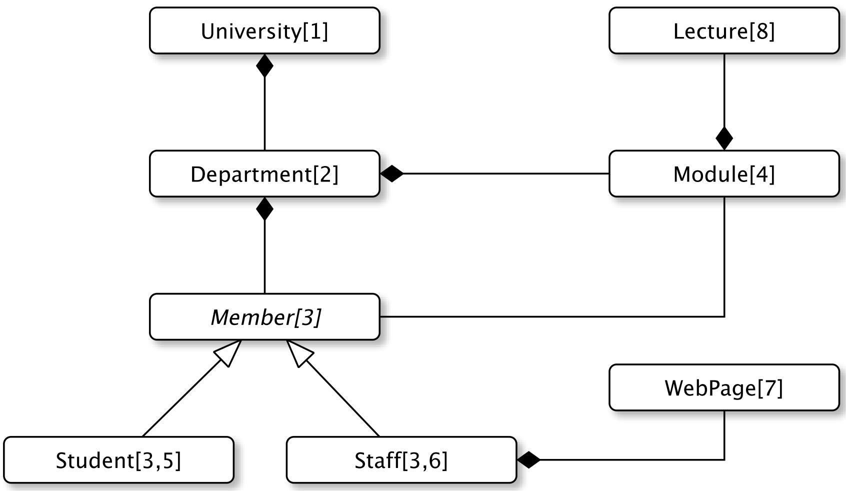


Figure 4.12.: Type System of the University metamodel

The type system for the *University* metamodel is partially depicted in Figure 4.12. As shown, type *Member* is assigned to the number 3, whereas *Student* is assigned the numbers 3 and 5, and *Staff* is assigned the numbers 3 and 6. Determining type compatibility, for example, between *Member* and *Student*, is solved by the relation $m(Member) = \{3\} \subset \{3, 5\} = m(Student)$. On the other hand, type *Student* is not compatible with type *Department* as $m(Department) = \{2\} \not\subseteq \{3, 5\} = m(Student)$.

After the type system is built for the query, a program traversal is performed, which processes every statement in every possible execution path of the transformation program. All the variables in the program are assigned to a CSP constraint. These constraints represent the type of a variable of the program, which will be matched with constraints representing the various uses of the variable. For example, for the statement:

```
University(U);
```

The variable U would be assigned the CSP constraint $m(typeOf(U)) \subset \{1\}$, according to the type system extracted in Figure 4.12. In terms of type information, the type of the expression $University(U)$ is *University*.

The constraints of the statements are then aggregated to determine the constraints of patterns. Thus, the pattern:

```
pattern university(U) = {
  University(U);
}
```

The constraint of the pattern is consequently the same as the constraint of the statement *University(U);* because it is the only statement contained in the pattern, which is $m(typeOf(U)) \subset \{1\}$.

By building the constraint network in a bottom-up manner, the constraint network is able to infer advanced type errors in the source code. For the following patterns:

```
pattern member(M) = {
  Member(M)
}


pattern studentsWithFirstClassResult(S) = {
```

```
    find member(S);
    Student.average_grade(S, G);
    check(G >= 70);
}
```

IncQuery is able to infer that, in line 6, the type of variable S is *Student* because of the property call in line 7. It is noteworthy that because of the type inference system, the users of IncQuery do not need to declare types unless it is absolutely necessary.

EMF-IncQuery does not provide any facilities based on the static analysis apart from error detection.

4.8. Related Work

Apart from the integrity checking tools mentioned in the previous sections, there are a number of tools that attempt to solve the same problem using formal methods. [125] presents EMFtoCSP, which is a tool for validating EMF models (annotated with OCL constraints) using constraint logic programming. EMFtoCSP translates EMF models along with their constraints (expressed in OCL) and the correctness properties to be checked into a constraint satisfaction problem (CSP). A constraint solver is then used to determine whether a solution for the CSP exists or not. If a solution is found, EMFtoCSP provides a valid instance of the input model to certify it. In this sense, EMFtoCSP may be a potential approach to statically analysing model management programs (OCL programs) by searching for *solution*s for the OCL constraints. However, as stated by the authors, EMFtoCSP employs a bounded verification strategy to ensure termination. Limits are set by restricting the number of instances per class and association and the domains of each attribute in the model, which may result in some problems when statically analysing complex programs which number of instances of types exceeding the restricted number set by the tool. UML2Alloy [126] uses the same approach which transforms UML/5OCL class diagrams into Alloy[3]. However, UML2Alloy desmonstrates limitations in direct manipulation of operations involving integers [125].

[3] `http://alloy.mit.edu/alloy/`

In [127], the authors present a plugin named OCL2Kodkod which is integrated to USE (UML-based Specification Environment), which is a UML and OCL tool. OCL2Kodkod, together with USE, provides a means for efficiently searching for model instances in large search spaces. OCL2Kodkod translates UML models and OCL constraints into the relational logic of Kodkod [128]. The formulas from the relation logic are then translated into boolean logic, and the resulting boolean fomulas are then searched by SAT solvers for boolean satisfiability (SAT). If the applied SAT solver finds a solution, the solution is translated back into a UML model. Similar to EMFtoCSP, OCL2Kodkod supports the search of valid UML models.

4.9. Review Findings

Whilst existing static analysis tools support the management of models defined in diverse modelling technologies, none of the available static analysis tools supports the analysis of programs that simultaneously manage models defined in different modelling technologies within a single program. In addition, the functions that the reviewed tools provide are only limited to error detection and auto-completion. None of the tools provide the facility to address scalability challenges to MDE.

Although the underlying tools/languages of the static analysis tools reviewed cover model management tasks such as model validation, model-to-model transformation, model-to-text transformation and model querying, in practice it is difficult to make use of the tools/languages in conjunction with each other due to their support for modelling technologies and their inconsistent syntax. Thus, it is possible that static analysis is not in place for a certain model management task throughout an MDE based development process because tools without static analysis support might be adopted. Thus, there is a need to have a static analysis framework that provides static analysis facilities to a broad range of model management languages, which share consistent syntax and the same means to interact with models defined in different technologies.

4.10. Chapter Summary

This chapter reviewed a number of static analysis tools in the context of MDE. In Section 4.1, the need for static analysis was introduced and a number of existing static analysis tools within the context of MDE were identified. Section 4.2 discussed the review strategy for the static analysis tools identified. Section 4.3 reviewed the built-in static analysis tools for Dresden OCL and Eclipse OCL. Section 4.4 reviewed the built-in static analysis tool for Eclipse ATL and a third-party static analyser, AnATLyzer, which provides an automated test case generation facility to detect runtime errors more efficiently. Section 4.5 and 4.6 reviewed the built-in static analysis tools for Acceleo and Xpand. Section 4.7 reviewed the built-in static analysis tools for EMF-IncQuery. The findings from the review of these tools were discussed in Section 4.9. Based on these findings, the thesis positions its research hypothesis, which will be discussed in Chapter 5.

5. Analysis and Hypothesis

This chapter presents the analysis and hypothesis of this thesis. Firstly, the analysis of the research problem is conducted, including the analysis of the target MDE platform for the research, the available infrastructure and the research methodology. This chapter also presents the hypothesis of this thesis and identifies the research objectives that need to be achieved to assess its validity.

5.1. Research Analysis

5.1.1. Research Challenges

In Chapter 2, two challenges to MDE (which are relevant to this thesis) were identified as follows:

- The need to ensure the correctness of model management programs; and
- The need to achieve scalability of model management tools when large-scale models are involved.

This thesis aims to tackle these challenges through static analysis of model management programs. In Chapter 4, a number of contemporary static analysis tools were identified and studied in terms of their ability to tackle these challenges. Throughout the study, a number of limitations were identified in the state of the art:

Simultaneous Diverse Model Management: Although most model management languages support the management of models defined in diverse modelling technologies, they do not support the management of models defined in different modelling technologies within a single program. On the other hand, languages which support the simultaneous management of models defined using different modelling technologies (e.g. languages of the Epsilon platform) do not provide built-in support for static analysis.

Multiple Model Management Language Support: The static analysis tools either support only a single model management language or a limited subset of languages in the model management language spectrum (discussed in Section 2.1.6). Within an MDE-based software development process, there are practical difficulties related to using independently developed model management languages. Such difficulties arise due to the diversity and inconsistency of the syntax of the languages. In addition, inconsistent assumptions and varying levels of support for different modelling technologies can often cause interoperability problems.

Support to Achieve Scalability of MDE: The static analysis tools reviewed focus on the detection of type-related runtime errors of model management programs. However, as discussed in Chapter 2, static analysis techniques can also help improve the performance of programs (e.g. by avoiding heavy and repetitive computation). In this context, none of the reviewed tools provides facilities that help address the scalability challenges in MDE.

Given the number of static analysis tools for model management languages that have emerged in recent years in the context of MDE, it is evident that the importance of static analysis in MDE is well understood. Contemporary static analysis tools within MDE aim to ensure the correctness of model management programs by detecting potential runtime errors. However, for the wider adoption of MDE, it is essential for it to support models defined in arbitrary modelling technologies and to support model management operations in which models defined in diverse modelling technologies are managed simultaneously (e.g. within a single model management program). In addition, as scalability has been identified as a major concern for the wider adoption of MDE, it is also desirable to investigate how static analysis techniques can be used to improve the performance of model management programs on large models.

5.1.2. Research Platform

To address the identified research challenge, it is necessary to indicate a set of target model management languages or a research platform. For this thesis, the Epsilon platform [36] is well positioned as a target platform. Epsilon provides the following features, which offer a promising basis for addressing the challenges summarised in Section 5.1.1:

- Epsilon provides an extensible model connectivity layer, EMC, which is able to manage models defined in diverse modelling technologies. Modelling technology-specific drivers can also be developed atop EMC to support arbitrary modelling technologies. In addition, Epsilon supports the simultaneous management of models defined in different modelling technologies within a single program;

- Epsilon provides a broad range of task-specific model management languages with consistent syntax, which are built atop a core language (the Epsilon Object Language [129]). It also enables the creation of further task-specific model management languages by extending EOL.

5.2. Research Hypothesis

The hypothesis of this thesis is as follows:

Reusable static analysis facilities can be used to identify errors in different types of model management programs (e.g. model transformations, validation constraints) that operate on multiple models defined using diverse modelling technologies, and to enhance the performance of programs operating on large models.

The objectives of the thesis are:

- To build a static analysis framework for the Epsilon platform, atop which reusable static analysis tools can be developed;

- To build a facility which supports the analysis of programs that manage models defined in diverse modelling technologies;
- To use the framework to develop static analysis tools for the Epsilon model management languages demonstrating its reusability and extensibility;
- To use the static analysis framework to develop facilities for analysis and automated optimisation of the performance of programs operating on large models.

Although this thesis positions Epsilon as its research platform, the outcomes of the thesis is not bound to Epsilon. Since EOL re-uses a large part of OCL's (Objec Constraint Language) language syntax, the static analysis techniques presented in this thesis can be applied to any language that re-uses OCL's language syntax without extensive changes. On the other hand, the means to address scalability through static analysis can be used as heuristics to solve similar problems for other model management languages/tools that inherit OCL's language syntax or execution semantics.

5.3. Research Scope

The purpose of this section is to establish the scope and boundaries of this work. Following the research hypothesis, the development of the static analysis framework on the Epsilon platform involves the following steps:

- Constructing the infrastructure of the static analysis framework, which includes building an analysable representation of programs of Epsilon's core language (EOL). Such a representation should be extensible in the sense that it can be extended to represent other languages that build on top of EOL, such as the Epsilon Transformation Language (ETL) and the Epsilon Validation Language (EVL);
- Constructing an extensible facility which is able to access the *metamodel*s of the models involved in a program. This facility should also be extensible in the sense that it can be extended to support arbitrary modelling technologies;
- Constructing a static analysis facility using static analysis techniques, such as abstract interpretation and lattice theory;

- Constructing a number of facilities based on the static analysis to address the scalability challenges in MDE, such as performance bottleneck detection, performance improvement of programs which operate on large models, etc.

Due to the high number of task-specific model management languages in Epsilon, a decision has been made to limit the scope of this work to the languages supporting the most recurring tasks, such as model querying, model-to-model transformation and model validation, and to provide guidelines on how to implement static analysers for the remainder of the languages of Epsilon.

5.4. Research Methodology

A typical software engineering process involving multiple analysis, design, implementation and testing iterations has been followed to evaluate the validity of the research hypothesis.

5.4.1. Iterative Analysis

In the analysis phase, an in-depth analysis of the Epsilon Model Connectivity (EMC, Section 6.1) and Epsilon Object Language (EOL, Section 6.4.1) was performed to study how the static analysis framework can be implemented to achieve the same extensibility as EMC and EOL in order to construct the infrastructure of the static analysis framework.

After the infrastructure of the static analysis framework was constructed, analysis was performed to discover which static analysis technique was best suited for the purpose of this research, in order to construct the static analysis framework.

After the static analysis framework was implemented, analysis was performed on the Epsilon Validation Language (EVL, Section 8.1) and the Epsilon Transformation Language (ETL, Section 8.2) in order to implement static analysers for these two languages.

Once the static analysis framework was constructed, analysis was performed to discover how the static analysis framework could be extended to implement facilities that provide automated performance analysis and optimisation to address the scalability challenges from various aspects.

5.4.2. Iterative Design and Implementation

Following the first analysis iteration, an extensible model connectivity layer fulfilling the purpose of static analysis was designed and implemented. Altogether, a *metamodel* of EOL was designed and implemented, together with a facility that transforms EOL programs into EOL models that conform to the EOL metamodel.

Following the second analysis iteration, a static analysis facility which performs analysis on EOL programs was designed and implemented.

Following the third analysis iteration, the static analysis framework was extended to add the modules in order to support the analysis of programs written in EVL and ETL.

Following the fourth analysis iteration, automated performance analysis and optimisation facilities were designed and implemented which address the scalability challenges in MDE from different aspects.

5.4.3. Iterative Testing

Throughout the design and implementation phases, several case studies have been used to assess the quality and usefulness of the proposed approach and the correctness of the implementation. Significant feedback has been provided by academic peers who have reviewed publications on several aspects of the framework. Errors and design defects were identified throughout the testing and were considered in future development iterations.

5.5. Chapter Summary

This chapter provided a detailed discussion on the selection of the target research platform, identified the research challenges, and also established the research hypothesis and the research methodology used to target the challenges and fulfil the research hypothesis. The following chapters present the static analysis framework which assess the validity of the research hypothesis.

6. Extensible Model Access and Model Management Language Infrastructure

This chapter presents the first analysis, design and implementation iteration of this research. As stated in Chapter 5, this iteration analyses the Epsilon Model Connectivity (EMC) layer and the Epsilon Object Language (EOL). This chapter then provides the design and implementation of the infrastructure of the static analysis framework.

6.1. Overview of the Epsilon platform

The design of Epsilon focuses on two main aspects with regards to MDE: modelling technologies and model management languages. With respect to modelling technologies, Epsilon is metamodel technology-agnostic [129]. Epsilon provides an abstract interface, named the *Epsilon Model Connectivity* layer (EMC), which enables the creation of modelling technology-specific drivers for arbitrary modelling technologies. EMC provides a set of interfaces which allow the languages of Epsilon to access models defined with different modelling technologies in a uniform way. Currently, Epsilon supports models described in EMF, plain XML, Meta Data Repository (MDR), CSV, etc.

With respect to model management languages, Epsilon provides a set of task-specific languages that are built atop a core language - the *Epsilon Object Language* (EOL) [129]. EOL reuses a significant part of the Object Constraint Language (OCL), but adds support for features such as imperative language constructs (statement sequences and groups), multiple model access, uniformity of function invocation, model modification, debugging and error reporting.

Currently, there is a broad range of task-specific model management languages implemented atop EOL, including the Epsilon Validation Language (EVL) for model validation, the Epsilon Transformation Language (ETL) for model-to-model transformation, the Epsilon Generation Language (EGL) for model-to-text transformation, the Epsilon Comparison Language (ECL) for model comparison, etc.

The Epsilon platform provides consistency, interoperability and extensibility. Consistency is achieved through the re-use of EOL - Epsilon languages have consistent syntaxes because they are built atop EOL. Interoperability is achieved by the abstract model interaction layer EMC. Extensibility is achieved by EMC and EOL - new technology-specific model drivers can be created by extending EMC and new model management languages can be created by extending EOL.

The architecture of the Epsilon platform is depicted in Figure 6.1.

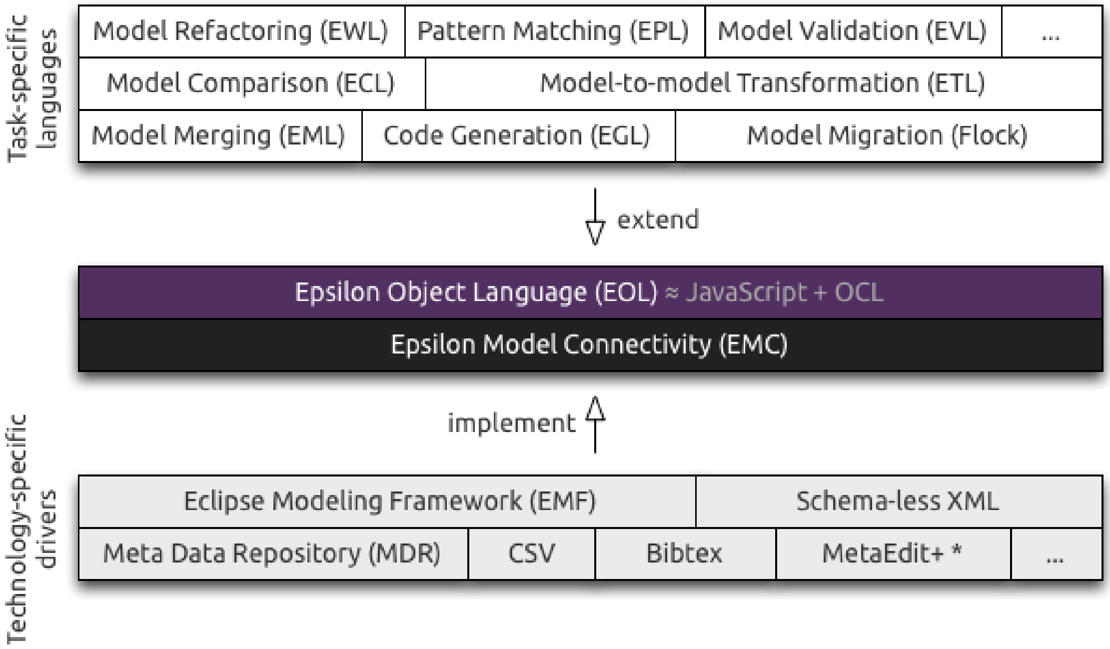


Figure 6.1.: The architecture of the Epsilon platform

6.2. The Epsilon Model Connectivity Layer

The Epsilon Model Connectivity layer (EMC) provides abstraction facilities over concrete modelling technologies such as EMF, XML, etc., and enables Epsilon programs to interact with models conforming to these technologies in a uniform manner. A graphical overview of the design is displayed in Figure 6.2.

EMC provides the *IModel* interface which abstracts away from concrete model representations. *IModel* provides a number of functions that enable model querying and modification. The *ModelRepository* acts as a container of models. It also enables Epsilon languages to manipulate models in a batch manner.

6.2.1. Loading and Persistence

The *load()* and *load(properties:Properties)* methods enable the model *drivers* which extend *IModel* to specify how a model is loaded onto memory. The *store()* and the *store(location:String)* methods are used to define how the model can be persisted from memory to a permanent storage location. [9].

6.2.2. Type-related Methods

In metamodelling architectures, there are typically two types of *type conformance* relationships. Assume a model element E from a model M and a type T from M's metamodel MM. E is said to have a *type-of* relationship with T if E is an instance of T. E is said to have a *kind-of* relationship with T if E is an instance of T or an instance of any sub-type(s) of T. With this definition, the operation *getAllOfType(type:String)* returns all the instances of *type* (provided in the parameter). The *getAllOfKind(type:String)* returns all the instances of *type* (provided in the parameter) and all instances of the sub-types of *type*.

The method *isTypeOf(element:Object, type:String)* returns true if the *element* has a *type-of* relationship with *type*. The method *isKindOf(element:Object, type:String)* returns true if the *element* has a *kind-of* relationship with *type*. The method *getTypeOf(element:Object)* returns the fully qualified name of the type with which the *element* has a *type-of* relationship. The *hasType(type:String)* method returns true if the model supports a type with the specified name (the parameter *type*). The method *isInstantiable(type:String)* returns true if a *type* defined in the metamodel is non-abstract.

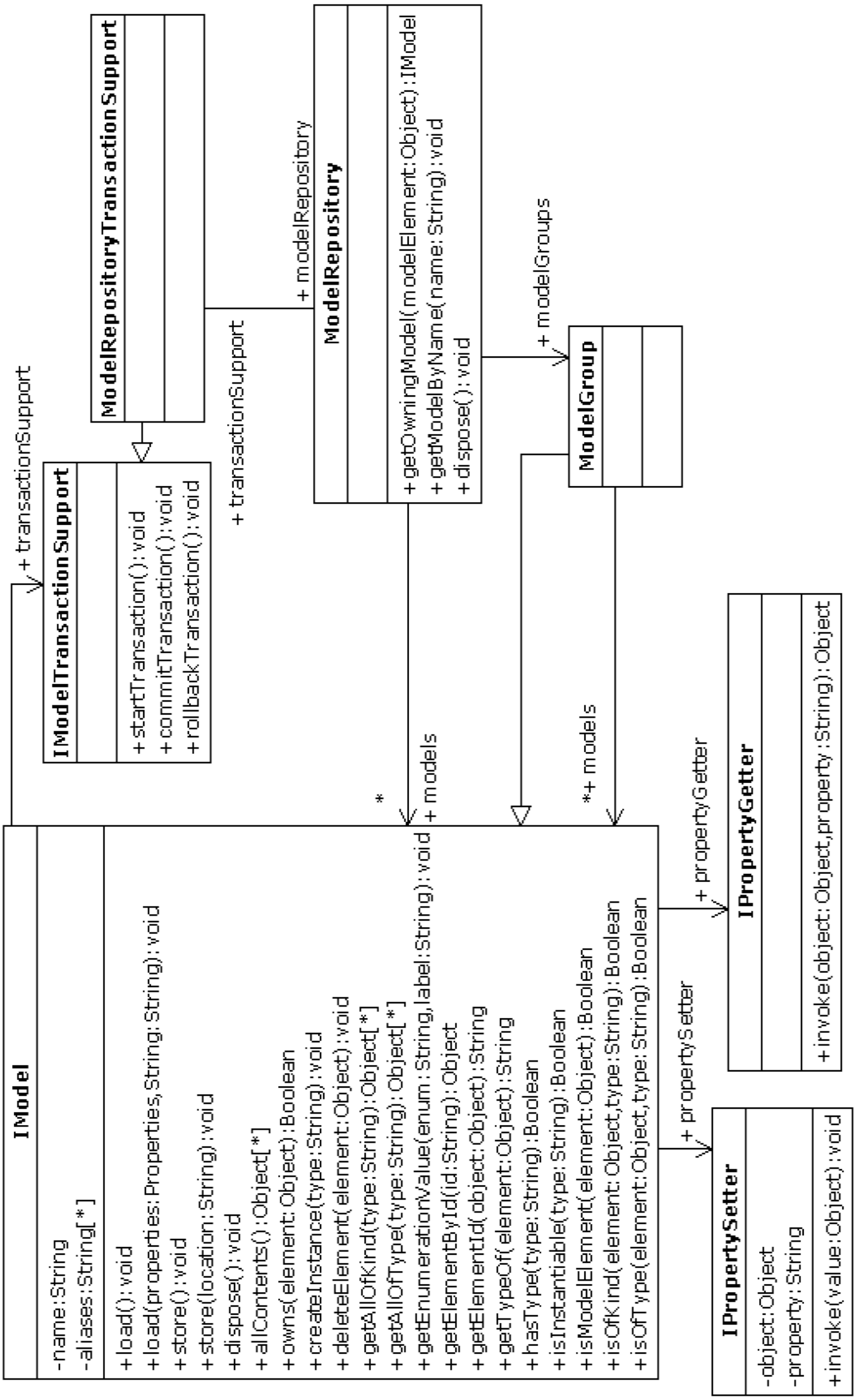


Figure 6.2.: The overview of the Epsilon Model Connectivity Layer from [9]

6.2.3. Model and contents

The method *allContents()* returns all the elements that a model contains. The method *owns(element:Object)* returns true if the *element* belongs to the model.

6.2.4. Creation, Deletion, and Modifications

Model elements are created and deleted using the *createInstance(type:String)* and *deleteElement(element:Object)* methods respectively.

To retrieve and set the values of the properties of its model elements, *IModel* uses its associated *propertyGetter (IPropertyGetter)* and *propertySetter (IPropertySetter)* respectively. Technology-specific drivers should also implement the *IPropertyGetter* and *IPropertySetter* interfaces and provide implementations for accessing and modifying the value of a property of a model element through their *invoke(element:Object, property:String)* and *invoke(value:Object)* operations.

6.2.5. ModelRepository

A model repository is a container for a set of models that need to be managed in the context of a task or a set of tasks. Apart from a reference to the models it contains, *ModelRepository* also provides the following methods:

- The method *getOwningModel(element:Object)* returns the model that owns a particular *element* (the parameter).

6.2.6. The ModelGroup

A *ModelGroup* is a group of models that have a common *alias*. *ModelGroup*s are calculated dynamically by the model repository based on model *alias*es given by model management operations. If two or more models share a common *alias*, the repository forms a new model group. The *ModelGroup* class implements the *IModel* interface. It also implements all the methods in the *IModel* interface, but in a batch manner. However, the *createInstance(type:String)* cannot be defined for a group of models, as it cannot be determined in which underlying model of the group the newly created element should belong.

6.3. Designing the Epsilon Static Analysis Model Connectivity Layer

The design of EMC makes minimal assumptions about the structure and the organisation of the underlying modelling technologies. This can be observed from EMC's deliberate avoidance of abstractions, such as *model element* (elements in a model), *type* (types in a metamodel) and *metamodel*. Instead, EMC uses *String* type for names of *type*s and Java *Object*s for model elements. This design decision promotes flexibility and extensibility - new technologies can be adapted by implementing technology-specific drivers by extending EMC. In addition, performance is also preserved - the lightweight approach of *IModel* (i.e. the use of *String* and Java *Object*s) avoids using wrapper objects for model elements and, therefore, reduces memory consumption, as opposed to using wrapper objects [9].

However, such a lightweight approach also introduces some challenges with respect to static analysis. EMC provides little support for inspecting the type structure of the *metamodel*(s) under question. The functions provided to make queries at the *metamodel* level are limited to the *hasType(type: String)* method and the *getTypeOf(element:Object)* method. Because EMC uses only Java *String* and Java *Object*, it is not possible to acquire the type hierarchy of the *metamodel*(s) under question (including the type inheritance structure, references between types, etc.). Therefore, to allow static analysis, an enhanced model connectivity layer needs to be devised.

6.3.1. Access to Metamodels

Model management programs mostly involve interacting with *model*s and their corresponding *metamodel*s. *EMC* refrains from defining such abstractions but essentially it interacts with artefacts at the model and metamodel levels. However, from the static analysis perspective, a static analysis facility is interested in the *type*s of expressions.

The types involved (for example in an EOL program) include (a subset of) the types provided in the EOL type systems and also (a subset of) the types provided by the underlying metamodel(s) (that the EOL program interacts with). Thus, a static analysis facility for model management programs is not interested in the *model*s involved, but rather the *metamodel*s of such *models*, in the sense that type names, legal features of types and their cardinalities are used to validate the correctness of the expressions with relation to their types.

6.3.2. Wrapping Metamodel Elements with Ecore

To enable static analysis, an enhanced model connectivity layer needs to be devised. Such a layer should provide *metamodel* level access, in order to obtain the information of the type structure in the *metamodel*(s) related to the analysis of a model management program.

A design decision was made to use EMF's Ecore as a wrapping layer to convert *metamodel*s (or unstructured models) defined by various modelling technologies into a common representation so that they can be accessed in a uniform way by the static analysis framework.

6.3.3. Epsilon Static Analysis Model Connectivity (ESAMC)

Based on the analysis of EMC, the Epsilon Static Analysis Model Connectivity (ESAMC) was created. The structure of the ESAMC is shown in Figure 6.3. Before a *metamodel* is accessed, regardless of its modelling technology, ESAMC requires that such *metamodel* is converted (or wrapped) to an Ecore metamodel, with the help of modelling technology-specific drivers that implement ESAMC.

The fundamental element of ESAMC is *IPackage*. *IPackage* is responsible for managing an *EPackage*. *IPackage* can be identified by its *name*. An *IPackage* may contain a number of *subPackage*s, and an *IPackage* may have a *superPackage*. In *IPackage*, a number of methods are provided to query the *type*s defined in a *metamodel*.

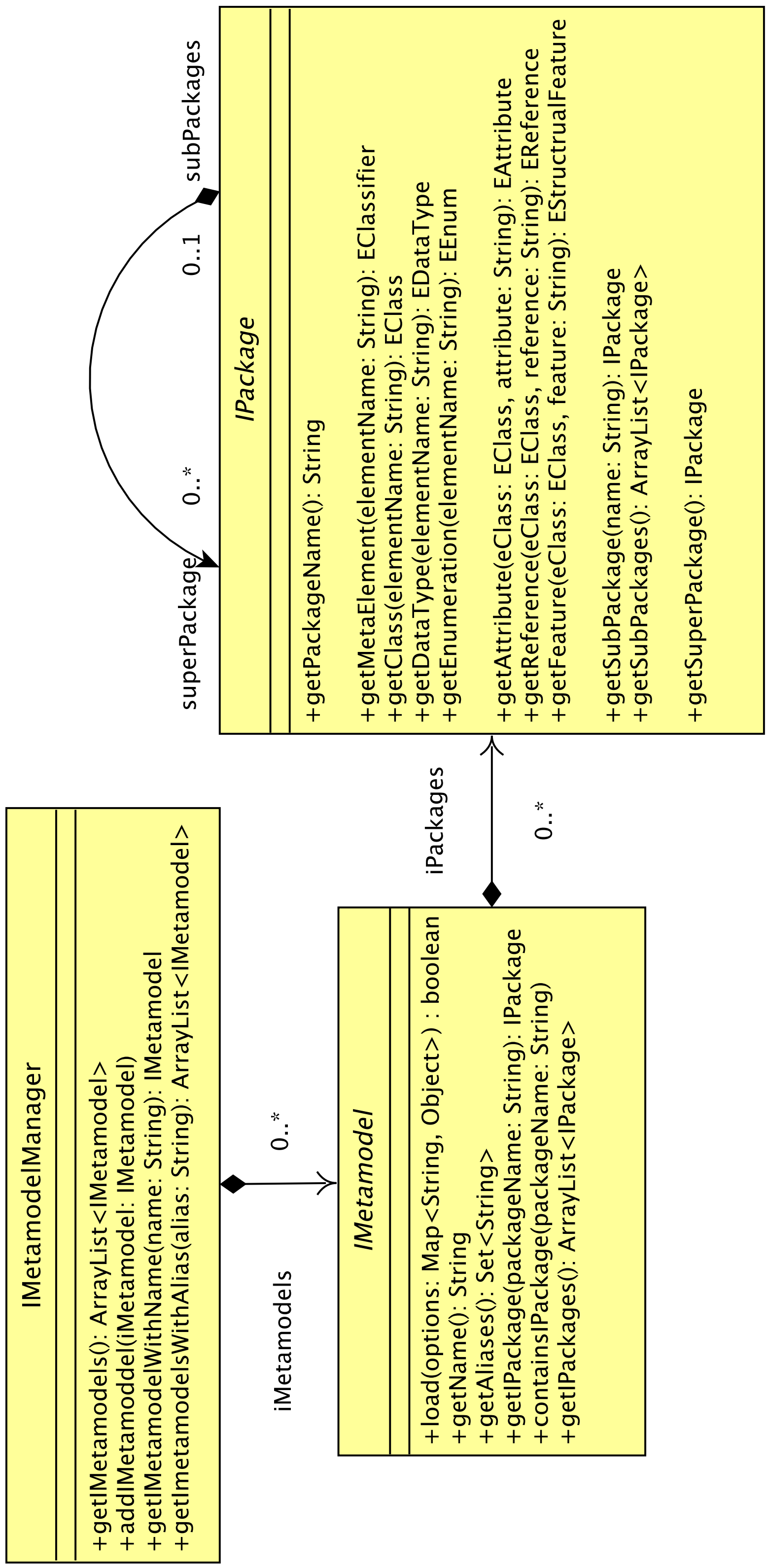


Figure 6.3.: The structure of ESAMC

- *getMetaElement(elementName: String)*, *getClass(elementName: String)*, *getDataType(elementName: String)* and *getEnumeration(elementName: String)* are used to fetch an *EClassifier*, *EClass*, *EDataType* or *EEnum* respectively by the *elementName* provided;
- *getAttribute(eClass: EClass, attribute: String)*, *getReference(eClass: EClass, reference: String)* and *getFeature(eClass: EClass, feature: String)* are used to fetch the corresponding *EAttribute*, *EReference* and *EStructuralFeature* defined by the *EPackage* with the parameters provided;
- *getSubPackage(name: String)* fetches any *IPackage* contained by the current *IPackageDriver*;
- *getSubPackages()* returns all the *IPackage*s contained by the current *IPackageDriver*;
- *getSuperPackage()* returns the parent *IPackage* of the current *IPackage*.

With such methods, the static analysis mechanism is able to fetch the *type*s defined in a *metamodel* and query its *attribute*(s) and *reference*(s). Because the *metamodel* is represented as *EPackage*s, the type structure inside the *metamodel* can be navigated from a given *type*.

Interface *IMetamodel* is used to represent a *metamodel*. It may contain a number of *IPackage*s. The *metamodel* is loaded with a method call to *load(options: Map<String, Object>)*, which takes a *Map* that contains loading options. The user of ESAMC can specify how the *metamodel* is loaded (for example, by looking for the underlying EPackage in the EPackage registry, etc.). An *IMetamodelDriver* can be identified by a *name* and a number of *aliases*. Methods are provided to get a specific *IPackage* by name.

The *IMetamodelManager* acts as a container to contain *IMetamodel*s and provides interfaces for retrieving *IMetamodel*s either by name or by aliases.

6.3.4. Summary

Compared to EMC, ESAMC is a read-only layer which only accesses the *metamodel*s of (structured) *models* (and the *model*s if they are unstructured, as discussed in Chapter 9). ESAMC is largely inspired by EMC in the sense that it provides a uniform measure to access metamodels defined in different modelling technologies. Modelling technology-specific drivers can be devised by extending the *IMetamodel* and *IPackage* interface. Unlike EMC, ESAMC requires that *metamodel*s defined in different modelling technologies to be *wrapped* (or *converted*) into Ecore *metamodel*s. Such an approach is necessary to provide a uniform way to access type hierarchy of *metamodel*s defined in different modelling technologies.

6.4. The Static Analysis Infrastructure for EOL

The Epsilon Object Language (EOL) is the core language of Epsilon. EOL provides a reusable set of common model management facilities atop which task-specific languages can be implemented. Epsilon's other task-specific model management languages (such as the Epsilon Validation Language, Epsilon Transformation Language, Epsilon Generation Language, etc.) are defined atop EOL. EOL can also be used as a standalone general-purpose model management language for tasks that do not fall into the patterns targeted by task-specific languages. EOL reuses the model navigation feature of OCL, but provides additional support for language features like multiple model access, statement sequencing and model modification capabilities. Since EOL is the core language of Epsilon and the basis of all other task-specific Epsilon languages, this thesis first focuses on EOL.

6.4.1. The Current Abstract Syntax Representation of EOL

The abstract syntax of EOL is implemented using an ANTLR-based [68] parser, in the form of Abstract Syntax Trees (ASTs). The ASTs are homogeneous trees: an AST node contains a *type* (of type *int*), a *text* which contains the content of the node, and a number of *children* which are of type AST node. For example, for the following EOL example program:

```
var a = 1;
var b = 2;
```

```
var c = a + b
```

Listing 6.1: An Example EOL program

the ANTLR parser parses the program into an AST as shown in Figure 6.4. At the top level is an AST node with *type* 61 and *text* “EOLMODULE”, it then contains another AST node with *type* 62 and *text* “BLOCK”. AST of *type* 62 then contains 3 *children*, and so on.

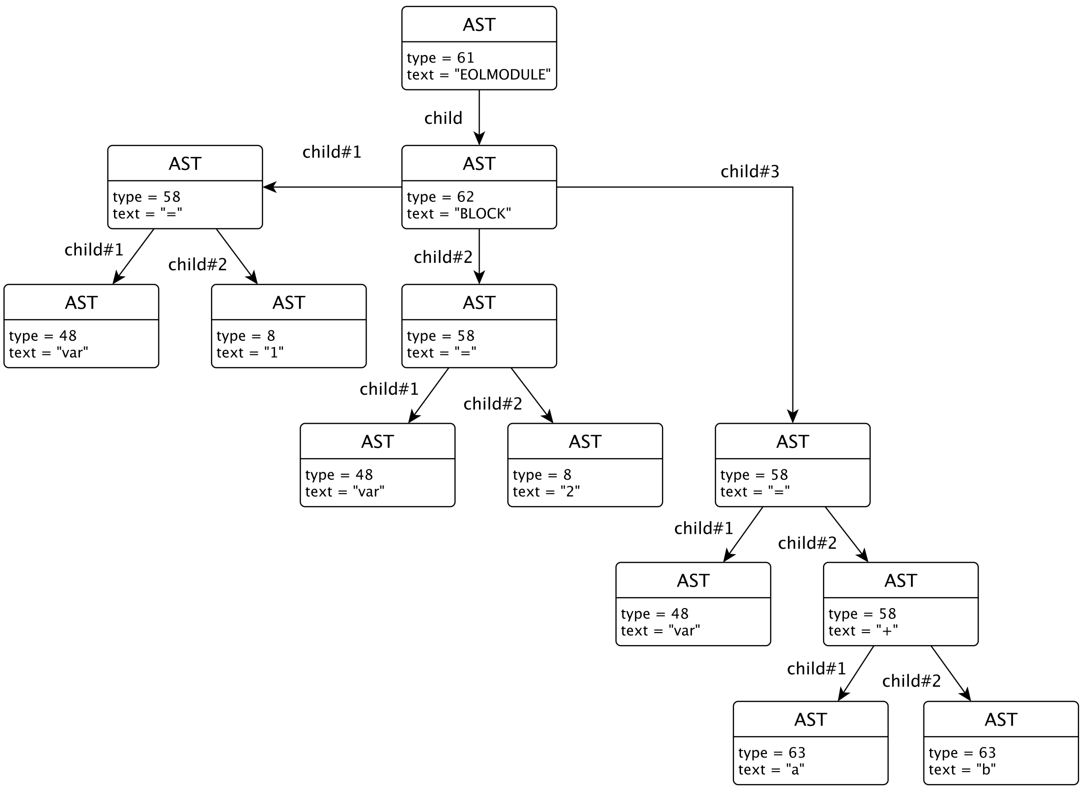


Figure 6.4.: An instance of EOL’s ANTLR-based AST

The EOL execution engine implements different *Executor*s for different *type*s of ASTs in order to execute an EOL program. However, for the purpose of this thesis, it is not desirable to perform static analysis on the ASTs for two major reasons. Firstly, the ASTs are homogeneous trees in the sense that all the nodes are of the same type: AST, which makes it difficult to express type hierarchy of all the types in EOL’s abstract syntax. In addition, the homogeneous nature of AST makes performing static analysis on it tedious and error-prone. Secondly, ANTLR is a choice of parsing technology that Epsilon adopts. However, Epsilon should not be tightly coupled to ANTLR. As Epsilon evolves over time, new parsing technologies may emerge, which can be potential choices for parsing EOL programs. Thus, the static analysis framework should avoid tight coupling with ANTLR by giving EOL a higher level representation of its abstract syntax.

Thus, in order to establish a static analysis framework for Epsilon, it is necessary to provide a heterogeneous representation of the abstract syntax of EOL. The design decision is to create the abstract syntax in the form of an Ecore metamodel.

6.5. Modelling the Epsilon Object Language

Following the discussions in the previous section, the first step of the research was to devise a metamodel for EOL. There are a number of choices of approaches/technologies at hand. In terms of approaches, there are two approaches that are feasible:

- To use Xtext (or similar tools) to define the grammar for EOL, Xtext then uses the grammar to generate a number of facilities such as parser, Ecore-based metamodels, static analysis infrastructure, etc. However, this approach involves perfoming redundant tasks such as defining the grammar (as Epsilon already defines a set of grammars for its parsers for different Epsilon languages). In addition, the static analysis infrastructure generated by Xtext binds the tool tightly to Xtext and provides little flexibility. Thus, this approach does not seem to be ideal for this research.

- To use existing modelling languages to define a metamodel of EOL, then translate EOL source code into instances of the EOL metamodel. This approach eliminates the redundant work and focus directly on the task of this research. In addition, this approach gives great amount of freedom to implement the static analysis framework. Therefore, this approach is adopted.

Although EOL has been inspired by OCL, the metamodels of the two languages are substantially different. For example, EOL does not support a number of OCL constructs such as *let* statement and *tuples*. On the other hand, EOL provides more languages constructs such as imperative statements and statement blocks. Thus, it is not possible to define the EOL metamodel as an extension of the OCL metamodel.

A detailed discussion on the EOL metamodel is provided in Appendix A, altogether with necessary concrete syntax examples of its elements.

6.6. Transformation from Homogeneous AST to Heterogeneous AST

With the EOL metamodel defined, the next step is to perform a model-to-model transformation, which transforms the homogeneous AST produced by EOL's ANTLR-based parser into a model that conforms to the EOL metamodel, as shown in Figure 6.5. For this purpose, the AST2EOL facility is created.

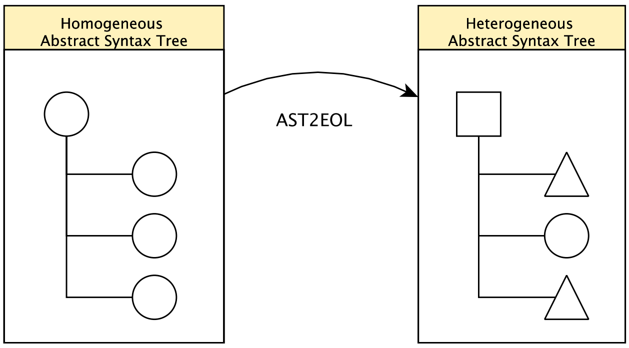


Figure 6.5.: The transformation from Homogeneous AST to Heterogeneous AST

The AST2EOL facility (implemented in Java) comprises several components, as shown in Figure 6.6. The *AST2EOLContext* provides a container of all the necessary facilities (hence the word *context*) needed during the AST2EOL transformation. The *AST2EOLContext* also acts as the centralised access control. It has a *create(AST ast)* method, which takes an ANTLR-based AST, and creates an *EOLElement*. The *AST2EOLContext* also keeps track of the *EOLElement*s created with their corresponding mapping to their ASTs, which is stored in the *traces* (of type *Map*).

The *AST2EOLContext* contains an *EOLElementCreatorFactory*, which is responsible for providing *EOLElementCreator*s during the AST2EOL transformation. *EOLEle-*

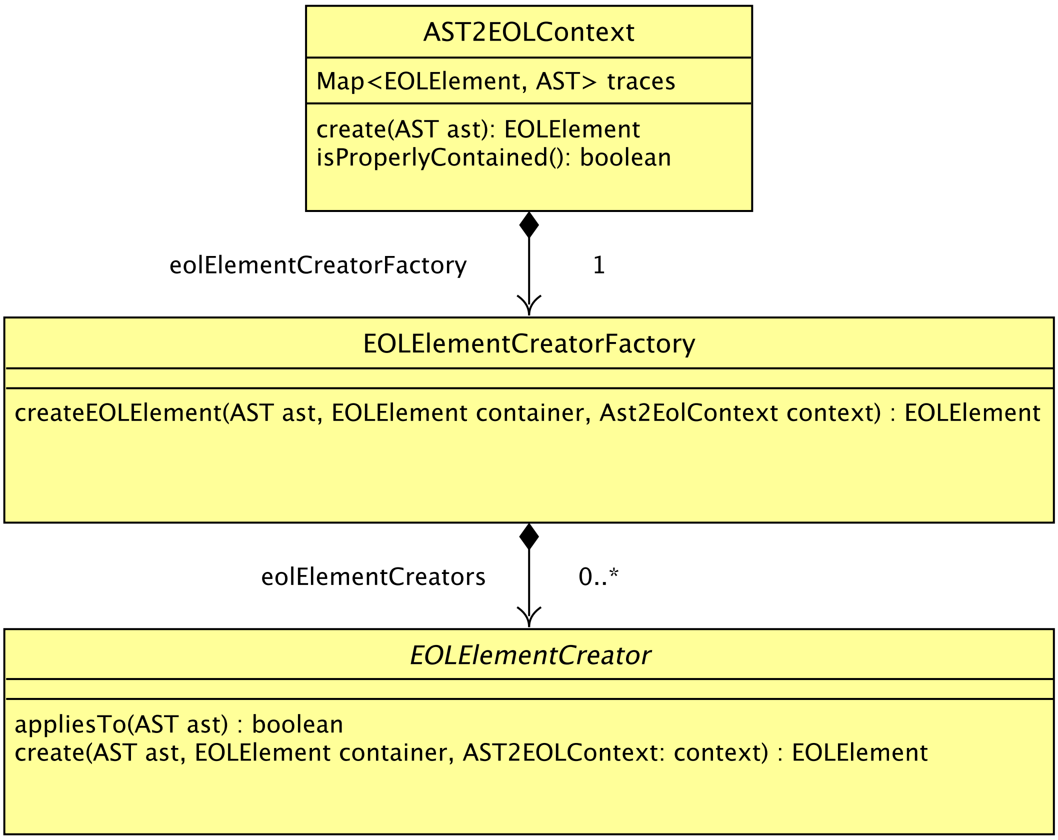


Figure 6.6.: The AST2EOLContext

mentCreators are one-to-one mappings to their counterparts in the EOL metamodel. The structure of *EOLElementCreator* and its sub classes is shown in Figure 6.7. Each *EOLElementCreator* contains two methods:

- The *appliesTo(AST ast)* acts as a guard, which checks the *type* and the children of the AST to determine if the *EOLElementCreator* is applicable to the AST:
- The *create()* method is responsible for creating the corresponding *EOLElement*. It takes three parameters: the *ast* is the AST in question, the *container* is the previously created *EOLElement* by another *EOLElementCreator*(s) which contains the *EOLElement* to be created, and the *context* is the *AST2EOLContext* which provides all the auxiliary facilities needed.

The details of individual *EOLElementCreator*s are not discussed in detail. During the

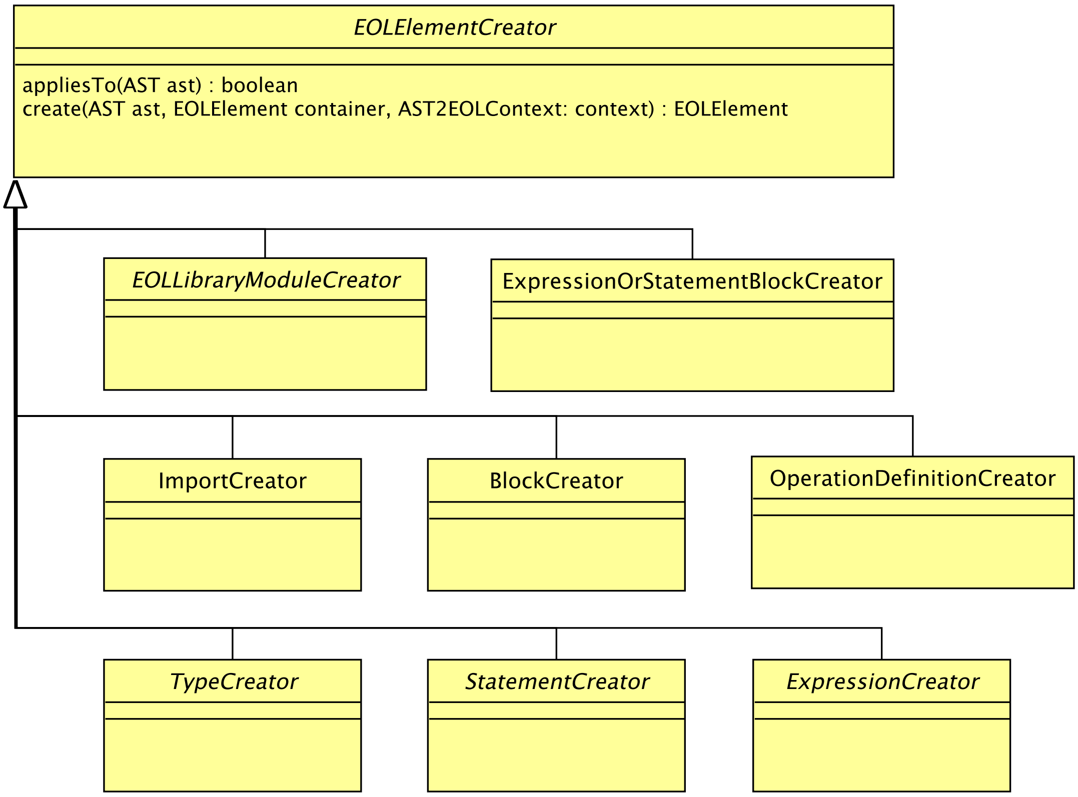


Figure 6.7.: The EOLCreator and its sub classes

transformation, the location information of ASTs is also copied over to target *EOLElements* so that locating *EOLElements* in Eclipse editors can be supported.

Because of the structure of the EOL metamodel, at the end of the transformation, the *EOLElements* created form a tree structure. The *isProperlyContained()* method in *AST2EOLContext* checks if all the *EOLElements* are properly contained.

With the AST2EOL facility in place, transforming the AST in Figure 6.4 (which is the representation of the EOL program in Listing 6.1) generates the EOL model in Figure 6.8. It is noteworthy that during the AST2EOL transformation, no type resolution is performed; hence, the types of variables a, b and c in the assignment statements are not resolved.

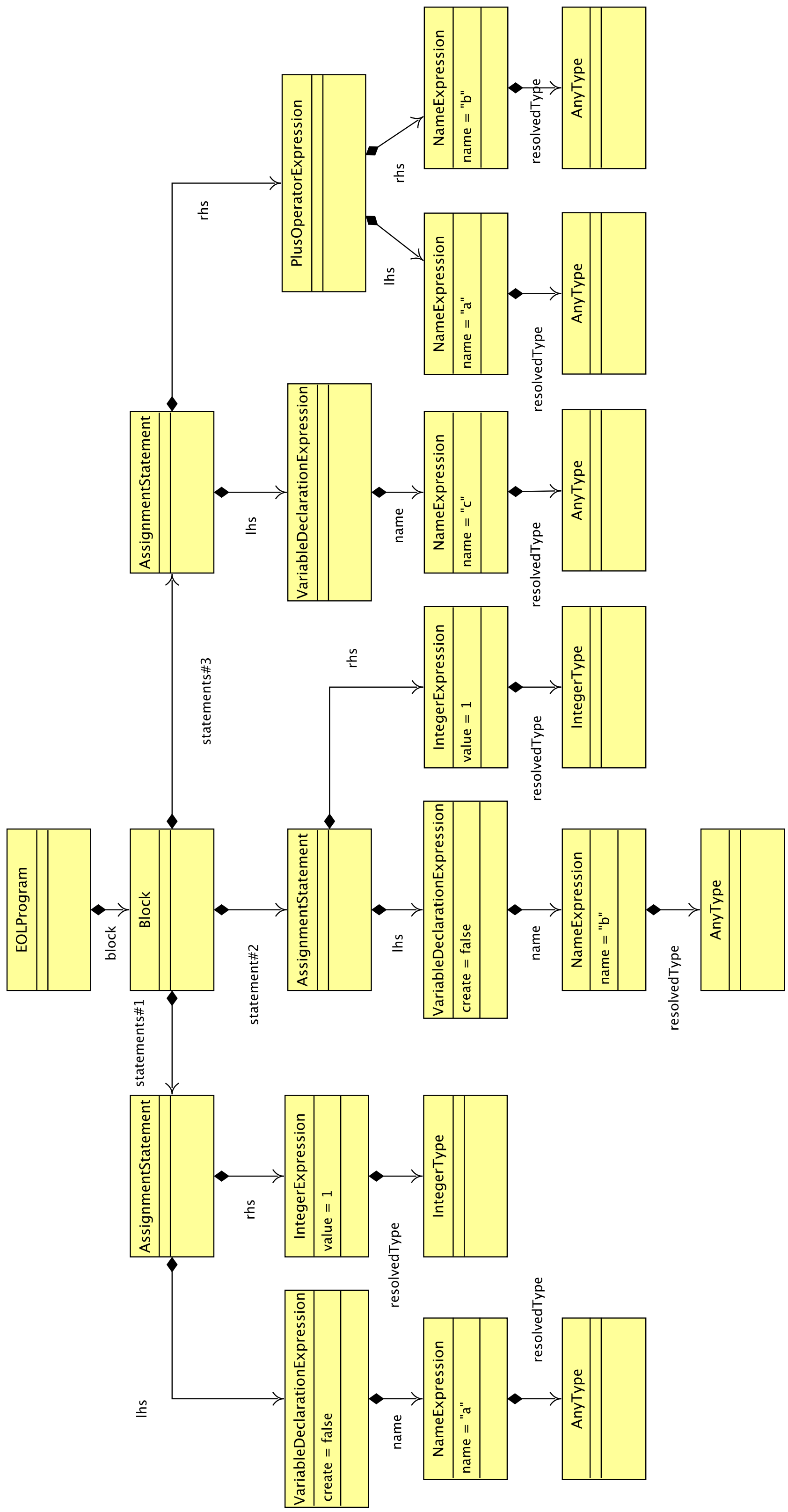


Figure 6.8.: The EOL model of the program in Listing 6.1

6.6.1. Summary

In this section, the AST2EOL facility was presented. AST2EOL is able to transform an AST produced by EOL's ANTLR-based parser into a model that conforms to the EOL metamodel.

6.7. Chapter Summary

This chapter presented the first analysis, design and implementation iteration of this thesis. In this chapter, the Epsilon Model Connectivity was reviewed and analysed. The analysis drew the conclusion that an enhanced model connectivity layer should be constructed. This chapter then moved onto the design and implementation of the Epsilon Static Analysis Model Connectivity layer (ESAMC), which provides a uniform layer for accessing metamodels defined in different modelling technologies. ESAMC comes naturally with an EMF driver as it is based in EMF. In Chapter 9, the ESAMC is extended and a plain XML model driver is created to evaluate the extensibility of ESAMC. This chapter then moved onto the design and implementation of the EOL metamodel. A detailed discussion of the design and the implementation of the EOL metamodel was then provided in Section 6.5. An AST2EOL transformation facility was then presented in Section 6.6, which transforms ANTLR-based ASTs into models that conform to the EOL metamodel.

The ESAMC and the EOL metamodel constitute the infrastructure of the Epsilon static analysis framework. The static analysis facilities for EOL, EVL and ETL are discussed in the following chapters.

6.8. Terminology

Epsilon: Epsilon stands for *Extensible Platform for Specification of Integrated Languages*, Epsilon is a platform on which MDE activities can be performed.

Epsilon Object Language (EOL): EOL is the core language of the Epsilon platform. EOL is inspired by the Object Constraint Language (OCL) which is used to express constraints on UML models. EOL provides additional language features such as imperative language constructs, multiple model access, uniformity of function invocation, model modification, etc.

Epsilon Model Connectivity layer (EMC): EMC is the model connectivity layer for the Epsilon platoform, it provides an abstract layer which enables the access of models defined in different modelling technologies in a uniformed way. EMC can be extended to build model drivers to access models defined in modelling technologies that are not currently supported by Epsilon.

Epsilon Static Analysis Framework: The Epsilon static analysis framework is the outcome of this research, it provides static analysis facilities for the languages of the Epsilon platform.

Epsilon Static Analysis Model Connectivity (ESAMC): ESAMC is the metamodel connectivity layer for the Epsilon static analysis framework, which acts similar to EMC. ESAMC can be exteded to access metamodels defined in different modelling technologies.

Abstract Syntax Tree (AST): AST in this chapter refers to the homogeneous abstract syntax tree produced by the Epsilon parser (an ANTLR-based parser).

EOL metamodel: The EOL metamodel refers to the abstract syntax of the Epsilon Object Language (EOL) represented in the form of a Ecore based metamodel.

AST2EOL: In the context of this thesis, the term AST2EOL refers to the transformation which transforms a homogeneous abstract syntax tree produced by the Epsilon parser to a model which conform to the EOL metamodel.

7. A Modular Static Analysis Framework for Epsilon

This chapter discusses the development iteration in which the static analyser for the Epsilon Object Language (EOL) was created. In Section 7.1, the design of the infrastructure of the EOL static analyser is presented. In Section 7.2, the EOLVisitor facility, which acts as the foundation of the EOL static analyser, is presented. In Sections 7.3 and 7.4, the design and the implementation of the EOL static analyser (the EOL variable resolution facility and the EOL type resolution facility) are discussed. The static analyser of EOL is essential to the Epsilon static analysis framework in the sense that it provides a baseline so that modules (static analysers for other Epsilon languages) can be developed atop it to extend its support for other Epsilon languages.

7.1. Infrastructure of the EOL Static Analyser

The EOL static analyser consists two main procedures when analysing an EOL programs for potential errors: *variable resolution* and *type resolution*. Consider an example program written in EOL:

```
var a: Integer = 1;
var b: Integer = 2;
var c: Integer = a + b;
c.println()
```

Listing 7.1: An example EOL program

A main objective for static analysis is to identify potential runtime errors with regards to type safety. In the program, in order to check the type of the expression $a + b$, it is necessary to acquire the types of a and b. In order to acquire the types of a and b, it is necessary to establish the link between a variable's declaration and its references. Therefore, for the program above, a link between the variable reference a in line 3 and the variable declaration *var a : Integer* in line 1 needs to be established, as do the links between the variable reference b and its declaration *var b: Integer* and the variable reference c and its declaration *var c: Integer*. This thesis refers to the establishment of the {variable declaration, variable reference} links as *Variable Resolution*.

With all the variables resolved, the next step is to resolve the types of the variable references, i.e. the variable references of a and b in line 3, and the variable reference of c in line 4. By looking at the variable declarations of these variables, it can be inferred that the types of a, b and c are all *Integer*s (since their types are declared at their variable declarations respectively). This thesis refers to this process as *Type Resolution*.

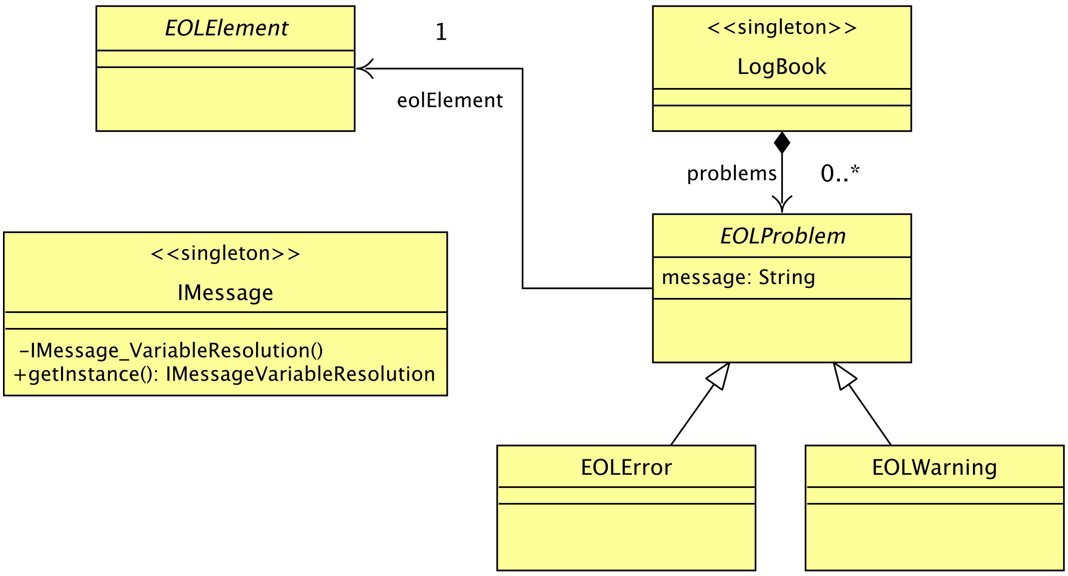


Figure 7.1.: The structure of the LogBook facility

Performing static analysis and detecting potential runtime errors, variable and type resolutions are essential. Variable resolution and type resolution are supported by the widely used Eclipse JDT (Java Development Tool) static analyser, which is a mature static analysis tool to detect errors in Java programs. However, the JDT static analyser is not suitable to be reused for static analysis of EOL, due to the following reasons.

Firstly, EOL programs operate on *models*, which requires performing type checks of expressions whose type is defined either in EOL or in the *metamodel* of the *model*s on which EOL operates on. Secondly, although EOL and Java share some language syntax, the type system of EOL and the operations provided for the type system are rather different to Java. Finally, Java is a statically typed language, where EOL is dynamically typed. Due to the above reasons, a decision was made to build an independent static analysis framework for Epsilon.

A *LogBook* facility is created to log the warnings and errors identified during the variable and type resolution processes. The structure of the *LogBook* is shown in Figure 7.1. *EOLProblem* (abstract) represents the problems that may arise during the variable and type resolutions. *EOLProblem* is associated with a *message* (of type String) and a reference to the *EOLElement* where the problem occurs. *EOLProblem*s are further categorised into *EOLError*s and *EOLWarning*s. *LogBook* contains a collection of *EOLProblem*s and provides functions (not discussed in detail) to add and extract *EOLProblem*s. For variable resolution and type resolutions, a number of warnings/errors have been identified and their corresponding messages are stored in the *IMessage* facility.

7.2. The EOLVisitor Facility

Section 6.6 discusses the AST2EOL facility which is able to transform an ANTLR-based AST into a model that conforms to the EOL metamodel. In order to perform static analysis on EOL models, a facility is needed which is capable of traversing instances of the EOL metamodel. To achieve this, a facility named *EOLVisitor* is created using a visitor generation framework, which is essentially a model-to-text transformation tool written in the Epsilon Generation Language (EGL). The transformation has been made available as an Eclipse plug-in under the EpsilonLabs open-source project[1]. The plug-in works on EMF generator models (.genmodel) and is able to generate an Eclipse plug-in which contains a visitor facility for any given genmodel. Figure 7.2 shows a screenshot, which illustrates how *EOLVisitor* can be generated from the EOL genmodel.

[1] `https://github.com/epsilonlabs/epsilonlabs/tree/master/com.googlecode.epsilonlabs.evg.updatesite`

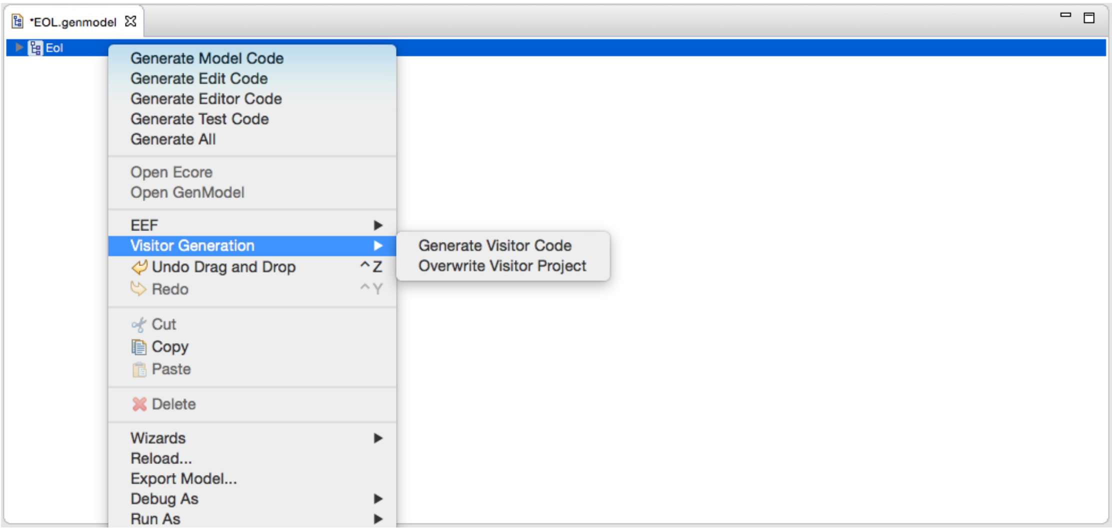


Figure 7.2.: Generating the EOLVisitor facility using visitor generation plug-in.

The structure of the *EOLVisitor* facility is shown in Figure 7.3. The core of the *EOLVisitor* facility is the *EOLVisitorController*, which acts as the centralised control and access point. *EOLVisitorController* contains a number of *EOLElementVisitor*s, one for each non-abstract element defined in the EOL metamodel. An *EOLElementVisitor* provides two methods:

- The *appliesTo(EOLElement element, T context)* method acts as a guard, which checks if the *EOLElementVisitor* is applicable to a given *EOLElement*;
- The *visit()* method performs the traversal of the applicable *EOLElement*. Developers who use *EOLElementVisitor* should implement their own algorithms inside the *visit()* method. The method provides a generic typed parameter so that the developers can develop their own *context* in order to achieve the desired functionality.

EOLVisitorController acts as the uniform access point for visiting an *EOLElement*. When *visit(Object o, T context)* is called, the appropriate *EOLElementVisitor* is selected and the *EOLElement* is visited according to the algorithm defined in each *EOLElementVisitor*.

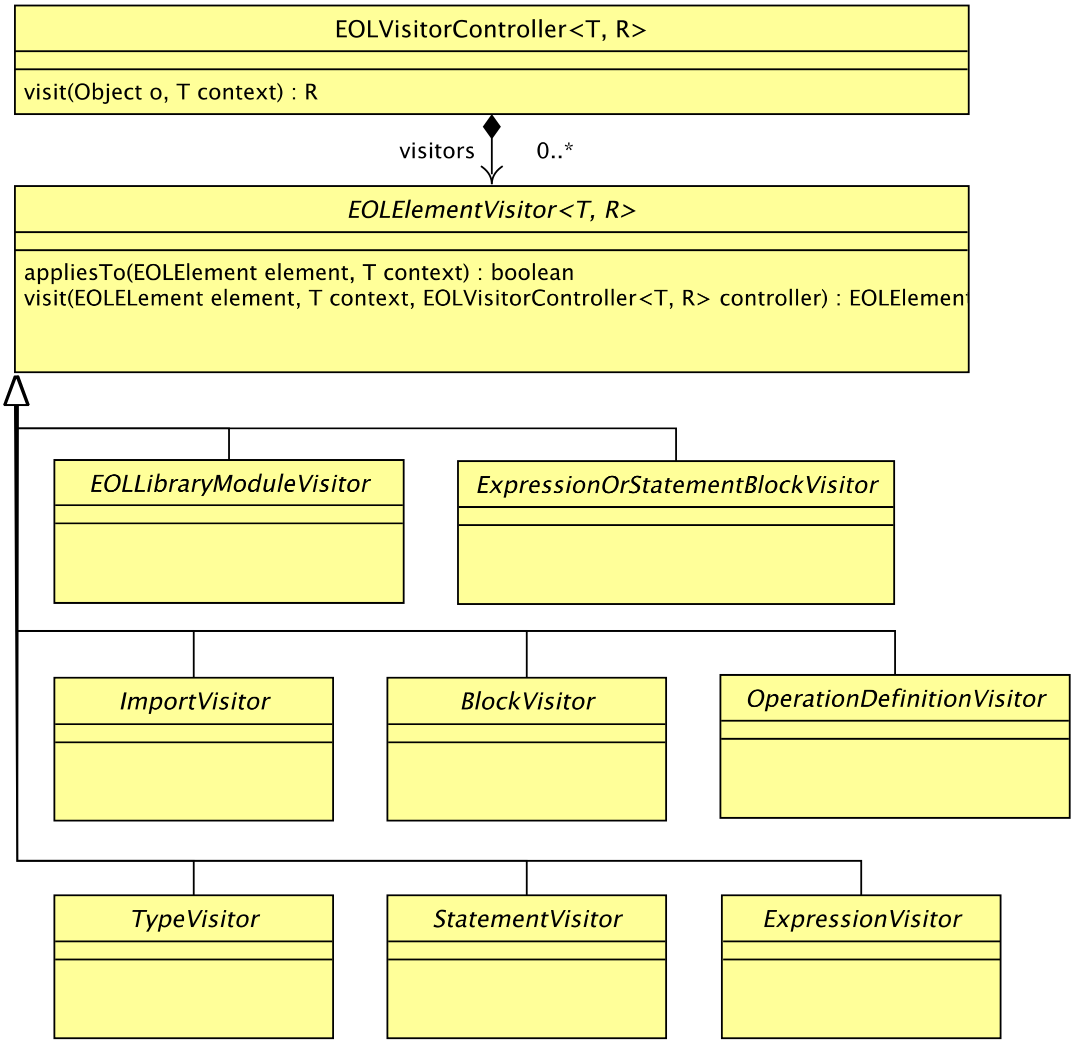


Figure 7.3.: The structure of EOLVisitor

7.3. The EOL Variable Resolution Facility

With the EOLVisitor facility in place, the next step is to develop the EOL variable resolution facility. The structure of this facility is shown in Figure 7.4. The *EOLVariableResolver* is the centralised access point of the facility. *EOLVariableResolver* contains an *EOLVariableResolutionController* (by extending the EOLVisitorController in the EOLVisitor facility), which contains a number of *EOLElementVariableResolvers* that

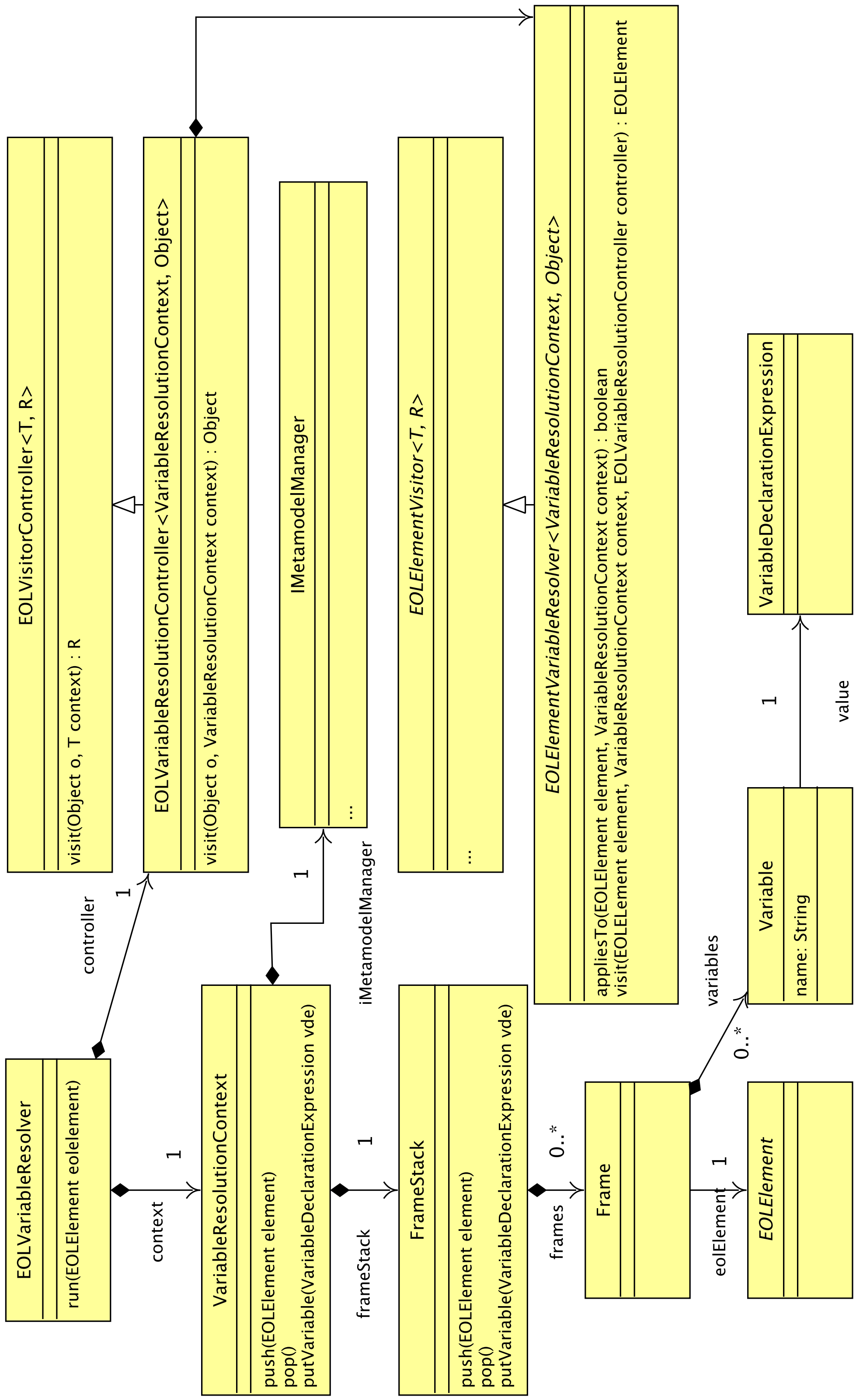


Figure 7.4.: The structure of the EOL Variable Resolution Facility.

are implemented by extending the EOLVisitor facility. Thus, when the *run(EOLElement eolElement)* method in *EOLVariableResolver* is executed, *EOLVariableResolutionController* traverses the *eolElement* provided using the *EOLElementVariableResolvers* in correspondence with each *EOLElement* encountered.

EOLVariableResolver also contains an *EOLVariableResolutionContext.* The main responsibility of the *EOLVariableResolutionContext* is to maintain the *FrameStack*, which represents the stack of scopes within an EOL program (e.g. the local scope within an *if* statement, etc.). Whenever the following *EOLElement*s are encountered, there is a need to push a new *Frame* onto the *FrameStack*:

- The *EOLModule* element. An *EOLModule* is the entry point of control when executing an EOL program;
- The *OperationDefinition* element, within an EOL program. Each *OperationDefinition* is visited once, and when it is visited, a corresponding *Frame* is pushed onto the *FrameStack*;
- Control flow constructs, such as *IfStatement*, *ForStatement*, *WhileStatement*, *SwitchStatement*, *SwitchCaseStatement* and *TransactionStatement*;
- *ExpressionOrStatementBlock* enclosed in control flow constructs;
- *FOLMethodCallExpression*, as first order logic method calls use lambda expressions, which declare *iterator*s within them (Section A.2.11).

The push() and pop() methods of the *StackFrame* are called within the corresponding *EOLElementVariableResolver*s for the *EOLElement*s mentioned above. Consider an example program shown in Listing 7.2. In line 1, a variable named *sequence* is declared which contains *Integer* values from -10 to 10. Line 2 extracts a random element from *sequence* and assigns the value of that element to a variable named *a.* Lines 3 to 9 are an *if* statement. In line 3, the value of *a* is examined; if a is less than 0, statements in lines 4 and 5 are executed, otherwise line 8 is executed. Lines 11 to 16 define an operation named *abs()* with context type *Integer* and return type *Integer* which calculates the absolute value of a given *Integer.*

```
1  var sequence : Sequence(Integer) = Sequence{-10..10};
2  var a : Integer = sequence.random();
3  if(a < 0) {
4    "a is negative".println();
5    a.abs().println();
6  }
7  else {
8    a.println();
9  }
10
11 operation Integer abs() : Integer {
12   if(self < 0) {
13     return -self;
14   }
15   return self;
16 }
```

Listing 7.2: An EOL Example to demonstrate the EOL Variable Resolution facility

The visualisation of all the *Frame*s inserted in the *FrameStack* is shown in Figure 7.5. It is worth noting that by the end of the variable resolution, the *FrameStack* during the variable resolution process on the program of Listing 7.2 is empty - the EOL variable resolution facility traverses an *EOLModule* in a top-down manner. When the traversal exits a control flow construct as discussed above, the top of the *FrameStack* is popped from it.

In Figure 7.5, when the *EOLModule* is visited, a *Frame* (*Frame:EOLModule*) is pushed onto the *FrameStack*. The *block* of the *EOLModule* is visited first, so that variables *sequence* and *a* are inserted in the *Frame:EOLModule*. When line 3 is encountered, the variable resolution pushes a *Frame* onto the stack named *Frame:IfStatement#ifBody*, and variable resolution takes place in lines 3-6 to link references of *a* (lines 3 and 5) to its declaration. After the *if* body is traversed, the *Frame:IfStatement#ifBody* is popped from the *FrameStack*, then the *Frame:IfStatement#elseBody* is pushed onto the

FrameStack, and variable resolution is performed by linking the reference of *a* in line 8 to its declaration. The *Frame:IfStatement#elseBody* is then popped from the *FrameStack* after line 9. In line 11, the operation *abs()* is encountered and a new *Frame* is pushed onto the *FrameStack*. As with all operation frames, the new frame has two associated variables: *self* is used to refer to the object that calls the operation and *_result* is used to refer to the result returned by the operation. For variable resolution, these two variables are inserted into the *Frame:OperationDefinition*. Operation *abs()* contains an *if* statement from line 12 to line 14; thus, a *Frame* is pushed onto the *FrameStack* by variable resolution.

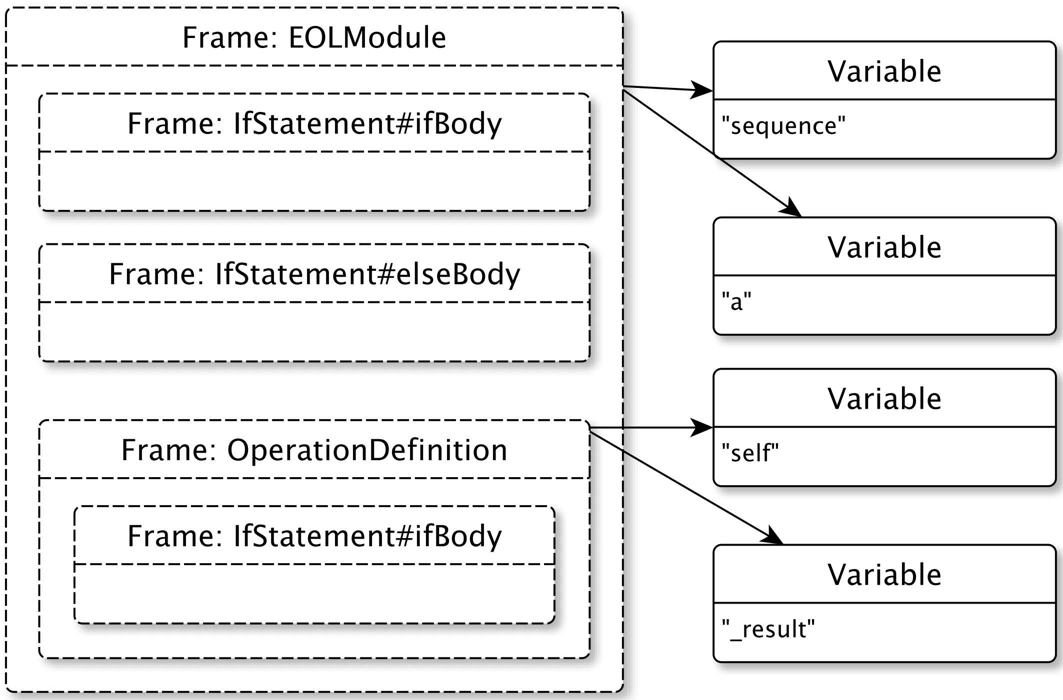


Figure 7.5.: StackFrame footprint of the program in Listing 7.2

The variable resolution process is responsible for establishing the link between a *NameExpression* (Section A.2.6) and a *VariableDefinitionExpression* (Section A.2.7), which essentially calculates the *resolvedContent* property of the *NameExpression* and the *references* property of the *VariableDeclarationExpression*.

EOL supports *variable shadowing* in the sense that a variable with a name that has been previously declared in a parent *Frame* can be declared in a child *Frame*. Consider the example in Listing 7.3 from [9]:

```
1 var i: Integer = 5;
2 var c : new Uml!Class;
3 //i = "somevalue";
4 if(c.isDefined()) {
5   var i: String;
6   i = "somevalue";
7 }
8 i.println();
```

Listing 7.3: Example of variable shadowing from [9]

In line 5, a variable named *i* (which is previously declared in line 1) is declared again and assigned a String value (line 6) inside the *if* statement (lines 4-7). This variable is only available within the scope of the *if* statement (and any sub-scopes inside the *if* statement). When the program exits the scope of the *if* statement (line 8), the variable is no longer available. Thus, when line 8 is executed, the output will be 5. This behaviour is handled by the *StackFrame*.

The Variable Resolution process captures variables that are declared but not referenced, as well as variables which are referenced but not declared. Consider the example:

```
1 var a: Integer;
2 b = 10;
```

In line 1, a variable named *a* is declared but never used in the program. The variable resolution generates an *EOLWarning* for this situation. In line 2, a reference named *b* is called but there is no variable named b declared. The variable resolution generates an *EOLError* for this situation.

The *VariableResolutionContext* also contains an *IMetamodelManager* which is responsible for managing the models declared in an EOL program. Consider the Example:

```
model University alias u
driver EMF {nsuri = "http://university/1.0"};
```

```
3 var a: Student = University!Student.all().first();
4 a.println();
```

In lines 1 and 2, a *ModelDeclarationStatement* is in place, which declares a model that conforms to the *University* metamodel mentioned in Section 2.1.6 (Figure 2.9). In line 3, a nested *MethodCallExpression* is in place. Consider only the *MethodCallExpression*:

```
University!Student.all()
```

The *target* of the *MethodCallExpression* is *University!Student*, which is in fact a *NameExpression*. The variable resolution is able to distinguish the differences between variable names and model element names. Thus, the *IMetamodelManager* is used to check if the model element name *University!Student* is legal; if not, an appropriate *EOLError* is generated.

7.4. Type Resolution

As previously discussed, the second stage of the static analysis process is to resolve the types of the *Expression*s within an EOL program. As with the variable resolution facility discussed above, the EOL type resolution facility is built by extending the EOLVisitor facility. The structure of the EOL type resolution facility is shown in Figure 7.6. *EOLTypeResolver* is the centralised access point of the facility. *EOLTypeResolver* contains an *EOLVariableResolutionController* (by extending *EOLVisitorController* in the EOLVisitor facility), which in turn contains a number of *EOLElementTypeResolvers* that are implemented by extending the EOLVisitor facility. When the *run(EOLElement eolElement)* method in the *EOLVariableResolver* is executed, the *EOLTypeResolution-Controller* traverses the *eolElement* provided using the *EOLElementTypeResolvers* in correspondence with each *EOLElement* encountered.

The *EOLTypeResolver* contains a *TypeResolutionContext*. The *TypeResolutionContext* acts as a container which provides the states of the different auxiliary facilities of the *EOLTypeResolver* during the analysis. The *TypeResolutionContext* contains the following important facilities:

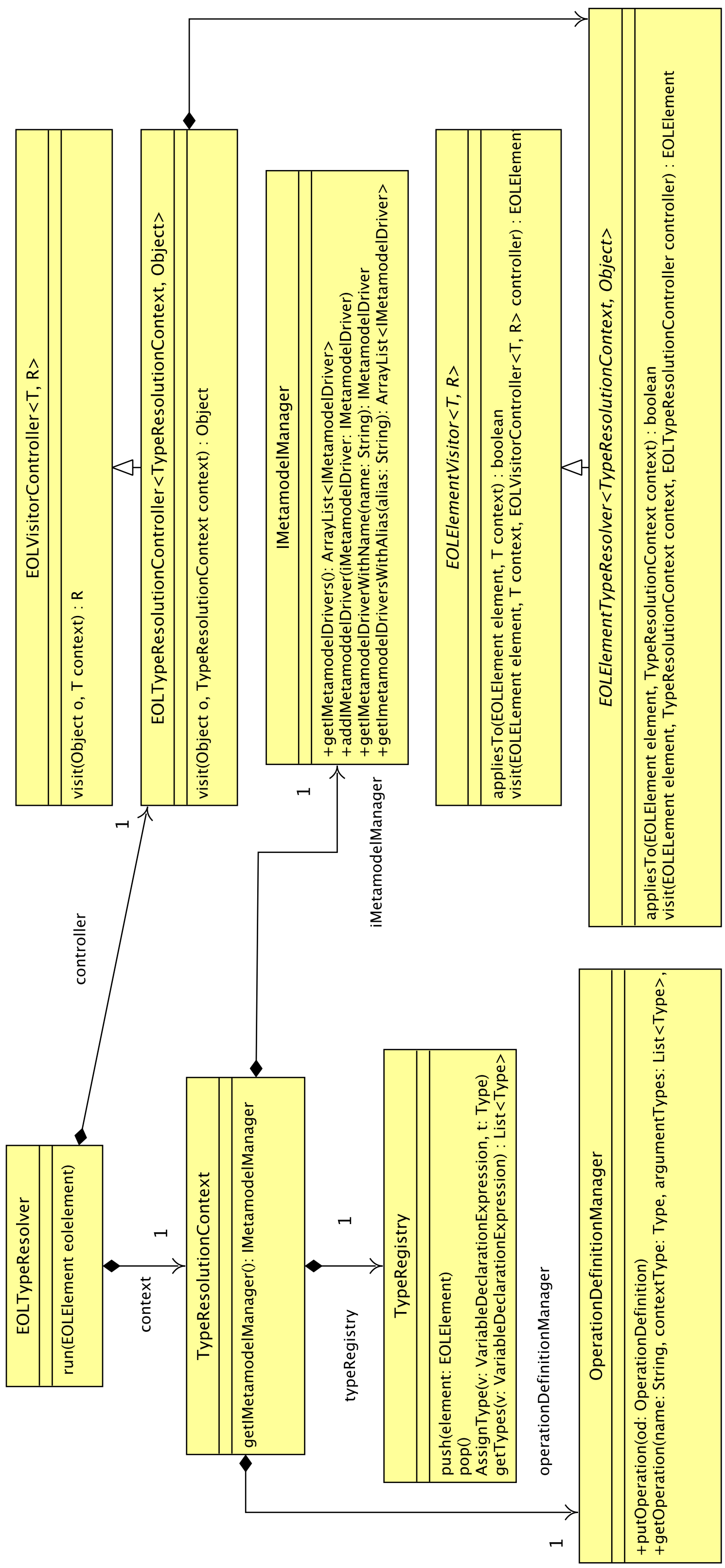


Figure 7.6.: The structure of the Type Resolution Facility

- An *IMetamodelManager*, which is responsible for accessing the metamodels involved in an EOL program (Section 6.3.3);
- An *OperationDefinitionManager*, which acts as a container of the *OperationDefinition*s both from the EOL standard library and from the definitions provided by the user.

7.4.1. Modelling the EOL Standard Library

In [114], the author addresses the rationale behind the modelling of the OCL standard library. In Section 4.2.3, a detailed discussion about modelling the standard library of a model management program is provided. The static analysis framework provides a model of the EOL standard library. The modelling of the EOL standard library is realised by defining the operation signatures of the EOL standard library, leaving the bodies of operations blank. For example, the *isDefined()* operation in the standard library can be specified as:

```
operation Any isDefined(): Boolean {}
```

However, modelling the EOL standard library exhibits some limitations, for the semantics of an operation cannot be accurately captured by existing EOL types. For example, EOL supports type propagation in its built-in *println()* operation. For example, consider the following program:

```
1 var a = "Hello World!";
2 a.println().split(" ").println();
```

In line 1, a variable *a* is assigned the value "Hello World". In line 2, method *println()* is called on *a* and then *split(" ")* is called afterwards. In this instance, the value of *a* is propagated through method call a.println() such that the call to split() is invoked on *a*.

Thus, there is a need to propagate the type of *a* through the call to println(). In [114], the author names such operation semantic as *self-variant* in the sense that the return type of the operation has the same type of context.

Therefore, two *PseudoType*s (Section A.4.8) are added to the EOL metamodel to enable more precise modelling of operations of the EOL standard library. They are

SelfType and *SelfContentType*.

SelfType

The *SelfType* is created to help model the operations where the return type of the operations should be the same as the context type, so that the type-related semantics of the operations is captured at the signature level. In the absence of *SelfType*, the signature of the built-in println operation would read:

```
operation Any println(): Any {}
```

This is reasonable given that *println()* is able to be called upon on any type, and returns whatever type it is called upon. However, this signature does not propagate the type of its context.

With *SelfType*, a new operation signature of *println()* can be written as:

```
operation Any println() : SelfType{}
```

so that when it comes to handling the operation call, it is known that this operation call should return the type of the context.

SelfContentType

SelfContentType is created to model the behaviour of operations that apply on collection types in EOL, and in the sense that their return types should be the same as the content type of their context. Consider the *at()* operation of the EOL standard library, which is used to retrieve an element from an ordered collection. In the absence of *SelfContentType* the signature of the built-in *at()* operation would read:

```
operation Collection at(index: Integer) : Any {}
```

However, the type-related semantics of this operation is lost given that this operation should return the type of the element at the *index*. *SelfContentType* can be used to resolve such trouble:

```
operation Collection at(index: Integer) : SelfContentType {}
```

Thus, when it comes to handling the operation call, there is an obvious indication that the operation call should return the *content type* of *self*, that is the content type of the collection that initiates this operation call.

7.4.2. Type Resolution: the Type Resolution Rule Solving Approach

The implementation of the static analysis adopts a rule-based approach using the lattice theory discussed in Chapter 3.

The first step of the type resolution process involves identifying all the *Expression*s in the EOL metamodel and resolving their types. These expressions are:

- *VariableDeclarationExpression* and *NameExpression*;
- *PrimitiveExpression* and its sub types;
- *MapExpression*, *CollectionExpression* and its sub types;
- *OperatorExpression* and its sub types in the EOL metamodel.
- *FeatureCallExpression* and its sub types. In particular, *PropertyCallExpression*, *MethodCallExpression* and *FOLMethodCallExpression*.
- *KeyValueExpression*; and
- *NewExpression*.

With the types of *Expression*s identified, the next step of the type resolution process involves checking for the type-correctness of the *Statement*s that encapsulate *Expression*s, which are:

- *AssignmentStatement*;
- *IfStatement*;
- *ForStatement*;
- *WhileStatement*;
- *SwitchStatement*

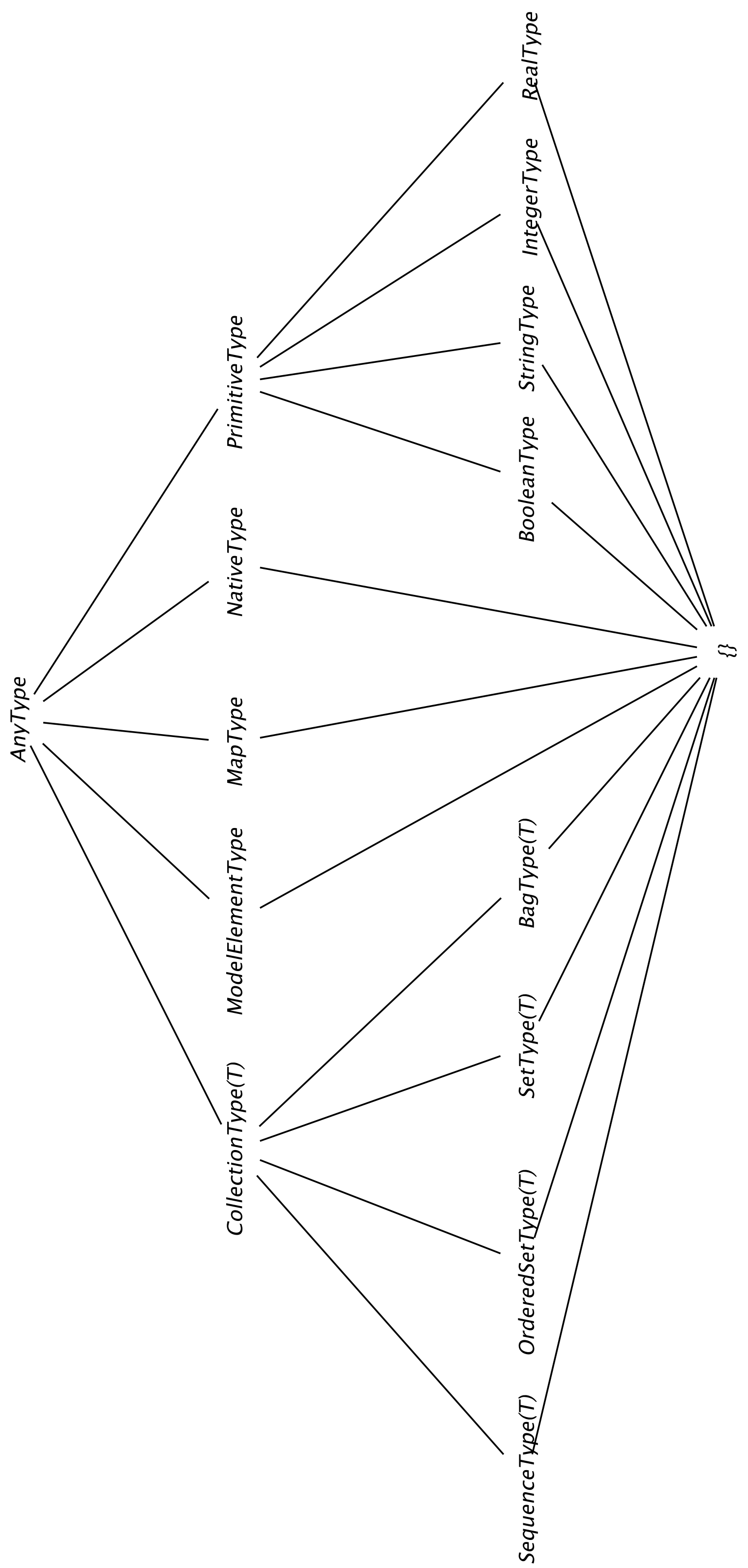


Figure 7.7.: The type lattice of EOL

The Type Set

To perform type resolution, it is necessary to identify the set of *Type*s of EOL. The set of all the *Type*s (denoted by T hereafter) is defined as:

$$\begin{aligned}
T = \{&AnyType,\\
&PrimitiveType\\
&RealType, IntegerType, StringType, BooleanType,\\
&CollectionType(T),\\
&SetType(T), OrderedSetType(T), SequenceType(T), BagType(T),\\
&ModelElementType, NativeType, MapType\}
\end{aligned}$$

The type set is a one-to-one mapping to the *Type*s in the EOL metamodel. It is to be noted that unlimited number of types can be derived from T due to the fact that *CollectionType*s can have nested *Type*s as their content types in an arbitrary number of levels.

The types in T form a lattice (poset) where the top of the lattice is *AnyType* and the bottom of the lattice is $\emptyset$ (although in practice all *Expression*s have an associated *Type*). The lattice is shown in Figure 7.7.

With each of the *Expression*s and *Statement*s identified previously, a set of type resolution rules are associated. The type resolution rules can be considered as an (abstract) interpretation of the semantics of the *Expression*s and *Statement*s. The type resolution rules are encompassed in *EOLElementTypeResolver*s, which extend the *EOLElementVisitor*s in the EOLVisitor facility. The EOL type resolution facility achieves type resolution by solving the type resolution rules (which is, in a sense, to compute the *fixed point* of the type) of an *EOLModule* in a bottom-up manner.

For each *Statement*, the *Expression*s contained within it are type-resolved (visited) by their corresponding *EOLElementTypeResolver*; and for each *Expression*, the *Expression*s contained within it are also type-resolved by their corresponding *EOLElementTypeResolver*s. Hence, solving the type resolution rules in this bottom-up manner resolves the types of all the *Expressions* within an EOL program.

Type resolution rules for VariableDeclarationExpression

Let V denote a *VariableDeclarationExpression*, $[V]$ denote the type of V. In EOL, a variable declaration has an optional type declaration, for example:

```
1 var aVar;
2 var anInt: Integer;
```

In line 1, an untyped variable is declared and, as such, the static analyser assumes that the type of the variable is *Any*. In line 2, a typed variable is declared. Let V^T denote the declared type of V. Therefore, the type resolution rules for *VariableDeclarationExpression* are defined as follows:

$$V^T = \emptyset \Rightarrow [V] = Any;$$
$$V^T = t \in T \Rightarrow [V] = t;$$

Type resolution rules for NameExpression

Let N denote a *NameExpression*, $[N]$ denote the type of N. Let $N \to V$ denote the variable declaration of N, $[N \to V]$ denote the type of $N \to V$. The type resolution rules for *NameExpression* are defined as follows:

$$(N \to V \neq \emptyset) \wedge ([N \to V] = t \in T) \Rightarrow [N] = t;$$

Sometimes, a *NameExpression* can be used to refer to types in either the EOL type system or in the metamodel of one of the models processed by the program. Let N_{text} denote the actual String value of an *NameExpression* and MM denote the set of model element types defined in the metamodel(s) involved in an EOL program. Thus, if a *NameExpression* does not have a corresponding variable declaration and its String value is a type defined in the EOL type system, the type of the *NameExpression* is resolved to whatever type N_{text} refers to:

$$(N \rightarrow V = \emptyset) \wedge (N_{text} \in T) \Rightarrow [N] = [N_{text}];$$

If a *NameExpression* does not have a corresponding variable declaration and its String value is a type defined in the metamodel(s) involved, the type of the *NameExpression* is resolved to *ModelElementType* (with its corresponding properties calculated):

$$(N \rightarrow V = \emptyset) \wedge (N_{text} \in MM) \Rightarrow [N] = ModelElementType;$$

It is worth noting that not all *NameExpression*s within an EOL program are visited (for example, the "property" of a *PropertyCallExpression*) - the *EOLElementTypeResolver* for each individual *EOLElement* determines which property for a given *EOLElement* should be visited.

Type resolution rules for PrimitiveExpressions

Let BE denote a *BooleanExpression*, $[BE]$ denote the type of BE. Let IE denote an *IntegerExpression*, $[IE]$ denote the type of IE. Let RE denote a *RealExpression*, $[RE]$ denote the type of RE. And finally, let SE denote a *StringExpression* and $[SE]$ denote the type of SE. Thus, the type resolution rules are defined as follows:

$$[BE] = Boolean;$$
$$[IE] = Integer;$$
$$[RE] = Real;$$
$$[SE] = String;$$

Type resolution rules for CollectionExpressions

Let $SetE$, $[SetE]$, $OSetE$, $[OSetE]$, $SeqE$, $[SeqE]$, $BagE$, $[BagE]$ denote a *SetExpression* and its type, an *OrderedSetExpression* and its type, a *SequenceExpression* and its type, and finally a *BagExpression* and its type. Let $[CollectionExpression^c]$ denote the content type of an *CollectionExpression*. Therefore, the type resolution rules are defined as follows:

$$([SetE^c] \neq \emptyset) \wedge ([SetE^c] \in T) \Rightarrow [SetE] = Set([SetE^c]);$$
$$[SetE^c] = \emptyset \Rightarrow [SetE] = Set(Any);$$
$$([OSetE^c] \neq \emptyset) \wedge ([OSetE^c] \in T) \Rightarrow [OSetE] = OrdredSet([OSetE^c]);$$
$$[OSetE^c] = \emptyset \Rightarrow [OSetE] = OrderedSet(Any);$$
$$([SeqE^c] \neq \emptyset) \wedge [SeqE^c] \in T \Rightarrow [SeqE] = Sequence([SeqE^c]);$$
$$[SeqE^c] = \emptyset \Rightarrow [SeqE] = Sequence(Any);$$
$$([BagE^c] \neq \emptyset) \wedge ([BagE^c] \in T) \Rightarrow [BagE] = Bag([BagE^c]);$$
$$[BagE^c] = \emptyset \Rightarrow [BagE] = Bag(Any);$$

Type resolution rules for Unary Operators

Let E denote the expression involved in a unary operator and $[E]$ denote the type of E.

Negative Operator ($-$). The $-$ operator is used to negate a Real or an Integer number. Therefore, the type resolution rules of the $-$ operator are:

$$[E] = Real \Rightarrow -[E] = Real;$$
$$[E] = Integer \Rightarrow -[E] = Integer;$$

Any other types apart from *Real* and *Integer* will result in a warning being reported if $[E]$ is *Any*, and an error being reported if E is of any other type. In addition:

$$[E] = Any \Rightarrow -[E] = Real;$$
$$[E] \in T \setminus \{Any, Real, Integer\} \Rightarrow -[E] = Real;$$

The type of $-[E]$ is bounded by *Real* to prevent type errors from being propagated further into the program.

Not Operator (*not*). The not operator *not* is used to negate a boolean value.

Therefore, the type resolution rules of the *not* operator are:

$$[E] = Boolean \Rightarrow not[E] = Boolean$$

Any other types apart from *Boolean* will result in a warning being reported if $[E]$ is *Any*, and an error being reported if $[E]$ is of any other type:

$$[E] \in T \setminus \{Boolean\} \Rightarrow not[E] = Boolean$$

Type resolution rules for Binary Operators

Let L and R denote the first and the second operands involved in a binary operator, and let $[L]$ and $[R]$ denote the types of L and R.

Plus Operator (+). The plus operator + is used to perform the summation of two values of type Real or Integer, String concatenations and collection aggregations. The type resolution rules of the + operator are defined as follows:

$$([L] = Integer) \wedge ([R] = Integer) \Rightarrow [L + R] = Integer;$$
$$([L] = Integer) \wedge ([R] = Real) \Rightarrow [L + R] = Real;$$
$$([L] = Real) \wedge ([R] = Real) \Rightarrow [L + R] = Real;$$

If $[L]$ and $[R]$ are both *String*, the operation is considered to be string concatenation, therefore:

$$([L] = String) \wedge ([R] = String) \Rightarrow [L + R] = String;$$

If $[L]$ is *Collection* and $[R]$ is *Collection*, the operation is considered to be collection aggregation, where the contents of the two collections are aggregated, in this case $[L+R]$ is the type of $[L]$. For the discussion, let $[L^c]$ and $[R^c]$ denote the content types of $[L]$ and $[R]$. The type of the plus operator expression is whatever the type it is for L. For the content type, If $[L^c]$ and $[R^c]$ are the same, then the content type of the expression is $[L^c]$. Otherwise, the Least Common Type (LCT) of $[L^c]$ and $[R^c]$ are computed. If there is a Least Common Type, then the content type is the computed Least Common Type, otherwise the content type is *Any*. Let the symbol $lct()$ denote the function that calculates the *Least Common Type* and LCT denote the LCT returned by $lct()$:

$$
\begin{aligned}
([L] = Collection^1) \wedge ([R] = Collection^2) \wedge ([L^c] = [R^c]) \\
\Rightarrow [L + R] = Collection^1(L^c); \\
([L] = Collection^1) \wedge ([R] = Collection^2) \wedge ([L^c] \neq [R^c]) \\
\Rightarrow [L + R] = Collection^1(lct([L^c], [R^c])) \neq null\,?\,LCT\ :\ Any);
\end{aligned}
$$

If $[L]$ and $[R]$ are not *Any*, *String* or *Collection*, and they do not relate in the type system, a warning is generated, but the type of $[L + R]$ is bounded by *String*.

$$
\begin{aligned}
([L] \in T \setminus \{Any, String, Collection\}) \wedge ([R] \in T \setminus \{Any, String, Collection\}) \\
\wedge([L] \notin [R]) \wedge ([L] \neq [R]) \Rightarrow [L + R] = String;
\end{aligned}
$$

Duplicated type resolution rules are omitted in this discussion (switching types of L and R).

Minus Operator ($-$), Multiply Operator ($*$) and Divide Operator ($/$). The minus operator $-$ is used to perform subtraction between two values of type Real or Integer. Because these operators share the same type resolution rules, they are discussed together. The operators $-$, $*$ and $/$ are represented as *op* in the rules. The type resolution rules of the $-$ operator are defined as follows:

$$
\begin{aligned}
([L] = Integer) \wedge ([R] = Integer) \Rightarrow [L\,op\,R] = Integer \\
([L] = Integer) \wedge ([R] = Real) \Rightarrow [L\,op\,R] = Real \\
([L] = Real) \wedge ([R] = Real) \Rightarrow [L\,op\,R] = Real
\end{aligned}
$$

The order of $[L]$ and $[R]$ does not influence the analysis, so duplicates are omitted.

$[L\,op\,R]$ is lifted to *Real* if the highest type between L and R is *Real*.

$$\begin{aligned}([L] = Any) \wedge ([R] \in \{Real, Integer\}) \Rightarrow [L\,op\,R] = Real\\ ([L] \in T \setminus \{Any, Real, Integer\}) \wedge ([R] \in T \setminus \{Any, Real, Integer\})\\ \Rightarrow [L\,op\,R] = Real\end{aligned}$$

If $[L]$ is *Any* and $[R]$ is either *Real* or *Integer*, then a warning is generated and $[L+R]$ is *Real*. If $[L]$ and $[R]$ are any other types, then an error is generated and $[L+R]$ is bounded by *Real*.

The multiply operator $(*)$ and the divide operator $(/)$ perform multiplication and division on *Integer*/*Real* values. Their type resolution rules are the same as for the minus operator $(-)$.

Equals Operator (=) and Not Equals Operator (<>). The equals operator (=) and the not equals operator (<>) are used to compare if L equals to (or not equals to) R. The = and <> operators are represented as *op* in the rules.The type resolution rules of the equals operator are defined as follows:

If $[L]$ and $[R]$ are equal or related with regard to inheritance, $[L\,op\,R]$ is of type Boolean:

$$([L] = [R]) \vee ([L] \in [R]) \vee ([R] \in [L]) \Rightarrow [L\,op\,R] = Boolean;$$

If $[L]$ and $[R]$ are not equal and are not related with regard to type inheritance, a warning is reported and $[L\,op\,R]$ is of type Boolean:

$$([L] \neq [R]) \wedge ([L] \notin [R]) \wedge ([R] \notin [L]) \Rightarrow [L\,op\,R] = Boolean;$$

Greater Than Operator (>), Greater Than Or Equal To Operator (>=), Less Than Operator (<), Less Than Or Equal To Operator (<=). These operators are used to compare *Real*/*Integer* values. The operators >, >=, <, and <= are represented as *op* in the rules.Their type resolution rules are defined as follows:

If $[L]$ and $[R]$ are *Integer* or *Real*, $[L + R]$ is *Boolean*.

$$(([L] \in \{Integer, Real\}) \wedge ([R] \in \{Integer, Real\})) \Rightarrow [L\, op\, R] = Boolean;$$

If $[L]$ is *Any*, and $[R]$ is *Integer* or *Real*, a warning is reported on L and $[L\, op\, R]$ is *Boolean*.

$$(([L] = Any) \wedge ([R] \in \{Integer, Real\})) \Rightarrow [L\, op\, R] = Boolean;$$

If $[L]$ (or) $[R]$ are not *Any*, *Integer* or *Real*, an error is reported, but $[L\, op\, R]$ is *Boolean*.

$$(([L] \in T \setminus \{Any, Integer, Real\}) \wedge ([R] \in \{Integer, Real\})) \\ \Rightarrow [L\, op\, R] = Boolean;$$

Duplicated type resolution rules (switching types of $[L]$ and $[R]$) are omitted.

And Operator (*and*), Or Operator (*or*), Exclusive Or Operator (*xor* and Implies Operator (*implies*)). These operators are logical operators which apply to *Boolean* values. The operators *and*, *or*, *xor*, and *implies* are represented as *op* in the rules. Their type resolution rules are defined as follows:

If $[L]$ and $[R]$ are both *Boolean*, then $[L\, op\, R]$ is *Boolean*.

$$(([L] = Boolean) \wedge ([R] = Boolean)) \Rightarrow [L\, op\, R] = Boolean;$$

If $[L]$ is *Any* and $[R]$ is *Boolean*, a warning is reported on $[L]$, then $[L\, op\, R]$ is *Boolean*.

$$(([L] = Any) \wedge ([R] = Boolean)) \Rightarrow [L\, op\, R] = Boolean;$$

If $[L]$ is not *Any* or *Boolean*, an error is reported on $[L]$, then $[L\, op\, R]$ is *Boolean*.

$$(([L] \in T \setminus \{Any, Boolean\}) \wedge ([R] = Boolean)) \Rightarrow [L\, op\, R] = Boolean;$$

Duplicated type resolution rules (switching types of $[L]$ and $[R]$) are omitted.

Type resolution rules for FeatureCallExpressions

In this section, the type resolution rules for *FeatureCallExpressions* are provided, which include *MethodCallExpression*, *PropertyCallExpression* and *FOLMethodCallExpression*.

MethodCallExpression in the EOL metamodel is used to represent a method call. To analyse the type of a *MethodCallExpression*, the first step is to acquire the respective *OperationDefinition*. Let *od* denote the *OperationDefinition* and *OD* denote the set of all available *OperationDefinitions* (both in the standard library and defined by the user within the EOL program under analysis). The *OperationDefinitionManager* (Figure 7.8) is responsible for identifying the appropriate operation given the *name*, the *context* type, the list of *parameter* types, and a boolean value to denote if the method call is initiated by the $\rightarrow$ operator or the . operator. The static analyser then looks at appropriate *OperationDefinitionContainers* for the operation. To compare the distance between types, the Dijkstra's Shortest Path algorithm [130] is used, which is not discussed in detail. Let MC and $[MC]$ denote a *MethodCallExpression* and its type, and $[od_{return}]$ denote the return type of OD. If *od* is found by the *OperationDefinitionManager*, the type of the method call should be the return type of the *od*. The return type of *od* is handled by the type resolution facility if it is a *PseudoType* (Section 7.4.1).

$$\exists od \in OD \Rightarrow [MC] = [od_{return}]$$

If no *OperationDefinition* is found, the type of the expression is *Any*, and an error is raised.

$$\neg\exists od \in OD \Rightarrow [MC] = Any$$

PropertyCallExpression in the EOL metamodel is used for feature navigation. For this discussion, let PC denote a *PropertyCallExpression*, PC_{tar} denote the target of PC, $PC_{property}$ denote the property to call on PC, whereas $[PC]$ and $[PC_{tar}]$ denote the types of PC and PC_{tar}. The type resolution rules for *PropertyCallExpression* are

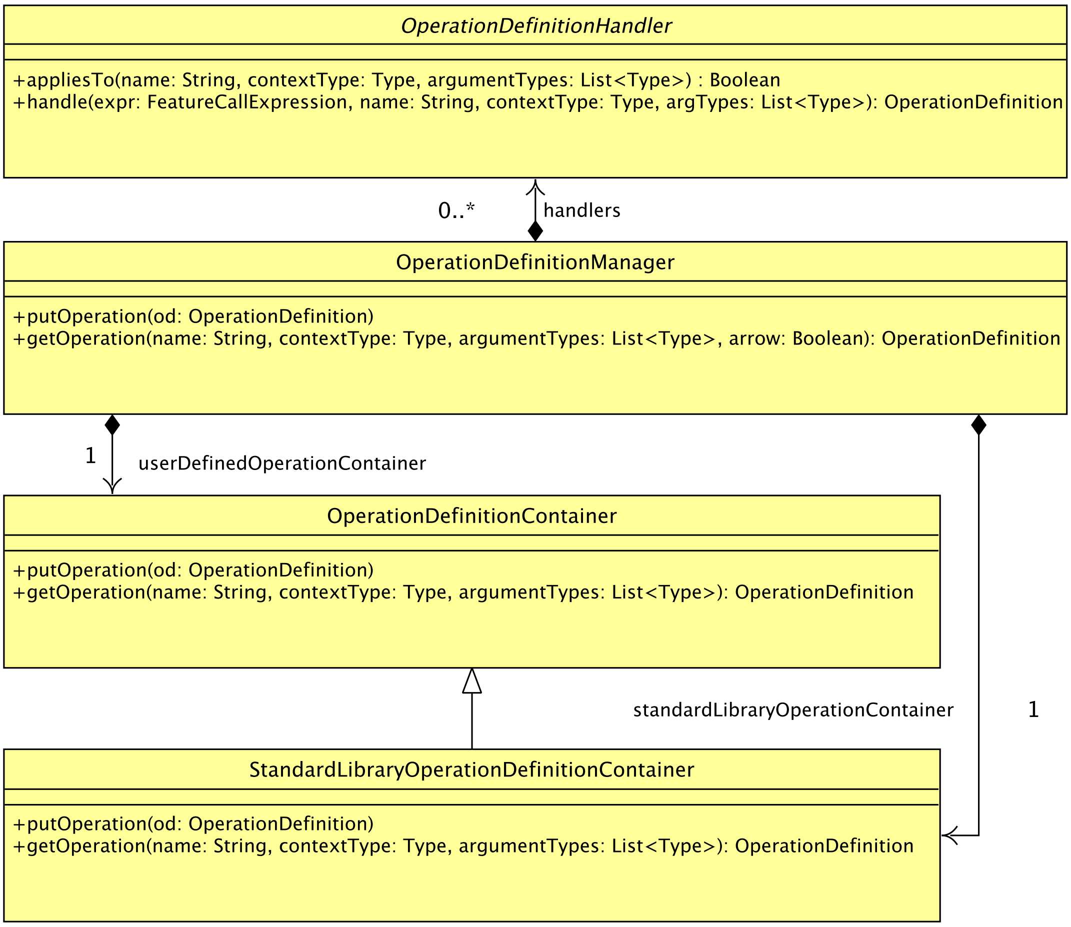


Figure 7.8.: Type structure OperationDefinitionManager

defined as follows[2]:

$$([PC_{tar}] = ModelElement) \wedge$$
$$(\exists\, feature \in ([PC_{tar}].features) : feature = PC_{property})$$
$$\Rightarrow [PC] = [feature]$$

If the target of the *PropertyCallExpression* is a *ModelElementType*, its features are inspected (via the ESAMC), and if a feature specified by the *property* of the *PropertyCallExpression* is found, it is returned by the ESAMC. The type of the *PropertyCallEx-*

[2]Note: the feature is retrieved by ESAMC

pression is calculated based on the returned feature, such that:

- if the upper bound of the feature is -1, and the feature is *ordered* and *unique*, an *OrderedSetType* is created. If the feature is *ordered* but not *unique*, a *SequenceType* is created. If the feature is not *ordered* but *unique*, a *SetType* is created. If the feature is not *ordered* but *unique*, a *BagType* is created. If the feature is of type *EDataType*, then the corresponding *PrimitiveType* is created as the *content type* of the collection type. If the feature is of type *EClass*, then the corresponding *ModelElementType* is created as the *content type* of the collection type;
- if the upper bound of the feature is 1, and the feature is of type *EDataType*, then the corresponding *PrimitiveType* is created. If the feature is of type *EClass*, then the corresponding *ModelElementType* is created.

In other cases, the target of the *PropertyCallExpression* is a collection of model elements. Consider the example:

```
1 var persons: Sequence(Person) = Person.allInstances();
2 var names: Sequence(String) = persons.first_name;
3 names.println();
```

In line 1, the collection of all *Person*s is extracted and in line 2, a property call is in place, which returns a *Sequence* of *String*s because it retrieves the property *first_name* on all *Person*s.

For this situation, the type resolution rule is defined as follows. Let $[PC_{tarCon}]$ denote the target content type. If the target type is *Collection*[3]:

$$\begin{aligned}
([PC_{tar}] = Collection)\wedge \\
([PC_{tarCon}] = ModelElement)\wedge \\
(\exists\, feature \in ([PC_{tarCon}].features) : feature = PC_{property}) \\
\Rightarrow [PC] = [PC_{tar}]([feature])
\end{aligned}$$

[3] Note: the feature is retrieved by ESAMC

The calculation of the type of *feature* follows the rules previously discussed.

In cases where the target type of the *PropertyCallExpression* is *Any*, a warning is reported, and the type of *PropertyCallExpression* is set to *Any*. If the target type of the *PropertyCallExpression* is neither *Any* nor *ModelElementType*, an error is reported and the type of *PropertyCallExpression* is set to *Any*.

$$[PC_{tar}] = Any \Rightarrow [PC] = Any;$$
$$[PC_{tar}] \in T \setminus \{Any, ModelElement\} \Rightarrow [PC] = Any$$

Finally, for extended properties (discussed in Section A.2.11), the type of the *PropertyCallExpression* is set to *Any*.

FOLMethodCallExpression in the EOL metamodel represents a first-order-logic method call. For the purpose of the discussion, let FOL and $[FOL]$ denote a *FOLMethodCallExpression* and its type, FOL_{tar} and $[FOL_{tar}]$ denote the target of FOL and its type, FOL_{iter} and $[FOL_{iter}]$ denote the iterator of FOL and its type, and FOL_{con} and $[FOL_{con}]$ denote the condition of FOL and its type. Let *CollectionType* denote the set of all the *CollectionType*s in EOL. Once again, the first-order-logic operations need to be matched by the OperationDefinitionManager. Let *od* denote the matched operation and OD denote the set of all operations. Let $[od_{return}]$ denote the return type of *od*.

Since first-order-logic operations in EOL can have non-trivial type semantics, a number of *OperationDefinitionHandler*s (Figure 7.8) is created. The type resolution rules will be discussed for each first-order-logic operation. All first-order-logic operations apply only to *Collection* values; hence, if the target of a first-order logic is not of *Collection* type, the *OperationDefinitionManager* is not able to locate the first-order-logic operation in the standard library.

aggregate(). The *aggregate()* operation returns a *Map* containing key-value pairs produced by evaluating the key and value expressions on each item of the collection that is of the type specified in the iterator. The type resolution rules for *aggregate()* are:

If an operation is found by *OperationDefinitionManager*, the type of FOL is *Map*

$$([FOL_{tar}] \in \mathit{CollectionType}) \wedge (\exists od \in OD) \Rightarrow [FOL] = Map$$

If no operation is found by the OperationDefinitionManager (normally due to $[FOL_{tar}]$ being not a *CollectionType*), the type of FOL is set to *Any* and an error is reported.

$$([FOL_{tar}] \notin \mathit{CollectionType}) \wedge (\neg\exists od \in OD) \Rightarrow [FOL] = \mathit{Any}$$

closure(). The *closure()* operation returns a collection containing the results of evaluating the transitive closure of the results provided by the expression. The type resolution rules for *closure()* are defined as follows:

If the target type is a collection, the type of FOL is the same as the target type, but the content type of FOL is the type of the condition of FOL ($[FOL_{con}]$):

$$([FOL_{tar}] \in \mathit{CollectionType}) \wedge (\exists od \in OD \Rightarrow [FOL]) \Rightarrow [FOL_{tar}]([FOL_{con}])$$

If no operation is found by *OperationDefinitionManager*, the type of FOL is set to *Any* and an error is reported.

$$\neg\exists od \in OD \Rightarrow [FOL] = \mathit{Any}$$

collect(). The *collect()* operation returns a collection containing the results of evaluating the expression (specified in the condition FOL_{con}) on each item of the collection that is of the specified type. The rules for *collect()* are defined as follows:

If the target type is *Collection*, the type of FOL is the same as its target type, but the content type of FOL is the type of the condition of FOL ($[FOL_{con}]$);

$$([FOL_{tar}] \in \mathit{CollectionType}) \wedge (\exists od \in OD \Rightarrow [FOL]) \Rightarrow [FOL_{tar}]([FOL_{con}])$$

If the target type is not *Collection*, the type of FOL is set to *Any* and an error is reported.

$$([FOL_{tar}] \in CollectionType) \wedge (\neg\exists od \in OD) \Rightarrow [FOL] = Any$$

exists(). The *exists()* operation returns true if there exists at least one item in the collection that satisfies the condition. The type resolution rules for *exists()* are defined as follows:

If the target type is *Collection*, the type of FOL is *Boolean*

$$([FOL_{tar}] \in CollectionType) \wedge (\exists od \in OD) \Rightarrow [FOL] = Boolean$$

If the target type is not *Collection*, the type of FOL is set to *Any* and an error is reported.

$$([FOL_{tar}] \in CollectionType) \wedge (\neg\exists od \in OD) \Rightarrow [FOL] = Any$$

forAll(). The *forAll()* operation returns true if all items in the collection satisfy the condition. The type resolution rules for *forAll()* are defined as follows:

If the target type is *Collection*, the type of FOL is *Boolean*

$$([FOL_{tar}] \in CollectionType) \wedge (\exists od \in OD) \Rightarrow [FOL] = Boolean$$

If the target type is not *Collection*, the type of FOL is set to *Any* and an error is reported.

$$([FOL_{tar}] \in CollectionType) \wedge (\neg\exists od \in OD) \Rightarrow [FOL] = Any$$

one(). The *one()* operation returns true if there exists *exactly* one item in the collection that satisfies the condition. The type resolution rules for *one()* are defined as follows:

If the target type is *Collection*, the type of FOL is *Boolean*

$$([FOL_{tar}] \in CollectionType) \wedge (\exists od \in OD) \Rightarrow [FOL] = Boolean$$

If the target type is not *Collection*, the type of FOL is set to *Any* and an error is reported.

$$([FOL_{tar}] \in CollectionType) \wedge (\neg\exists od \in OD) \Rightarrow [FOL] = Any$$

reject(). The *reject()* operation returns a sub-collection containing only items in the (target) collection that do not satisfy the condition. The type resolution rules for *reject()* are defined as follows:

If the target type is *Collection*, the type of FOL is the same as its target, but the content type of $[FOL]$ should be the type of the iterator:

$$([FOL_{tar}] \in CollectionType) \wedge (\exists od \in OD) \Rightarrow [FOL] = [FOL_{tar}]([FOL_{iter}])$$

If the target type is not *Collection*, the type of FOL is set to *Any* and an error is reported.

$$([FOL_{tar}] \in CollectionType) \wedge (\neg\exists od \in OD) \Rightarrow [FOL] = Any$$

select(). The *select()* operation returns a sub-collection containing only items in the (target) collection that satisfy the condition. The type resolution rules for *select()* are:

If the target type is *Collection*, the type of FOL is the same as its target, but the content type of $[FOL]$ should be the type of the iterator:

$$([FOL_{tar}] \in CollectionType) \wedge (\exists od \in OD) \Rightarrow [FOL] = [FOL_{tar}]([FOL_{iter}])$$

If the target type is not *Collection*, the type of FOL is set to *Any* and an error is reported.

$$([FOL_{tar}] \in CollectionType) \land (\neg\exists od \in OD) \Rightarrow [FOL] = Any$$

selectOne(). The *selectOne()* operation returns an item in the (target) collection that satisfies the condition. The type resolution rules for *selectOne()* are defined as follows:

If the target type is *Collection*, the type of FOL is the same as the type of its iterator:

$$([FOL_{tar}] \in CollectionType) \land (\exists od \in OD) \Rightarrow [FOL] = [FOL_{iter}]$$

If the target type is not *Collection*, the type of FOL is set to *Any* and an error is reported.

$$([FOL_{tar}] \in CollectionType) \land (\neg\exists od \in OD) \Rightarrow [FOL] = Any$$

sortBy(). The *sortBy()* operation returns a copy of the collection which is sorted by evaluating the expression specified in the condition. The type resolution rules for *sortBy()* are defined as follows:

If the target type is *Collection*, the type of FOL is the same as its target:

$$([FOL_{tar}] \in CollectionType) \land (\exists od \in OD) \Rightarrow [FOL] = [FOL_{tar}]$$

If the target type is not *Collection*, the type of FOL is set to *Any* and an error is reported.

$$([FOL_{tar}] \in CollectionType) \land (\neg\exists od \in OD) \Rightarrow [FOL] = Any$$

KeyValueExpression. The key value expression has two types: the key type and the value type. Let $KeyVal$ and $[KeyVal]$ denote the *KeyValueExpression* and its type,

K and $[K]$ denote the key and its type, and V and $[V]$ denote the value and its type.

$$[K] \in T, [V] \in T \Rightarrow [KeyVal] = \{[K] \rightarrow [V]\}$$

NewExpression. The new expression is used to create an instance of a type. Let NE and $[NE]$ denote a new expression and its type, and $[NE_{target}]$ denote the target type to be instantiated. Thus, the type resolution rule of *NewExpression* is:

$$(\exists t \in T : [NE_{target}] = t) \Rightarrow [NE] = t$$

An error is reported if the target type to be created cannot be found in the EOL type system (including model element types, which are managed by ESAMC), or if the target type is not instantiable (abstract/interface);

AssignmentStatement. The assignment statement encapsulates two *Expression*s, the left hand side expression and the right hand side expression. The left hand side expression should be only of type *PropertyCallExpression*, *NameExpression* or *VariableDeclarationExpression*. For the purpose of the discussion, let L and $[L]$ denote the left hand side expression and its type, and R and $[R]$ denote the right hand side expression and its type:

$$[R] \subseteq [L]$$

An error is reported if $[L]$ and $[R]$ are incompatible.

IfStatement. The if statement encapsulates potentially three blocks of statements, the if-body, a number of optional else-if-bodies and an optional else-body. Let if_{con} and $[if_{con}]$ denote the if condition and its type; $elseIf_{con}$ and $[elseIf_{con}]$ denote an else-if condition and their type; $ELSEIF_{con}$ denote the set of all else-if conditions. Thus, the type resolution rule for *IfStatement* is:

$$([if_{con}] \in \{Boolean, Any\}) \wedge$$
$$(\forall elseIf_{con} \in ELSEIF_{cond} : [elseIf_{c}on] \in \{Boolean, Any\})$$

Warnings are reported for conditions that are of type *Any*, and errors are reported for all conditions that are not of types *Boolean* or *Any*.

ForStatement. The for statement encapsulates an iterator and a condition (domain, discussed in Section A.3.3), where the condition should be a *Collection* and the type of the iterator should be the content type of the type of the condition. Let *iter* and [*iter*] denote the iterator and its type, *cond* and [*cond*] denote the condition and its type, $[cond_{content}]$ denote the content type of the condition, and *CollectionType* denote the set of all *Collection* types:

$$([cond] \in \{CollectionType \cup \{Any\}\}) \wedge ([iter] \in \{Any, [cond_{content}]\})$$

A warning is reported if the type of the condition is *Any*, and an error is reported if the type of the condition is neither *Any* or *Collection*. A warning is reported if the type of the iterator is *Any*, and an error is reported if the type of the iterator is different from the content type of the condition type.

WhileStatement. The while statement encapsulates a condition, where the type of the condition should be *Boolean*. Let *cond* and [*cond*] denote the condition and its type:

$$([cond] \in \{Boolean, Any\})$$

A warning is reported if the type of the condition is *Any*, and an error is reported if the type of the condition is neither *Boolean* nor *Any*.

SwitchStatement. The switch statement encapsulates a switch expression, a number of cases and a default case. For each case, there is a case expression which is used to compare with the expression in the switch. Let *expr* and [*expr*] denote the switch expression and its type, *case* and [*case*] denote a case expression and its type, and $CASE$ denote the set of all cases in the switch statement:

$$\forall\, case \in CASE : [case] \in \{[expr], Any\}$$

Warnings are reported if the switch expression, or the case expressions, are *Any*, and errors will be reported if types are not compatible.

7.5. Warnings/Errors Detectable by the EOL Static Analyser

The warnings and errors detectable by the EOL static analyser are listed below.

7.5.1. Warnings/Errors Detectable by the EOL Variable Resolver

- Warning: a warning is issued if a defined variable v is not referenced anywhere in the program;
- Warning: a warning is issued if an attempt to define a variable with a name that has been used previously to define another variable;
- Error: an error is issued if within a model declaration statement md, a name n has been used as the name of another model declaration statement;
- Error: an error is issued if a name n cannot be resolved to a previously defined variable;

7.5.2. Warnings/Errors Detectable by the EOL Type Resolver

- Error: an error is issued for attempts to instantiate non-instantiable types;
- Error: an error is issued if model types are used without supporting model declaration statements;
- Error: an error is issued if the driver of a model declaration statement is not supported;
- Error: an error is issued if a name n cannot be resolved to a type (model element type/EOL type);
- Error: an error is issued if a metamodel cannot be found using the identifier provided in a model declaration statement;

- Error: an error is issued if the type of an expression contained in a logical operator (*and*, *or*, *xor*, *not* and *implies*) is not Boolean or Any;

- Error: an error is issued if the type of any expression contained in a comparison operator ($>$, $\geq$, $<$, $\leq$) is not Integer, Real or Any;

- Error: an error is issued if the type of any expression contained in $-$, $*$ and $/$ operators is not Integer or Real;

- Warning: a warning is issued if the types of the expressions contained in the $+$ operator are not compatible;

- Warning: a warning is issued if the targets of the operation call to all operations (EXCEPT *asBoolean()*, *asInteger()*, *asReal()*, *asString()*, *asOrderedSet()*, *asSequence()*, *asSet()*, *asBag()*, *err()*, *errln()*, *format()*, *hasProperty()*, *ifUndefined()*, *isDefined()*, *isKindOf()*, *isTypeOf()*, *isUndefined()*, *owningModel()*, *print()*, *println()*) are of type Any;

- Error: an error is issued if the targets of the operation calls to *charAt()*, *concat()*, *endsWith()*, *firstToLowerCase()*, *firstToUpperCase()*, *isInteger()*, *isReal()*, *isSubstringOf()*, *length()*, *isReal()*, *isSubstringOf()*, *length()*, *matches()*, *pad()*, *replace()*, *split()*, *startsWith()*, *subString()*, *toCharSequence()*, *toLowerCase()*, *toUpperCase()*, *trim()* are NOT of type Any or String;

- Error: an error is issued if the targets of the operation calls to *abs()*, *ceiling()*, *floor()*, *log()*, *log10()*, *max()*, *min()*, *pow()*, *round()* are NOT of type Any, Real or Integer;

- Error: an error is issued if the targets of the operation calls to *iota()*, *to()*, *toBinary()* and *toHex()* are NOT of type Any or Integer;

- Error: an error is issued if the targets of the operation calls to *add()*, *addAll()*, *clear()*, *clone()*, *concat()*, *count()*, *excludes()*, *excludesAll()*, *excluding()*, *excludingAll()*, *flatten()*, *includes()*, *includesAll()*, *including()*, *includingAll()*, *isEmpty()*, *product()*, *random()*, *remove()*, *removeAll*, *size()*, *sum()*, *at()*, *first()*, *second()*, *third()*, *fourth()*, *last()*, *indexOf()*, *invert()*, *removeAt()* are NOT of type Any or Collection;

- Error: an error is issued if the targets of the operation calls to first order logic operations calls (*aggregate()*, *closure()*, *collect()*, *exists()*, *forAll()*, *one()*, *reject()*, *select()*, *selectOne()*, *sortBy()*) are NOT of type Any or Collection;
- Error: an error is issued if the targets of the operation calls to *all()*, *allInstances()*, *allOfKind()*, *allOfType()*, *createInstance()*, *isInstantiable()* are NOT of type ModelElementType
- Warning: a warning is issued when collection expression *ce* does not have a content type;
- Error: an error is issued when abort statement is used outside of a transaction statement;
- Error: an error is issued when a return statement is detected outside an operation;
- Error: an error is issued when the types of the left hand side and the right hand side of an assignment statement are not compatible;
- Error: an error is issued when the left hand side of an assignment statement is an invalid expression (i.e. expressions that are NOT name expressions, property call expressions or variable declaration expressions);
- Error: an error is issued when two operations with the same signature are defined in the same program;
- Error: an error is issued when an invoked operation cannot be found withinin either the EOL standard library or the user defined operations;
- Error: an error is issued when accessing an undefined property in a property call expression;
- Error: an error is issued when a call to first order logic operation has no target;

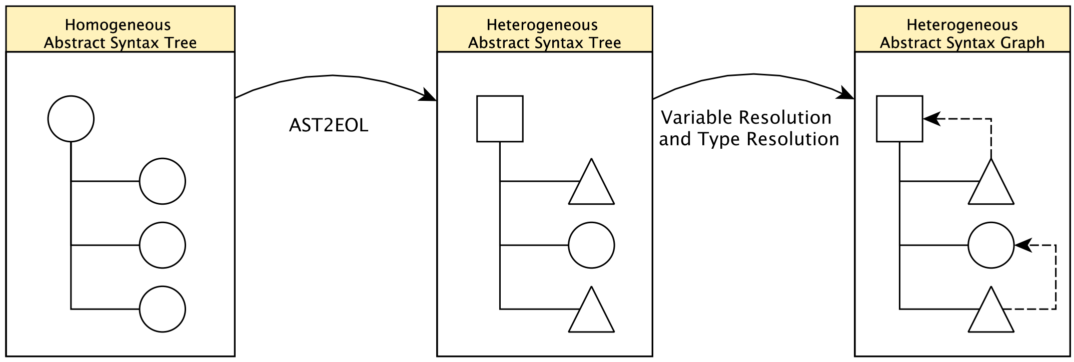


Figure 7.9.: The transformation from Homogeneous Abstract Syntax Tree to Heterogeneous Abstract Syntax Graph

7.6. Chapter Summary

In this chapter, the design and implementation of the core facilities of the Epsilon static analysis framework were presented. In Section 7.1 and 7.2, the design and implementation of the infrastructure of the EOL static analyser were discussed and the EOLVisitor facility was presented. In Section 7.3, the variable resolution facility of the EOL static analyser was presented. In Section 7.4, the type resolution facility of the EOL static analyser was presented, as well as the details of the static analysis approach adopted and the detailed type resolution rules for each EOL language construct.

With the EOL static analyser constructed, the next step is to design and implement the static analysis facilities for other Epsilon languages. In Chapter 8, the design and implementation of the static analysers for the Epsilon Validation Language (EVL) and the Epsilon Transformation Language (ETL) will be presented.

The variable resolution facility and the type resolution facility constitute the infrastructure of the Epsilon static analysis framework. This infrastructure, together with the EOL metamodel, provide a means to transform a homogeneous abstract syntax tree (ANTLR-based AST) to a heterogeneous abstract syntax tree (Ecore-based EOL model that conforms to the EOL metamodel), and eventually to a heterogeneous abstract syntax graph (variable resolved and type resolved EOL model that conforms to the EOL metamodel), shown in Figure 7.9. Using the heterogeneous abstract syntax graph, facilities that aim at addressing the scalability challenges in MDE can be constructed, which will be discussed in Chapter 9.

7.7. Terminology

LogBook The LogBook facility is used to record warnings and errors that arise during the static analysis process.

Variable Resolution: The term Variable Resolution refers to the process to solve the reaching definition problem, i.e. to establish the links between a variable declaration and its references. Problems (warnings/errors) arise during the variable resolution are recorded by the LogBook facility.

Type Resolution: The term Type Resolution refers to the process to resolve the types of the expressions in the EOL program. Type resolution happens after the variable resolution and it also resolves the links between an operation definition and corresponding calls to it. Problems (warnings/errors) arise during the type resolution are recorded by the LogBook facility.

8. Extending the Epsilon Static Analysis Framework

This chapter discusses the development iteration where the established Epsilon Static Analysis Framework, discussed in Chapter 7, is extended to support static analysis for the Epsilon Validation Language (EVL) [73] and the Epsilon Transformation Language (ETL) [61]. In Section 8.1, the static analyser for EVL is discussed. The EVL metamodel is discussed in Section 8.1.1, the AST2EVL transformation is discussed in Section 8.1.2, the EVL variable resolution facility is discussed in Section 8.1.3 and the EVL type resolution facility is discussed in Section 8.1.4. In Section 8.2, the static analyser for ETL is discussed. The ETL metamodel is discussed in Section 8.2.1, the AST2ETL transformation is discussed in Section 8.2.2, the ETL variable resolution facility is discussed in Section 8.2.3 and the EVL type resolution facility is discussed in Section 8.2.4. For ETL, a transformation rule dependency calculation facility is discussed in detail in Section 8.2.5, which is useful for transformation analysis and potentially for optimising ETL transformations.

8.1. The EVL Static Analyser

This section discusses the design and implementation of the static analyser for the Epsilon Validation Language (EVL) [73]. The EVL static analyser is built by extending the EOL static analyser. The EVL static analyser includes the EVL metamodel, the AST2EVL transformation, the EVL visitor framework, the EVL variable resolution facility and the EVL type resolution facility, which are discussed in what follows.

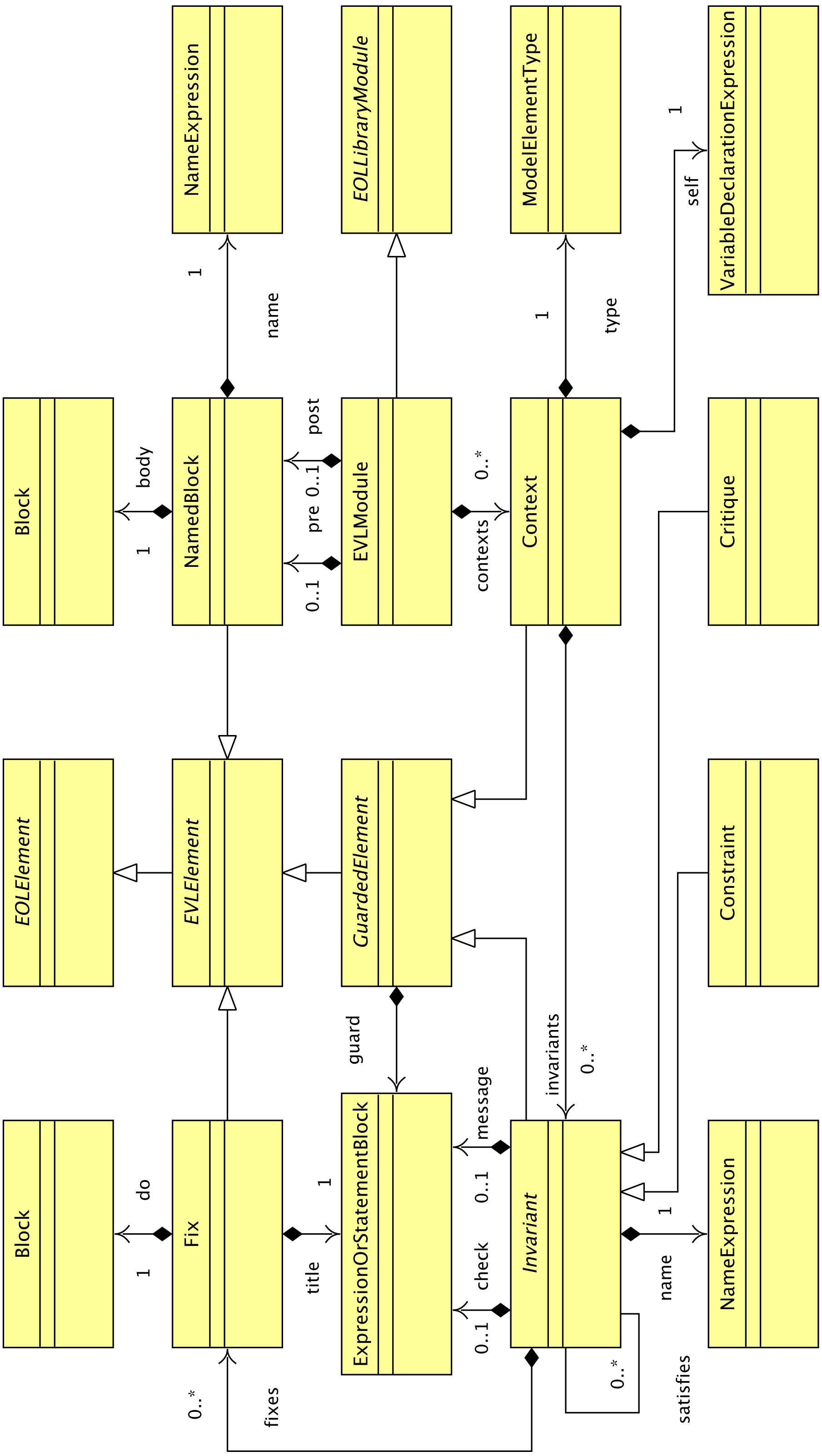


Figure 8.1.: The EVL Metamodel

8.1.1. The EVL Metamodel

To support the static analysis of EVL programs, the EVL metamodel was created by extending the EOL metamodel. The structure of the EVL metamodel is shown in Figure 8.1 (metamodel elements with dashed lines are elements reused from the EOL metamodel). The elements introduced in the EVL metamodel are:

EVLModule

An EVL program is represented by the *EVLModule* element, which is a subtype of *EOLLibraryModule* in the EOL metamodel. An *EVLModule* contains a number of constraints organised in *Context*s, which are discussed further in this section. It also contains an optional *pre* statement block and an optional *post* statement block, which are executed before and after the module's constraints respectively. The *pre* and *post* blocks are **NamedBlock** elements, which contain a *name* (of type *NameExpression* in EOL), and a *body* (of type *Block* in EOL).

GuardedElement

In EVL, a **Context** specifies the kind of instances in which the contained *Invariant*s will be evaluated. *Context* and *Invariant* can define an optional *guard* (of type *ExpressionOrStatementBlock*) to limit their applicabilities only to elements that satisfy a condition. Thus, they are categorised as *GuardedElement* elements.

A *Context* contains a *type* (of type *ModelElementType*) to specify the model element type to which it is applicable. A *Context* has a variable named *self* (of type *VariableDeclarationExpression*) which is used to refer to the object that is being validated by a *Context*. A *Context* also contains a number of *invariant*s (of type *Invariant*), which model elements are required to satisfy.

An **Invariant** is a *GuardedElement*. An *Invariant* contains the following properties:

- a *name* (of type *NameExpression*) that identifies it;
- a *check* (of type *ExpressionOrStatementBlock*) to express the properties to validate;

- a *message* (of type *ExpressionOrStatementBlock*) to provide detailed user feedback to describe the reason an *Invariant* has failed for a particular model element;
- a number of *fixes* (of type **Fix**) which support semi-automatic fixing of elements on which the invariant has failed;
- *satisfies*, which is a list of references to other *Invariant*s that the *Invariant* under question depends on. The *satisfies* property is calculated during static analysis (discussed in Section 8.1.4).

*Invariant*s are further categorised into **Constraint**s, which are critical errors that invalidate the model, and **Critique**s which are non-critical errors that do not invalidate the model, but should be addressed to enhance the quality of the model [73].

Fix

A *Fix* defines a *title* (of type *ExpressionOrStatementBlock*) to allow the user to specify a context-aware title [9]. A *Fix* defines a *do* (of type *Block*) that describes the fix which repairs the inconsistency in the model.

Concrete Syntax

Listings 8.1, 8.2, 8.3 and 8.4 demonstrate the concrete syntax of the EVL *Context*, *Invariant Fix*, *pre-* and *post-blocks* discussed above.

```
context <name> {
  (guard (:Expression) | ({StatementBlock}))?
  (invariant)*
}
```

Listing 8.1: Concrete Syntax of an EVL context [9].

```
(@lazy)?
(constraint|critique) <name> {
  (guard (:Expression) | ({StatementBlock}))?
  (check (:Expression) | ({StatementBlock}))?
```

```
  (message (:Expression) | ({StatementBlock}))?
  (fix)?
}
```

Listing 8.2: Concrete Syntax of an EVL invariant [9].

```
fix {
  (guard (:Expression) | ({StatementBlock}))?
  (title (:Expression) | ({StatementBlock}))?
  do {
    StatementBlock
  }
}
```

Listing 8.3: Concrete Syntax of an EVL fix [9].

```
(pre|post) <name> {
  block
}
```

Listing 8.4: Concrete Syntax of an EVL fix [9].

8.1.2. The AST2EVL Transformation and the EVLVisitor Framework

An AST2EVL transformation facility was created by extending the AST2EOL transformation facility (Section 6.6). A set of *EVLElementCreator*s (in accordance to the additional *EVLElement*s created in the EVL metamodel) were created, which create *EVLElement*s from ASTs generated by the EVL parser.

The EVLVisitor Framework was generated based on the EVL metamodel using the mode-to-text transformation discussed in Section 7.2. The EVLVisitor Framework acts as the infrastructure for the EVL variable resolution and type resolution facilities, which were created by extending the EOL variable resolution (Section 7.3) and type resolution facilities (Section 7.4) respectively.

8.1.3. The EVL Variable Resolution Facility

The implementation of the EVL variable resolution facility involves creating a number of *EVLElementVariableResolvers* atop the EOL variable resolution facility (as well as by extending the EVLVisitor framework). Apart from the *EOLElementVariableResolvers* discussed in Section 7.3, the EVL variable resolution facility comprises the following *EVLElementVariableResolvers*:

- *EVLModuleVariableResolver*, which is used to resolve the variables of an EVL module. The resolution process is driven by the *EVLModuleVariableResolver*, in the sense that it visits the *import*s, *pre* block, *context*s, *operationDefinition*s and the *post* block in sequential order;
- *ContextVariableResolver*, which is used to resolve variables when a *Context* in the EVL module is encountered;
- *InvariantVariableResolver*, which is used to resolve variables when an *Invariant* is encountered in the EVL module;
- *FixVariableResolver*, which is used to resolve variables when a *Fix* is encountered in the EVL module;
- *NamedBlockVariableResolver*, which is used to resolve the *pre* and *post* blocks in an EVL module.

These *EVLElementVariableResolvers* direct the control of the variable resolution. In addition to the scoping rules of the EOL variable resolution facility, the EVL resolution facility defines additional scoping rules. For example, the variables defined in the *guard* of a *Context* are accessible throughout the whole scope of the *Context*. The same principle applies to variables defined in the *guard* of an *Invariant*, so that the variables are accessible throughout the whole scope of the *Invariant*.

8.1.4. The EVL Type Resolution Facility

The EVL type resolution facility was implemented atop the EOL type resolution facility (Section 7.4). The implementation of the EVL type resolution facility involves creating a number of *EVLElementTypeResolvers* (by extending the EVLVisitor framework):

- *EVLModuleTypeResolver*;
- *ContextTypeResolver*;
- *InvariantTypeResolver*;
- *FixTypeResolver*;
- *NamedBlockTypeResolver*;

These *EVLElementTypeResolvers* drive the type resolution of different *EVLElement*s, which are one-to-one mappings to the newly created *EVLElement*s in the EVL metamodel. In addition, a number of operations are introduced in the EVL standard library:

- *satisfies(invariant: String)*;
- *satisfiesAll(invariants: Sequence(String))*;
- *satisfiesOne(invariants: Sequence(String))*.

```
context Lecturer {

  constraint DefinesNames {
    check: self.first_name.isDefined() and
                self.last_name.isDefined()
    message: "Lecturer’s names must be defined"
  }

  constraint DefinesID {
    guard: self.satisfies("DefinesNames")
    check: self.staff_id.isDefined()
  }
}
```

Listing 8.5: An example EVL program

These operations are used to capture dependencies between *Invariant*s. Using these operations, an *Invariant* can specify in its *guard* the *Invariant*s that need to be satisfied in order for it to be executed. Listing 8.5 demonstrates how *satisfies()* is used. Lines 1-13 declare two *constraint*s for the *context Lecturer*. Lines 3-7 define a *constraint* named *DefinesNames* which checks if a *Lecturer*'s *first_name* and *last_name* are defined. In line 9-11, a *constraint* named *DefinesID* is defined, which checks if the *staff_id* of a *Lecturer* is defined. To promote modularity, in line 10, the *satisfies* operation is named, which states that for "DefinesID" to be meaningful, the lecturer should first satisfy "DefinesNames". In other words, the *constraint* "DefinesID" depends on the *constraint* "DefinesNames".

A number of *OperationDefinitionHandler*s (discussed in Section 7.4.2) has been created to handle calls to these operations. The type resolution rules of the operations are defined as follows:

satisfies(invariant: String)

This operation takes a String and returns a *Boolean* value. Let MC and $[MC]$ denote the *MethodCallExpression* (that calls *satisfies()*) and the type of it, and arg and $[arg]$ denote the argument and its type. The type resolution rule of this operation is:

$$(arg \in Expression) \wedge ([arg] = String) \Rightarrow [MC] = Boolean$$

The parameter *invariant* can be any *Expression*, which must evaluate to a value of type *StringType*. An error will be reported if the type of the *invariant* is not *StringType*. For the static analysis to calculate the constraint dependencies, ideally the parameter *invariant* should be of type *StringExpression*. If the *invariant* is not a *StringExpression*, at compile time, the static analyser is not able to compute the value of such an expression. A warning will be reported if *invariant* is not a *StringExpression* to notify the developer that the constraint dependency cannot be resolved.

The *satisfies()* handler looks for the *Invariant* by the value provided in the parameter of the method call. If an *Invariant* with the name is found, the found *Invariant* is then added to the *satisfies* property of the current *Invariant*. If no *Invariant* is found, an error will be reported.

satisfies(invariantsAll: Sequence(String)) and satisfiesOne(invariantsOne: Sequence(String))

These operations take a *Sequence* of *String* values and return a *Boolean* value. Let MC and $[MC]$ denote the *MethodCallExpression* (that calls these methods) and the types of them, ARG denote the set of all the arguments, and arg and $[arg]$ denote each argument and its type. The type resolution rule of these operations is:

$$(\forall\, arg \in ARG : arg = Expression \wedge [arg] = String) \Rightarrow [MC] = Boolean$$

Each element in the parameter of *invariantsAll* or *invariantsOne* can be of any *Expression*, which must evaluate to a value of type *StringType*. An error will be reported if any element in the parameter *invariantsAll* or *invariantOne* is not *StringType*. For the static analysis to calculate the constraint dependencies, ideally each element in the parameter *invariantAll* or *invariantOne* should be of type *StringExpression*. A warning will be reported if this is not the case, to notify the developer that the constraint dependency cannot be resolved.

For the method call to *satisfiesAll()*, the *satisfiesAll()* handler iterates through the parameter *invariantsAll* and looks for *Invariant*s by the value provided in the parameter of the method call. If an *Invariant* is found, the found *Invariant* is added to the *satisfies* property of the current *Invariant*. If no *Invariant* is found, an error will be raised.

For the method call to *satisfiesOne()*, the handler adopts an optimistic approach which copies the behaviour of the *satisfiesAll()* handler. This approach results in all the *Invariant*s found to be added to the *satisfies* property of the current *Invariant*. Thus, the constraint dependency is only an approximation because of the uncertainty introduced by the *satisfiesOne()* operation.

The EVL Type Resolution facility inherits all the features of the EOL Type Resolution facility (Section 7.4). However, there are additional checks for *Invariant*:

- the *check* of an *Invariant* should be either an *Expression* with type *BooleanType*, or a *Block* of *Statement*s. However, if *check* contains a *Block* of *Statement*s, for each *ReturnStatement*, the returned value must have a *BooleanType*. The same principle applies to the *guard* of *Context*, *Invariant* and *Fix*;

- the *message* of an *Invariant* should be either an *Expression* with type *StringType*, or a *Block* of *ExpressionStatement*s, for which the type of each *Expression* encapsulated in the *ExpressionStatement* should be *StringType*. The same principle applies to the *title* of *Fix*;
- the *fix* of an *Invariant* should not contain any *ReturnStatement* (that returns a value) in the sense that fixes do not return values in EVL (*ReturnStatement*s that do not return anything, e.g. *return;* is acceptable). Thus, an error is reported if a *ReturnStatement* (that returns a value) is detected within a *Fix*.

8.2. The ETL Static Analyser

This section presents the design and implementation of the static analyser for the Epsilon Transformation Language (ETL) [61]. The ETL static analyser is built by extending the EOL static analyser. The ETL static analyser includes the ETL metamodel, the AST2ETL transformation facility, the ETLVisitor framework, the ETL variable resolution facility and the ETL type resolution facility, which are discussed in this section.

8.2.1. The ETL Metamodel

To support the static analysis of ETL programs, the ETL metamodel was created by extending the EOL metamodel. The structure of the ETL metamodel is shown in Figure 8.2. The elements introduced in the ETL metamodel (atop the EOL metamodel) are the following:

ETLModule

An ETL transformation program is organised in an *ETLModule*. The *ETLModule* is a subtype of *EOLLibraryModule* in the EOL metamodel. An *ETLModule* contains a number of *pre* and *post* **NamedBlock**s, which are executed before and after an ETL program's transformation rules respectively.

TransformationRule

An *ETLModule* contains a number of *TransformationRules*, which are used to represent the transformation rules in an ETL program. A *TransformationRule* contains a number of properties:

- an optional *annotationBlock* (of type *AnnotationBlock*), which provides annotation(s) about the *TransformationRule*. For example, a *TransformationRule* may be declared as *abstract*, *lazy*, *primary* or *greedy*, using appropriate annotations;
- attributes *abstract*, *lazy*, *primary* and *greedy* are used to denote if a *TransformationRule* is abstract, lazy, primary or greedy. Such attributes decide the rule execution scheduling and are provided in the *annotationBlock* of a *TransformationRule*. These attributes are calculated during the static analysis. The semantics of these attributes is discussed in [61];
- a *source* (of type *FormalParameterExpression*) denoting the type of instances in which the *TransformationRule* is applicable;
- a number of *targets* (of type *FormalParameterExpression*) denoting the type of the target elements to be created by the *TransformationRule*;
- a number of *extends* (of type *NameExpression*), to denote if a *TransformationRule* extends the behaviour of other *TransformationRules*. A *TransformationRule* also contains a number of *resolvedParentRules*, which are a collection of references to the current rule's parent rules, and are calculated during the static analysis:
- an optional *guard* (of type *ExpressionOrStatementBlock*) limiting the applicability of the *TransformationRule*;
- a *body* (of type *Block*) which contains the logic of the transformation;
- a number of *TransformationRuleDependency*(-ies), which are used to capture the transformation dependency graph of an ETL program, as discussed in Section 8.2.5.

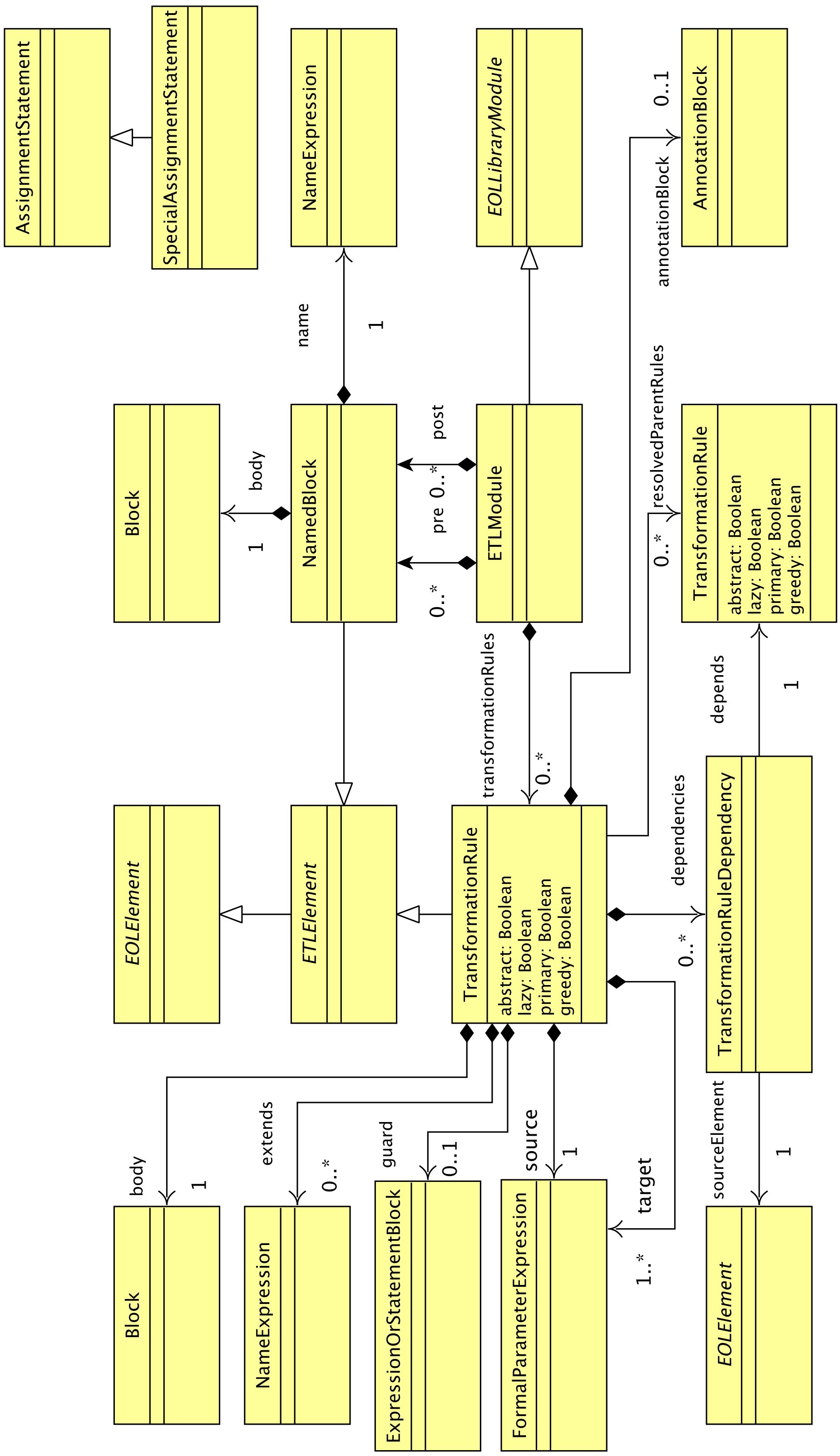


Figure 8.2.: The ETL Metamodel

SpecialAssignmentStatement

In ETL, transformation rules can depend on each other by using the operation *equivalent()* and *equivalents()* provided in the ETL standard library. In addition, there is also a special assignment operator (::=) which is equivalent to making the call to *equivalent()* on the right hand side of the assignment operator. To capture this behaviour, the *SpecialAssignmentStatement* is created by extending the *AssignmentStatement* in EOL.

Concrete Syntax

Listing 8.6 and Listing 8.7 demonstrate the concrete syntax of the ETL *TransformationRule* and *Pre/Post* Blocks discussed above.

```
(@abstract)?
(@lazy)?
(@primary)?
rule <name>
transform <sourceParameterName>:<sourceParameterType>
to <targetParameterName>:<targetParameterType>
        (,<targetParameterName>:<targetParameterType>)*
(extends <ruleName>(, <ruleName>)*)? {
  (guard (:expression)|({statementBlock}))?
  statement+
}
```

Listing 8.6: Concrete Syntax of a Transformation Rule [61]

```
(pre|post) <name> {
  statement+
}
```

Listing 8.7: Concrete Syntax of Pre and Post Blocks [61]

8.2.2. The AST2ETL Transformation and the ETLVisitor Framework

An AST2ETL transformation facility was created by extending the AST2EOL transformation facility (Section 6.6). A set of *ETLElementCreator*s (in accordance to the additional *ETLElement*s created in the ETL metamodel) were created which, in turn, create *ETLElement*s from ASTs generated by the ETL parser.

The ETLVisitor Framework was generated based on the ETL metamodel using the model-to-text transformation discussed in Section 7.2. The ETLVisitor framework acts as the infrastructure for the ETL variable resolution and type resolution facilities, which were created by extending the EOL variable resolution (Section 7.3) and type resolution facilities (Section 7.4).

8.2.3. The ETL Variable Resolution Facility

The implementation of the *EVLVariableResolver* involves creating a number of *EVLElementVariableResolver*s, which are used to direct the resolution process:

- *ETLModuleVariableResolver*, which is used to perform variable resolution for an ETL module. The resolution process is driven by the *ETLModuleVariableResolver* in the sense that it visits the ETL module's model declarations, imports, pre blocks, transformation rules, operation definitions and post blocks in sequential order;

- *TransformationRuleVariableResolver*. The resolution is delegated to the *TransformationRuleVariableResolver* whenever a *TransformationRule* is encountered in the ETL module. The *TransformationRuleVariable* puts its *source* and *target*s in the current active *Frame* in the *FrameStack* (Section 7.3) and performs the resolution;

- *NamedBlockVariableResolver*, which is used to perform the variable resolution for the *pre* and *post* blocks of the ETL module.

These *ETLElementVariableResolvers* direct the control of the variable resolution. There are additional scoping rules added by the ETL variable resolution facility. For example, variables defined in the *pre* block of an *ETLModule* are available throughout the entire *ETLModule*; variables defined in the *guard* of a *TransformationRule* are available throughout the scope of the *TransformationRule*.

8.2.4. The ETL Type Resolution Facility

The ETL type resolution facility was implemented atop the EOL type resolution facility (Section 7.4). The implementation of the ETL type resolution facility involves creating a number of *ETLElementTypeResolvers* (by extending the ETLVisitor framework). The *ETLElementTypeResolvers* are one-to-one mappings to the newly created *ETLElement*s in the ETL metamodel. In addition, a number of operations are introduced in the ETL standard library: the **equivalent()** and the **equivalents()** operations.

```
1  rule Lecturer2Person
2  transform l : University!Lecturer
3  to p : SocialNetwork!Person {
4    p.first_name = l.first_name;
5    p.last_name = l.last_name;
6    p.knows = l.students.equivalent();
7  }
8
9  rule Student2Person
10 transform s: University!Student
11 to p: SocialNetwork!Person {
12   p.first_name = s.first_name;
13   p.last_name = s.last_name;
14   p.knows = s.tutor.equivalent();
15 }
```

Listing 8.8: A model-to-model transformation written in Epsilon Transformation Language

The principle behind the *equivalent()* and *equivalents()* operations is to resolve target elements that have been (or can be) transformed from source elements by other rules. Consider the ETL transformation example (involving the *University* metamodel in Figure 2.9, Section 2.1.6, and the *SocialNetwork* metamodel in Figure 2.10, Section 2.1.6) in Listing 8.8. In lines 1-7, a *TransformationRule* is in place, which transforms a *Lecturer* (denoted as *l*) in the *University* model into a *Person* (denoted as *p*) in the target

SocialNetwork model. In line 6, *equivalent()* is called on the *students* property of l, which denotes that the contents in the *students* property of l, which have been or can be transformed, should be resolved. The ETL transformation engine is responsible for handling the call to *equivalent()*. Thus, if the ETL transformation engine is not able to find any objects that have been transformed with relation to $l.students$, in Listing 8.8, the *TransformationRule Student2Person* (lines 9-15) will be called to resolve the *students* property of l.

equivalents() and *equivalent()* are semantically similar to each other. When *equivalents()* is called on a single-valued object, the ETL execution engine inspects the established transformation trace, invokes *all* applicable transformation rules (if necessary) to calculate the counterparts of the element in the target model, and returns a *Sequence* containing the elements. The order of the elements in the result respects the order of the applicable transformation rules defined within the ETL module. When *equivalents()* is invoked on a collection of objects, it returns a *Sequence* containing *Sequence*s that contain the counterparts of the source elements contained in the collection.

When *equivalent()* is called on a single-valued object, the look up of transformation rules in the transformation trace is the same as for *equivalents()*. The difference is that only the first element of the respective result that would have been returned by *equivalents()* is returned by the call to *equivalent()*. When *equivalent()* is invoked on a collection, a flattened *Sequence* (of the result that would have been returned by *equivalents()*) is returned.

The ETL type resolution facility implements dedicated handlers for *equivalent()* and *equivalents()*. During the type resolution process, the ETL type resolution facility establishes a transformation trace by matching the type of the *source* of all *TransformationRule*s.

equivalent(rule : String ..)

The *equivalent()* operation can be invoked with a number of parameters specifying from which rules the source should be resolved. Let MC and $[MC]$ denote the call to *equivalent()* and its type, $rule$, $[rule]$ and $RULE$ denote each matched *TransformationRule*, the type of the (first) *target* of that *TransformationRule* and the set of all matched *TransformationRule*s. Since there may be any number of *TransformationRule*s, and a *TransformationRule* may have any number of *target*s, the notation (*index*) is used to refer to an element in a collection with *index*.

If *equivalent()* is called on a single-valued object, the type of the call should be the type of the first *target* of the first matched *TransformationRule*:

$$[MC] = [RULE(0)]$$

If *equivalent()* is called on a collection, the type resolution collects all matched *TransformationRule*s which are applicable for the call to *equivalent()*, then a computation is performed on all the first *target*s of such transformation rules. The objective is to compute the *Least Common Type* (LCT) of all the *target* types. If a LCT is found, then the type of the call to *equivalent()* is $Sequence(LCT)$; otherwise, it is $Sequence(Any)$. Let the symbol $lct()$ denote the function that calculates the *Least Common Type* and LCT denote the LCT returned by $lct()$:

$$[MC] = Sequence(lct([RULE])) \neq null\,?\,LCT\ :\ Any)$$

For all matched *TransformationRule*s on the call to *equivalent()*, a *RuleDependency* is created to link the call to *equivalent()* to the matched *TransformationRule*. The *RuleDependency* is added to the *dependencies* property of the current rule.

equivalents(rule : String ..)

The *equivalents()* operation can be invoked with a number of parameters specifying from which rules the source should be resolved. Let MC and $[MC]$ denote the call to *equivalents()* and its type, and *rule*, $[rule]$ and $RULE$ denote each matched *TransformationRule*, the type of the first *target* of that *TransformationRule* and the set of all matched *TransformationRule*s.

If *equivalents()* is called on a single-valued object, the type resolution collects all matched *TransformationRule*s which are applicable for the call to *equivalents()*. The

Least Common Type (LCT) is computed among all the types of the first *target* of all the *TransformationRule*s. Let the symbol $lct()$ denote the function that computes the *Least Common Type* and LCT denote the LCT returned by $lct()$:

$$[MC] = Sequence(lct([RULE])) \neq null\,?\,LCT\,:\,Any)$$

If *equivalents()* is called on a collection, the type resolution collects all matched *TransformationRule*s which are applicable to the call to *equivalents()*. The *Least Common Type* (LCT) is computed among all the types of the first *target* of all the *TransformationRule*s. Let the symbol $lct()$ denote the function that computes the *Least Common Type* and LCT denote the LCT returned by $lct()$:

$$[MC] = Sequence(Sequence(lct([RULE])) \neq null\,?\,LCT\,:\,Any))$$

For all matched *TransformationRule*s on the call to *equivalents()*, a *RuleDependency* is created to link the call to *equivalents()* to the matched *TransformationRule*. The *RuleDependency* is added to the *dependencies* property of the current rule.

The Special Assignment Operator

The special assignment operator ::= is an alias for invoking the equivalent() operation without parameters on the right hand side of an assignment. As such, its type resolution semantics reuses the semantics of equivalent(). As an assignment operator, the left and right hand side should be of compatible types. Let L and $[L]$ denote the left hand side expression and its type of the ::= operator, R and $[R]$ denote the right hand side expression and its type of the ::= operator; $CollectionType$ denote the set of all collection types in EOL; and $rule$, $[rule]$ and $RULE$ denote each matched *TransformationRule*, the type of the *target* of that *TransformationRule* and the set of all matched *TransformationRule*s. The type constraints are:

If R is a model element, the type resolution is the same as the call to equivalent():

$$[L] = [RULE(0)]$$

If R is a collection, the type resolution is the same as the call to equivalent():

$$[MC] = Sequence(lct([RULE])) \neq null\,?\,LCT\ :\ Any)$$

For all matched *TransformationRules* on the call to *equivalent()*, a *RuleDependency* is created to link the call to *equivalent()* to the matched *TransformationRule*. The *RuleDependency* is added to the *dependencies* property of the current rule:

8.2.5. Transformation Rule Dependency Analysis

The ETL static analyser is able to construct a rule dependency graph. For illustration purposes, a simple example is presented to demonstrate how the calculation is performed. The ETL static analyser provides an Eclipse plug-in to visualise transformation rule dependency graphs and is discussed together with the example. Then, the discussion moves on to a more realistic example, which examines the static analysis on an existing OO2DB transformation.

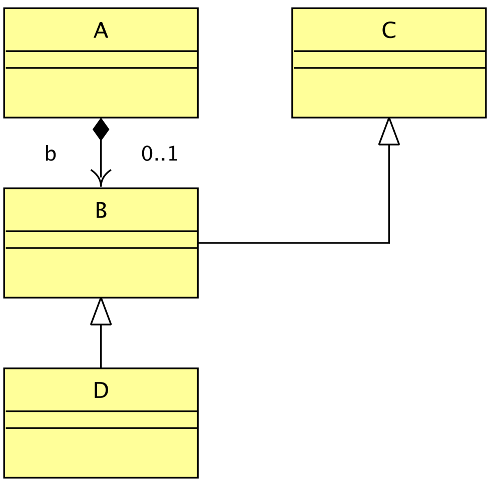


Figure 8.3.: The Source Metamodel

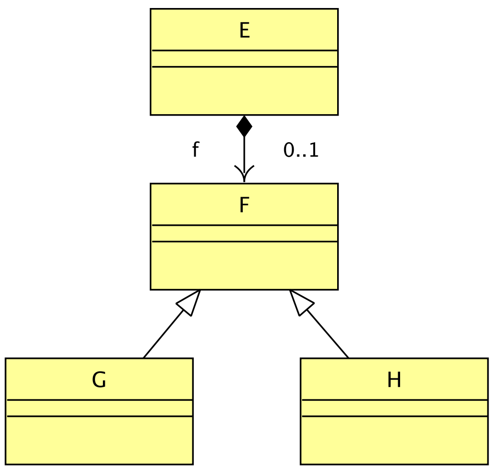


Figure 8.4.: The Target Metamodel

To illustrate the transformation rule dependency calculation, a simple example is provided first. In this example, two simple Ecore metamodels named *Source* and *Target* (in Figure 8.3 and Figure 8.4) are devised for demonstration purposes. In the *Source* metamodel, A has a *single-valued* reference to B, B extends C, and D extends B. In the

Target metamodel, E has a *single-valued* reference to F, and G and H extend F.

```
1 rule A2E
2 transform a : Source!A
3 to e : Target!E {
4   e.f = a.b.equivalent();
5 }
6
7 rule B2F
8 transform b : Source!B
9 to f : Target!F { }
10
11 rule B2G
12 transform b : Source!B
13 to g : Target!G { }
14
15 rule C2F
16 transform c : Source!C
17 to f: Target!F { }
18
19 rule D2G
20 transform d: Source!D
21 to g: Target!G { }
```

Listing 8.9: Example ETL transformation

In Listing 8.9, an ETL transformation is created to illustrate how the transformation dependency graph is calculated. In lines 1-5, a transformation rule named *A2E* is created, which transforms instances of A to instances of E. In line 4, equivalent() is called to resolve the element $a.b$, which is a single-valued reference. Lines 7-10 define a transformation rule which transforms instances of B to instances of F. Lines 11-13 define a transformation rule to transform instances of B to instances of G. Lines 15-17 define a transformation rule to transform instances of C to instances of F. Lines 19-21 define a transformation rule to transform instances of D to instances of G.

equivalent() on single-valued elements

This section illustrates the transformation rule dependency resolution for operation *equivalent()* on single-valued elements. To visualise the transformation rule dependency graph, an Eclipse plug-in is developed using Eclipse Zest [131], which allows the creation of nodes and edges for visualisation. The visualisation of the transformation dependency graph of the program in Listing 8.9 is shown in Figure 8.5. Since the property b of element A is *single-valued*, the call to *equivalent()* resolves to the first transformation rule that transforms instances of B, which in this case is the transformation rule *B2F*.

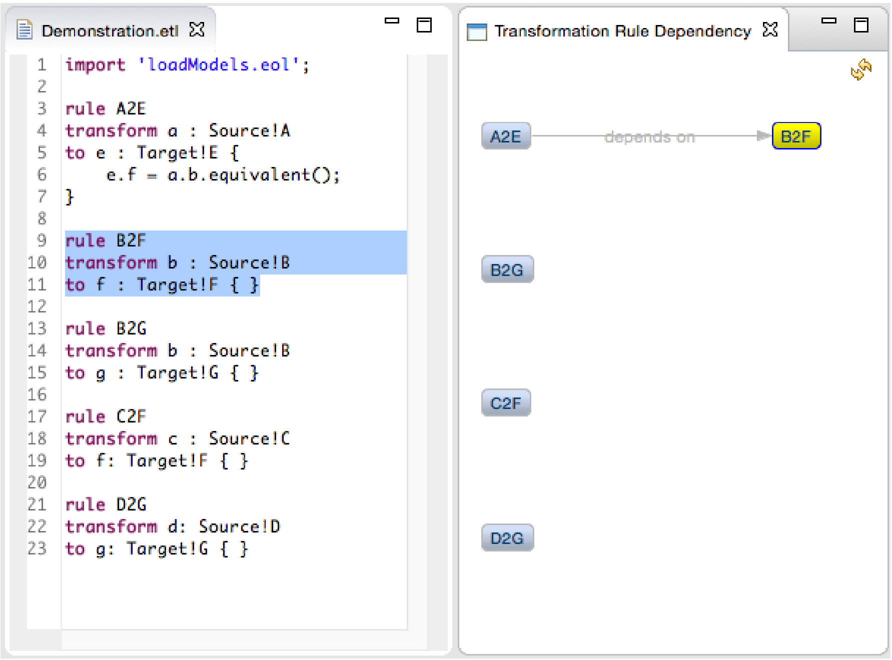


Figure 8.5.: $A2E \rightarrow B2F$ dependency

ETL provides the keyword *primary* for developers to use in the *annotation* of a transformation rule. The keyword *primary* gives priority to the transformation rule when scheduling the transformation (by the ETL transformation engine) and resolving source elements. Thus, if the rule *B2G* is declared as *primary*, as shown in Figure 8.6, the transformation dependency is resolved to rule *B2G*.

If the keyword *primary* is used on transformation rule *C2F*, as shown in Figure 8.7,

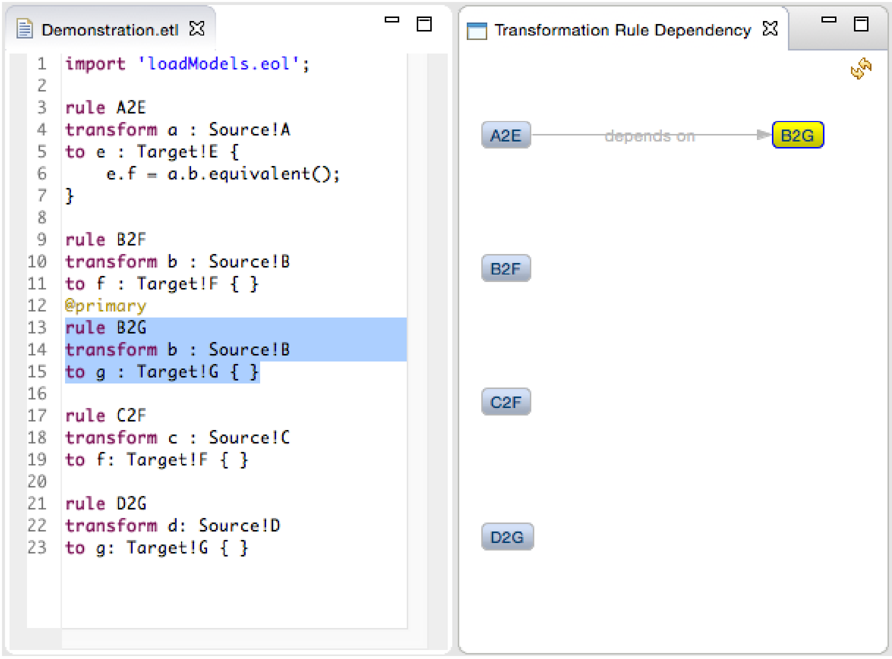


Figure 8.6.: *A2E* → *B2G* dependency

Rule *C2F* will not be resolved because in the *Source* metamodel, *C* is the supertype of *B*. In order for the *equivalent()* operation to call rule *C2F*, the keyword *greedy* should be used. Semantically, the keyword *greedy* means that the transformation rule will transform all objects, which are either instances of the type of the *source* of the applicable transformation rule or instances of its subtypes.

equivalents() on single-valued elements

When operation *equivalents()* is applied on a single-valued element, rule dependencies are resolved to all applicable transformation rules. Figure 8.9 illustrates the transformation rule dependency graph for the call to *equivalents()* on *a.b* (*a.b* is a single-valued element). However, since the expression:

```
a.b.equivalents()
```

is of type Collection, a type mismatch error is detected in line 6 by the ETL static analyser (since *e.f* is single-valued). This error is rectified in Figure 8.10 by getting the *first* element of the expression *a.b.equivalents()*.

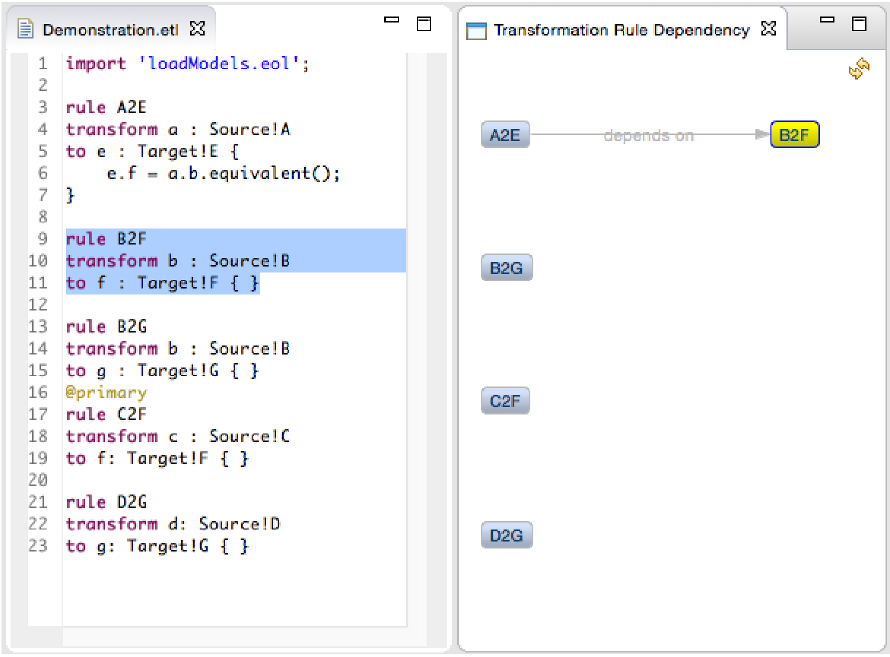


Figure 8.7.: Declaring *C2F* as *primary*

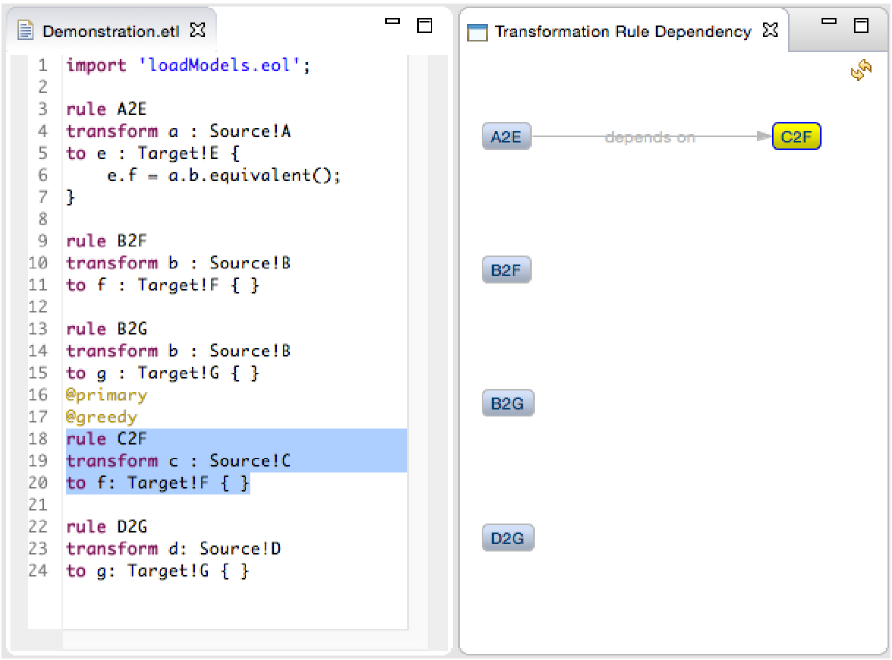


Figure 8.8.: Declaring *C2F* as *primary* and *greedy*

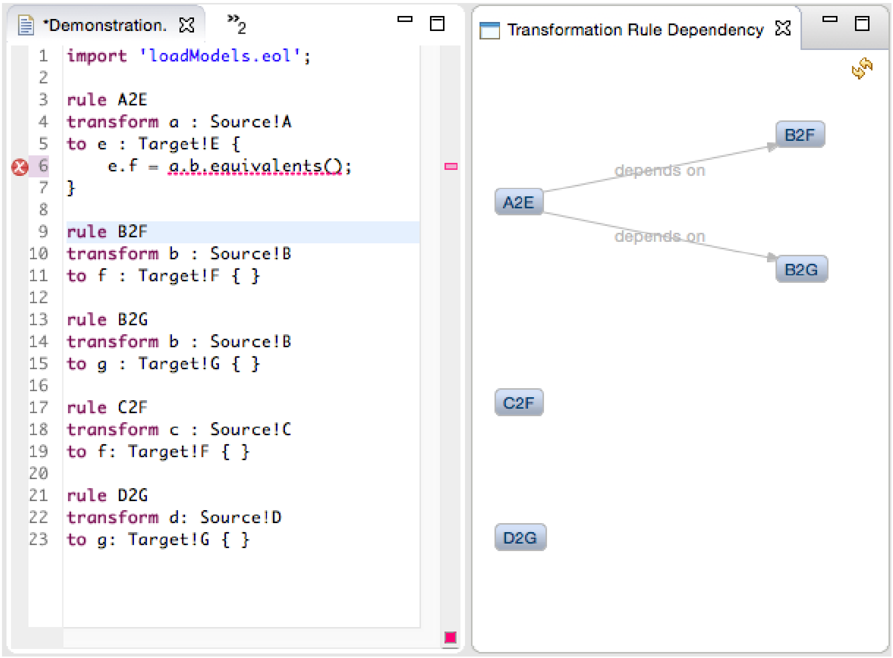


Figure 8.9.: Error detection on the call to *equivalents()*

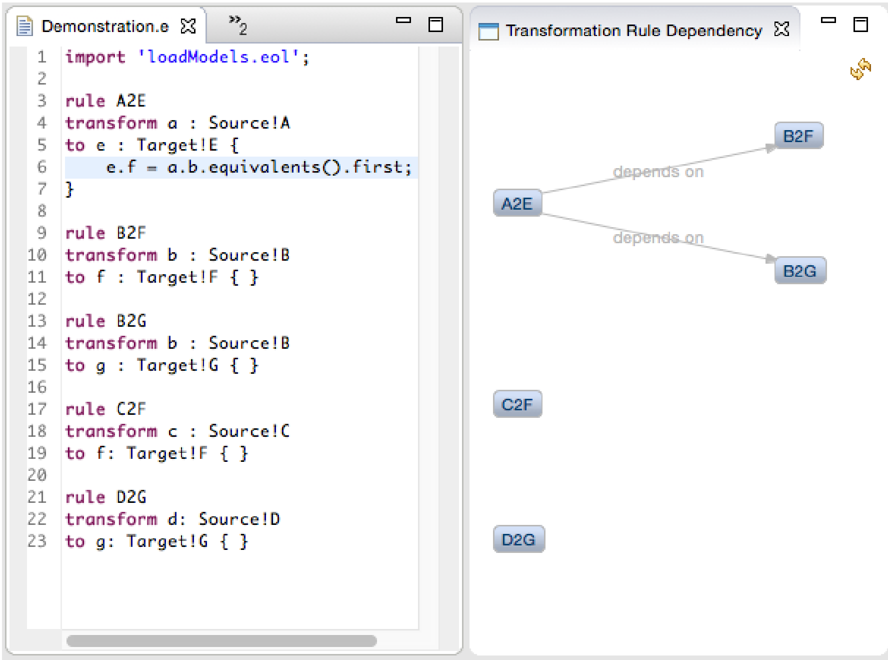


Figure 8.10.: Calling equivalents() on single-valued elements

equivalent() and equivalents() on collections

Although the semantics of equivalent() and equivalents() on collections is different, their transformation rule dependency resolutions are the same. To illustrate this behaviour, the *Source* metamodel and the *Target* metamodel are **modified**: the cardinalities of *A.b* and *E.f* are changed to *many*. Thus, when the call to *equivalent()* is made on *a.b* in Figure 8.11, the transformation rule dependencies are resolved to all applicable transformation rules.

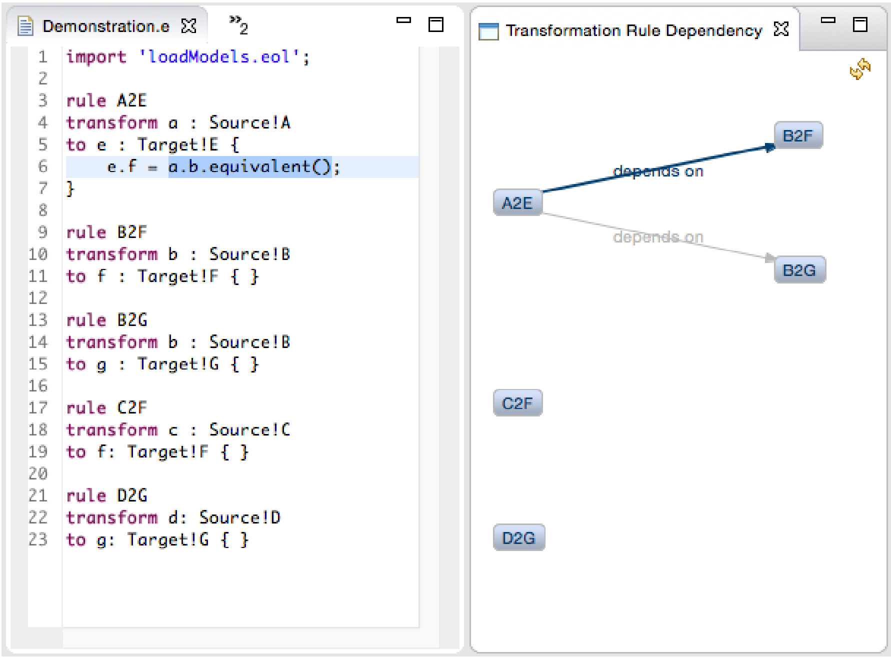


Figure 8.11.: Calling equivalents() on collections

Analysing OO2DB

In a more realistic and complex example, the transformation rule dependency calculation facility is used to analyse the ETL transformation *OO2DB* in the Epsilon Examples[1]. The *OO2DB* transformation is used to transform models that conform to the *OO* metamodel, which is a metamodel containing constructs related to Object-Oriented design, to models that conform to the *DB* metamodel, which is a metamodel containing constructs related to Relational Database (Emfatic Specifications provided in Appendix B).

The *OO2DB* transformation contains four transformation rules:

- *Class2Table*, which transforms an instance of *Class* to an instance of *Table*;

[1] `https://www.eclipse.org/epsilon/examples/index.php?example=org.eclipse.epsilon.examples.oo2db`

- *SingleValuedAttribute2Column*, which transforms a single-valued attribute of a *Class* to a column of a *Table*;
- *MultiValuedAttribute2Table*, which transforms a multi-valued attribute of a *Class* to a column of a *Table*;
- *Reference2ForeignKey*, which transforms a reference of a *Class* to a foreign key in a *Table*.

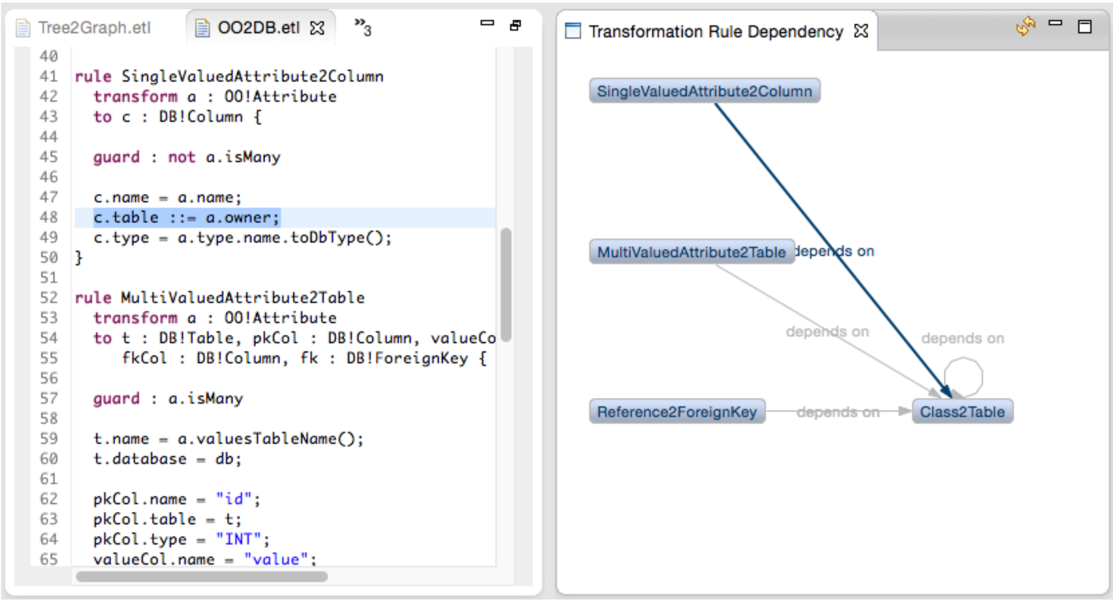


Figure 8.12.: OO2DB Transformation Rule Dependency Graph.

The transformation rule dependency graph of *OO2DB* is shown in Figure 8.12. It is noteworthy that the transformation graph supports navigating back to the source code, so that in Figure 8.12, when the dependency between *SingleValuedAttribute2Column* and *Class2Table* is selected, the tool navigates to the source code where it happens. In the following code:

```
c.table ::= a.owner;
```

c is the *Column* that the transformation creates and *Column* has a reference named *table*, which refers to the *Table* to which the *Column* belongs. *a* is the source *Attribute* to be transformed and *Attribute* has a reference named *owner*, which refers to the *Class* to which the *Attribute* belongs. The special operator ::= is used to resolve *a.owner*, which in turn calls the transformation rule *Class2Table* to create the corresponding *Table*.

For rules *MultiValuedAttribute2Table* and *Reference2ForeignKey*, the rule dependencies are resolved in the same manner. On the other hand, the rule *Class2Table* depends on itself, because when transforming a *Class*, it also recursively calls *Class2Table* to create instances of the super *Class* of the current *Class* that is being transformed.

8.3. Chapter Summary

This chapter presented the design and implementation of the EVL and ETL static analysers, which are both built by extending the EOL static analyser. The implementation of these two static analysers demonstrates the extensibility of the Epsilon static analysis framework, in the sense that the EOL metamodel, the EOL variable resolution facility and the EOL type resolution facility are reused in their entireties to create the EVL and ETL static analysers.

8.4. Terminology

Epsilon Validation Language (EVL): The Epsilon Validation Language (EVL) contributes the model validation capabilities to the Epsilon platform. Using EVL, invariants can be expressed which are validated against the models. EVL is built atop EOL.

EVL metamodel: The EVL metamodel refers to the abstract syntax of the Epsilon Validation Language (EVL) represented in the form of a Ecore based metamodel. The EVL metamodel is built by extending the EOL metamodel.

AST2EVL: In the context of this thesis, the term AST2EVL refers to the transformation which transforms a homogeneous abstract syntax tree produced by the Epsilon parser (for EVL) to a model which conform to the EVL metamodel.

EVL static analyser: Built atop the EOL static analyser, the EVL static analyser provides a variable resolution facility (which extends the EOL variable resolution facility) and a type resolution facility (which extends the EOL type resolution facility). Each facility contains their own rules to analyse EVL programs.

Epsilon Transformation Language (ETL): The Epsilon Transformation Language (ETL) contributes model-to-model transformation capabilities to the Epsilon platform. ETL is a hybrid transformation language which provides declarative and imperative language constructs. ETL is built atop EOL

ETL metamodel: The ETL metamodel refers to the abstract syntax of the Epsilon Transformation Language (ETL) represented in the form of a Ecore based metamodel. The ETL metamodel is built by extending the EOL metamodel.

AST2ETL: In the context of this thesis, the term AST2ETL refers to the transformation which transforms a homogeneous abstract syntax tree produced by the Epsilon parser (for ETL) to a model which conform to the ETL metamodel.

ETL static analyser: Built atop the EOL static analyser, the ETL static analyser provides a variable resolution facility (which extends the EOL variable resolution facility) and a type resolution facility (which extends the EOL type resolution facility). Each facility contains their own rules to analyse ETL programs.

9. Evaluation

Chapter 6 discussed the infrastructure of the core Epsilon static analysis framework, Chapters 7 and 8 discussed the EOL, EVL and ETL static analysers. In this chapter, the evaluation of the static analysis framework is carried out.

The evaluation of the extensibility of the Epsilon Static Analysis Model Connectivity layer (ESAMC) is presented in Section 6.3.3, where the implementaiton of an additional modelling technology driver (for schema-less XML models) is discussed. Model queries and transformations involving transforming models defined in both EMF and schema-less XML simultaneously will be evaluated to further validate the extensibility of ESAMC and the Epsilon static analysis framework. The evaluation of the EOL, EVL and ETL static analyser is then carried out by analysing existing queries and transformations. This chapter then points out the limitation of the Epsilon static analysis framework which hopefully will be addressed in future work.

9.1. Extending ESAMC: A Schema-less XML Driver

Chapter 6 highlighted the need for the Epsilon Static Analysis Model Connectivity layer (ESAMC). To perform static analysis on programs written in Epsilon languages, there is a need for the Epsilon static analysis framework to provide support for accessing metamodels defined in different modelling technologies. In the previous chapters, the example programs presented all interacted with models defined in EMF. This section evaluates the extensibility of ESAMC by presenting a schema-less XML driver (hereby referred to as XML driver), which is an extension of the ESAMC to support the analysis of programs that manage models defined in schema-less XML documents (hereby referred to as XML models).

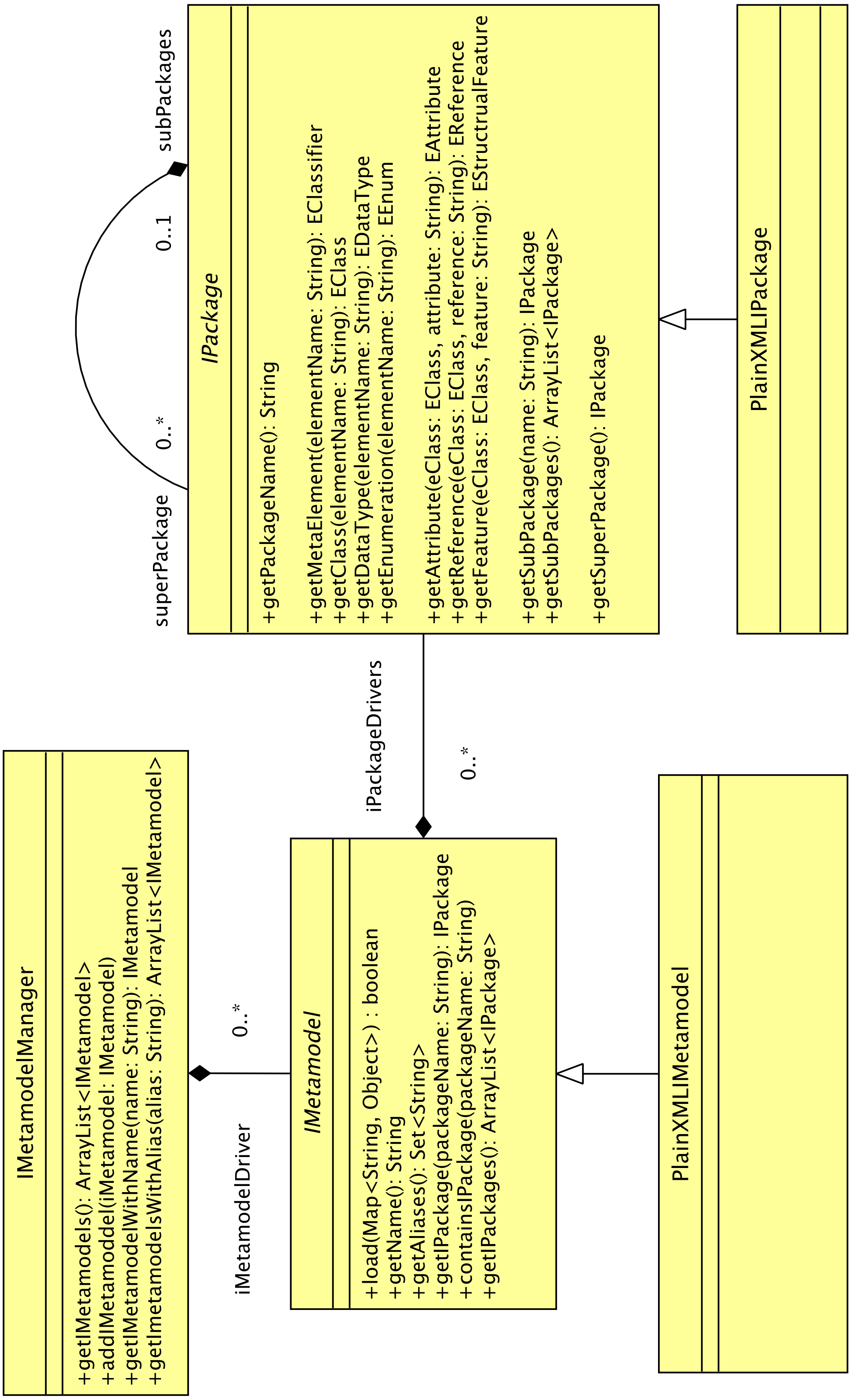


Figure 9.1.: The XML driver for ESAMC

The majority of contemporary MDE tools focus on 3-level metamodelling architectures (Section 2.1.5) where models conform to metamodels which are defined in terms of architecture/framework-specific metamodelling languages such as MOF or Ecore. Thus, most contemporary model management languages/tools require models to be defined by such architectures. In practice, however, many modelling tools do not use MOF/Ecore to manage and store their models. In [112], the authors identify the need for Epsilon to provide support of the management of XML models.

In order to support the static analysis of programs that manage XML models, it is necessary to implement a corresponding XML model driver for *ESAMC*. Therefore, an XML driver was developed by extending ESAMC as illustrated in Figure 9.1, where *PlainXMLIMetamodel* was created by extending *IMetamodel*, and *PlainXMLIPackage* was created by extending *IPackage*.

9.1.1. Epsilon's Rules of Accessing Schema-less XML

This section discusses the syntax that Epsilon uses to query XML models. Epsilon provides a set of naming conventions for accessing different constructs within an XML model. Listing 9.1 provides an example XML document which describes a *library* and its *books*.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<library>
  <book title="EMF Eclipse Modeling Framework" pages="744">
    <author>Dave Steinberg</author>
    <author>Frank Budinsky</author>
    <author>Marcelo Paternostro</author>
    <author>Ed Merks</author>
    <published>2009</published>
  </book>
  <book title="Eclipse Modeling Project:
      A Domain-Specific Language (DSL) Toolkit" pages="736">
    <author>Richard Gronback</author>
```

```
    <published>2009</published>
  </book>
  <book title="Official Eclipse 3.0 FAQs" pages="432">
    <author>John Arthorne</author>
    <author>Chris Laffra</author>
    <published>2004</published>
  </book>
</library>
```

Listing 9.1: An Example Plain XML Document

Epsilon provides a set of prefixes that enable developers to access and query XML models in a concise manner. Firstly, it uses the *t_* prefix to emulate types in XML models, which is necessary given that XML models do not have corresponding metamodels. In the example EOL program in Listing 9.2, the *t_* prefix is used in line 2 to access the type *book*.

```
1 model XMLDoc alias xml driver XML {path = "library.xml"};
2 var books = t_book.all;
3 for(b in books) {
4   b.a_title.println();
5   for(author in b.c_author) {
6     author.text.println();
7   }
8   b.i_pages.println();
9 }
```

Listing 9.2: An example EOL program that manages a XML model

To access attributes, Epsilon provides five prefixes. The *a_* prefix is used to access an attribute's string value (*a* is the shorthand for *attribute*). The *b_* prefix is used to access an attribute but the type of its value is explicitly cast to Boolean (*b* is the shorthand for *boolean*). The *i_*, *r_*, *s_* prefixes work in the similar way, and cast the type of the attribute in question to Integer, Real and String respectively. Atop these prefixes, for every *type* in the XML model, there is a *tagName* attribute which returns the name of the *element* tag in the XML model, and a *text* attribute which returns the content between the opening tag and the closing tag of an *element*. Listing 9.2 illustrates the usage of the *a_* prefix in line 4, the *text* attribute in line 6 and the *i_* prefix in line 8.

To access references, Epsilon provides two prefixes. The *e_* prefix is used to access a single-valued containment reference, where the *c_* prefix is used to access a multi-valued containment reference. Listing 9.2 illustrates the usage of the *c_* prefix in line 5. For each *element* in the XML document, there is also a reference named *parentNode* which accesses the container of the current *element* under question. Finally, there is a reference named *children* which accesses the contained *element*s of the *element* under question.

9.1.2. Constructing Ecore metamodels from XML models

To compensate for the lack of a metamodel, or of an equivalent artefact from which a metamodel can be inferred (e.g. an XML Schema), the XML driver is able to create an Ecore metamodel by analysing a sample XML document.

To access an XML model, the EOL developer needs to provide its location, from which *PlainXMLIMetamodel* can infer an Ecore *metamodel*. Consider the example XML document in Listing 9.1. From this document, *PlainXMLIMetamodel* infers an Ecore metamodel using the algorithm provided in Algorithm 1-3. Thus, for the example XML document provided in Listing 9.1, the construction of the Ecore metamodel follows the steps below;

- Lines 2-20 form the *root* of the document, from which an EClass named *t_library* is created. As there is no *attribue* in line 2, no attribute is added to the *t_library* EClass. However, EOL supports querying the element name by the *tagName* attribute and the value of the *element* by the *text* attribute. Thus, *EAttributes* *tagName* of type *EString* with cardinality 1, and *EAttribute text* of type *EString* with cardinality 1, are created and added to the *t_library* EClass;
- The children nodes of *root* are iterated. The name of the first child node is *book* in line 3-9. An *EClass* named *t_book* is created and an *EReference* named *book*

```
let document = the XML Document parsed by DOM;
let root = the root of the document; let p = EPackage created with the name of the document;
createEClass(root);
function createEClass(element: Element) : EClass
    let name = "t_" + element.nodeName;
    let result = EClass to be created;
    /* create an attributeMap for this EClass, which is used to check if there are
       reocurring attributes                                                      */
    let attributeMap = new Map<String, EAttribute>;
    if p contains EClass with name then
        result = p.getEClass(name);
    end
    else
        result = create new EClasss with name;
        p.add(result);
    end
    let tagName = new EAttribute with name "tagName" of type EString;
    tagName.upperBound = 1;
    let text = new EAttribute with name "text" of type EString;
    text.upperBound = 1;
    add tagName and text to result;
    foreach attribute attr in element do
        createEAttribute(result, attr, attributeMap);
    end
    /* create an referenceMap for this EClass, which is used to check if there are
       reocurring attributes                                                      */
    let referenceMap = new Map<String, EReference>;
    foreach node in element.getChildNodes() do
        if node instance of Element then
            createEReference(result, node, referenceMap)
        end
    end
end
foreach EClass eClass in p do
    if result.allReferences().size() == 1 then
        let childrenReference = new reference with name "children";
        childrenReference.upperBound = -1; childrenReference.eType = type of the first
        EReference;
        add childrenReference to result;
    end
    else
        let childrenReference = new reference with name "children";
        childrenReference.upperBound = -1;
        add childrenReference to result;
        let leastCommonType = the least common super type of the eTypes of all EReferences
        of result;
        if leastCommonType != null then
            childrenReference.eType = leastCommonType;
        end
        else
            childrenReference.eType = null;
        end
    end
end
```

Algorithm 1: Ecore Metamodel Creation from Plain XML (1 of 3)

of type *t_book* with cardinality 1 is added to *t_library*. It is worth noting that the name of the reference does not contain a *prefix*, because the XML driver only uses

```
function createEReference(eClass: EClass, reference: Node, referenceMap: Map)
    if node instance of Element then
        let refType = createEClass(node);
        let reference = get reference from referenceMap by the name of refType;
        /* if reference is null, it means it has not ocurred before            */
        if reference == null then
            /* reference names are not created with prefixes                   */
            reference = new reference with node name and refType;
            reference.upperBound = 1;
            add reference to referenceMap;
        end
        else
            reference.upperBound = -1;
        end
        add reference to eClass;
        /* calculate the parentNode, if there is a parentNode for thie EClass,
           check types, if types don't match, set eType of parentNode to null  */
        let parentNode = get reference from EClass;
        if parentNode == null then
            parentNode = new EReference with name "parentNode";
            parentNode.eType = eClass;
        end
        else
            eType = parentNode.eType;
            if eType != EClass then
                parentNode.eType = null;
            end
        end
        parentNode.upperBound = 1;
        add parentNode to refType;
    end
end
```

Algorithm 2: Ecore Metamodel Creation from Plain XML (2 of 3)

prefixes internally for attributes and references for type comparison and cardinality comparison;

- The *attribute*s of *book* in line 3 are then iterated, which results in the creation of an *EAttribute* named *title* of type *EString* with cardinality 1, and the creation of *EAttribute* named *pages* of type *EInt* with cardinality 1. The created *EAttribute*s are added to the *EClass t_book*. Default *EAttributes tagName* and *text* are also created. Apart from all the *EAttribute*s, a default *EReference* named *parentNode* of type *t_library* with cardinality 1 is created and added to *t_book*;

- The children nodes of *book* in line 3 are then iterated, resulting in the creation of an *EClass* named *t_author* with its default *EAttribute*s *tagName* and *text*. Then an *EReference* is created with its name set to *author* and its *eType* to *t_author* and

```
function createEAttribute(eClass: EClass, attribute: Node, attributeMap: Map)
    let attrName = attr.getNodeName(0);
    let value = attr.getNodeValue();
    let eAttribute = new EAttribute with attrName;
    eAttribute.upperBound = 1; if value.equals("true") or value.equals("false") then
        eAttribute.setEType(EBoolean);
    end
    else if value instanceof Integer then
        eAttribute.setEType(EInt);
    end
    else if value instanceof Float then
        eAttribute.setEType(EFloat);
    end
    else if value instanceof Double then
        eAttribute.setEType(EDouble);
    end
    else if value instanceof String then
        eAttribute.setEType(EString);
    end
    /* if attributeMap contains the attribute, set cardinality to many          */
    if attributeMap contains attrName then
        let _eAttribute = eClass.getEAttribute(attrName);
        /* if existing attribute has the same type as the attribute inferred, set
           cardinality to many                                                  */
        if _eAttribute.eType.equals(eAttribute.eType) then
            if _eAttribute.upperBound == 1 then
                _eAttribute.upperBound = -1;
            end
        end
        /* if existing attribute has a different type, change the existing
           attribute to the type inferred, set cardinality to many              */
        else
            let _eType = _eAttribute.eType;
            _eAttribute.eType = __eType;
            _eAttribute.upperBound = -1;
        end
    end
    /* if attributeMap does not contain the attribute, add to attributeMap      */
    else
        add eAttribute to attributeMap;
    end
    add eAttribute to eClass;
end
```

Algorithm 3: Ecore Metamodel Creation from Plain XML (3 of 3)

added to *t_book*. For *EClass t_author*, a default *EReference* named *parentNode* of type *t_book* with cardinality 1 is created and added to *t_author*;

- Because *t_author* appears more than once within the *EClass t_book*, the cardinality of the *EReference* named *author* for *t_book* is changed to -1 (unbounded);
- An *EClass* named *t_published* is created with its default *tagName* and the *text EAttribute*. An *EReference* named *parentNode* of type *t_book* with cardinality 1 is

created and added to *t_published*;

- When the *book* in line 10 is processed, it is determined that the *EReference* named *book* for *EClass t_library* appears more than once, therefore, the cardinality of the *EReference* is changed to -1 (unbounded);

- The rest of the elements are processed in a similar manner. At the end of the file, for each *EClass* calculated, its *children EReference* is computed. If the *EClass* under question has only one *EReference*, the *children*'s eType is the eType of the *EReference*; if the *EClass* has more than one *EReferences* (and the eType of these *EReferences* are different), the *children's* eType is *Any*.

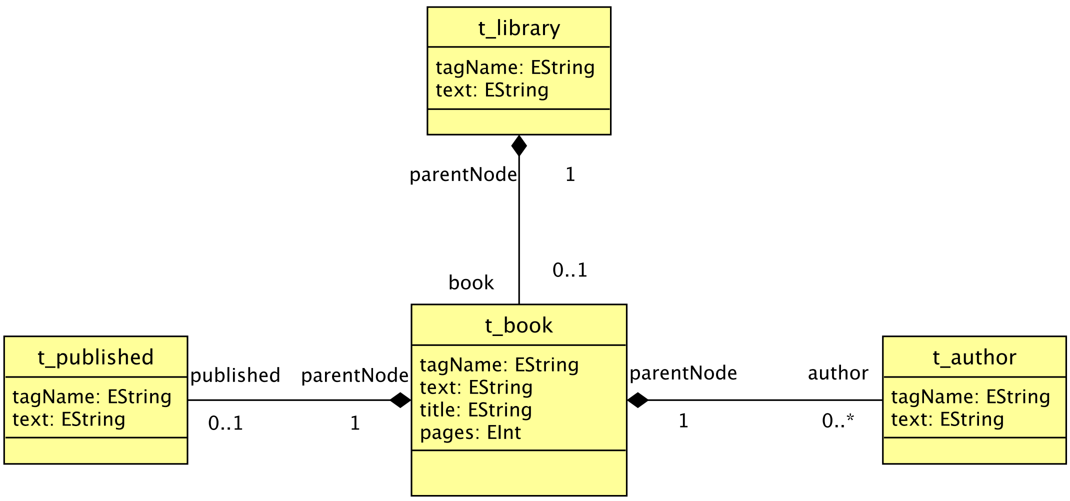


Figure 9.2.: Ecore metamodel generated from XML document in Listing 9.1

The inferred Ecore metamodel is shown in Figure 9.2. It is noteworthy that all the *EReferences* generated are containment references (they directly contain elements instead of referring to elements defined elsewhere). The only exception is the *parentNode* reference for all *EClass*es, which is introduced to support navigation to an *element*'s container.

In addition, the generation of the Ecore metamodel is based on the assumptions that, for an XML document, each *element* in the DOM is mapped to an *EClass*, the *attributes* defined within the *element* tag are mapped to *EAttributes*, and any *elements* contained with the *element* are mapped to *EReferences*.

Because there is no typing scheme in XML models, the creator of the XML models has great flexibility. As such, there are a number of corner cases which the XML driver has to cater for. The first corner case is when attributes are of different types as illustrated in Listing 9.3. Up until line 9, the inferred type of the *EAtrribute pages* of *EClass t_book* is *Integer* because the value of the *pages* attribute in line 3 can be parsed as an Integer. However, as the value of the *pages* attribute in line 11 is not a valid integer, the XML driver needs to change the type of the *EAttribute pages* of *EClass t_book* to *EString*. If the developer tries to access the *pages* attribute of *t_book* as an Integer, they will receive an error from the static analyser.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <library>
3    <book title="EMF Eclipse Modeling Framework" pages="744">
4      <author>Dave Steinberg</author>
5      <author>Frank Budinsky</author>
6      <author>Marcelo Paternostro</author>
7      <author>Ed Merks</author>
8      <published>2009</published>
9    </book>
10   <book title="Eclipse Modeling Project:
11       A Domain-Specific Language (DSL) Toolkit" pages="many">
12   ...
13   </book>
14 </library>
```

Listing 9.3: An example corner case 1

The second corner case is when a type is contained in different types of parents. Listing 9.4 illustrates such case, in lines 3-5, *EClass* named *t_book* is created, an *EReference* named *parentNode* is created with its *eType* set to *t_library*. However, later in the XML file, when lines 6-9 are processed, the *t_book* element has a different container. As such the type of the *parentNode EReference* needs to be set to Any.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<library>
  <book title="EMF Eclipse Modeling Framework" pages="744">
    ...
  </book>
  <aisle title="box of MDE" id="1">
    <book title="Eclipse Modeling Project:
      A Domain-Specific Language (DSL) Toolkit" pages="many">
  </aisle>
  ...
</library>
```

Listing 9.4: An example corner case 2

9.1.3. Integration of the Plain XML Driver with the EOL Static Analyser

Misuses of prefixes result in warnings being produced by the EOL static analyser. Figure 9.3 illustrates a number of misuses of prefixes and how the static analyser reacts to them. In line 5, the *i_* prefix is used to access the *EAttribute title* for the type *t_book*. However, in the extracted Ecore metamodel, the *eType* of *EAttribute title* is *EString*, thus, a warning is produced. In line 9, the *b_* prefix is used to access *EAttribute pages*. However, *EAttribute pages* is of type *EInt*, accessing a *Integer* value using the *b_* is considered a misuse, therefore a warning is produced.

```
1  model XMLDoc alias xml driver XML {path = "library.xml"};
2
3  var books = t_book.all;
4  for(b in books) {
5      b.i_title.println();
6      for(author in b.c_author) {
7          author.text.println();
8      }
9      b.b_pages.println();
10 }
```

Figure 9.3.: Warnings generated by the EOL static analyser for misuses of prefixes.

The EOL static analyser is also able to detect errors when the developer attempts to access features of an element that are not defined in the sample XML model. In Figure 9.4, errors are injected in the program such that in line 5, an attempt is made to access a feature named *titles* (which is not defined in the XML model). A similar error is injected in line 9. The EOL static analyser detects these illegal property calls and generates appropriate errors markers.

```
1  model XMLDoc alias xml driver XML {path = "library.xml"};
2
3  var books = t_book.all;
4  for(b in books) {
5      b.i_titles.println();
6      for(author in b.c_author) {
7          author.text.println();
8      }
9      b.r_page.println();
10 }
```

Figure 9.4.: Errors generated by the EOL static analyser for accessing undefined features.

The EOL static analyser is also able to detect the misuse of prefixes for the cardinalities of features. In Figure 9.5, an error is injected in line 6. Instead of using the *c_* prefix (for accessing collections), the *e_* prefix is used, which is used to access single-valued references. The EOL static analyser detects this and generates an error.

```
1  model XMLDoc alias xml driver XML {path = "library.xml"};
2
3  var books = t_book.all;
4  for(b in books) {
5      b.i_title.println();
6      for(author in b.e_author) {
7          author.text.println();
8      }
9      b.r_pages.println();
10 }
```

Figure 9.5.: Errors generated by the EOL static analyser for accessing features with inappropriate prefixes.

9.1.4. Analysing EOL programs that manage models defined in EMF and schema-less XML

One of the contributions of this thesis is the support for static analysis of model management programs that manage models defined in diverse technologies. In this section, the static analyser is used to analyse a model transformation program written in the Epsilon Transformation Language (ETL), which transforms a schema-less XML-based *university* model (as shown in Listing 9.5) to an EMF-based model which conforms to the *University* model presented in Section 2.1.6 (Figure 2.9).

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <university name="UoY">
3    <department name="CS">
4      <student first_name="Cathy" last_name="Smith"/>
5      <student first_name="Carl" last_name="Smoka"/>
6      <lecturer first_name="Tom" last_name="Brown"/>
7      <lecturer first_name="Leo" last_name="James"/>
8      <module name="MODE">
9      </module>
10     <module name="TPOP">
11     </module>
12   </department>
13   <department name="Psychology">
14   </department>
15 </university>
```

Listing 9.5: A schema-less XML based university model

In Listing 9.5, a sample *university* model is displayed. In line 2, a *university* element is defined with the name *UoY*, and line 3 defines a child element *department* with the name *CS*. Lines 4 and 5 define two child elements (*student*) of element *department*, with their *first_name*s and *last_name*s. Lines 6 and 7 define two child elements (*lecturer*) of element *department*, with their *first_name*s and *last_name*s. Lines 8-11 define two *module*s with their names. Lines 13-14 define another department named *Psychology* which contains no children.

An ETL program (Listing 9.6) is created to transform the model defined in Listing 9.5 to an EMF model that conforms to the *University* metamodel presented in Section 2.1.6 (Figure 2.9). In lines 5-10 of Listing 9.6, a transformation rule is defined which transforms each *university* element in the XML model to a *University* by copying the *name* of the *university* element in the XML model, and by transforming its *department*s into instances of *Department* in the *University* metamodel (via the *equivalent()* method call in line 9). In lines 11-17, a transformation rule is defined which transforms each *department* element in the XML model into a *Department* in the *University* EMF model, by copying the *name* attribute of the *department* element in the XML model and by transforming the *student* and *lecturer* child elements of the *department* element in the XML model into instances of *Student* and *Lecturer* in the EMF model. Lines 18-23 define a rule which transforms each *student* element in the XML model into a *Student* in the EMF model. Finally, lines 24-39 define a rule which transforms each *lecturer* element in the XML model into a *Lecturer* in the EMF model.

```
1  model XMLDoc driver XML
2                 {path = "university.xml"};
3  model University driver EMF
4                 {nsuri = "http://university/1.0"};
5  rule xml_university2University
6  transform xml_u : XMLDoc!t_university
7  to e : University!University {
8    e.name = xml_u.s_name;
9    e.departments = xml_u.c_department.equivalent();
10 }
11 rule xml_department2Department
12 transform xml_d : XMLDoc!t_department
13 to d : University!Department {
14   d.name = xml_d.s_name;
15   d.members.addAll(xml_d.c_student.equivalent());
16   d.members.addAll(xml_d.c_lecturer.equivalent());
17 }
18 rule xml_student2Student
19 transform xml_s : XMLDoc!t_student
```

```
to s : University!Student {
  s.first_name = xml_s.a_first_name;
  s.last_name = xml_s.a_last_name;
}
rule xml_lecturer2Lecturer
transform xml_l : XMLDoc!t_lecturer
to l : University!Lecturer {
  l.first_name = xml_l.a_first_name;
  l.last_name = xml_l.a_last_name;
}
```

Listing 9.6: An ETL M2M transformation which transforms an XML model to an EMF model

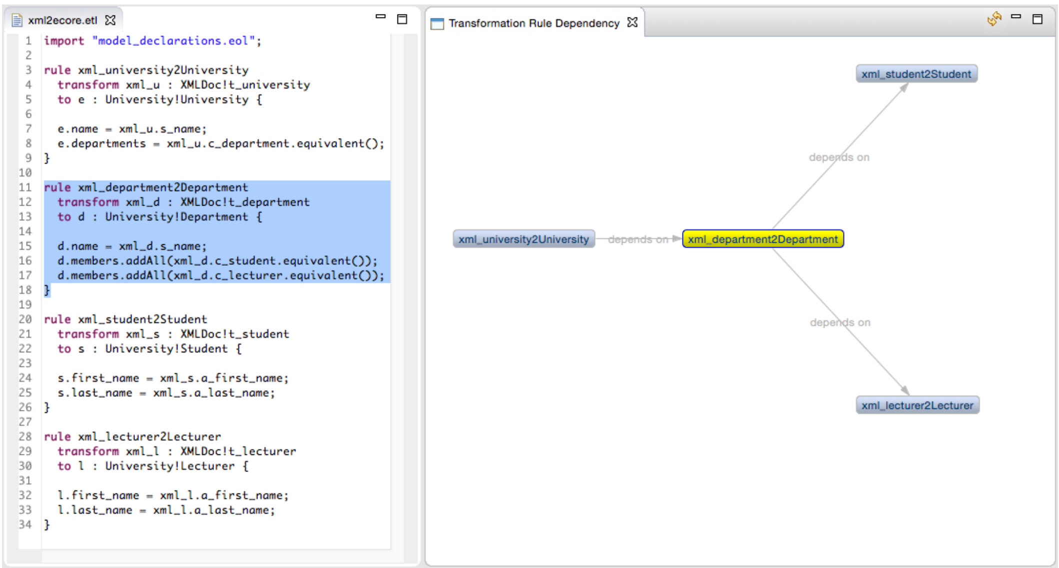


Figure 9.6.: Transformation rule dependency graph for the program in Listing 9.6.

The ETL static analyser is able to analyse the transformation and compute the transformation rule dependency graph (Discussed in Section 8.2.5) illustrated in Figure 9.6. Transformation rule *xml_university2University* delegates the creation of *Department* elements to transformation rule *xml_department2Department*, which in turn delegates the creation of *Student* and *Lecturer* elements to rules *xml_student2Student* and *xml_lecturer2Lecturer*.

The example above illustrates a copy transformation: the XML model has a structure that is similar to the EMF *University* model. To further evaluate the XML driver, another example is provided, which transforms an XML based *tree* model into an EMF based *Graph* model.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<tree name="root">
  <tree name="node1">
    <tree name="node2">
    </tree>
    <tree name="node3">
    </tree>
  </tree>
</tree>
```

Listing 9.7: A XML based tree model

The XML based *tree* model is shown in Listing 9.7. It is composed of *tree* elements, which have an attribute called *name*. A *tree* element can contain an arbitrary number of *tree* children.

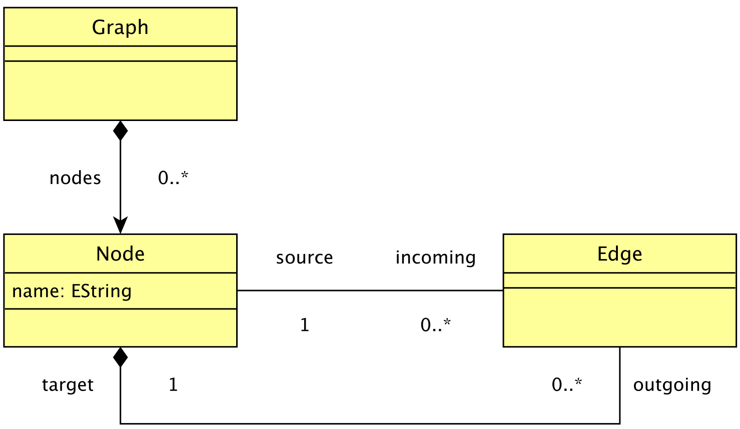


Figure 9.7.: The Graph metamodel

The EMF-based *Graph* metamodel is provided in Figure 9.7. A *Graph* type is composed of *Node*s, which have a number of *incoming* and *outgoing* *Edge*s. An *Edge* connects two *Node*s, identified as *source* and *target*.

Listing 9.8 shows the ETL transformation which transforms *tree* elements into *Node*s and *Edge*s that connect the *Node*s. In line 7, the name of the *tree* element (xml_t.s_name) is copied and assigned to the name of the *Node* created. If the *tree* element has a *parentNode*, the transformation creates an *Edge* (line 9) and resolves the *parentNode* by calling the *equivalent()* method (line 10). The *Edge* then connects the resolved *Node* and *n*.

```
1  import "declare_models.eol";
2
3  rule tree2Graph
4  transform xml_t : XMLDoc!t_tree
5  to n : Graph!Node {
6
7    n.name = xml_t.s_name;
8    if(xml_t.parentNode.isDefined()) {
9      var e: new Graph!Edge;
10     e.source = xml_t.parentNode.equivalent();
11     e.target = n;
12   }
13 }
```

Listing 9.8: Transforming an XML-based *tree* model into an EMF based *Graph* model

Figure 9.8 shows the computed the transformation rule dependency graph for the program. To illustrate that the static analysis checks for misuse of *prefix*es in XML models, when the *prefix* of *name* in line 7 is changed to *i_*, an appropriate warning is produced.

9.1.5. Summary

In this section, the extensibility of the ESAMC was evaluated through the implementation and evaluation of the XML driver. The implementation of the XML driver demon-

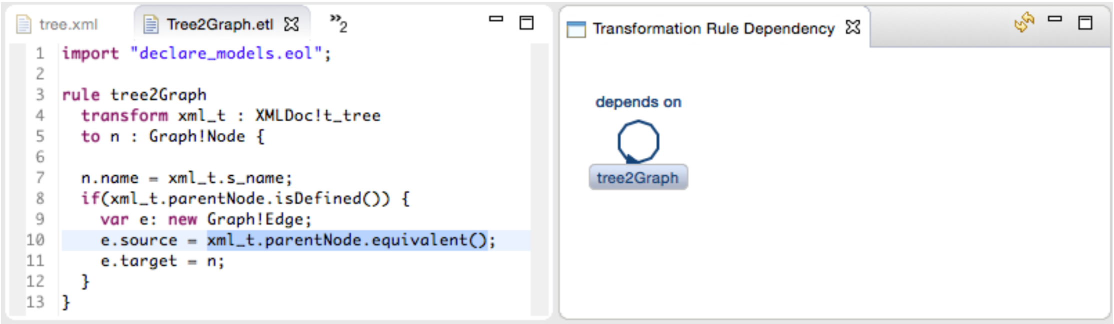


Figure 9.8.: Transformation rule dependency graph for the program in Listing 9.8.

strates that the Epsilon static analysis framework is able to analyse programs that manage models defined in different modelling technologies within a single model management program.

9.2. Evaluating the EOL, EVL and ETL Static Analysers

This section discusses the evaluation of the capabilities of the EOL, EVL and ETL static analysers in terms of their abilities to detect errors. The evaluation of the Epsilon static analysis framework is carried out in three different stages.

9.2.1. Evaluating the EXL Metamodels and the AST2EXL Transformations

The first step of the evaluation was conducted by evaluating the EOL, EVL and ETL metamodels and the AST2EOL, AST2EVL and AST2ETL transformations. The EOL, EVL and ETL metamodels have been maintained throughout the entire development process as an increasing number of programs written in these languages from the Epsilon labs[1] have been analysed by the static analysis framework. The EOL metamodel was developed based on the study of the EOL parser of Epsilon. As previously mentioned, Epsilon defines an ANTLR-based grammar for EOL, which was used as a guide for the development of the EOL metamodel. The EOL metamodel is iteratively validated by comparing its structure with the EOL grammar provided by Epsilon.

A set of unit tests have also been developed for each *EOLElementCreator* within the

[1] `https://github.com/epsilonlabs/epsilonlabs`

AST2EOL transformation facility[2]. These unit tests were executed as regression tests whenever changes were made to the EOL metamodel and the AST2EOL transformation. In addition, a pretty printer facility[3] was also developed to evaluate the EOL metamodel and the AST2EOL facility. The pretty printer facility was built on the EOL visitor framework. Essentially it is a model-to-text transformation, which transforms EOL models (obtained from the AST2EOL transformation) into Strings. Manual code reviews are performed on the Strings generated by the pretty printer to verify that Strings generated are identical to the original EOL programs.

The complex EuGENia transformation[4] was also used to evaluate the EOL metamodel and the AST2EOL facility. The validation approach presented above were also applied to the components for the EVL and ETL static analysers.

9.2.2. Evaluating the EOL Static Analyser

A set of EOL programs have been created to evaluate the EOL static analyser[5]. These EOL programs target the operations defined in the standard library and cover different scenarios in which the operations are called, within which warnings/errors are deliberately injected in the EOL code to test the capabilities of the EOL static analyser. These programs have also been used as regression tests whenever the EOL static analyser was changed.

To test the EOL static analyser in realistic scenarios, it is also useful to analyse existing EOL programs that have been proved to work correctly and are used extensively. Thus, a number of existing EOL programs (from Epsilon labs and the examples provided by Epsilon) have been analysed using the EOL static analyser. Among the EOL programs, the most complex and mature program analysed was the EuGENia transformation. Although the EuGENia transformation has been heavily tested, a number of warnings and errors were still found by the EOL static analyser.

[2] `https://github.com/epsilonlabs/epsilonlabs/tree/master/org.eclipse.epsilon.eol.ast2eol`
[3] `https://github.com/epsilonlabs/epsilonlabs/tree/master/org.eclipse.epsilon.eol.visitor.printer`
[4] `https://epsilonblog.wordpress.com/2009/06/15/eugenia-polishing-your-gmf-editor/`
[5] `https://github.com/epsilonlabs/epsilonlabs/tree/master/StaticAnalysis/org.eclipse.epsilon.static.analysis.tests`

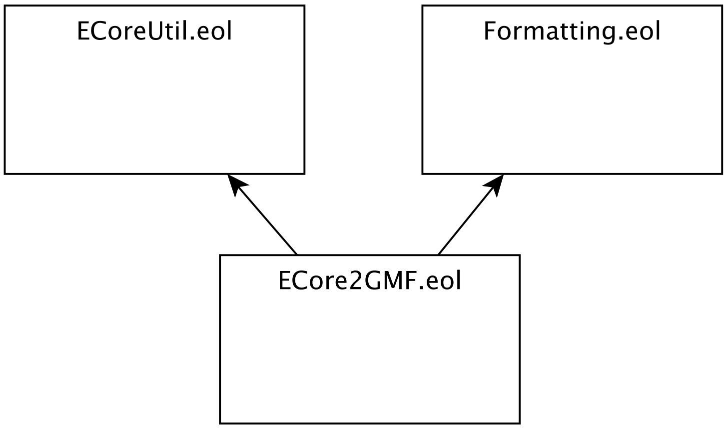


Figure 9.9.: The structure of the EuGENia transformation.

EuGENia is a tool that automatically generates the .gmfgraph, .gmftool and .gmfmap models needed to implement a GMF editor from a single annotated Ecore metamodel. The EuGENia transformation comprises three parts. The actual transformation is described in the ECore2GMF.eol file, which imports two additional EOL files: ECoreUtil.eol which provides utility operations for Ecore, and Formatting.eol which provides formatting operations that convert entities in Ecore into entities in GmfGraph.

Analysing ECoreUtil.eol

The EOL static analyser generates 17 warnings and 2 errors after analysing ECoreUtil.eol (467 lines of code). The warnings are all related to expressions having *AnyType* (for example, the warning in line 168 in Figure 9.10, in which the expression *featureName* is of type *AnyType*). In lines 154-164, an error is produced as the *getNodeSize()* operation with exactly the same signature is already defined in lines 142-152. This feedback is illustrated in Figure 9.10. In line 413, an error is reported to the property call

```
self.eAllStructuralFeatures
```

In this expression, the type of the variable *self* is ECore!EModelElement (as *self* is an instance of the context type of the operation. In the ECore metamodel, model element type *EModelElement* does not have a feature named “eAllStructuralFeatures”, hence the error is reported. This code works fine at runtime as the operation only happens to be

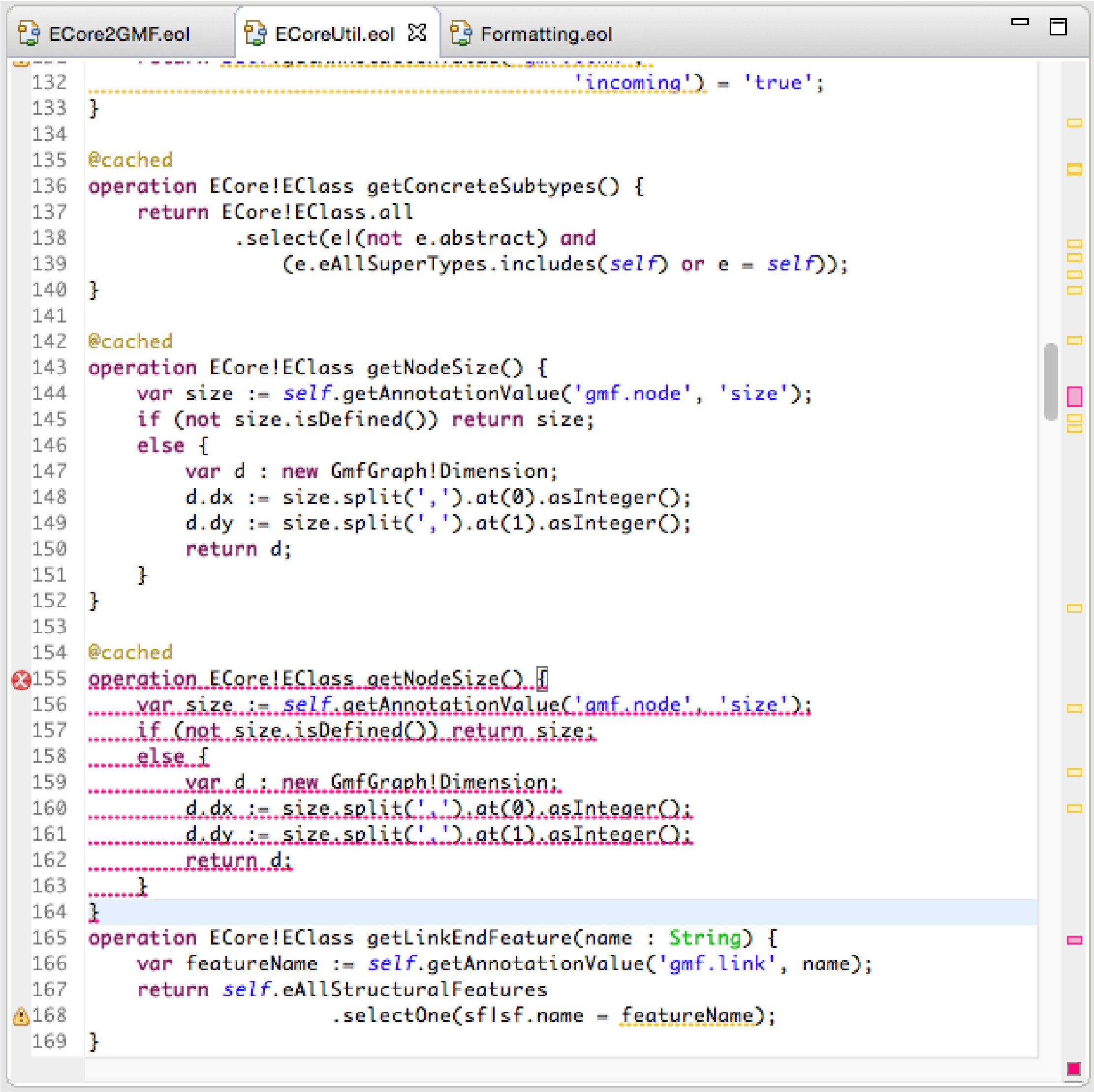


Figure 9.10.: Analysis on ECoreUtil.eol (1 of 2)

invoked on instances of EClass (which is a sub-type of *EModelElement*). This feedback is illustrated in Figure 9.11.

Analysing Formatting.eol

Analysing Formatting.eol (267 lines of code) uncovers 2 warnings and 4 errors. The warnings are both related to expressions of type *AnyType*. The first two errors are in Figure 9.12, and are reported because the variable *figureGallery* cannot be located by

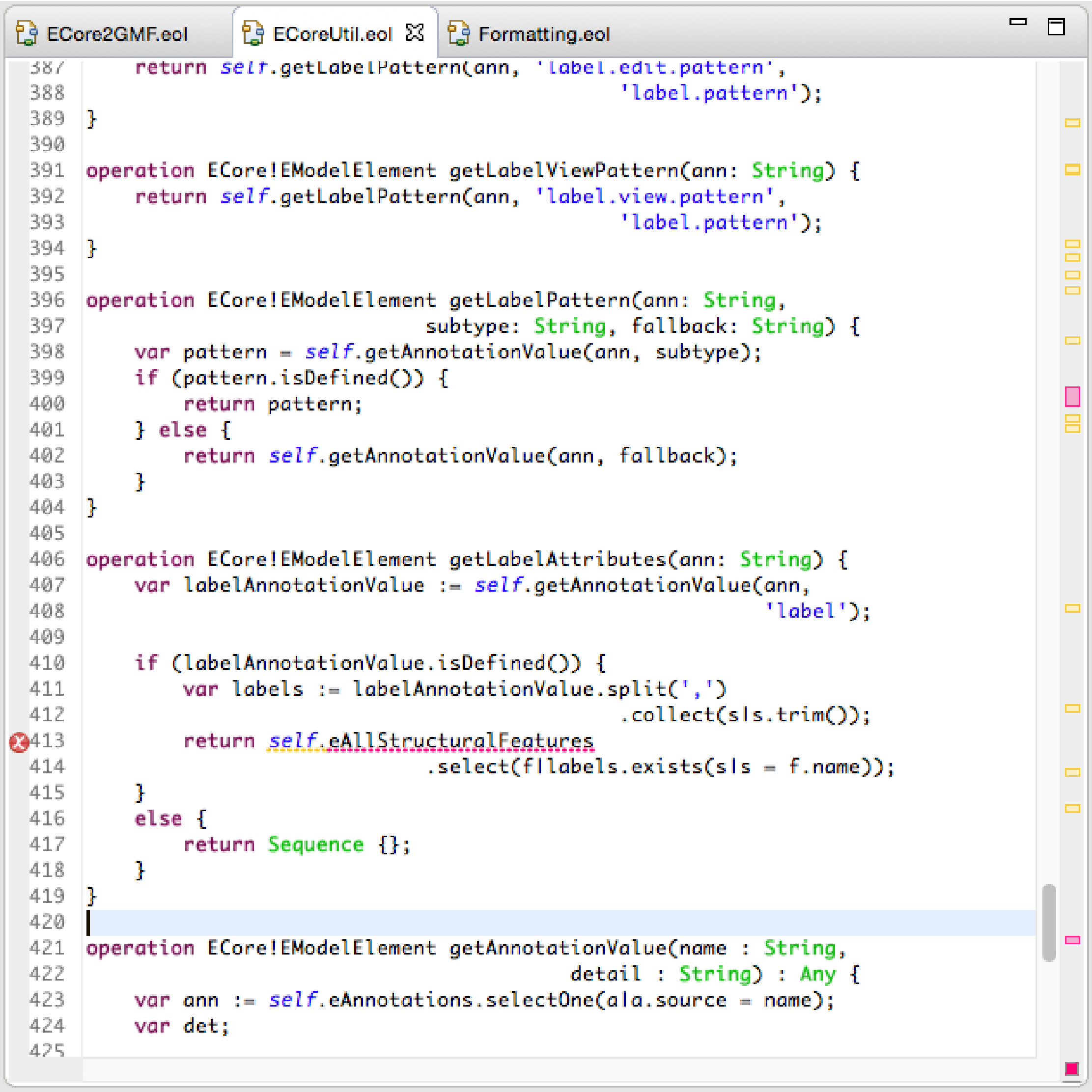


Figure 9.11.: Analysis on ECoreUtil.eol (2 of 2)

the static analyser. Performing manual code inspection reveals that the variable *figureGallery* is defined in ECore2GMF.eol. Figure 9.9 shows that ECore2GMF.eol imports Formatting.eol, but not the other way around, hence, as a standalone EOL file analysed by the static analyser, as far as the static analyser is concerned, the variable *figureGallery* cannot be located.

This error reveals a potential problem in the EOL runtime because it is not ideal for an imported program to depend on the program which imports it.

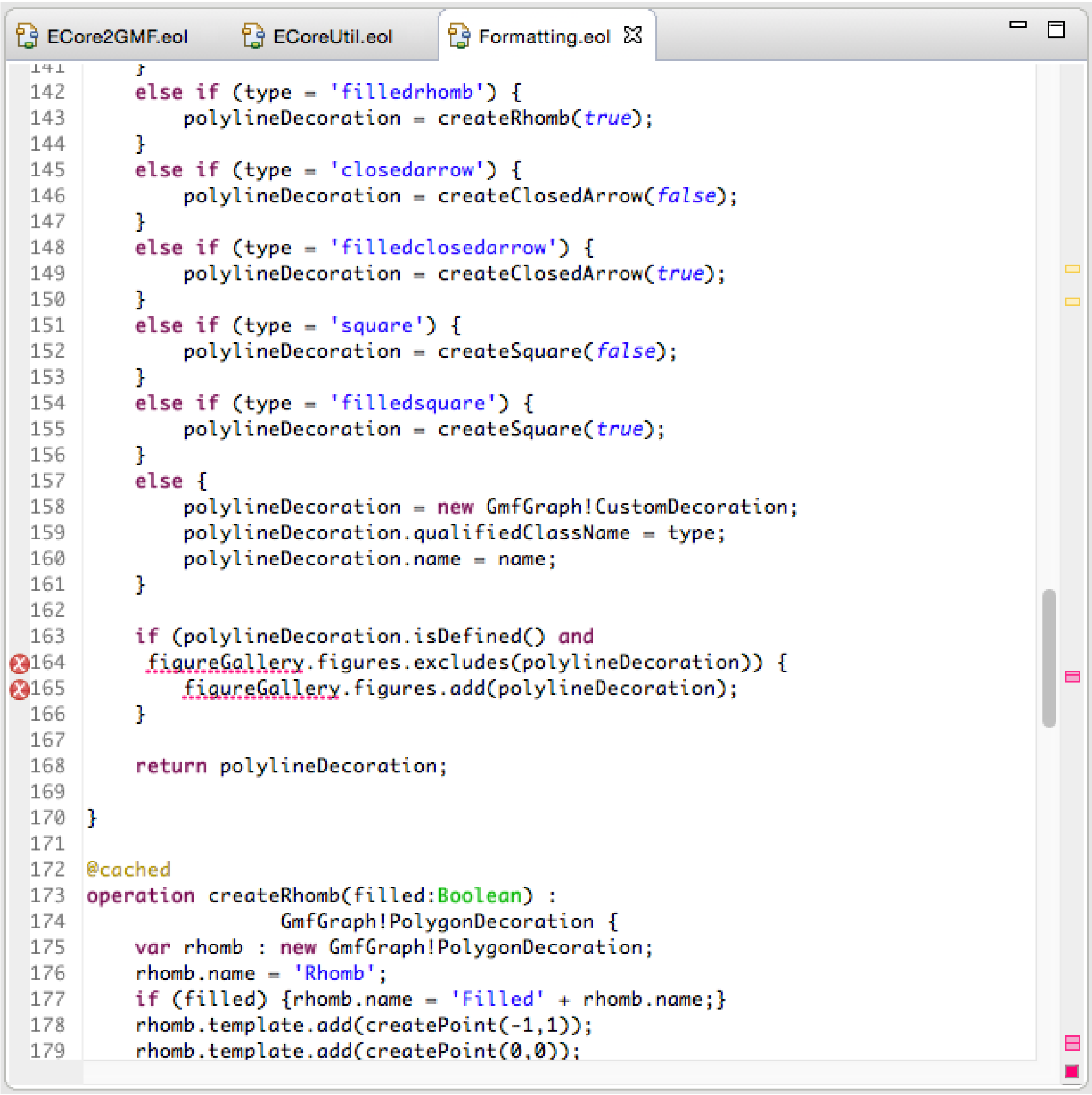
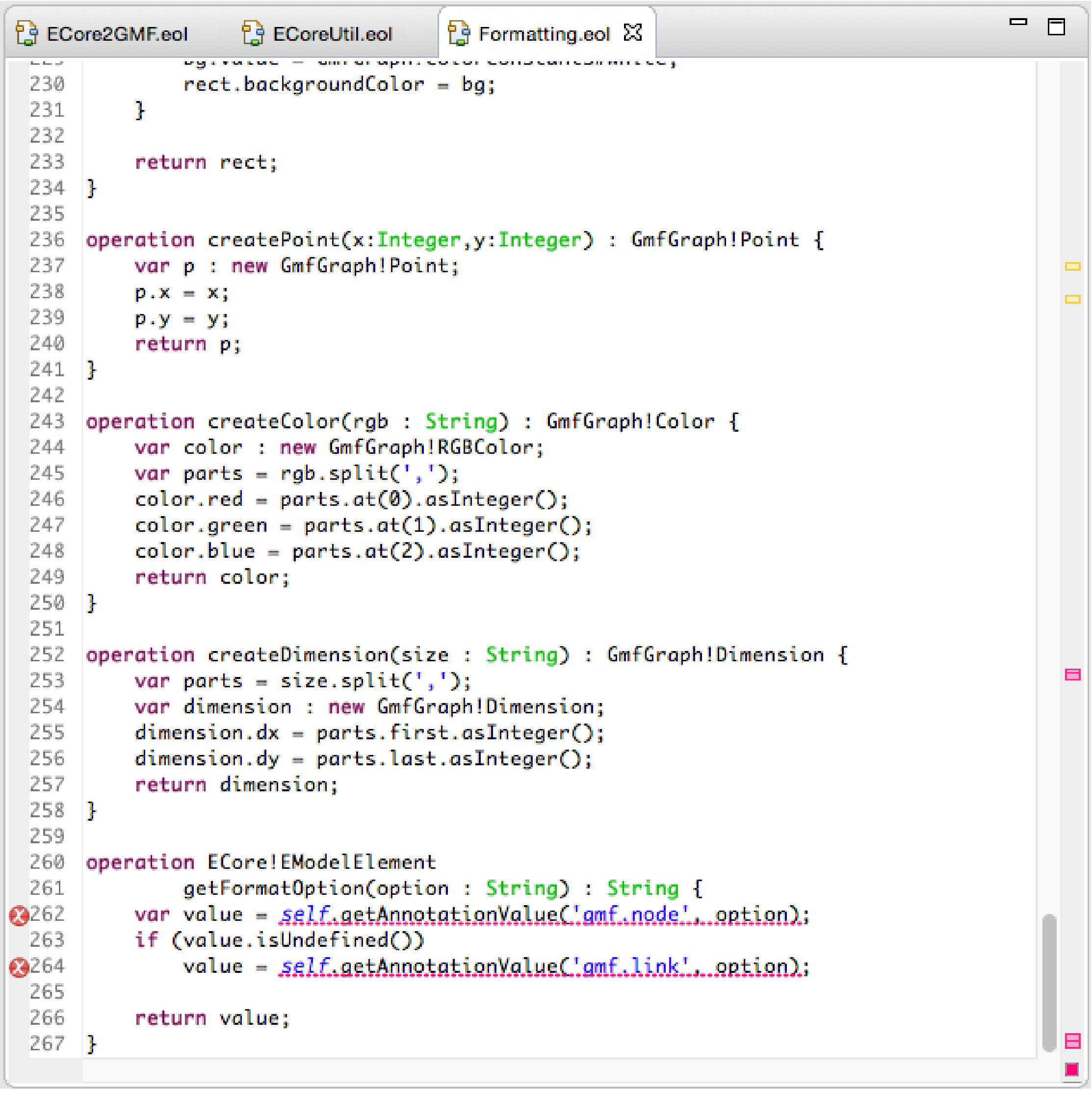


Figure 9.12.: Analysis on Formatting.eol (1 of 2)

The rest of the errors for Formatting.eol are of the same nature as illustrated in Figure 9.13. In lines 262 and 264, the program attempts to call the operation *getAnnotationValue()*, which is not defined in Formatting.eol, but in ECoreUtil.eol.

Analysing ECore2GMF.eol

The analysis of ECore2GMF.eol (614 lines of code) generates 97 warnings and 1 error. The warnings are all related to expressions of type *AnyType*. The error generated is




```
        rect.backgroundColor = bg;
    }

    return rect;
}

operation createPoint(x:Integer,y:Integer) : GmfGraph!Point {
    var p : new GmfGraph!Point;
    p.x = x;
    p.y = y;
    return p;
}

operation createColor(rgb : String) : GmfGraph!Color {
    var color : new GmfGraph!RGBColor;
    var parts = rgb.split(',');
    color.red = parts.at(0).asInteger();
    color.green = parts.at(1).asInteger();
    color.blue = parts.at(2).asInteger();
    return color;
}

operation createDimension(size : String) : GmfGraph!Dimension {
    var parts = size.split(',');
    var dimension : new GmfGraph!Dimension;
    dimension.dx = parts.first.asInteger();
    dimension.dy = parts.last.asInteger();
    return dimension;
}

operation ECore!EModelElement
        getFormatOption(option : String) : String {
    var value = self.getAnnotationValue('gmf.node', option);
    if (value.isUndefined())
        value = self.getAnnotationValue('gmf.link', option);

    return value;
}
```

Figure 9.13.: Analysis on Formatting.eol (2 of 2)

shown in Figure 9.14. In lines 530 and 531, the property "referencedChild is called on variable *self*. The property call is illegal because *self* is of type *GmfMap!NodeReference*. Although *GmfMap!NodeReference* is an abstract type, it has two sub types, which are *GmfMap!ChildReference* and *GmfMap!TopNodeReference*. Of the two sub types, only *GmfMap!ChildReference* defines a feature named "referencedChild". Hence, the source code is incorrect, as if an instance of *GmfMap!TopNodeReference* is involved in the operation call, a runtime error will be thrown.

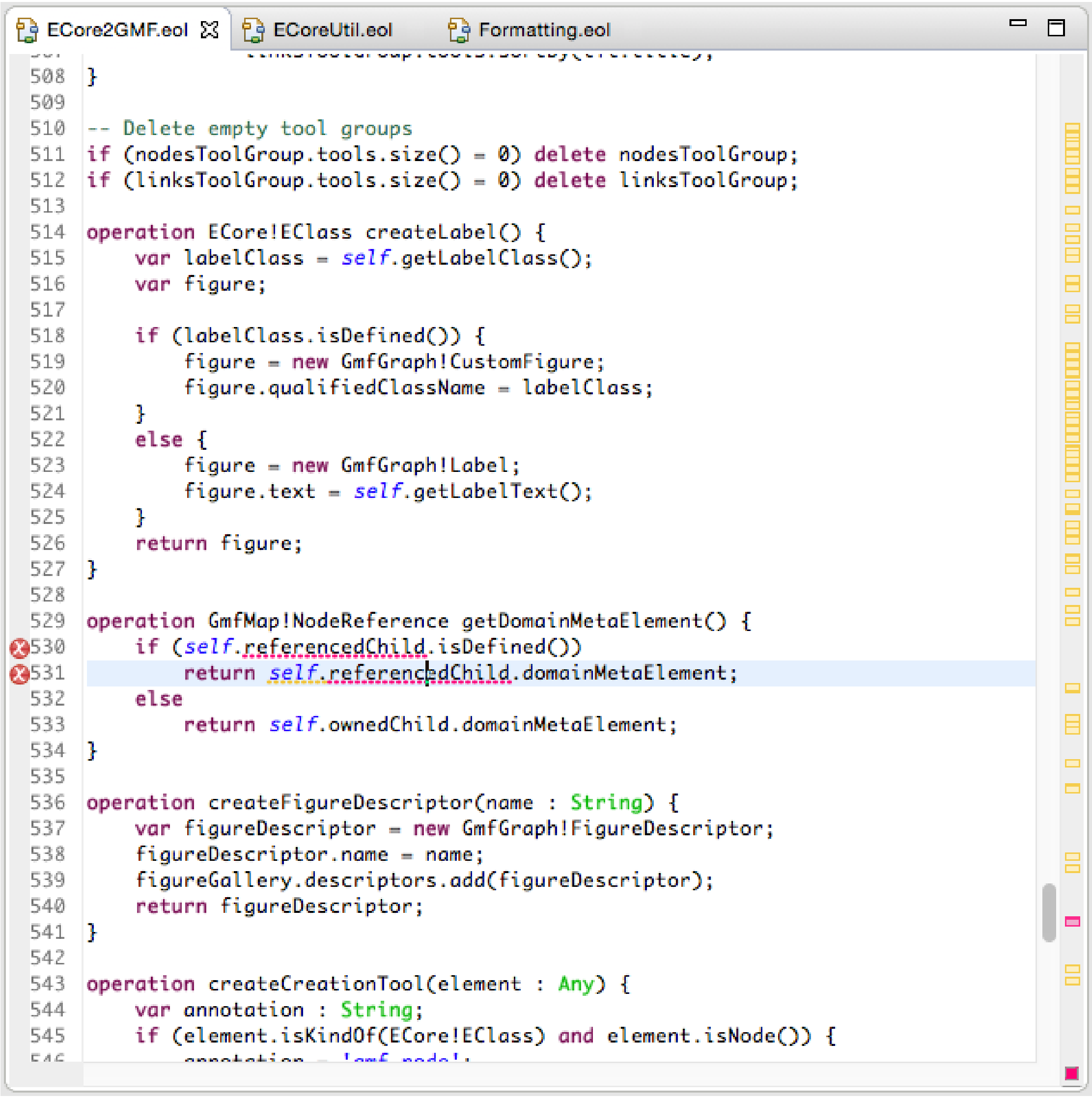


Figure 9.14.: Analysis on ECore2GMF.eol

9.2.3. Evaluating the EVL Static Analyser

The evaluation of the EVL static analyser is carried out by analysing a set of example validation rules provided in the Epsilon project's Git repository. In this section, an example which validates an OO model with EVL[6] (OO metamodel provided in Appendix B) is discussed. Analysing the EVL program in the example generates just one warning

[6] `http://www.eclipse.org/epsilon/examples/index.php?example=org.eclipse.epsilon.examples.validateoo`

and no errors. This section evaluates the EVL static analyser by injecting errors in the example code and show that such errors can be identified by the EVL static analyser.

The first type of error is to inject non-boolean expressions in the *guard* and *check* of an *Invariant* as shown in Figure 9.15. In lines 17-18, injected errors convert the resolved type of the expressions in the *guard* and *check* into String. This is detected by the static analyser which produces an error message stating that the *guard* and *check* can only contain expressions that evaluate to a Boolean value.

```
validateOO.evl    loadModels.eol
1  import "loadModels.eol";
2
3  context OO!NamedElement {
4
5      // Every NamedElement must define a name
6      constraint HasName {
7      check : self.name <> ""
8      message : "Element " + self + " must define a name"
9      }
10 }
11
12 context OO!Feature {
13
14     // The name of a feature (attribute,referecne,parameter)
15     // should start with a lower case letter
16     critique NameMustStartWithLowerCase {
17         guard : self.satisfies("HasName") + "(Injected Error)"
18         check : (self.name.substring(0,1) =
19                 self.name.substring(0,1).toLowerCase()) + "(Injected Error)"
20     }
21 }
22
23 context OO!Class {
24
25     // The name of a class should start with
26     // an upper case letter
27     critique NameShouldStartWithUpperCase {
28         guard : self.satisfies("HasName")
29         check : self.name.substring(0,1) =
30                 self.name.substring(0,1).toUpperCase()
31         message : "The name of class " + self.name +
32                 " should start with an upper-case letter"
33         fix {
34             title : "Rename class " + self.name + " to " +
35                             self.name.firstToUpperCase()
36             do {
37                 self.name = self.name.firstToUpperCase();
38             }
```

Figure 9.15.: Injecting errors to validateOO.evl

Figure 9.16 illustrates errors injected to the *message* of an *Invariant*. In line 8, the expression of the *message* is changed to a boolean value. This is detected by the static analyser which produces error messages stating that only expressions that evaluate to a String value is allowed within the *message*. In line 17, the parameter of the method call *satisfies()* has been changed from “HasName” to “Hasname”. The static analyser produces a warning as the names of the *Invariants* are case sensitive and no *Invariant* named “Hasname” can be found.

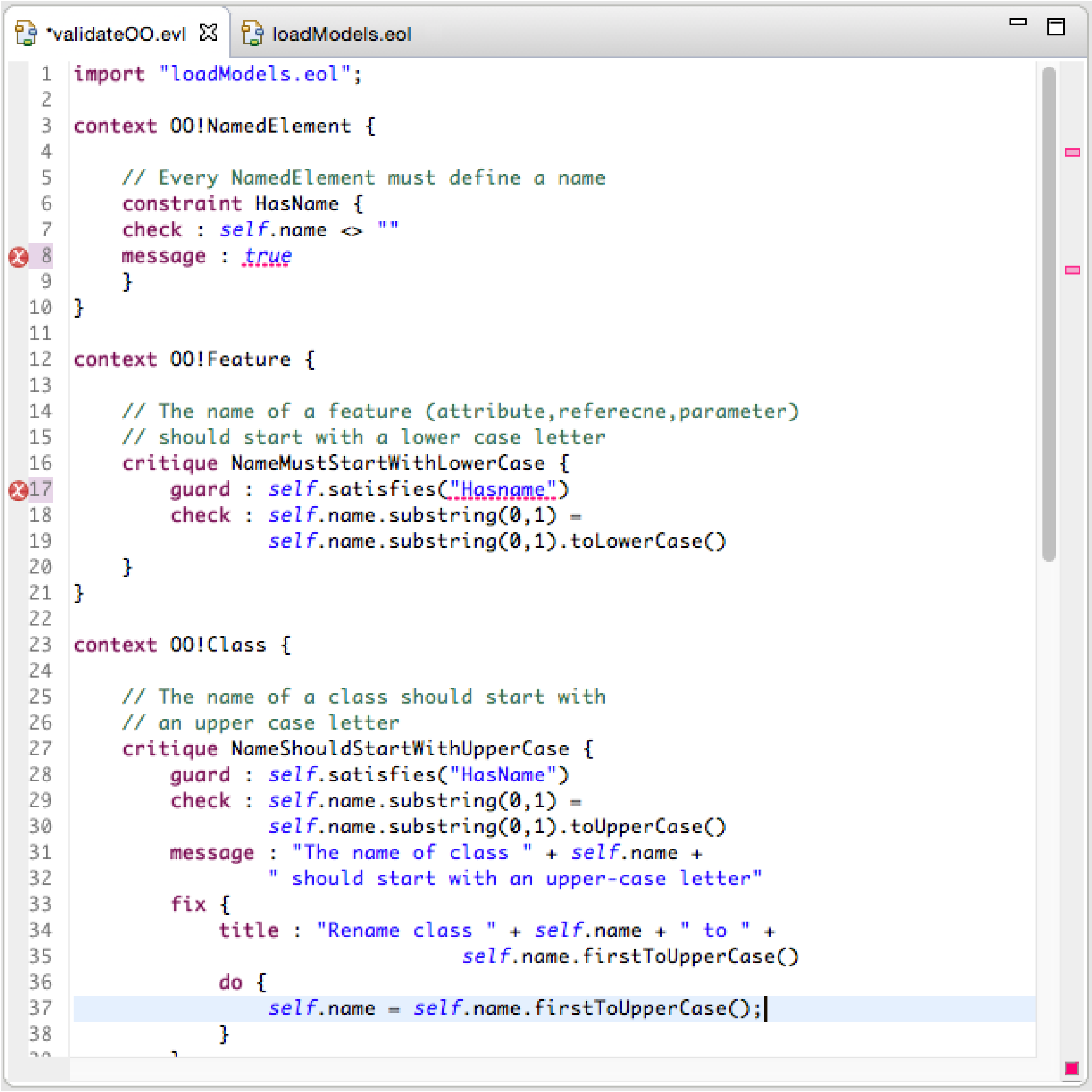


Figure 9.16.: Injecting errors to validateOO.evl

An injected error which requires more complex analysis is shown in Figure 9.17. In line 3, the *type* of the *context* is changed from *OO!NamedElement* to *OO!Class*, which triggers the error in line 17. In line 17, the method call *satisfies()* requires that *self* should satisfy the *constraint* named "HasName" which is defined in line 6. The EVL static analyser checks if the *constraint* "HasName" applies to the same type as the *critique* (line 16) that calls the *satisfies()* method. In this case, the static analyser will check if OO!Class is a super class of OO!Feature (which is not the case). The error is then produced in line 17 indicating that the *constraint* "HasName" does not apply on instances of OO!Feature.

9.2.4. Evaluating the ETL Static Analyser

In Section 8.2, the ETL static analyser was used to analyse the OO2DB transformation example provided in Epsilon's Git repository[7]. In addition, the transformation rule dependency graph calculation were discussed. In this section, example ETL transformations with injected errors are analysed by the ETL static analyser to demonstrate how different kinds of errors are detected. The created examples reuse the metamodels introduced in Section 8.2, which are however provided again for readability in Figures 9.18 and 9.19.

The ETL static analyser inherits all the features of the EOL static analyser, thus examples of errors in ETL that share the same nature with EOL errors are not discussed in this section. The first type of error is to check that the expression(s) in *guard*s of transformation rules are of type Boolean. Figure 9.20 illustrates an injected error in line 6, where the expression within the *guard* of transformation rule A2E is a String expression. The ETL static analysis is able to detect this error and produce an appropriate marker on the offending expression.

In ETL, a transformation rule may inherit another transformation rule (using the *extends* keyword). For the discussion, let R denote a transformation rule and R' denote another transformation rule which *extends* R, let S and T denote the *source* and the *target*(s) of the transformation rule R, and S' and T' denote the *source* and the *target*(s)

[7] `http://www.eclipse.org/epsilon/examples/index.php?example=org.eclipse.epsilon.examples.oo2db`

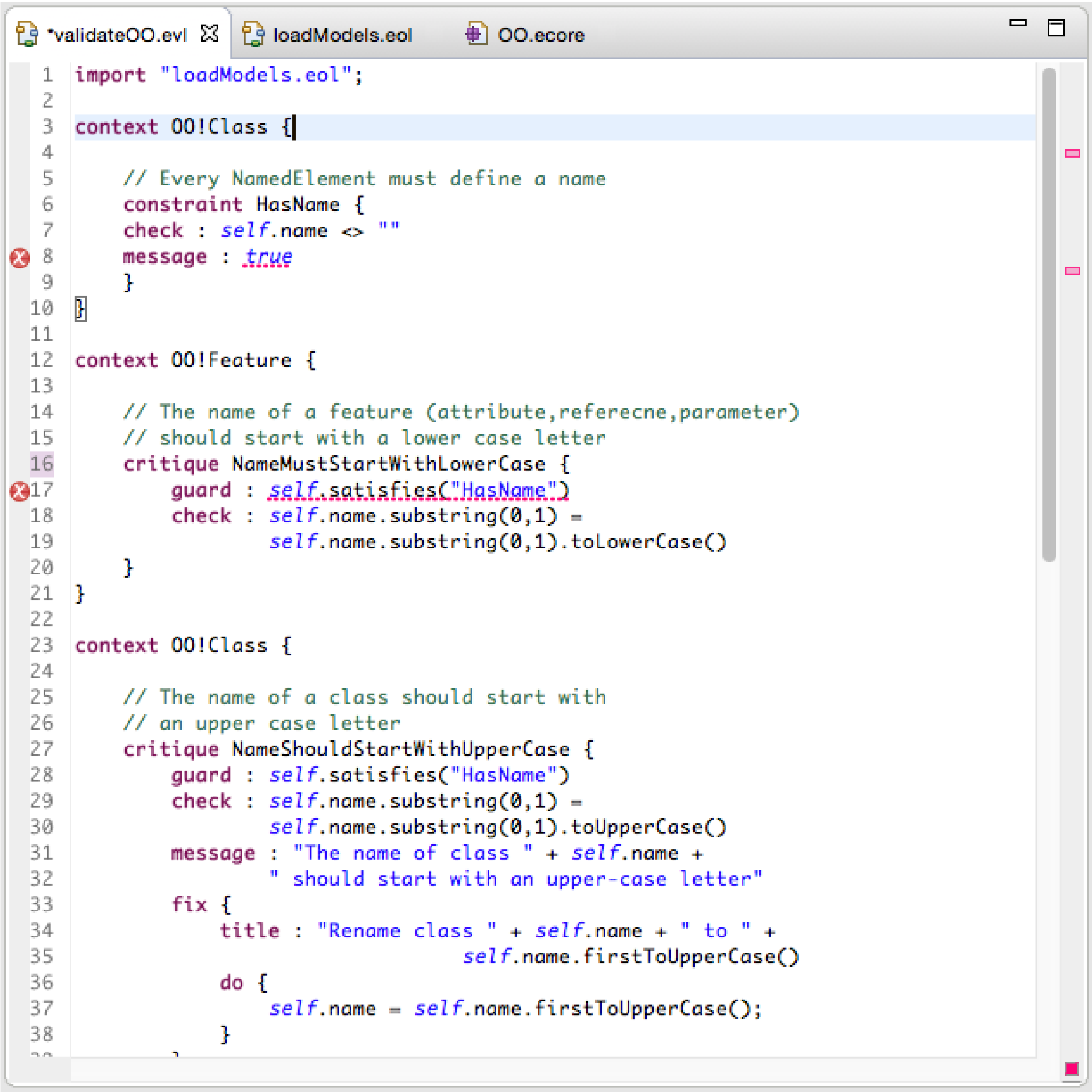


Figure 9.17.: Injecting errors to validateOO.evl

of the transformation rule R'. In a rule inheritance relationship, S' should be a sub type of S and each T' in R' should be a sub type of the corresponding Ts in R. To ensure the integrity of the inheritance relationship between transformation rules, the ETL static analyser should check if the rules satisfy this constraint.

An injected error with regard to transformation rule inheritance is demonstrated in Figure 9.21. In the example ETL module, three transformation rules are defined, rule A2E transforms instances of A in the source model into instances of E in the target

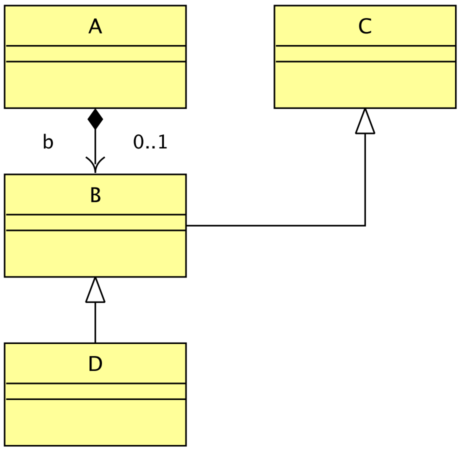


Figure 9.18.: The Source Metamodel

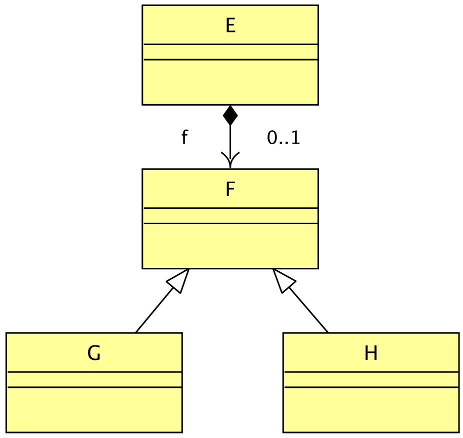


Figure 9.19.: The Target Metamodel

```
import 'loadModels.eol';

rule A2E
transform a : Source!A
to e : Target!E {
    guard: "hello world"
    e.f = a.b.equivalent();
}
```

Figure 9.20.: Checkikng the type of the expression in *guard*.

model, rule C2H transforms instances of C into instances of H, rule B2G transforms instances of B into instances of G. Rule B2G inherits the transformation rule C2H by using the *extends* keyword. For the inheritance relationship, the ETL static analyser checks if the type of the *source* of rule B2G is a sub type of the type of the *source* of rule C2H, and the *target* type(s) of B2G are sub types of the *target* type(s) of rule C2H in their respective order. In this example, since G in the target metamodel is not a sub type of H, an error is reported in the offending line 16.

In Section 8.2.4, the rule resolution operations (*equivalent()*, and *equivalents()* and the special assignment operator ::=) were discussed. During the rule resolution process, if no applicable rule is found, there is a need to inform the developer about it. Figure 9.22 shows a program that exhibits this error. In the ETL program, there are two rules, rule A2E, which transforms instances of A into instances of E and rule C2H, which transforms

```
import 'loadModels.eol';

rule A2E
transform a : Source!A
to e : Target!E {
    e.f = a.b.equivalent();
}

rule C2H
transform c : Source!C
to h : Target!H { }

rule B2G
transform b : Source!B
to g : Target!G
extends C2H { }
```

Figure 9.21.: Checking the correctness of rule inheritance.

```
1  import 'loadModels.eol';
2
3  rule A2E
4  transform a : Source!A
5  to e : Target!E {
6      e.f = a.b.equivalent();
7  }
8
9  rule C2H
10 transform c : Source!C
11 to h : Target!H { }
```

Figure 9.22.: Checkikng the type of the expression in *guard*.

instances of C into instances of H. In line 6, a call to the operation *equivalent()* is in place, which tries to resolve the equivalent model element for the result of the expression *a.b* in the target model. However, inspecting the whole ETL program reveals that there is no such rule that transforms instances fo B, hence, the ETL static analyser produces an error in line 6.

9.3. Limitations of the Epsilon Static Analysis Framework

One major limitation of the Epsilon static analysers is that they do not support resolving the types for *native* method calls, i.e. calls to native Java methods. Statically analysing Java code is outside the scope of this work. Therefore, the responsibility of ensuring the correctness of *native* expressions are delegated to the users of Epsilon languages.

Another limitation of the Epsilon static analyser is that it does not support the verification of OCL recursive expressions, i.e. the call to the *closure()* operation.

9.4. Chapter Summary

In this chapter, the extensibility of the Epsilon Static Analysis Model Connectivity was evaluated by means of the implementation of the schema-less XML driver. Transformations involving both EMF models and schema-less XML models were analysed to evaluate Epsilon static analysers. The EOL, EVL and ETL static analysers altogether with the AST2EOL, AST2EVL, AST2ETL transformations have also been evaluated by analysing widely used transformations. The limitations of the Epsilon static analysis framework were also discussed in this chapter, which can be considered goals for future work.

10. Applications of the Epsilon Static Analysis Framework

In this chapter, the discussion moves onto the applications of the Epsilon static analysis framework related to performance analysis and optimisation of model management programs. These applications, although developed atop the Epsilon platform, can be ported to other model management tools to achieve the same functionalities.

In the first application, based on the Epsilon static analysis framework, a facility named Sub-Optimal Performance Pattern Detection (SPPD) was developed which detects performance bottlenecks within EOL programs. In the second application, the Epsilon static analysis framework was used in conjunction with the Epsilon execution engine to improve the performance of programs that manage very large models. Finally, a sophisticated XMI model parser was developed atop the Epsilon static analysis framework which is able to partially load large XMI-based models with the help of static analysis.

10.1. Sub-Optimal Performance Pattern Detection

In the context of processing large models in MDE, it is essential to ensure that model management programs are written in an optimised manner in terms of performance. Sub-optimal performance patterns within model management programs can result in severe performance issues. This section presents the *Sub-optimal Performance Pattern Detector* (SPPD), which is built by leveraging the Epsilon static analysis framework.

With SPPD, sub-optimal performance patterns in a program written in Epsilon languages (for example, an EOL program) can be detected and reported to developers statically (i.e. without the need for runtime monitoring).

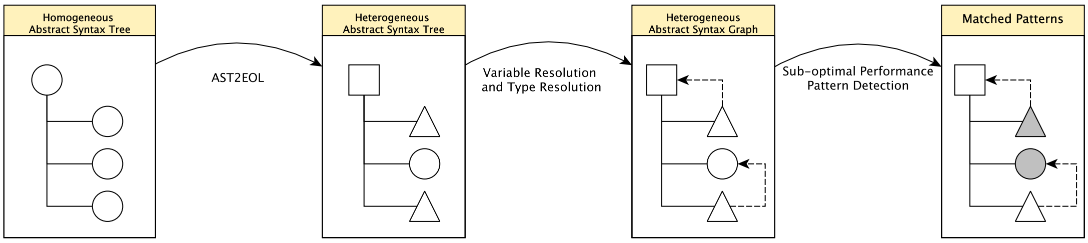


Figure 10.1.: Detecting sub-optimal performance patterns from Abstract Syntax Trees.

10.1.1. Design of SPPD

Chapter 7 explained how static analysis works by converting the Homogeneous Abstract Syntax Tree (HoAST) of a program written in EOL (ANTLR-based AST) into the Heterogeneous Abstract Syntax Tree (HeAST), which is essentially a model that conforms to the EOL metamodel. Based on the HeAST, variable resolution and type resolution are performed which enrich the HeAST (by generating inter-related links such as variable-reference links and by calculation features of expressions such as the resolved types), and turn it into a Heterogeneous Abstract Syntax Graph (HeASG). SPPD uses the Epsilon Pattern Language (EPL) to define patterns which are matched against the fully resolved HeASG. This process is depicted in Figure 10.1. If any matches are found, corresponding warnings are produced to prompt the developer of source code segments that can be potentially improved.

10.1.2. Sub-Optimal Performance Patterns

In this section, a number of sub-optimal performance patterns in the context of large scale model manipulation are presented.

The examples provided are based on a minimal *Library* metamodel illustrated in Figure 10.2. The Library metamodel contains two types, *Author* and *Book*. An *Author* has a *first_name*, a *surname* and a number of published *Book*s, and a *Book* has a *name* and an *Author*. The association between *Author* and *Book* is bidirectional, they are *books* and *authors* respectively.

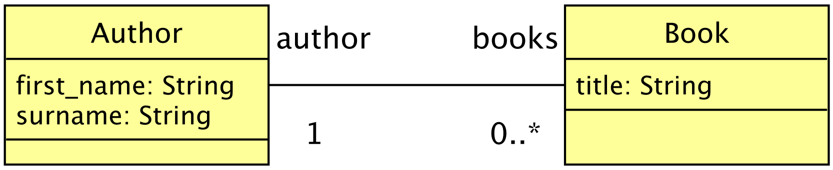


Figure 10.2.: The Library metamodel

Inverse navigation

A frequent operation in EOL is to retrieve all model elements of a specific type by using the *.all* property call (equivalent to *allInstances()* and *allOfKind*, which retrieves all instances of a type and its sub types) which can be a computationally expensive activity as models grow in size. By analysing the metamodel of the model under question, bidirectional relationships between model elements can be used to avoid such expensive computations.

```
1 var a = Author.all.first;
2 var books = Book.all.select(b|b.author = a);
3 var aBook = Book.all.selectOne(b|b.author = a);
```

The listing above demonstrates a sub-optimal pattern that can lead to degraded performance. In line 1, an *Author* is retrieved from the model. In line 2, all instances of type *Book* are retrieved and then a *select*() operation is performed to find the books that are written by *Author a*. However, since the relationship between *Author* and *Book* is bidirectional, this can be replaced by the (more efficient) statement:

```
var books = a.books;
```

This is also the case for the *selectOne* operation in line 3, which can be rewritten as:

```
var aBook = a.books.first();
```

Compound select operations

Another sub-optimal pattern is the presence of compounded *select*() operations on the same collection. For example:

```
var authors = Author.all.select(a|a.first_name =
        'William').select(a|a.surname = 'Shakespeare');
```

Listing 10.1: Compounded select operations

All of the *Author*s are retrieved first, then a *select*() operation is performed to select all *Author*s whose *first_name*s are *William*, and finally another *select*() operation is performed to select all *Author*s whose *surname*s are *Shakespeare*. The complexity of this operation is $O(n + m)$ where n is the number of *Author*s in the model under question, and m is the number of the results returned by the first *select*() operation. However, the condition of both the *select* operations can be put together to form a single *select* operation, which can be written as:

```
var authors = Author.all.select(a|a.first_name =
        'William' and a.surname = 'Shakespeare');
```

The complexity of this operation is $O(n)$ as the collection of the *Author*s is only traversed once.

Collection element existence

In some cases, checking existence of an element inside a collection can be written in inefficient ways.

```
1 if(Book.all.select(b|b.name = "EpsilonBook").size() > 0) {
2   "There is a book called EpsilonBook".println();
3 }
```

Listing 10.2: Collection element existence

Listing 10.2 demonstrates such a scenario. In line 1, the condition of the *if* statement retrieves all instances of *Book*, then selects those with the name *EpsilonBook*, then evaluates if the size of the result is greater than 0. This operation eventually checks for the existence of a book named *EpsilonBook*. Thus, this operation can be more efficiently re-written as:

```
Book.all.exists(b|b.name = "EpsilonBook")
```

Select the first element in a collection

Listing 10.3 demonstrates another example of sub-optimal EOL code.

```
var anEpsilonBook = Book.all.select(b|b.name = "EpsilonBook").first();
```

Listing 10.3: Select an element in a collection

A *select*() operation is performed on all instances of *Book* to select all books with the name *EpsilonBook*, then a $first$ operation is performed to select the first item of the collection returned by *select*. This can be more efficiently re-written as:

```
var anEpsilonBook = Book.all.selectOne(b|b.name = "EpsilonBook");
```

to avoid traversing all instances of *Book*.

10.1.3. Pattern Implementation: Inverse Navigation

This section demonstrates the implementaion of the detection facility for the first pattern discussed above (Inverse Navigation). Detectors for the remaining patterns are implemented in a similar way.

The Abstract Syntax Graph

Figure 10.3 illustrates a fragment of an EOL model which represents the statement below:

```
Book.all.select(b|b.author = a);
```

Firstly, invocations of the *select*() operation in the EOL metamodel are represented by instances of *FOLMethodCallExpression*. A *FOLMethodCallExpression* has a *name* (of type *NameExpression*) and an *iterator* (of type *FormalExpressionParameter*). In this case, the *name* is *select* and the *iterator* is *b*.

The *select*() operation has a condition, which in this case is an instance of *EqualsOperatorExpression*. The *lhs* (left hand side) of it is an instance of *PropertyCallExpression*, whose *target* (of type *NameExpression*) is *b* and *property* (of type *NameExpression*) is *author*; the *rhs* (right hand side) of it is *a* (an instance of *NameExpression*). Both the

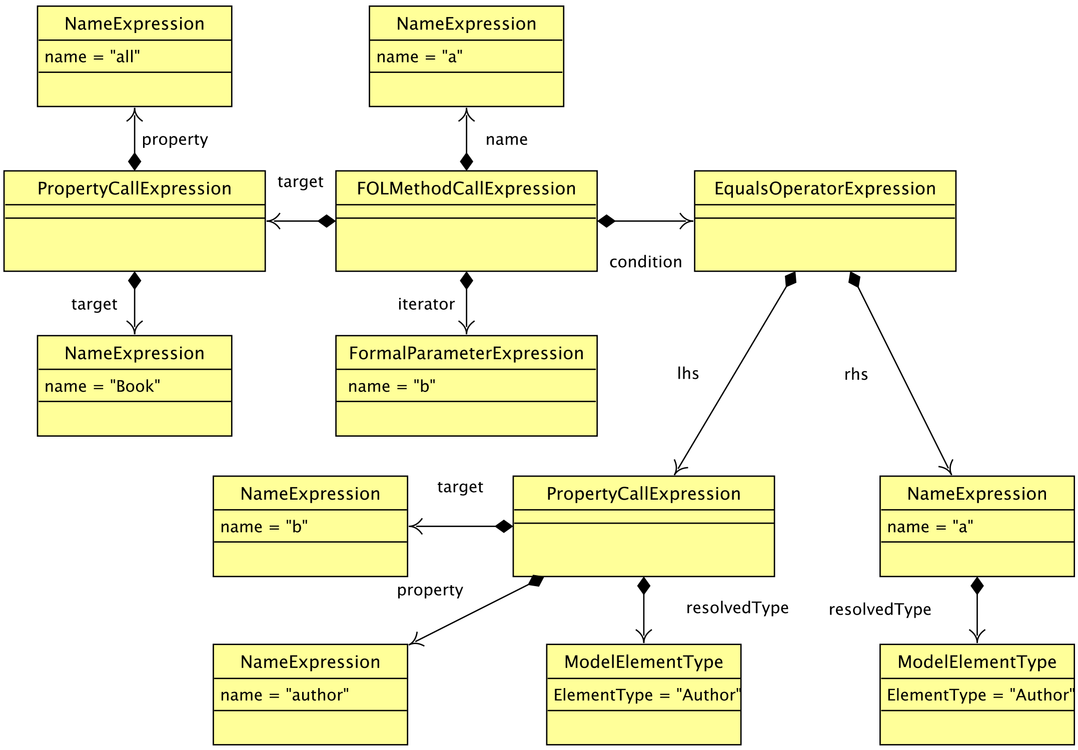


Figure 10.3.: The model representation for $Book.all.select(b|b.name = a)$

lhs and *rhs* of the *EqualsOperatorExpression* have *resolvedTypes*, in this case, they are both *Author* (*ModelElementTypes*).

The *target* of the *FOLMethodCallExpression* is an instance of *PropertyCallExpression* with its *target* being *Book* (of type *NameExpression*) and its *property* being *all* (of type *NameExpression*). The types of these expressions, as well as with some irrelevant details are omitted for the purpose of the discussion.

The EPL pattern

In Listing 10.4, an EPL pattern is defined to match occurrences of the pattern described above. In lines 3-7, a *guard* is defined to look for a *FOLMethodCallExpression* the name of which is either *select* or *selectOne*, the type of the condition should be *EqualsOperatorExpression*, its target should be an instance of *PropertyCallExpression*, and the property of the *PropertyCallExpression* should be *all*.

In line 11, a *guard* is defined to look for an instance of *EqualsOperatorExpression* in the condition of the *FOLMethodCallExpression* found previously, the *lhs* of which should be an instance of *PropertyCallExpression*.

Lines 13-15 specify that the *resolvedType* of the *lhs* should be an instance of *ModelElementType*. In lines 17-19, it specifies that the *resolvedType* of the *rhs* should be an instance of *ModelElementType*, too. In lines 21-25, it specifies that the type of the *lhs* and the *rhs* should be the same.

Lines 26-37 perform the matching of the pattern. This part firstly fetches the *EReference* from the *lhs* of the condition (in this case, 'b.author', it is an *EReference* because as previously discussed, all metamodels are converted to EMF metamodels by ESAMC for uniformity). The *EReference* is then inspected; if it is not null and it has an *eOpposite* reference, the pattern continues to check if the type of the *eOpposite* of the reference is the type of the *rhs* of the condition (in this case, Author).

In lines 40-47, a helper method is defined to help look for an *EReference* given an *EClass* and a name.

At runtime, the EPL execution engine matches the provided patterns against a given EOL model (which is acquired from the AST2EOL transformation discussed in Section 6.6). Pattern matches are then collected by SPPD and corresponding warnings are produced in the EOL editor in Eclipse.

```
1  pattern InverseNavigation
2  folcall : FOLMethodCallExpression
3    guard: (folcall.method.name = 'select'
4      or folcall.method.name = 'selectOne')
5      and folcall.conditions.isTypeOf(EqualsOperatorExpression)
6      and folcall.target.isTypeOf(PropertyCallExpression)
7      and folcall.target.property.name = 'all',
8
9  condition : EqualsOperatorExpression
10   from: folcall.condition
11   guard: condition.lhs.isTypeOf(PropertyCallExpression)
```

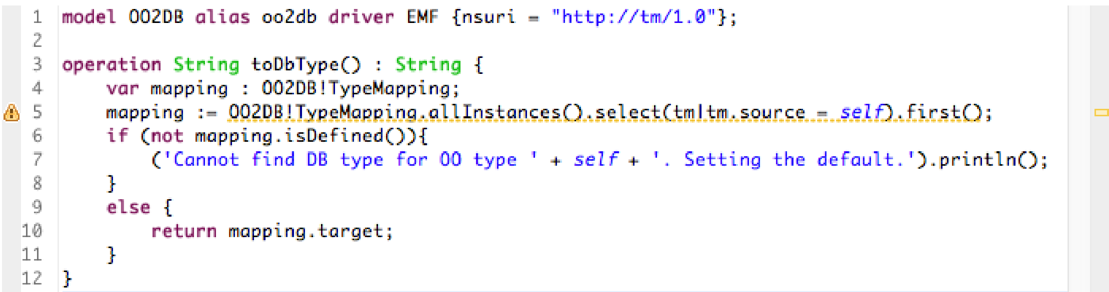
```

lhs : PropertyCallExpression
  from: condition.lhs
  guard: lhs.resolvedType.isTypeOf(ModelElementType),

rhs : NameExpression
  from: condition.rhs
  guard: rhs.resolvedType.isTypeOf(ModelElementType),

lhsType : ModelElementType
  from: lhs.resolvedType,
rhsType : ModelElementType
  from: rhs.resolvedType
  guard: lhsType.ecoreType = rhsType.ecoreType {
    match {
    var r = getReference(lhs.target.resolvedType.ecoreType,
    lhs.property.name);
    if(r.upperBound = 1 and
        r.eOpposite <> null and r <> null) {
      if(r.eOpposite.eType =
          lhs.target.resolvedType.ecoreType) {
        return true;
      }
    }
    return false;
  }
}

operation getReference(class: Any, name:String)
{
```

```
  for(r in class.eReferences) {
    if(r.name = name)
      return r;
  }
  return null;
}
```

Listing 10.4: EPL pattern for inverse navigation

10.1.4. Evaluation

To validate the applicability of SPPD in practice, a number of EOL files on GitHub[1] were analysed by SPPD to check for existing defined sub-optimal patterns. GitHub returned 245 matched EOL programs, however, only 47 of them are complex enough for SPPD to analyse in a meaningful way. Among all the 47 EOL files, 32 of them contained sub-optimal performance patterns. For example, a file from the Jet2Egl project[2] in Figure 10.4. In line 5, a sub-optimal pattern which calls *first()* after *select()* (instead of calling *selectOne()*) was detected. This pattern is very common in complex queries and is the most occurring pattern for all the EOL files analysed.



```
1  model OO2DB alias oo2db driver EMF {nsuri = "http://tm/1.0"};
2
3  operation String toDbType() : String {
4      var mapping : OO2DB!TypeMapping;
5      mapping := OO2DB!TypeMapping.allInstances().select(tm|tm.source = self).first();
6      if (not mapping.isDefined()){
7          ('Cannot find DB type for OO type ' + self + '. Setting the default.').println();
8      }
9      else {
10         return mapping.target;
11     }
12 }
```

Figure 10.4.: Detecting sub-optimal pattern for OO2DB.

Another common found sub-optimal pattern is the call to *select()* followed by *size()*, which should be replaced by *exists()* instead, Figure 10.5 illustrates analysing an EOL

[1] https://github.com/search?utf8=%E2%9C%93&q=select+extension%3Aeol&type=Code&ref=searchresults

[2] https://github.com/majicmoo/EGLStuff/tree/72be6e6f503e94db052ca4cc25fbe55a8aa97822/Jet2Egl

program named TestScenario.eol which was developed in the context of the CATMOS project (Capability Acquisition Tool with Multi-objective Search) [3].

```
for (desired in desiredMeasurements)
{
    if (providedMeasurements.select(x|x.name == desired.name).size() == 0
                                            and desired.script.size() == 0)
    {
        "Error".println();
        "No matching measurement for ".print();
        desired.println();
        "Alternative problem: Missing script".println();

        errorCount = errorCount + 1;
    }
}
```

Figure 10.5.: Detecting sub-optimal pattern for TestScenario.eol.

10.2. An efficient computation strategy for the call to allInstances()

As models involved in MDE processes get larger and more complex [16, 17], model query and transformation languages are being stressed to their limits [27, 132]. One of the most commonly-used and computationally-expensive operations that model query and transformation engines support is the ability to retrieve collections of instances of a particular type/kind regardless of their location in a model (i.e. OCL's *allInstances()*).

This section discusses existing strategies for computing such collections of instances, to highlight their advantages and shortcomings. Based on the discussion, this section presents a novel computation strategy, which uses static analysis and metamodel introspection to pre-compute and cache the results of calls to *allInstances()* within programs in an efficient way.

10.2.1. Background

The majority of contemporary model query and transformation languages provide support for retrieving collections of all model elements that are instances of a particular type-

[3] `https://github.com/Frankablu/CATMOS/blob/8c8b33cc2f0297fd5ffe184b76fb954eac3c81ed/catmos_gui/runtime-LoadTool/Tool/Scripts/testScenario.eol`

/kind. For example, OCL, QVTr, ATL, and Acceleo provide the built-in *allInstances()* operation which can be invoked on a type to return a set containing all its instances (e.g. *Person.allInstances()*), Epsilon's EOL provides the *getAllOfType()* and *getAllOfKind()* operations, and QVTo the *objects(type : Type)* and *objectsOfType(type : Type)* operations that operate in a similar way. In this section, all such operations are collectively referred to as *allInstances()*.

For file-based EMF models, a naive strategy to implement *allInstances()* is to navigate the (loaded) in-memory model element containment tree upon invocation, to collect and return all instances of the requested type. Repeatedly traversing the containment tree to fetch all instances of the same type for multiple invocations of the operation on that type is clearly inefficient, so the majority of model query and transformation engines provide support for caching and reusing the results of previous invocations of the operation (which is a simple task for side-effect free languages but requires some additional book-keeping for languages that can mutate the state of a model).

When a query (or a transformation) contains a large number of calls to *allInstances()* for different types, instead of traversing the containment tree for each of these calls/types on demand, it can be more efficient for the execution engine to pre-compute and cache all these collections in one pass at start-up instead (normally referred to as *greedy caching*, as not all of the caching results are needed). This typically incurs a higher upfront cost and increase the memory footprint. However, for a sufficiently high number of invocations on different types, it is very likely to pay off eventually – particularly as models grow in size.

Overall, when more than one calls to *allInstances()* are made for different types in the context of a query, the on-demand approach is sub-optimal in terms of performance. On the other hand, if a query only calls *allInstances()* on a small number of types (compared to the total number of types in the metamodel), greedy caching is wasteful.

10.2.2. Program- and Metamodel-Aware Instance Collection

Given in-advance knowledge of the metamodel of a model, and the types on which *allInstances()* is likely to be invoked in the context of a query (e.g. obtained through static analysis of the query itself) operating on that model, it is possible to pre-compute the results of these invocations by traversing the contents of the model only once.

This section demonstrates the proposed algorithms and their supporting data structures with reference to EOL. For conciseness, the discussion is also restricted to EOL queries operating on a single EMF-based model which conforms to an Ecore metamodel comprising exactly one EPackage. However, the proposed approach is trivially portable to other query and transformation languages of a similar nature, and to queries that involve more than one models conforming to multi-*EPackage* metamodels.

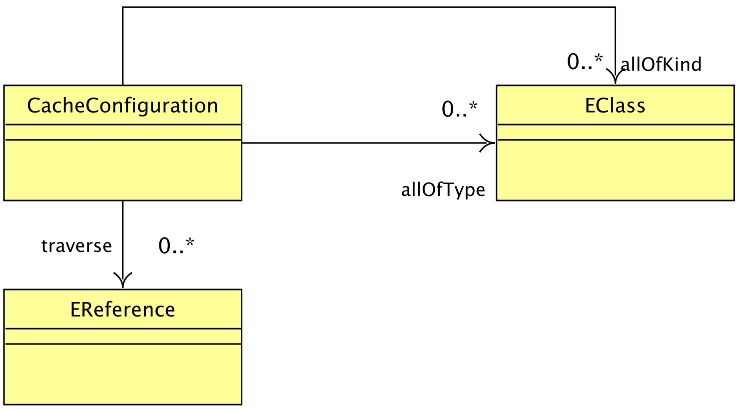


Figure 10.6.: Cache Configuration Metamodel

10.2.3. Cache Configuration Model

Figure 10.6 demonstrates a data structure (in the form of a metamodel) of the cache configuration, an instance of which needs to be populated at compilation time (e.g. by statically analysing the query of interest and by inspecting the metamodel of models on which it will be executed) in order to facilitate efficient execution of *allInstances()* at runtime.

CacheConfiguration acts as a container for the *EClasses* of the model's metamodel of which that the execution engine may need to retrieve all instances in the context of the query of interest. *EClasses* of interest can be linked to a *CacheConfiguration* through the latter's *allOfKind* and *allOfType* references (EOL, like QVTo, support distinct operations for computing all direct and indirect instances of a given type). The *traverse* reference in Figure 10.6 is discussed in Section 10.2.5.

10.2.4. Query Static Analysis: The Type-Aware Strategy

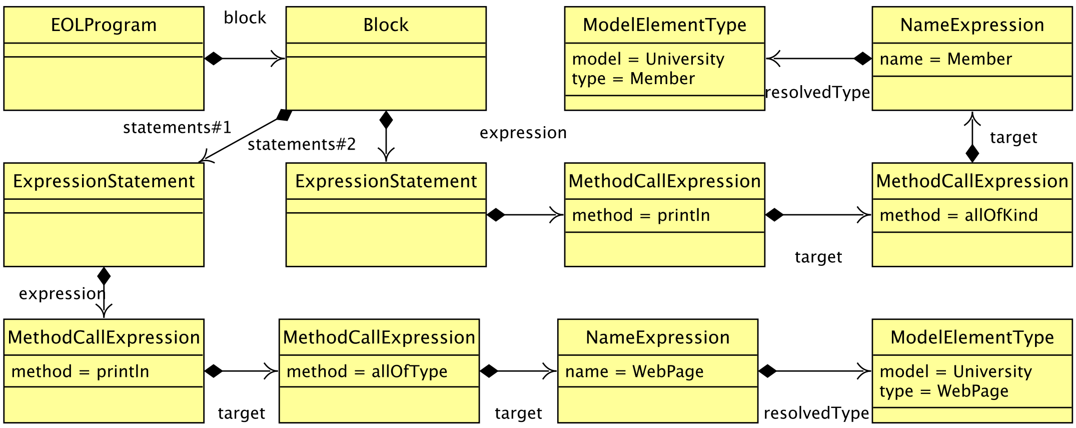


Figure 10.7.: The Abstract Syntax Graph of the EOL program of Listing 10.5

The first step of the process is to generate an initial version of the cache configuration model by statically analysing the query of interest. For this purpose, an extension of the Epsilon static analysis framework was created for automatic cache configuration extraction. Figure 10.7 demonstrates the variable-resolved and type-resolved abstract syntax graph of the example EOL program below;

```
WebPage.allOfType().println();
Member.allOfKind().println();
```

Listing 10.5: An Example EOL Program

This program operates on models conforming to the *University* metamodel (introduced in Chapter 2 and provided again in Figure 10.8). To compute the initial version of the cache configuration model, the automatic cache configuration extraction facility (backed by EOLVisitor facility discussed in Section 7.2) goes through the abstract syntax graph and locates instances of:

- *MethodCallExpression* for which the name of the method called is *allOfKind()*, *allOfType()*, *allInstances()* (alias of *allOfKind()*), the resolved type of their target expression is *ModelElementType*, and which have no parameter values;

- *PropertyCallExpression* for which the name of the property is *all* (alias of the *allOfKind() operation*), and the resolved type of their target expression is *ModelElementType*.

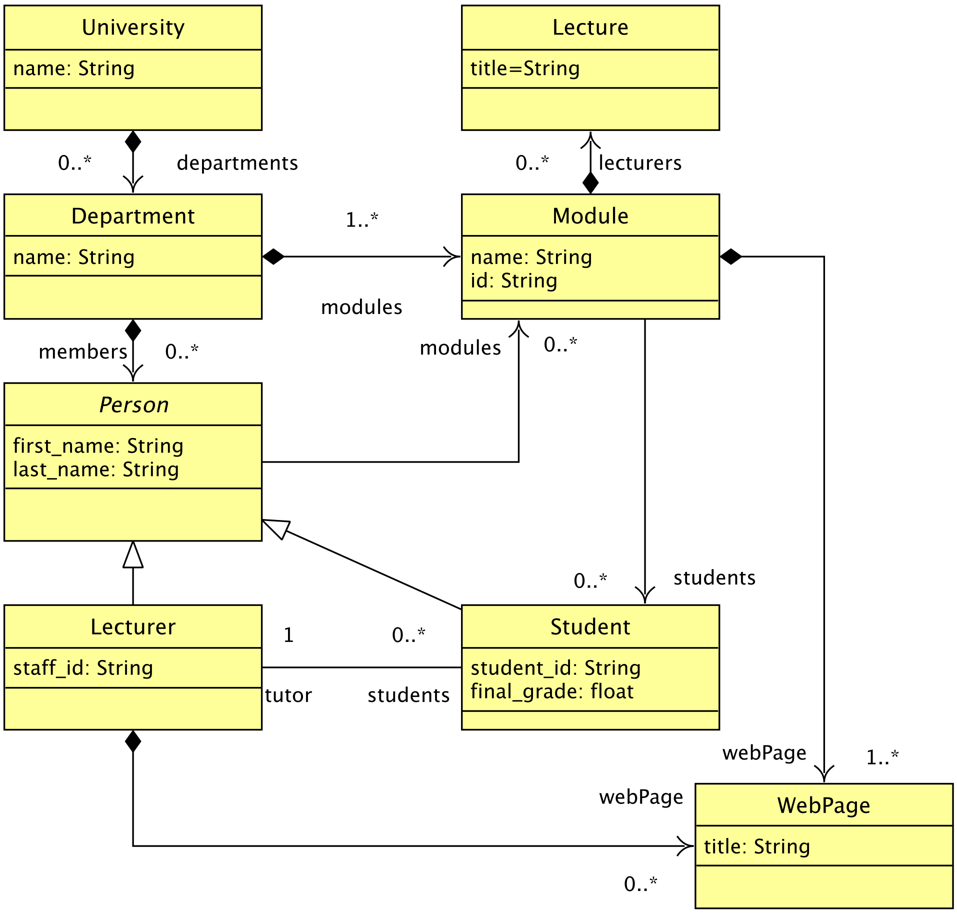


Figure 10.8.: A simple university metamodel

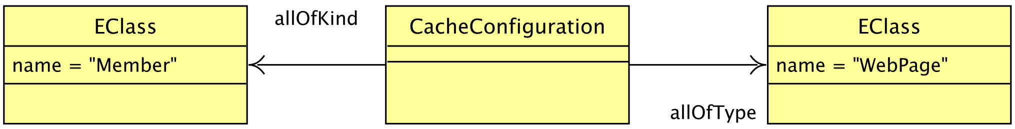


Figure 10.9.: Initial Extracted Cache Configuration Model

Having identified the EOL constructs of interest, a *CacheConfiguration* is automatically extracted. For each call to *allOfType()* the *allOfType* reference is populated in the *CacheConfiguration* with its corresponding *EClass*. Similarly, for all other calls of interests, the respective references of the *CacheConfiguration* is populated. The initial extracted cache configuration model after running the example is illustrated in Figure 10.9. This approach is referred to as the *Type-Aware* strategy because it extracts all of the types that need to be cached in a program.

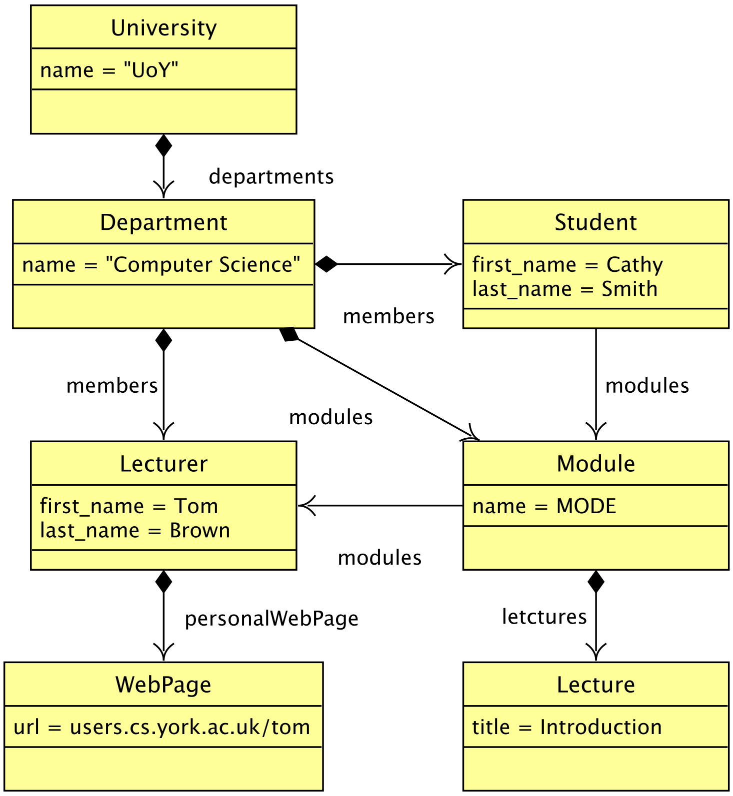


Figure 10.10.: An University Model

10.2.5. Reference Pruning: The Type-and-Reference-Aware Strategy

Following the process discussed above, the execution engine can be made aware of all the calls to *allInstances()* it needs to perform, and to pre-compute and cache upfront (*WebPage.allOfType()* and *Member.allOfKind()* in the running example). The next step is to collect the model elements of interest in one pass and as efficiently as possible. A straightforward collection strategy would involve navigating the entire model containment tree, assessing whether each model element is of one of the types of interest and, if so, adding it to the appropriate cache(es).

However, by inspecting the example model in Figure 10.10, it is observed that traversing the containment closure of the *modules* reference of the "Computer Science" Department model element is guaranteed not to reveal any model elements of interest (according to the metamodel of Figure 10.8, *Module*s can only contain *Lecture*s and neither of these types of elements are of interest to the query). This observation can be generalised and exploited to prune the subset of the containment tree that the engine will need to visit in order to populate the caches of interest.

```
let cm = the initial version of the configuration cache model;
let p = the EPackage that the model conforms to;
let refs = empty list of EReferences;

foreach non-abstract EClass c in p do
    foreach containment EReference r of c do
        call planTraversal(r);
    end
end

function planTraversal(r : EReference)
    let types = transitive closure of r's type and all its sub-types;
    if types includes any of the EClasses in cm then
        add r to refs;
    end
    else
        foreach containment EReference tr of each of the types do
            planTraversal(tr)
        end
    end
end
```

Algorithm 4: Containment Reference Selection Algorithm

To achieve this, it is needed to analyse the metamodel and compute the subset of containment references that can potentially lead to elements of interest. The proposed algorithm for this purpose is illustrated in Algorithm 4. It is worth noting that the algorithm has been simplified for presentation purposes and that implementations of the algorithm need to make use of memoisation to avoid infinite recursion that can be caused by circular containment references of no interest. Adding the computed containment references that need to be traversed at runtime to the (incomplete) cache configuration

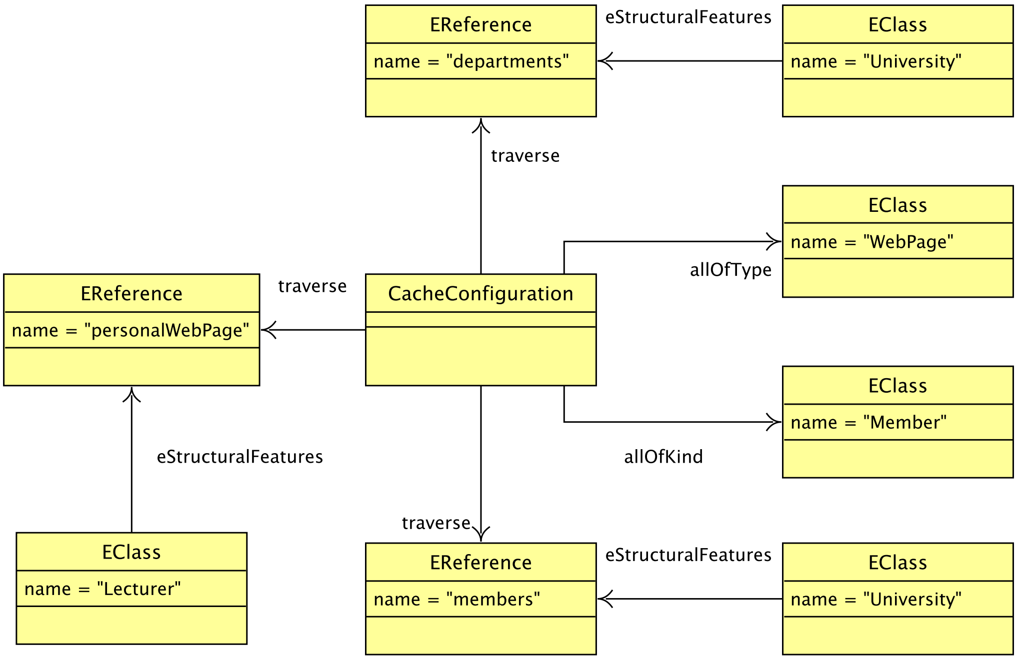


Figure 10.11.: Complete Cache Configuration Model

model of Figure 10.9, produces the (complete) configuration model of Figure 10.11. This approach is referred to as the *Type-and-Reference-Aware* strategy because not only it gives an idea of what *Types* to cache, but it also give an idea of what *references* that need to be traversed in the model level in order to get the instances of the *Types* that need to be cached.

10.2.6. Instance Collection and Caching

Having computed the cache configuration model, the final step includes traversing only the identified containment references of the in-memory model at runtime in a top-down recursive manner to collect and cache the elements of interest.

For example, with reference to the example model of Figure 10.10, the instance collection process starts at the top-level *:University* element. The element's EClass is not linked to the cache configuration via one of its *allOfType* or *allOfKind* references, and as such the element is not cached. Navigating the university's *departments* reference reveals a *:Department* element, which also does not need to be cached. The process does not need to navigate the department's *modules* reference as it is not linked to the cache configuration via the latter's *traverse* reference, and as such it proceeds with its *members* reference. Traversing the *members* reference reveals an instance of *Student* and an instance of *Lecturer*, both of which are cached in preparation for the *Member.allOfKind()* invocation. Similarly, the *webpage* reference of :Lecturer is traversed and reveals a *:WebPage*, which is also cached in preparation for the *WebPage.allOfType()* invocation.

10.2.7. Benchmark Results

This section presents the benchmarking results of the work presented in Section 10.2. For the benchmarking, the computation approaches have been integrated with Epsilon runtime in the sense that the Epsilon EMF model driver has been modified so that it works with the static analysis and pre-caches the results of query in different approaches. For comparison purposes, the benchmarks are performed on four different strategies for computing the calls to *allInstances()* (based on the Epsilon platform);

1. Lazy (on-demand) approach (denoted by L), the lazy approach is the default approach for Epsilon;

2. Greedy approach (denoted by G), the greedy approach naively pre-computes the results all possible calls to *allOfType()* and *allOfKind()* and caches them in memory (discussed in Section 10.2.1);

3. Type-Aware approach (denoted by T), the Type-Aware approach makes use of static analysis as discussed in Section 10.2.4 but does not prune references and as such it needs to visit the entire containment tree at runtime. It is included in this benchmark only to assess the additional benefits of reference pruning; and

4. Type-and-reference-aware (denoted by TR), the Type-and-Reference-Aware approach makes use of the further static analysis approach discussed in Section 10.2.5, which prunes containment references in the sense that only part of the model is visited in memory at runtime.

The benchmarks were performed on a computer with Intel(R) Core(TM) i7 CPU @

2.3GHz, with 8GB of physical memory, running OS X Yosemite. The version of the Java Virtual Machine used was 1.8.0_31-b13. The results are in seconds.

For the benchmarks, models of varying sizes obtained from reverse engineered Java code in the 2009 GraBaTs contest[4] are used. These models, named set0, set1, set2, set3 and set4 (9.2MB, 27.9MB, 283.2MB, 626.7MB, 676.9MB respectively) are stored in XMI 2.0 format and have been used for various benchmarks for different tools [87, 81].

Model Element Coverage

To quantify model coverage in the benchmarks, the numbers of elements in each data set are counted, and then EOL programs which (approximately) exercise 20%, 40%, 60%, 80% and 100% of the number of elements for each data set are automatically generated with their respective calls to *allOfKind()* (and/or) *allOfType()*. An example generated EOL program is provided in Listing 10.6.

```
var size = 0;
var methodInvocation = MethodInvocation.all.first();
size = size + MethodInvocation.all.size();
var qualifiedName = QualifiedName.all.first();
size = size + QualifiedName.all.size();
...
size.println();
```

Listing 10.6: An example of generated EOL program for model element coverage

All generated EOL programs are then executed and the performance of the four different approaches is measured in terms of the time it takes to load the models and the time it takes to execute the programs.

Results

The obtained results are presented in Table 10.2.7 and Table 10.2.7. Acronyms L, G, T and TR are used to denoted the aforementioned approaches (Lazy, Greedy, Type-

[4]GraBaTs2009: 5th Int. Workshop on Graph-Based Tools, `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/`

Perc.	L Load	L Exec.	G Load	T Load	TR Load	* Exec.	Imp.G Total	Imp.T Total	Imp.TR Total	Imp.* Exec.
	sec.		sec.	sec.	sec.	sec.	%	%	%	%
					Set0					
20%	0.552	0.015	0.652	0.554	0.572	0.001	-15.17%	2.12%	-1.06%	93.33%
40%	0.555	0.007	0.631	0.572	0.561	0.002	-12.63%	-2.14%	-0.18%	71.43%
60%	0.549	0.012	0.645	0.571	0.573	0.003	-15.51%	-2.32%	-2.67%	75.00%
80%	0.543	0.026	0.652	0.573	0.576	0.005	-15.47%	-1.58%	-2.11%	80.77%
100%	0.552	0.141	0.638	0.623	0.619	0.013	6.06%	8.23%	8.80%	90.78%
Perc.					**Set1**					
20%	1.643	0.606	1.856	1.653	1.672	0.01	17.03%	26.06%	25.21%	98.35%
40%	1.596	0.595	1.875	1.736	1.711	0.011	13.92%	20.26%	21.41%	98.15%
60%	1.587	0.556	1.843	1.786	1.773	0.013	13.39%	16.05%	16.66%	97.66%
80%	1.611	0.571	1.86	1.787	1.788	0.017	13.98%	17.32%	17.28%	97.02%
100%	1.606	0.626	1.866	1.852	1.852	0.021	15.46%	16.08%	16.08%	96.65%
Perc.					**Set2**					
20%	14.159	2.244	17.169	14.802	14.809	0.007	-4.71%	9.72%	9.68%	99.69%
40%	14.061	4.402	17.979	16.587	16.613	0.015	2.54%	10.08%	9.94%	99.66%
60%	14.456	3.305	16.96	16.276	15.851	0.02	4.40%	8.25%	10.64%	99.39%
80%	15.151	5.685	18.145	17.724	18.217	0.03	12.77%	14.79%	12.43%	99.47%
100%	15.223	6.2	17.32	17.769	17.839	0.036	18.98%	16.89%	16.56%	99.42%

Table 10.1.: Benchmark results for Lazy, Greedy, Type-Aware, Type-and-Reference-Aware caching for Set0, Set1 and Set2 (* in the table represents the results for G,T and TR collectively).

Aware and Type-and-Reference-Aware respectively). Since the execution time of the EOL programs for *G*, *T* and *TR* is practically the same[5], only one result for all three of these approaches is presented under the *** column.

Acronym *Imp.* denotes the performance improvement of a certain approach, *Load* denotes the time it takes to load the models, whereas *Exec.* denotes the time it takes to execute the EOL programs. Finally, *Total* denotes the time it takes to load the model and execute an EOL program for a single experiment.

[5]This is expected as all three strategies populate all caches required before the EOL program executes.

Perc.	L Load	L Exec.	G Load	T Load	TR Load	* Exec.	Imp.G Total	Imp.T Total	Imp.TR Total	Imp.* Exec.
	sec.		sec.	sec.	sec.	sec.	%	%	%	%
					Set3					
20%	34.199	8.706	38.096	34.17	33.753	0.017	11.17%	20.32%	21.29%	99.80%
40%	31.786	9.756	37.552	35.086	34.809	0.028	9.54%	15.47%	16.14%	99.71%
60%	31.835	12.222	37.528	36.516	35.662	0.045	14.72%	17.01%	18.95%	99.63%
80%	32.417	11.456	39.301	39.302	37.795	0.068	10.27%	10.26%	13.70%	99.41%
100%	35.872	13.7	38.659	40.779	40.513	0.071	21.87%	17.59%	18.13%	99.48%
Perc.					**Set4**					
20%	36.133	7.586	43.745	39.477	37.278	0.018	-0.10%	9.66%	14.69%	99.76%
40%	37.99	12.973	43.515	41.044	41.01	0.039	14.54%	19.39%	19.45%	99.70%
60%	36.457	14.131	44.883	42.348	41.055	0.05	11.18%	16.19%	18.75%	99.65%
80%	37.782	11.762	41.932	44.038	45.168	0.065	15.23%	10.98%	8.70%	99.45%
100%	37.617	14.563	44.813	46.914	43.406	0.078	13.97%	9.94%	16.67%	99.46%

Table 10.2.: Benchmark results for Lazy, Greedy, Type-Aware, Type-and-Reference-Aware caching for Set3 and Set4(* in the table represents the results for G,T and TR collectively).

From the benchmarks, it is observed that with the *Greedy*, *Type-Aware* and *Type-and-Reference-Aware* approaches, programs execute significantly faster than with the *Lazy* approach. These approaches require more time upfront to load the models due to the overhead incurred by their respective caching logic; such overhead affects the performance for small data sets (set 0 in this case).

However, as the sizes of models grow, these approaches provide marginal benefits in terms of the time it takes to load a model and to execute an EOL program (total time). In general, *TR* provides better performance but for some cases in which *TR* needs to visit elements deep in the containment tree, *T* and *G* marginally outperform it. In terms of memory footprint, the three approaches behave very similarly and incur a small linear overhead compared to *L*.

10.2.8. Summary

This section discussed a novel approach for computing and caching of the results of calls to *allInstances()* (and similar) operations based on static analysis of programs written in EOL. In Section 10.2.7, the benchmarking results for running the *lazy* approach, the *greedy* approach, the *type-aware* approach and the *type-and-reference-aware* approach were provided and compared. The benchmark results reveals that the proposed strategy exhibits significant performance improvements.

10.3. SmartSAX: Towards Partial Loading of Large XMI Models

According to [27], one significant emerging concern for scalability in the context of MDE is the scalability of tools when accessing large models. As discussed in Section 2.2.1, XML Metadata Interchange (XMI) is an XML-based model interchange format standardised by the OMG. It is the default model persistence format of the widely used Eclipse Modelling Framework (EMF) [6] and is supported (typically as an import/export option) by the majority of UML modelling tools. However, as the size of XMI-based models increase, the general consensus is that working with such models does not scale in terms of loading time and memory consumption [31, 81], because XMI parsers typically need to load an XMI model into memory in its entirety before any queries can be evaluated against it.

In response to this limitation, several database-backed model persistence prototypes have been proposed for storing very large models, such as Morsa [31], Neo4EMF [81], MongoEMF [32], EMF Fragments [133] and Hawk [87]. The general idea behind these tools is that they are able to load only the parts of a model that are needed for the task at hand (e.g. to compute particular queries), so that large models can be accessed efficiently both in terms of loading time and memory consumption. On the downside, these model representation formats are non-standard and, as such, they can adversely impact tool interoperability.

This section presents *SmartSAX*, an alternative solution which is able to "partial" load XMI-based models.The hypothesis of *SmartSAX* is that partially loading XMI-based models is feasible and beneficial in terms of improving loading performance and lowering memory footprint. In particular, given in-advance, the knowledge of the parts of an XMI model that are needed to compute a particular query/transformation (e.g. obtained through static analysis of the query/transformation itself), an intelligent XMI parser can skip irrelevant elements while reading the model file and (selectively) only load/process the elements of interest into memory.

10.3.1. Background

This section briefly discusses the background of *SmartSAX*. This section provides an overview of how SAX parser works, how EMF implements SAX to load XMI-based models, and the default algorithm for parsing XMI-based models.

Java SAX Parser

A Java SAX (Simple API for XML) XML parser is an event-based XML parser that operates by going through an XML file/stream and invoking callback methods on a *listener/handler* object (which subclasses SAX's *DefaultHandler* built-in class) when it encounters certain structural elements of the XML file. For example, the parser invokes the handler's *startDocument()* method when the start of the XML document is encountered, its *startElement()* method when the start of an element is encountered, etc. It is the responsibility of the handler to extract the information it needs from these elements (by implementing the callbacks methods where appropriate).

Default XMI Parsing Algorithm

To illustrate the default XMI parsing algorithm implemented by EMF, the *University* metamodel is used, provided in Figure 10.12. A sample XMI representation of a model that conforms to the *University* metamodel is provided in Figure 10.13. This model contains a *University* element, which in turn contains a *Department* (*Computer Science*) element. Under *Computer Science*, there are two members: a *Lecturer* (Tom Brown) and a *Student* (Cathy Smith). Tom has a webPage while Cathy does not. Under *Computer*

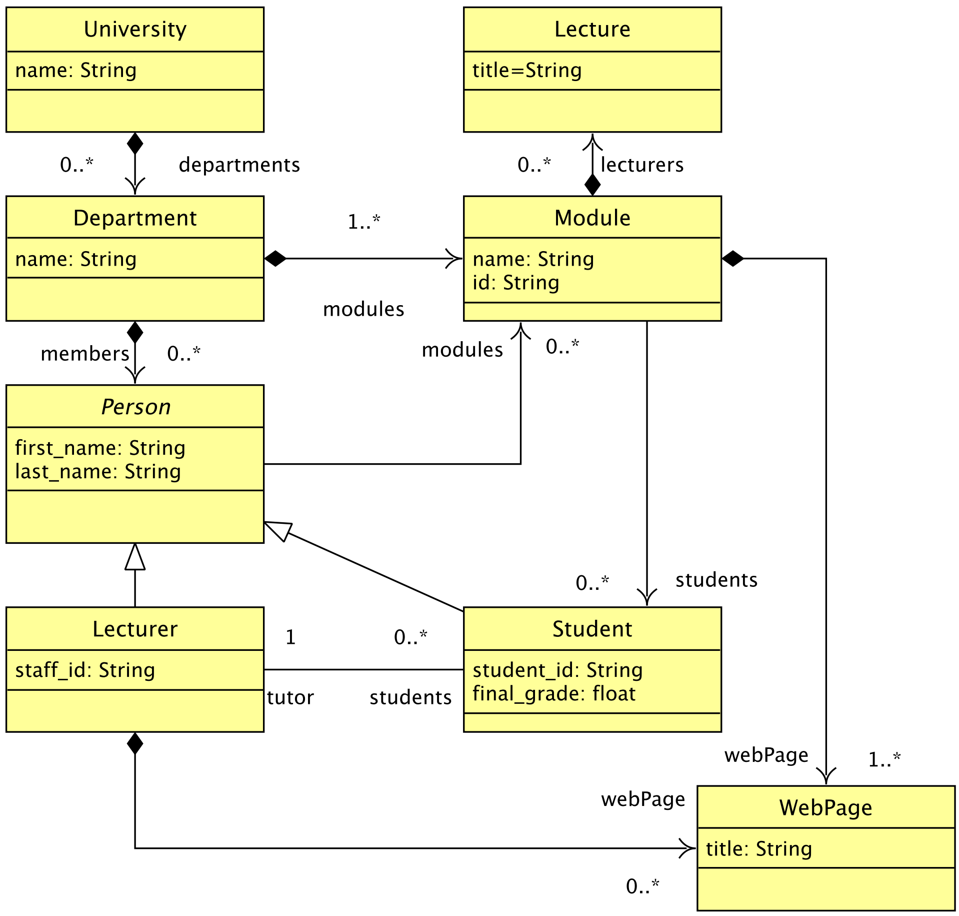


Figure 10.12.: The University Metamodel

Science, there is also a *Module* element: *MODE*, which also has a webPage. Finally, both Tom and Cathy are involved in MODE (see *modules="e6"* in lines 6 and 10).

EMF's XMI parser, in particular, the *SAXXMIHandler* component, maintains a stack of model elements (*EObjects* in EMF's terminology) to keep track of its position in the XMI document. This is needed in order to determine what *EObjects* to create next, as illustrated in the lower part of Figure 10.13. When line 1 of the XMI file is read, the callback method *startElement()* is triggered, the <*university*> element is handled and a new instance of *University* (with its *name* attribute set to *UoY*) is created. The new *EObject* is pushed into the object stack. When line 4 is read, the parser processes the top of the stack (*peekObject* in EMF's terminology) together with the <*departments*>

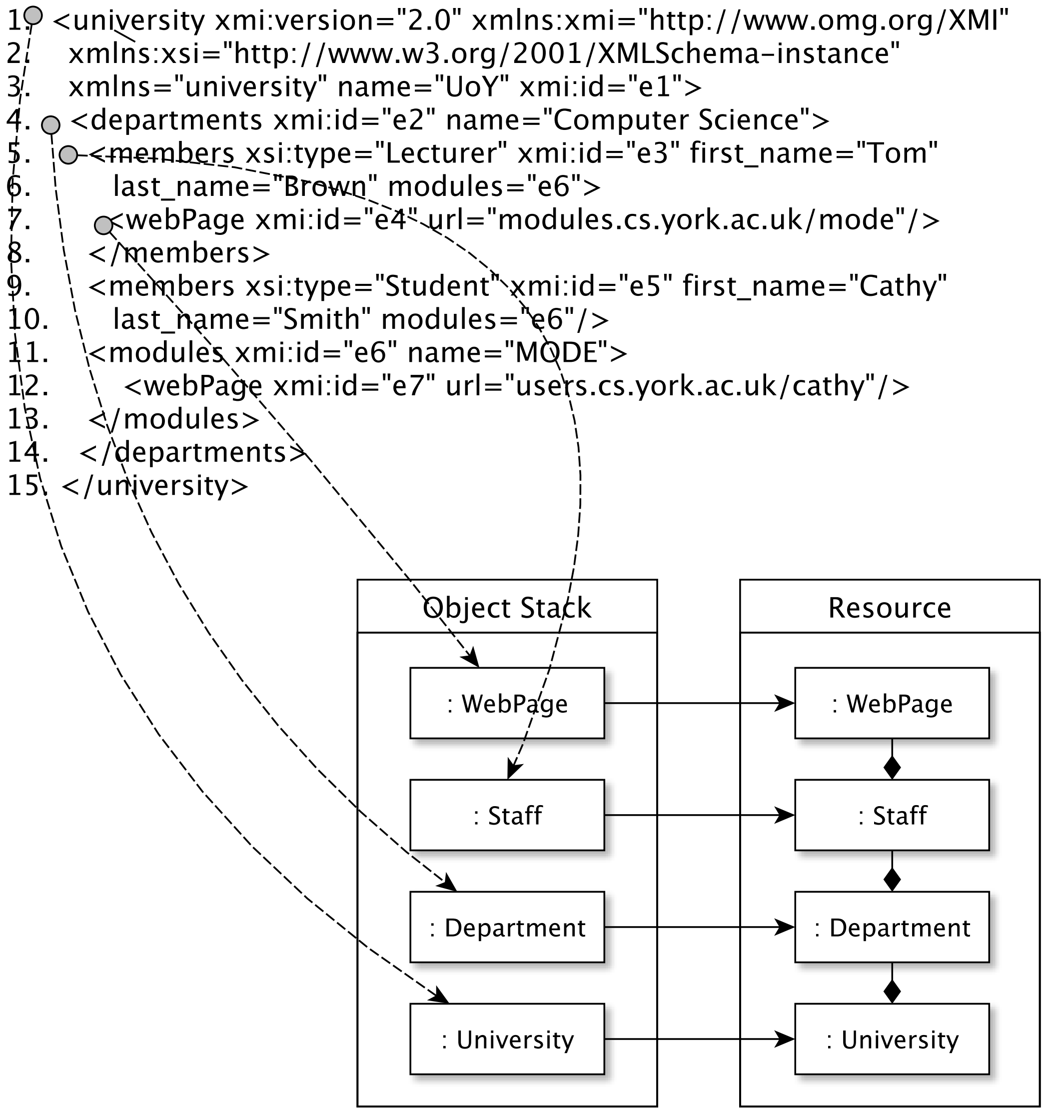


Figure 10.13.: Parsing a University XMI model using EMF's built-in XMI parser.

element and decides that an instance of *Department* should be created and added to the *departments* reference of the *University* model element. The created instance of *Department* is also pushed into the object stack. The same principle is applied when line 5 is read: the element <*members*> is handled and an instance of *Lecturer* is created, added to the *members* reference of the *Department* and pushed into the stack. When an element tag ends (e.g. in line 8), the top element of the object stack is popped.

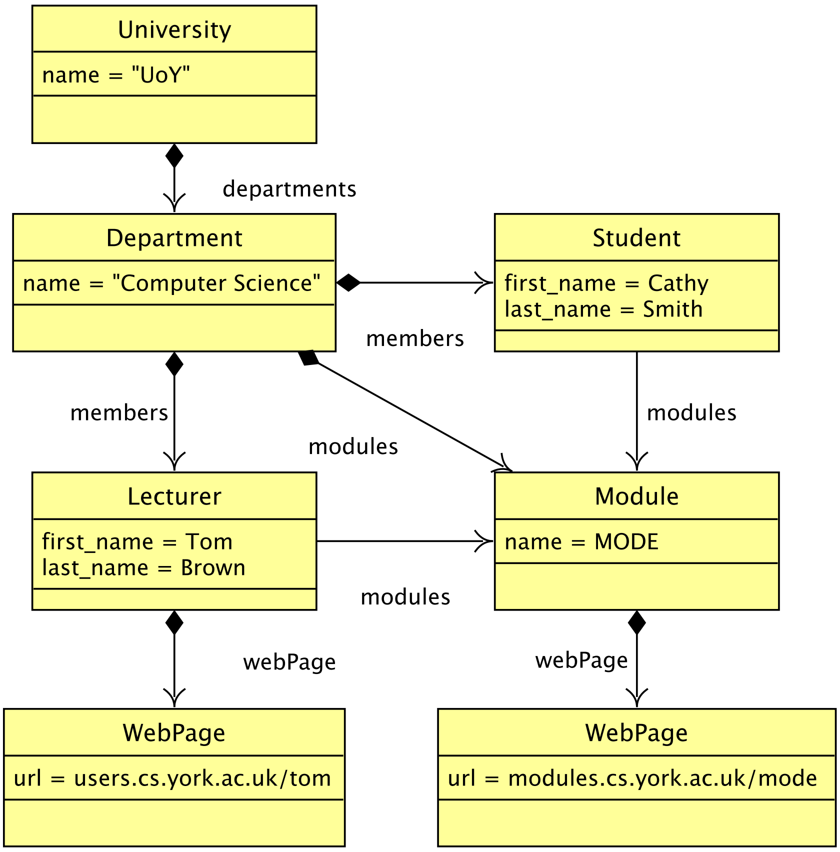


Figure 10.14.: An example University model.

Once all XML elements have been processed, a tree structure has been constructed in memory and resolution of non-containment references (e.g. *Member.modules*) takes place to transform the tree into the graph shown in Figure 10.14.

10.3.2. Partial XMI Loading

While parsing the entire contents of an XMI-based model into an in-memory object graph is often necessary (for example, when it is not known in advance which elements of the model will need to be accessed by a program/user), there are also cases where only parts of the model need to be loaded, and precise information about which parts are relevant/needed can be provided in advance. For example, when a model is loaded in order to be queried by a program (e.g. a set of OCL constraints or an M2M/M2T transformation), it is possible to detect through static analysis which parts of the model the program is likely to exercise. In such cases, loading parts of the model that the program is guaranteed not to access is inefficient both in terms of loading time and in terms of memory footprint.

To improve support for working with large XMI-based models in such scenarios, in the following sections, an algorithm that is used to load only *EObject*s of interest into memory and ignore irrelevant XML elements are explained in detail.

10.3.3. Effective Metamodel

Partial loading is typically associated with analysing model management programs. Suppose an EOL program p, which manipulates an underlying model m, which conforms to its metamodel mm. *SmartSAX* needs some in-advance information for it to partial-load XMI models, that contains information on which part(s) of m are needed by p. Such information, in [23], is referred to as the *effective metamodel* of mm extracted from p. *Effective metamodel* is a subset of the underlying metamodel under question. There are also cases that the *effective metamodel* extracted from a program is identical to the underlying metamodel. In this case, it means that the program exercises instances of all types in the underlying metamodel and all attributes and references of the types are accessed by the program as well.

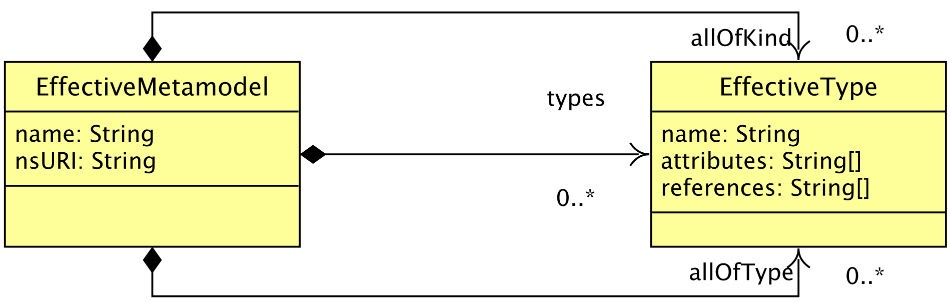


Figure 10.15.: Effective Metamodel Representation

Figure 10.15 illustrates how effective metamodels are represented in *SmartSAX*. Let symbols p, m, mm and em denote respectively the EOL program under question, the model that p manages, and metamodel of m and the effective metamodel extracted from analysing p. The base construct of the proposed *Effective Metamodel* structure is the *EffectiveType*, which represents a type in mm, *EffectiveType* contains a *name*, a collection of *attributes* and a collection of *references* of interest. The em of m is represented by *EffectiveMetamodel*, which has a *name*, an *nsURI* (mm's globally unique identifier), and three collections of *EffectiveType* (*types*, *allOfKind* and *allOfType*).

allOfKind, allOfType and types

The *allOfKind* and *allOfType* references are used to specify the types, instances of which need to be loaded by the parser. For example, if *allOfKind* of *Person* is declared in the effective metamodel, it implies that instances of both *Lecturer* and *Student* should be loaded (as they have the *kind-of* relationship with *Person*). In contrast, if *allOfType* of *Person* is declared in the effective metamodel, only instances of *Person* should be loaded (in this case no element will be loaded since *Person* is abstract). The *types* reference specifies types, instances of which should be loaded only when they appear under containment references of interest. For example, in the effective metamodel of Figure 10.16, it specifies that all instances of *Lecturer* need to be loaded regardless of their positions; but only instances of *WebPage* that are contained in containment references of interest will be loaded (for example, *Lecturer.webPage*).

Attributes and References

In each *EffectiveType*, names of the attributes and references that need to be populated can be declared. For example, if it is declared that only the *first_name* attribute is needed for *Student* elements. The partial loading parser will only populate the value of the *first_name* attribute of *Student*s (and not any other attributes or references of *Student*).

10.3.4. Automated Effective Metamodel Extraction

Atop the Epsilon static analysis framework, an *Automated Effective Metamodel Extraction* facility (AEME) is constructed. AEME works with the variable-resolved and type-resolved EOL Heterogeneous Abstract Syntax Graph and extracts the effective metamodel by looking at operation calls to *all()*, *allOfKind()*, *allOfType()* and *allInstances()*. If such operation calls are found, their corresponding *target* types are recorded and added to the *Effective Metamodel*. AEME also looks for *property* calls that for which the *target* expression of the *property* call is of *ModelElementType* (T). If this is the case, the *property* is added to either the *attribute* or the *reference* of T.

```
let EM = new effective metamodel;
foreach method call expression do
    if the type of the target of the method call is a model element type then
        if the name of the method is all(), allOfKind() or allInstances() then
            let ET = create new / retrieve existing effective type for the method call's target
            model element type;
            if ET is already under EM's allOfType reference then
                // allOfKind is a superset of allOfType
                move ET under EM's allOfKind reference;
            end
            else
                add ET under EM's allOfKind reference;
            end
        end
        else if the name of the method is allOfType() then
            let ET = create new / retrieve existing effective type for the model element type;
            if ET is not already under EM's allOfType or allOfKind reference then
                add ET under EM's allOfType() reference;
            end
        end
    end
end
```

Algorithm 5: Effective Metamodel Extraction Algorithm (1 of 2)

```
for(s in Lecturer.allOfKind()) {
  ("First name:" + s.first_name).println();
  ("Last name:" + s.last_name).println();
  ("Web page:" + s.webPage.url).println();
  ("Number of modules taught:" + s.modules.size()).println();
}
```

Listing 10.7: An example EOL Program

Algorithm 5 and 6 show the algorithm for matching operation calls and property calls which populates the *Effective Metamodel* automatically. These algorithms, in principle, can also be applied to other model management languages such as OCL and ATL, so that they can work seamlessly with SmartSAX.

foreach *property call expression* **do**
  if *the type of the target of the property call is a model element type* **then**
    if *the name of the property is* all **then**
      // .all is a shorthand notation for *.all()/.allInstances()/.allOfKind()* // treat this as a call to *allOfKind()* – see Algorithm 5
    end
    else
      let ET = create new / retrieve existing effective type for the model element type;
      if *ET is not already under the EM's* types, allOfKind *or* allOfKind *references* **then**
        add ET under EM's *types* reference;
      end
      if *the property is an attribute* **then**
        add the property to ET's attributes (if not already there)
      end
      else if *the property is a reference* **then**
        add the property to ET's references (if not already there)
      end
    end
  end
  else if *the type of target of the property call is a collection type* **then**
    if *the content type of the collection type is a model element type* **then**
      let ET = create new / retrieve existing effective type for the model element type;
      if *ET is not already under the EM's* types, allOfKind *or* allOfKind *references* **then**
        add ET under EM's *types* reference;
      end
      if *the property is an attribute* **then**
        add the property to ET's attributes (if not already there)
      end
      else if *the property is a reference* **then**
        add the property to ET's references (if not already there)
      end
    end
  end
end

Algorithm 6: Effective Metamodel Extraction Algorithm (2 of 2)

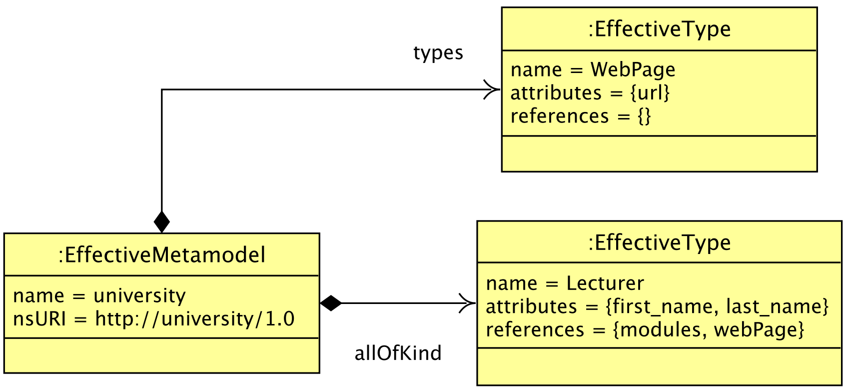


Figure 10.16.: Automatically-extracted Effective Metamodel from Listing 10.7

Applying Algorithm 5 and 6 on the EOL program in Listing 10.7 produces the effective metamodel illustrated in Figure 10.16.

10.3.5. Effective Metamodel Reconciliation

Effective metamodels specified manually, or extracted through static analysis of model management programs, can be incomplete. For example, the effective metamodel shown in Figure 10.16 specifies that the underlying program is interested in loading all of the instances of *Lecturer*, and in turn the *first_name* and *last_name* attributes and the *webPage* and *modules* references. It also specifies that for loaded instances of *WebPage*, their *url* attributes should be populated.

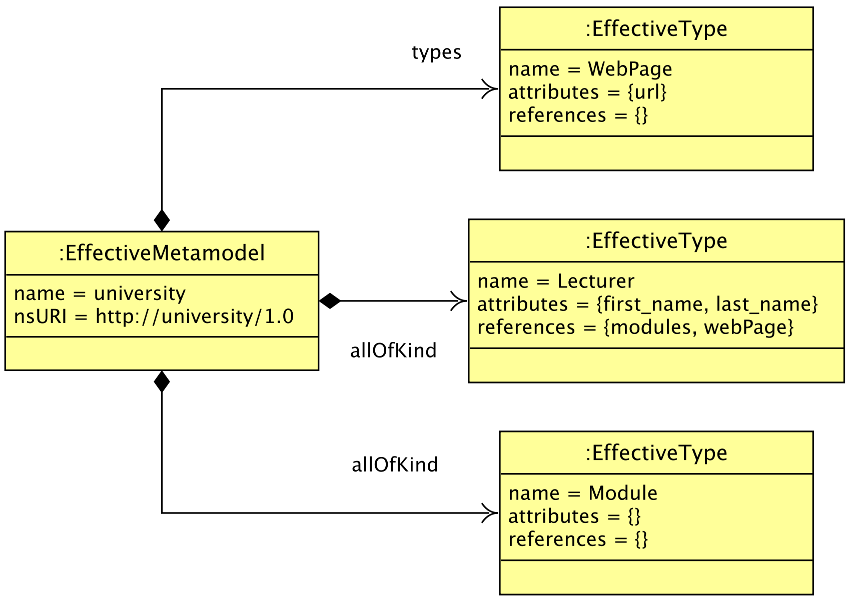


Figure 10.17.: Reconciled Version of the Effective Metamodel of Figure 10.16

As the extracted effective metamodel does not include the *Module* type (because the *modules* feature of type *Lecturer* is a non-containment reference), the parser will not load any instances of *Module*, and as such the *modules* reference of all loaded *Lecturer* elements will be empty. To address such cases, in this step, the AEME automatically reconciles a provided (potentially incomplete) effective metamodel by adding *allOfKind* relationships to the types of declared non-containment references. If an *allOfType* relationship already exists for that type, it is converted to an *allOfKind* relationship. The reconciled effective metamodel for this example, where the missing *allOfKind* relationship has been added for the *Module* type, appears in Figure 10.17.

10.3.6. Partial XMI Loading Algorithm

As discussed above, EMF's built-in XMI parser maintains a stack of *EObjects* to determine what type of *EObject* it needs to create when it encounters a new XML element, and where[6] it should place the new element in the containment hierarchy. While on one hand it is desirable to re-use as much of the (by far non-trivial) functionality of the existing XMI parser as possible; creating all *EObject*s in order to maintain the stack is not efficient in terms of time and memory consumption, and defeats the purpose of partial loading. Thus, a solution is proposed by SmartSAX to use a *placeholder cache*, which contains an empty/placeholder *EObject* for each type that is not declared in the effective metamodel. As such, when the parser encounters an XML element that it wishes to *skip* (means to read but not to process), it can fetch the corresponding placeholder *EObject* for that type from the cache and put it in the object stack, instead of creating an new *EObject*.

On the other hand, when it encounters an XML element of a type that is included in the effective metamodel under an appropriate *allOfType* or *allOfKind* reference, or an element that belongs to a containment reference of interest and is included in the effective metamodel under a *types* reference, it creates a new *EObject*. If the top element of the stack is not an placeholder *EObject* and the containment reference is included in the effective metamodel, it puts the new object under the containment reference; otherwise it adds it a top level element in the resource (model). Finally, the parser adds the new *EObject* to the top of the stack.

Figure 10.18 illustrates a snapshot of the state of *SmartSAX* at the point where it has parsed the University XMI model up to line 7, with reference to the reconciled effective metamodel shown in Figure 10.17. When the parsing starts, SmartSAX populates the *Placeholder Cache* with one placeholder *EObject* for each type of the full metamodel

[6] i.e. under which containment reference

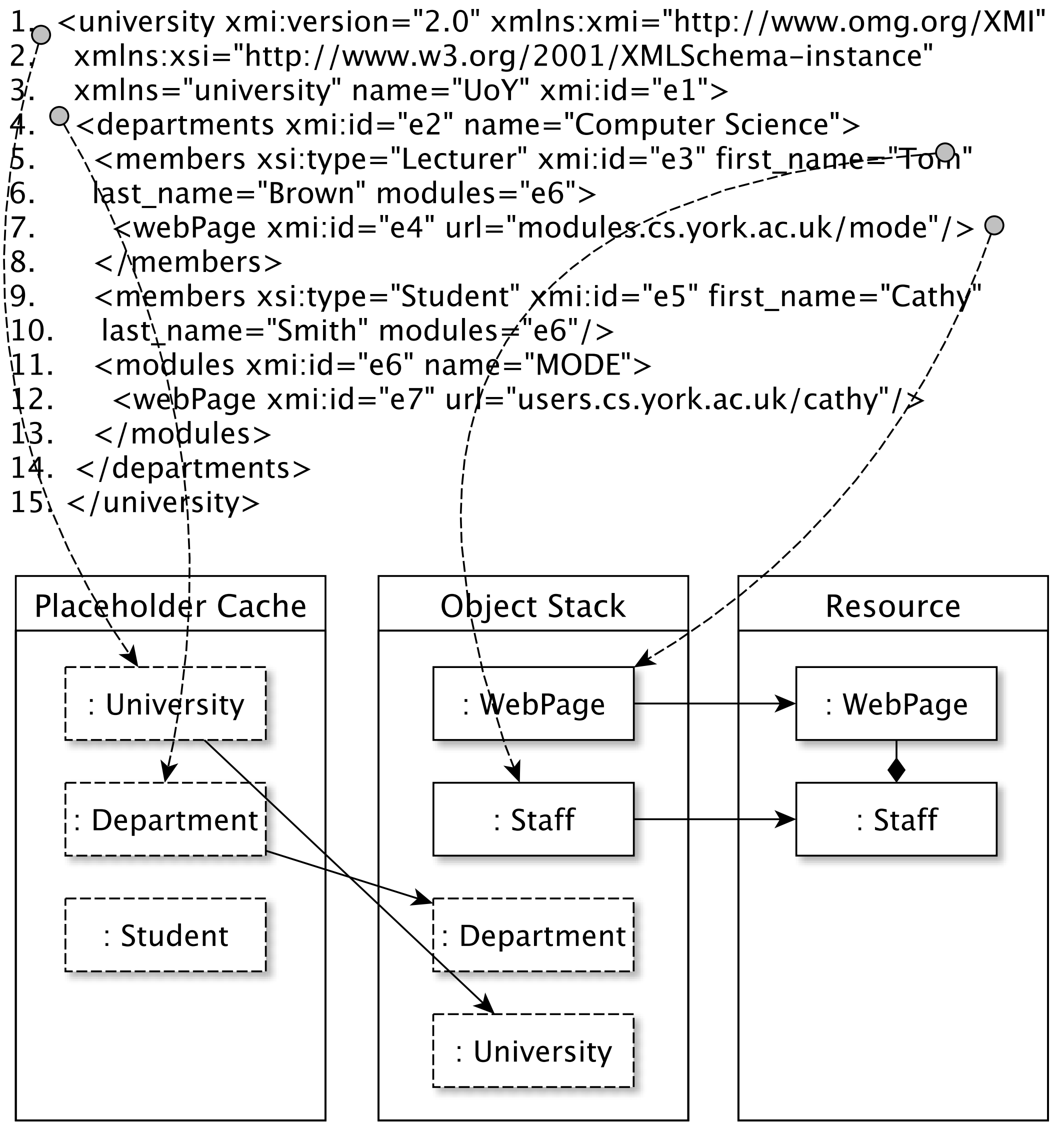


Figure 10.18.: Parsing the University model with SmartSAX.

that is not included in the effective metamodel[7] as illustrated on the bottom-left corner of Figure 10.18. It then handles the XML elements it encounters as follows:

- When the <*university*> element in line 1 is encountered, SmartSAX checks the effective metamodel, and determines that instances of *University* do not need to

[7]Another approach would be to populate the cache in a lazy manner - i.e. to only create placeholder *EObjects* the first time they are needed.

be loaded. Therefore, SmartSAX fetches the placeholder *EObject* of *University* from the placeholder cache and pushes it on to the object stack.

- When the <*departments*> element in line 4 is encountered, SmartSAX determines that the type of the object that should be instantiated is *Department*. However, according to the effective metamodel, instances of *Department* do not need to be loaded, so SmartSAX fetches the placeholder *EObject* of *Department* from the placeholder cache and pushes it on to the object stack.

- When SmartSAX encounters the <*members*> element in line 5 it determines that it needs to create a new instance of *Lecturer* (as *Lecturer* is part of the effective metamodel). After creating the new instance, it consults the effective metamodel and populates the values of its *first_name* and *last_name* attributes. Then it looks at the element on the top of the stack (currently the placeholder instance of *Department*), it detects that it is a placeholder, and as such adds the populated instance of *Lecturer* to the resource as a top-level element.

- When the <*webPage*> element in line 7 is encountered, SmartSAX determines that it needs to create an instance of *WebPage* and place it in the *webPage* containment reference of the top element of the stack. It also populates the *url* attribute of the new instance with the value of the respective attribute of the XML element.

- When </*webPage*> is encountered in line 7, the top object of the stack is popped (the current top element is now *Tom*)

- When </*members*> is encountered in line 8, the top object of the stack is popped (the current top element is now *Computer Science*)

- When the <*members*> element is encountered in line 9, SmartSAX determines that it needs to create an instance of *Student*. Since the *Student* type is not part of the effective metamodel, it fetches the *Student* placeholder object and puts it at the top of the stack.

- When </*members*> is encountered in line 10, the top object of the stack is popped (the current top element is still *Computer Science*).

- When the <*modules*> element is encountered in line 11, SmartSAX determines that it needs to create an instance of *Module* and since the effective metamodel declares that all instances of *Module* need to be loaded, it creates a fresh *EObject* (but does not populate any of its attributes/references as none of these need to be loaded according to the effective metamodel). Since the top element in the stack is a placeholder, it adds the new *Module* instance to the resource as a top-level element and also pushes it on to the stack.

- When the <*webPage*> element is encountered in line 12, SmartSAX determines that it maps to an instance of *WebPage* that should be placed under the *webPage* containment reference of the top element of the stack (which is currently the *MODE* module). Since the *WebPage* type is part of the *types* reference of the effective metamodel, and its containment reference (*Student.webPage*) is not of interest, the parser fetches the placeholder *WebPage* object and pushes it on to the stack.

- Each of the last three lines (13-15) cause the parser to pop the top element of its stack – thus ending up with an empty stack.

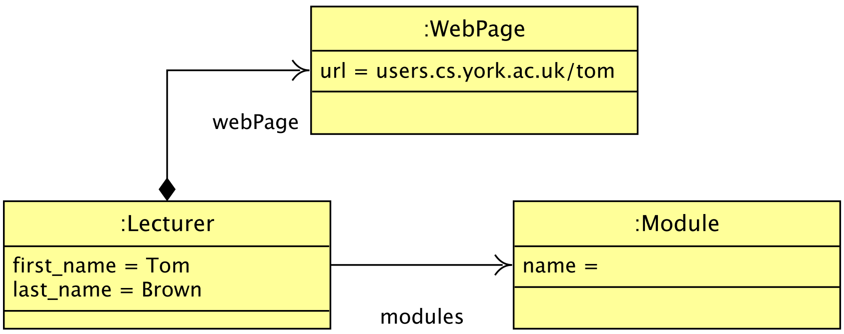


Figure 10.19.: The Partially Loaded Model

Since all required objects have been loaded, the last step of the algorithm involves resolving non-containment references (in this case, link *Tom* to the *MODE* module, through its *modules* reference). The obtained partially-loaded model appears in Figure 10.19. Note how the *name* attribute of the loaded *Module* is empty, as the value of this attribute is not of interest according to the effective metamodel of Figure 10.17.

```
let stack = new stack of model elements;
let cache = new set of model elements;
let model = new model;
let elements = new stack of xml elements;
let EM = the defined/extracted effective metamodel;
let referencesToHandle = non-containment references to resolve after file is fully read;
Procedure startElement(xmlElement)
    push xmlElement to elements;
    let peekModelElement = peek top model element of stack;
    if peekModelElement is nil then
        // We are at a root element
        let type = find a model element type for the tag name of the xmlElement;
        let modelElement = createModelElement(type);
        push modelElement to the stack;
    end
    else
        let peekModelElementType = the type of peekModelElement;
        if a feature needs to be created based on peekModelElementType and xmlElement then
            handleFeature(xmlElement);
        end
        else if a (top level) model element to be created then
            let type = find a model element type for the tag name of the xmlElement;
            let modelElement = createModelElement(type);
            push modelElement to the stack;
        end
    end
Procedure endElement(element)
    pop the top model element from the stack;
    pop the top model element from the elements;
```

Algorithm 7: Partial Loading Algorithm 1 of 3

The partial loading algorithm illustrated above is also presented in an example-independent manner in Algorithms 7, 8 and 9.

10.3.7. Benchmark Results

This section presents the results of benchmarks of SmartSAX for its partial loading algorithm presented in Section 10.3.6 to evaluatate the scalability and practicality of the proposed approach. Benchmarks were performed on a computer with Intel(R) Core(TM) i7 CPU @ 2.3GHz, with 8GB of physical memory, running OS X Yosemite. The version of the Java Virtual Machine used was 1.8.0_31-b13. Results are in seconds and Megabytes.

For the benchmarks, models of varying sizes obtained from reverse engineered Java code in the 2009 GraBaTs contest[8] are used. These models, named set0, set1, set2, set3 and set4 (9.2MB, 27.9MB, 283.2MB, 626.7MB, 676.9MB respectively) are stored in XMI 2.0 format and have been used for various benchmarks for different tools [87, 81].

[8]GraBaTs2009: 5th Int. Workshop on Graph-Based Tools, `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/`

```
let stack = new stack of model elements;
let cache = new set of model elements;
let model = new model;
let elements = new stack of xml elements;
let EM = the defined/extracted effective metamodel;
let referencesToHandle = non-containment references to resolve after file is fully read;
Procedure createModelElement(type)
    let modelElement = instance to be created/retrieved;
    if EM contains type under allOfKind/allOfType then
        modelElement = create an instance of the type;
        add modelElement to the resource;
    end
    else
        modelElement = create/retrieve cache object from the cache by type;
    end
    return modelElement;
Procedure handleObjectAttributes(eObject)
    foreach attribute in the current xmlElement do
        let name = name of the attribute;
        let value = the value of the attribute;
        if shouldHandleFeature(eObject, attribute) then
            setFeatureValue(eObject, name, value);
        end
    end
Procedure setFeatureValue(eObject, name, value)
    let feature = identify feature based on eObject and name;
    if feature is single valued then
        set value to feature;
    end
    else
        add value to feature;
    end
```

Algorithm 8: Partial Loading Algorithm 2 of 3

Loading Unit Coverage

To quantify partial loading in the benchmarks, this thesis uses the concept of *loading units*. For this purpose, three types of *loading units* are identified: objects (model elements), attribute values, and reference values. For example, the partially-loaded model in Figure 10.19, there are 8 loading units (3 objects, 3 attribute values (excluding *Module.name*) and 2 reference values), while the fully-loaded model of Figure 10.14 consists of 24 loading units (7 objects, 9 attribute values and 8 reference values).

With regard to the experiment, the first step is to count the number of loading units in each of the models (from set0 to set4). Then, EOL programs which achieve coverage of 20%, 40%, 60%, 80%, and 100% of the loading units for models from set0 to set4 were constructed. Finally, set0 to set4 were loaded using the effective metamodel extracted from the EOL programs and SmartSAX, the performance in terms of loading time and memory consumption were recorded and compared with the performance of the built-in

```
Procedure handleFeature(xmlElement)
    let peekModelElement = peek top model element of stack;
    let peekModelElementType = the type of peekModelElement;
    let feature = the feature that is to be created based on peekModelElementType and
    xmlElement;
    if feature is an attribute then
        handleObjectAttributes(peekModelElement);
    end
    else
        //feature is a reference
        if feature is a containment reference then
            let eType = the type of the reference;
            let modelElement = createModelElement(eType);
            if modelElement is null then
                if em contains eType under types then
                    if shouldHandleFeature(peekModelElement, feature) then
                        let modelElement = create an instance of the eType;
                        add modelElement to the resource;
                        setFeatureValue(peekModelElement, feature name, modelElement);
                    end
                end
                else
                    let modelElement = create/retrieve cache object from the cache for
                    typeToCreate;
                end
            end
            else
                add modelElement to the resource;
                setFeatureValue(peekModelElement, feature name, modelElement);
            end
            push modelElement to the stack;
        end
        else
            if shouldHandleFeature(peekModelElement, feature) then
                add feature to referencesToHandle;
            end
        end
    end
Procedure shouldHandleFeature(eObject, feature)
    let effectiveType = retrieve effective type from EM based on eObject;
    if effectiveType is not nil then
        if effectiveType contains the name of feature then
            return true;
        end
    end
    return false;
```

Algorithm 9: Partial Loading Algorithm 3 of 3

EMF XMI parser on the same models.

Percentage	Set0					
	Norm.(T)	Norm.(M)	Part.(T)	Part.(M)	Part.Impr(T)	Part.Impr(M)
20%	0.65	161	0.31	87	52.29%	60.11%
40%	0.65	161	0.47	96	28.57%	39.98%
60%	0.65	161	0.51	97	25.61%	38.91%
80%	0.65	161	0.60	126	6.08%	21.37%
100%	0.65	161	0.71	162	-4.12%	-0.63%
Percentage	**Set1**					
20%	1.97	419	0.92	128	53.46%	69.27%
40%	1.97	419	1.36	258	31.67%	38.54%
60%	1.97	419	1.51	285	23.83%	31.81%
80%	1.97	419	1.64	316	14.88%	23.08%
100%	1.97	419	2.01	416	-3.97%	-1.08%
Percentage	**Set2**					
20%	18.11	1891	6.49	893	64.12%	52.81%
40%	18.11	1891	9.67	1172	47.22%	38.24%
60%	18.11	1891	11.71	1389	35.72%	26.60%
80%	18.11	1891	14.10	1487	22.37%	21.35%
100%	18.11	1891	18.90	1893	-3.97%	-0.09%
Percentage	**Set3**					
20%	35.8	2285	17.25	836	50.94%	63.36%
40%	35.8	2285	21.21	1111	40.55%	51.25%
60%	35.8	2285	29.19	1221	18.94%	46.42%
80%	35.8	2285	33.23	1636	7.38%	26.42%
100%	35.8	2285	37.29	2300	-3.63%	-0.13%
Percentage	**Set4**					
20%	39.58	2501	15.25	584	60.47%	76.69%
40%	39.58	2501	25.91	1689	32.99%	33.03%
60%	39.58	2501	30.26	1807	24.62%	30.03%
80%	39.58	2501	35.47	2074	11.48%	18.59%
100%	39.58	2501	42.58	2560	-7.63%	-2.23%

Table 10.3.: Partial Loading GraBaTs models

Norm.(T)	Normal Loading Time
Norm.(M)	Normal Loading Memory
Part.(T)	Partial Loading Time
Part.(M)	Partial Loading Memory
Part.Impr(T)	Partial Loading Improvement in Time
Part.Impr(M)	Partial Loading Improvement in Memory

Table 10.4.: Terms explained for Table 10.3

Results

The obtained results are presented in Table 10.3. The terms of each column are explained in Table 10.4. From the benchmarks it is observed that for SmartSAX, the resource consumptions (time and memory) are linear with respect to the loading unit coverages.

For set0, SmartSAX demonstrate significant resource consumption improvements both in terms of time and memory. For 100% coverage, SmartSAX takes slightly more time and memory due to the upfront cost for effective metamodel extraction and reconciliation. During the parsing, the (redundant) comparisons with the effective metamodel also costs more time.

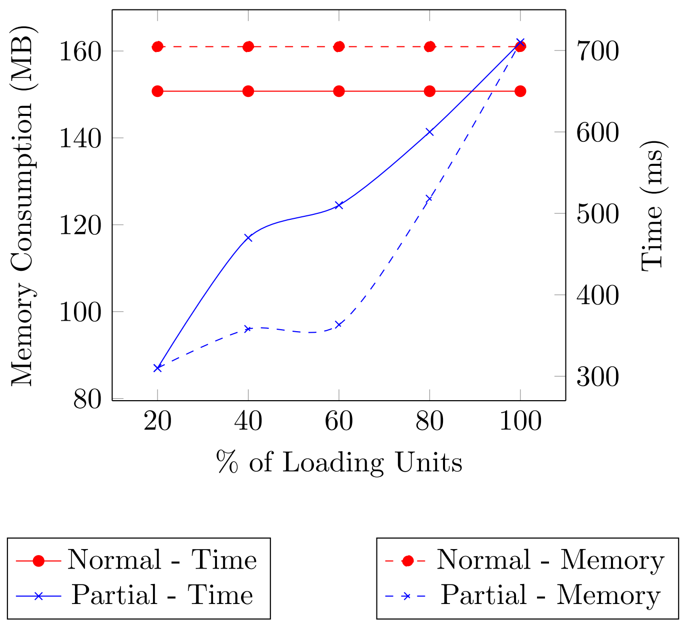


Figure 10.20.: Benchmark results for Set 0

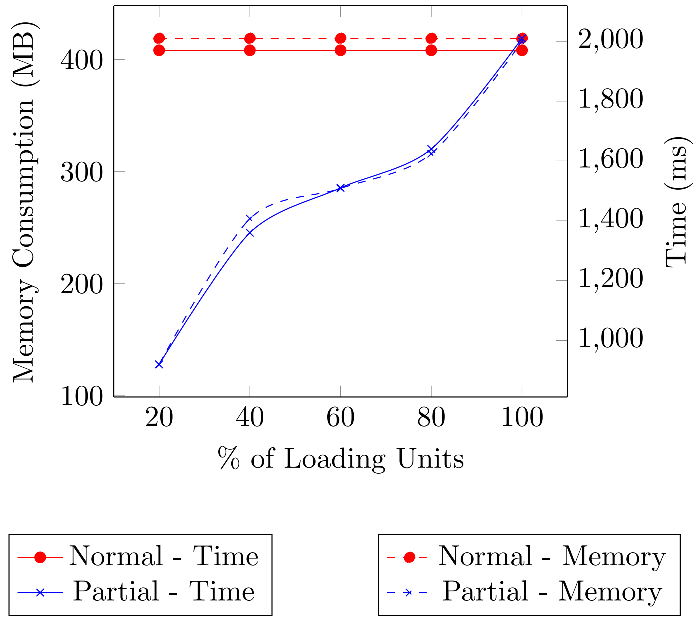


Figure 10.21.: Benchmark results for Set 1

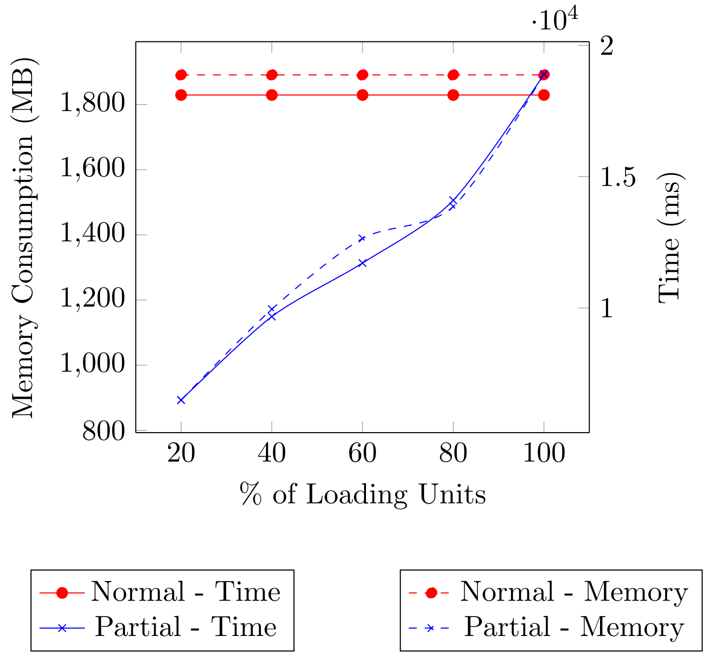


Figure 10.22.: Benchmark results for Set 2

For set 1 to set 4, significant improvements (at least 10% time improvement and 20% memory improvement up to 80% of coverage) are observed. The obtained results are plotted in Figures 10.3.7-10.3.7 for all five data sets. It is worth noting that the time

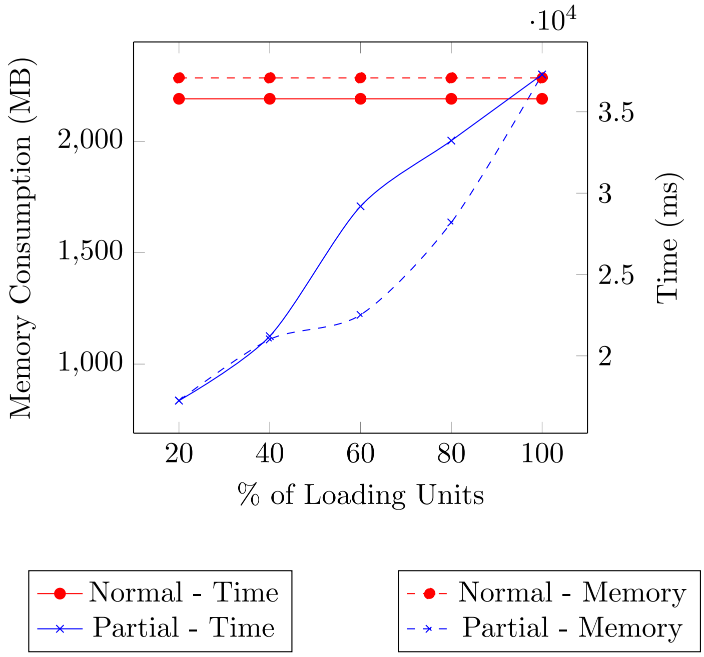


Figure 10.23.: Benchmark results for Set 3

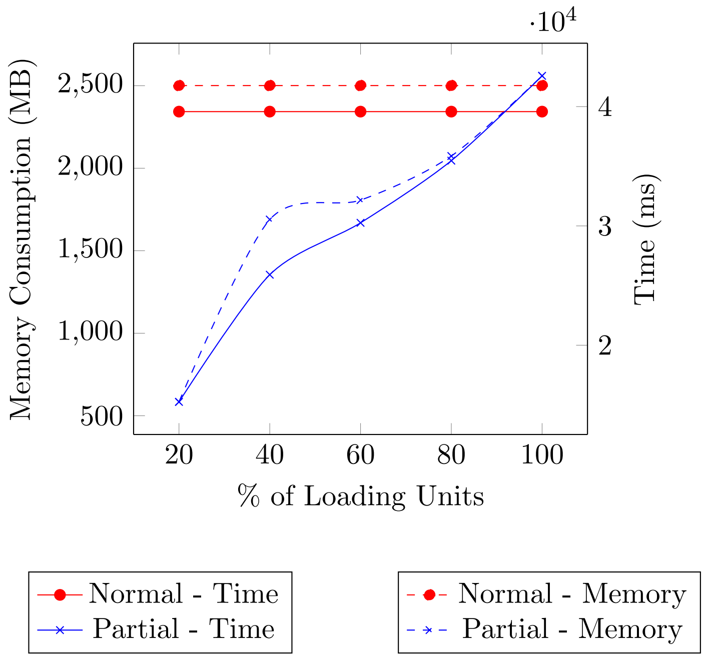


Figure 10.24.: Benchmark results for Set 4

consumption and memory consumption are not proportional as some *attributes* of the *EObjects* contain relatively large amount of Strings therefore taking more memory. Such situation can be perceived for set0, set3 and set4.

GraBaTs query

To further assess the performance benefits it delivers, SmartSAX was used in conjunction with Epsilon and the Epsilon static analysis framework to execute a complex query (proposed in the context of the Grabats 2009 competition), which detects singletons in the reverse-engineered Java models (set0 - set4). The Epsilon static analysis framework was used to extract the effective metamodel from the query, which was then used by SmartSAX to load models set0 - set4. The query is then executed on the models. The loading time, execution time and memory consumption for both full loading and partial loading were measured and are illustrated in Table 10.5 (the header labels in Table 10.5 are explained in Table 10.6).

	Set0	Set1	Set2	Set3	Set4
Norm.L(T)	0.65	1.98	16.85	38.58	40.71
Norm.E(T)	0.11	0.12	2.77	7.24	7.42
Norm.L(M)	161	419	1891	2268	2444
Part.L(T)	0.17	0.37	10.20	25.07	27.57
Part.E(T)	0.01	0.03	1.83	5.12	5.32
Part.L(M)	32	52	578	1524	1515
Part.L.Impr(T)	71.65%	81.43%	39.47%	35.02%	32.28%
Part.E.Impr(T)	89.38%	71.55%	33.95%	29.29%	28.31%
Part.L.Impr(M)	80.00%	87.53%	69.48%	32.81%	38.01%

Table 10.5.: Partial Loading and GraBaTs Models and Executing GraBaTs Query

For all data sets, SmartSAX demonstrates substantial benefits for loading. Set0 is loaded 71.65% faster than and set4 is loaded 32.28% faster compared to the full loading algorithm. It also takes less to compute the query on partially loaded models (as there are fewer elements to traverse in the model). The query on the partially loaded set0 is computed 89.38% faster and the query on the partially loaded set4 is computed 28.31% faster compared to their fully-loaded equivalents. In terms of memory consumption, partial loading set0 consumes 80.00% less memory and set 4 consumes 38.1% memory.

Norm.L(T)	Normal Loading Time
Norm.E(T)	Normal Execution Time
Norm.L(M)	Normal Loading Memory
Part.L(T)	Partial Loading Time
Part.E(T)	Partial Execution Time
Part.L(M)	Partial Loading Memory
Part.L.Impr(T)	Partial Loading Improvement in Time
Part.E.Impr(T)	Execution Improvement for Partial Loading in Time
Part.L.Impr(M)	Partial Loading Improvement in Memory

Table 10.6.: Terms explained for Table 10.5

10.3.8. Limitations

There are two noteworthy limitations for *SmartSAX*. First, it requires elements referenced from non-containment references to have IDs that do not depend on their position in the containment hierarchy (i.e. *intrinsic ID*s or *extrinsic ID*s instead of *fragment path* IDs [6] such as *//@departments.0/@modules.0*) as partial loading re-arranges the order of the objects loaded into memory, so fragment path becomes incorrect. Second, it does not support propagating changes made to the partially-loaded model back to its original XMI source (i.e. it is only useful for read-only operations on models).

10.3.9. Summary

This section discussed a novel approach for partial-loading XMI-based models by combining the Epsilon static analysis framework with an enhanced SAX parser which is built by extending EMF's existing SAX parser. The benchmark results presented in Section 10.3.7 indicate that *SmartSAX* brings significant improvements to both loading time and memory consumption.

10.4. Chapter Summary

This chapter presented three applications of the Epsilon static analysis framework which aim for performance analysis and optimisation of model management programs. Section 10.1 presented a facility for detecting sub-optimal performance patterns that may exist in EOL programs. Section 10.2 discussed how the Epsilon static analysis framework can be used to optimise the caching strategy for calls to *allInstances()*, And Section 10.3 discussed how static analysis can be used to achieve partial-loading of large XMI models to reduce both loading time and memory consumption. Although developed for the Epsilon platform, these applications can be ported to other model management tools (and to be used in conjunction with their corresponding static analysis facilities) to achieve same functionalities.

10.5. Terminology

Sub-Optimal Performance Pattern Detection (SPPD): SPPD is an extention of the Epsilon static analysis framework and is an application of static analysis. SPPD allows the developers to express performance bottleneck patterns so that these patterns can be matched against the programs they wish to check. Patterns can be expressed in *Epsilon Pattern Language* (EPL) or Java although the EPL approach is cleaner and easy to comprehend.

Epsilon Pattern Language (EPL): The Epsilon Pattern Language contributes pattern matching capabilities to the Epsilon platform. Using EPL, model patterns can be expressed which can be matched against models. EPL is built atop EOL.

The allInstances() operation: In model query and transformation languages, an important task to perform is to retrieve collections of model elements of a particular type/kind. OCL, QVTr and ATL provides the built-in allInstances() operation to achieve this function.

Effective metamodel: In the context of model management, the term *effective metamodel* refers to the footprint of a model management operation on the metamodel it manages. In theory, a model management operation only accesses a part of a model. Extracting the effective metamodel gives an idea of how much of a model is accessed by a model management operation.

11. Conclusions

This thesis has investigated the topic of automated analysis and validation of programs - as a means for error detection and performance optimisation of model management programs. In particular this thesis investigated the validity of the following research hypothesis:

Reusable static analysis facilities can be used to identify errors in different types of model management programs (e.g. model transformations, validation constraints) that operate on multiple models defined using diverse modelling technologies, and to enhance the performance of programs operating on large models.

To explore the thesis hypothesis, the following research objectives were identified:

- To build a static analysis framework for the Epsilon platform atop which, reusable static analysis tools can be developed;
- To build a facility which supports the analysis of programs that manage models defined in diverse modelling technologies;
- To use the framework to develop static analysis tools for the Epsilon model management languages to demonstrate its reusability and extensibility;
- To use the static analysis framework to develop facilities for analysis and automated optimisation of the performance of programs operating on large models.

The remainder of this chapter summarises the contributions of the thesis in relation to the research hypothesis and the research objectives, and provides a direction towards potential future extensions of the current state of the research.

11.1. Review Findings

In Chapter 2, a background review of Model Driven Engineering was performed. The review covered the terminology and principles of MDE, including the concepts of models, modelling languages and metamodels. The review then moved onto common model management operations in an MDE-based development process, a non-exhaustive list of which includes:

- Model-to-model Transformation
- Model-to-text Transformation
- Text-to-Model Transformation
- Model Validation
- Model Comparison
- Model Merging

Existing languages and tools that support these tasks were also identified and reviewed. The Eclipse Modelling Framework (EMF) and the Epsilon platform, which are closely related to the aims of this thesis were then reviewed. The review on MDE identified a number of challenges, including the need to ensure the correctness of model management programs, and the need to address a number of scalability challenges.

In Chapter 3, a background review of static analysis was performed. A number of characteristics of static analysis were introduced and a number of static analysis techniques were also reviewed.

In Chapter 4, a comparative study of a number of static analysis tools within the context of MDE was performed. To compare such static analysis tools, a number of criteria were identified. The review concluded that existing static analysis tools had a number of limitations:

- They are built to target only independently developed model management languages that support a limited range of model management tasks;
- They lack support for performing static analysis on programs that manage models defined in different modelling technologies;
- They do not provide support for runtime performance optimisation based on the results of static analysis.

11.2. Proposed Solution and Prototype

In Chapter 5, the research hypothesis was stated and a number of research objectives were identified. The following sections summarise the components of the Epsilon static analysis framework developed to achieve the research objectives.

11.2.1. Epsilon Static Analysis Model Connectivity Layer (ESAMC)

The Epsilon platform provides an abstract layer to support models defined in diverse modelling technologies, the Epsilon Model Connectivity layer (EMC). However, as discussed in Chapter 6, EMC is not suitable for static analysis as it does not provide support for inspecting the type hierarchy of the metamodels of models involved in model management operations. Therefore, an enhanced version of EMC, the Epsilon Static Analysis Model Connectivity layer (ESAMC), was designed and implemented. ESAMC natively supports Ecore, and provides interfaces for developing modelling technology specific drivers. The extensibility of ESAMC was validated in Section 9.1 where an XML driver for ESAMC was created, which is able to infer meta element structures from XML documents.

11.2.2. EOL Metamodel, the AST2EOL Transformation and the EOLVisitor Facility

For EOL programs to be analysed, there is a need to convert them into structures that can be easily queried and traversed. Epsilon executes EOL programs by first parsing them into homogeneous ANTLR-based Abstract Syntax Trees (ASTs). Section 6.4 provided a detailed discussion explaining that such ASTs are not suitable for static analysis. Thus, a design decision was made to create a metamodel for EOL using EMF's Ecore. Section 6.5 presented the language constructs of EOL and their corresponding meta elements in the EOL metamodel.

With the EOL metamodel in place, there was a need to convert the ANTLR-based ASTs produced by the Epsilon parser to instances of the EOL metamodel. For this purpose, an AST2EOL transformation facility was created in Java and was presented in Section 6.6. A facility for generating visitors from the EOL metamodel (EOLVisitor) was presented in Section 7.2.

11.2.3. EOL Static Analyser

The EOL static analyser was built by extending the EOLVisitor facility and was presented in Chapter 7. It contains two main facilities, the EOL variable resolution facility (presented in Section 7.3) which is responsible for establishing links between variable declarations and their references, and the EOL type resolution facility (presented in Section 7.4) which is responsible for resolving types of expressions within EOL programs. The EOL type resolution algorithm adopts the *lattice* theory to represent the EOL type system and a rule-based type resolving approach to resolve types of the expressions in the program.

11.2.4. EVL and ETL Static Analyser

The EVL static analyser, presented in Section 8.1, was built by extending the EOL static analyser. Beyond type checking, the EVL static analyser is also able to compute invariant dependency graphs, which may be used in the future work to optimise EVL program execution. The ETL static analyser, presented in Section 8.2, was also built by extending the EOL static analyser. Beyond type-checking capabilities, a transformation rule dependency graph computation facility was also built atop the ETL static analyser and was discussed in Section 8.2.5.

11.2.5. Applications of Static Analysis

As pointed out in Chapter 5, apart from error detection, the Epsilon static analysis framework also supports the implementation of facilities for performance analysis and optimisation of model management programs. The sub-optimal performance pattern detection (SPPD, presented in Section 10.1) is an application of the Epsilon static analysis framework, which enables the developers to define known code patterns that can cause performance degradation. Such patterns are then searched for in Epsilon programs so that performance bottlenecks can be discovered early in the development process.

Another scalability issue discovered from the study of Epsilon (and other querying languages such as OCL) is the execution time to compute the results for calls to the *allInstance()* operation which computes a set of all instances of a given type in a model. The drawbacks of the existing computation strategy of such operations (for contemporary tools) were identified in Section 10.2. This thesis then proposed a number of more efficient computation strategies for computing and caching the results of such operations using the results of the static analysis. The computation strategies are backed by the Epsilon static analysis framework's cache configuration extraction facility (discussed in Section 10.2.3). The static analysis is integrated with the EOL runtime and benchmark results for running EOL programs that interact with very large models were presented in Section 10.2.7.

SmartSAX, discussed in detail in Section 10.3, is another application of static analysis, which adds support for partial XMI model loading by leveraging the results of static analysis of model management programs. SmartSAX exploits the *effective metamodel*s extracted from model management programs and achieves partial loading of XMI-based models (based on the *effective metamodel*s) to improve resource consumption (time and memory) during model loading.

11.3. Evaluation Results

In Chapter 9, the validity of the proposed hypothesis presented in Section 5.2 was assessed. In Section 9.1, the extensibility of the ESAMC was demonstrated by the XML driver created atop it and example ETL transformations transforming XML models into EMF models were also provided. From this perspective, the hypothesis is validated in the sense that the Epsilon static analysis framework is able to analyse programs that simultaneously manage models defined using different modelling technologies.

Section 9.2 evaluated the EOL, EVL and ETL static analysers in terms of their ability to detect runtime errors. The evaluation covered testing plans and a number of examples which illustrate analysing widely used and complex programs were also presented.

11.4. Applications

Section 10.1 presented an extension of the Epsilon static analysis framework which aims at finding sub-optimal performance patterns in programs written in Epsilon languages. This facility addresses the scalability issues from the perspective of programming styles. It was used to evaluate a number of EOL programs found on GitHub and was able to detect (potentially) sub-optimal code from these programs.

Section 10.2 presented an extension of the Epsilon static analysis framework which aims at efficient computation of queries over models that contain millions of model elements. This facility addresses the scalability issues from the perspective of execution optimisation. Using the efficient computation strategy, resource consumption during program execution are significantly improved.

Section 10.3 presented an extension of the Epsilon static analysis framework (SmartSAX), which aims at enabling partial loading of XMI-based models that contain millions of model elements. This facility addresses the scalability issues from the perspective of model loading. The benchmark results of SmartSAX showed that partial loading significantly reduces the time and memory consumption compared to normal loading, and partial loading is applicable to general programs.

11.5. Summary of Contributions

This section summarises the contributions of this project to the Epsilon platform, and to model management in general.

11.5.1. Contributions to Epsilon

To investigate and assess the validity of the hypothesis of this thesis, a static analysis framework for languages the Epsilon platform was developed. This contributed the following facilities to Epsilon:

- Ecore-based EOL, EVL and ETL metamodels, which formalise the respective languages' abstract syntaxes;
- AST2EOL, AST2EVL and AST2ETL transformations, which transform ANTLR-based homogeneous abstract syntax trees into instances of EOL, EVL and ETL metamodels;
- Epsilon Static Analysis Model Connectivity layer (ESAMC), an enhanced version of Epsilon Model Connectivity layer (EMC) that provides interfaces for accessing metamodels defined in different modelling technologies in a uniform way;
- EOL, EVL and ETL static analysers, which formalise the scoping rules for variable resolution, and the type resolution semantics of the respective languages.

11.5.2. Contributions to Model Management

In terms of contributions that are not bound to Epsilon, this thesis demonstrated that:

- Meaningful static analysis of programs that involve models defined in diverse modelling technologies is feasible and practical.
- The results of static analysis can be used to reason about and to automatically optimise the performance of model manamgement programs operating on large models. More specifically by leveraging the results of static analysis:

  - Sub-optimal performance pattterns can be identified;

  - Efficient computation and caching strategies can be defined for computationally-expensive operations such as collecting all instances of a type in a model (*allInstances()*);

  - Partial loading of XMI models can be achieved.

11.6. Future Work

Static analysis in the context of MDE, as illustrated in this thesis, has great potential to develop applications that solve a broad range of existing/emerging problems. This section provides a number of research directions for future work.

11.6.1. Extending the EOL Static Analyser

The EOL static analyser can be extended in the future in a number of ways.

- The EOL static analyser can be extended to support the analysis of EOL programs that manage models defined in modelling technologies beyond EMF and schema-less XML. This is made possible by the Epsilon Static Analysis Model Connectivity (ESAMC) layer; drivers for other modelling technologies can be built atop ESAMC.
- This work aims to establish a static analysis framework for model management programs. However, some of the active research topic in the context of static analysis are not extensively explored. For example, EOL is a dynamically typed language, as pointed out in Section 9.3, the EOL static analyser adopts a naive approach - it makes best guess of the type of expressions of *Any* type. However, the type of an expression of *Any* type gets significantly more difficult to guess when its type changes in condition statements (such as *if* and *switch* statements). In the future work, the EOL static analyser should be able to handle this situation - the resolution of *Any* type using union types or similar approach. However, such tolerance to the use of *Any* type should also have its boundaries - it should not encourage the abuse of *Any* type, i.e. that the developers should not use an expression of type *Any* and assign values of different types to it. The boundary of tolerance would be (or at least should respect) that an expression with type *Any* to be assigned values of types that are inherently related - an warning/error should be issued for assignments to expressions of *Any* type which do not respect this boundary. There are at least two advantage if this approach is adopted: the static analyser is able to resolve types of most of the expressions (or possibly all),

and that the code gets clean and more clear and therefore is easier to comprehend and maintain.

- As previously discussed, the EOL static analyser should in the future provide support for type resolution of *Native* (Java) expressions. For variables (objects) of *Native* types, EOL supports direct method invocations (methods defined in their respective Java classes). Therefore to further help the developer detect potential errors, it is beneficial to expand the static analysis support to *Native* types.
- Most static analysis tools provide content assistance/completion facilities for developers. Although not a research contribution, implementing a content assist tool for EOL would be arguably beneficial to EOL developers.

11.6.2. Extending the Epsilon Static Analysis Framework

Currently, the static analysis framework supports the analysis of programs written in EOL, EVL and ETL. In the future work, supports for EGL (Epsilon Generation Language), ECL (Epsilon Comparison Language) EML (Epsilon Merging Language) and EWL (Epsilon Wizard Language) can be implemented to expand the static analysis to programs written in achieving different model management goals.

In addition, the analysis of programs should not be bound to one program at a time. In a typical MDE-based development process, model mangement programs are used in complex workflows. For example, before a model ($m1$) is used as the input model of a model-to-model transformation, it typically needs to be validated by a model validation operation ($mv1$). When the transformation is completed, the output model ($m2$) may also be validated by another model validation operation ($mv2$). While individually analysing the model management operations may discover potential runtime errors, analysing the model management operations collectively may discover more runtime problems such as dead code (transformation rules that are not executed) or inconsistencies (transformation rules that disregard the constraints described in $mv1$ or $mv2$). Therefore, expanding the analysis scope across different types of model management operations can be beneficial.

11.6.3. Additional Applications

In this thesis, three applications of the Epsilon Static Analysis Framework were presented and discussed. In the future work, several research additional directions can be pursued:

- Scheduling of model transformation rules. As discussed in Chapter 8, computing the rule dependency graph for model transformation is beneficial for the parallel execution of model transformations. However, in reality model transformation rules can typically get complicated making transforamtion scheduling a hard problem. To pursue this path, search-based meta-heuristics may be adopted to find an optimal solution for rule scheduling (either applied dynamically or statically).

- New model-management-operation-driven model persistence format. Analysing models/metamodels involved in a model management operation can be beneficial if more freedom is given to the persistence technique on how the structure of the models can be re-arranged within the persistence format so that partial loading/saving can be achieved (in the sense that the entirety of the model does not need to be fully loaded, and that any changes made to the loaded model can be propagated back to the persistence). This means that every model-metamodel pair may have a different structure of how model elements are persisted (with look-up information persisted at the beginning of each persisted file).

11.6.4. Porting to OCL-like languages and tools

As previously described, EOL re-uses a large amount of language syntax of OCL. This makes it possible for (principles used in) the Epsilon Static Analysis Framework and its applications to be ported to languages that use OCL syntax. In order to achieve this, additional work is required to implement a *bridge* (or via OCL's *pivot* metamodel) between other OCL-like language and EOL. Alternatively, the ideas of the static analysis and its applications for the Epsilon platform can be re-implemented for other OCL-like languages independently.

Appendices

A. The Abstract Syntax of Epsilon Object Language

This section presents the abstract syntax of EOL. It is worth noting that the EOL metamodel also includes elements which are created for the purpose of static analysis. The syntax of EOL is presented in a top-down manner. Although this section aims at organising the introduction of the EOL abstract syntax, the inter-related nature of EOL's abstract syntax makes it difficult to introduce one concept at a time. Thus, a convention is adopted, *if a concept needs to be explained in detail, but is necessary to be introduced before its detailed discussion, in the first instance of its occurrence it is emphasised in **bold** font and with a forward reference to its detailed discussion.* Abstract Syntax elements marked as *conceptual* do not have their corresponding concrete syntax counterparts in EOL, but are essential for static analysis.

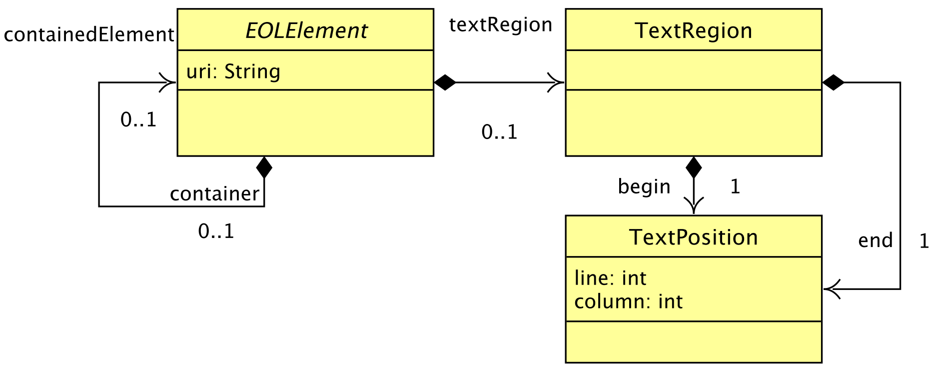


Figure A.1.: The structure of *EOLElement*

A.1. EOLElement

The structure of *EOLElement* is depicted in Figure A.1. Element *EOLElement* is the fundamental element of EOL. *EOLElement* is abstract, and is extended by other EOL elements. Before moving on to the features of *EOLElement*, two basic constructs should be discussed: **TextPosition** and **TextRegion**.

- Element *TextPosition* is created to represent the position of a character in an EOL program. It contains two attributes: *line* and *column* which are *int*, and represent the coordinates of a character in an EOL program.
- Element *TextRegion* is created to represent a region of text in an EOL program. It contains two references: *start* and *end* which are of type *TextPosition*, and represent the start and the end coordinates of a region of text in an EOL program.

EOLElement contains the following features:

- *container*, of type *EOLElement*, is used to establish a link between an *EOLElement* and its containing *EOLElement*.
- *region*, of type *TextRegion*, is used to denote the text region of an *EOLElement* in its containing file.
- *uri*, of type String, is used as the unique identifier of the *EOLElement*, which contains information such as file path etc. so that an *EOLElement* can be located with the *uri* in the file system.

EOLElement is the base type of all constructs in the EOL metamodel. The direct sub-types of *EOLElement* are shown in Figure A.2. **EOLLibraryModule** models the EOL Library Modules [9] in EOL, and is discussed in Section A.1.1. **Import** is used to model the EOL imports which are used to import other EOL programs (discussed in Section A.1.2). **OperationDefinition** is used to model the operation definitions in EOL contained in an *EOLLibraryModule* (discussed in Section A.4.9). **Statement** is used to model the statements in an EOL program, and is discussed in Section A.3. **Block** is used to model a block of *Statement*s, and is discussed in section A.1.4. **Expression**

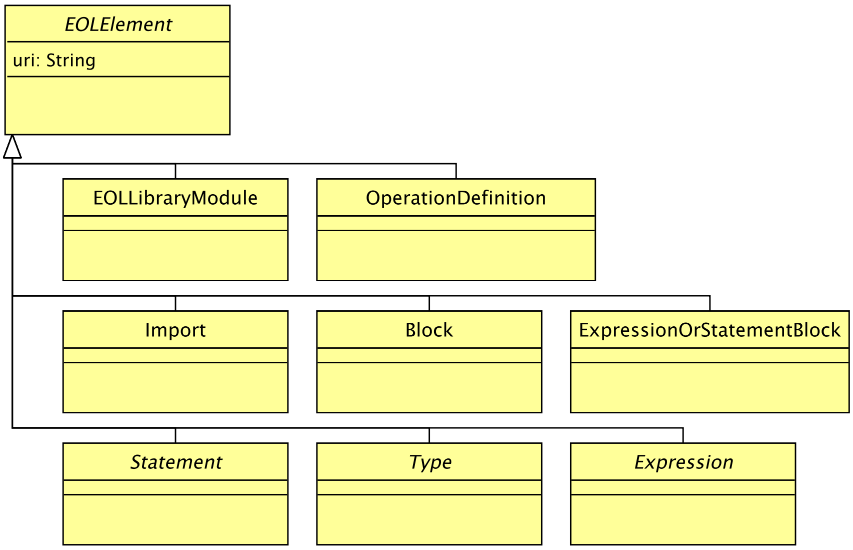


Figure A.2.: Sub-types of *EOLElement*

is used to model expressions that EOL is capable of describing, and is discussed in Section A.2. **Type** is used to model the type system of EOL, and is discussed in Section A.4. **ExpressionOrStatementBlock** is used to model the construct in EOL which may contain a single *Expression* or (exclusively) a single *Block*, and is discussed in Section A.1.5.

A.1.1. EOLLibraryModule

EOLLibraryModule (abstract) is used to denote a *module* of Epsilon. In Epsilon, a *module* is an abstract concept that represents a program. For example, an EOL program is an EOL *module*. *EOLLibraryModule* can be extended to create other modules. This is discussed later in Section A.1.3. In particular, an *EOLLibraryModule* contains:

- A number of *Imports*, which are used to denote imported *EOLLibraryModules*, where the path of the imported module is denoted by the *imported* property of *Import* (Section A.1.2).
- A number of **ModelDeclarationStatements**. A *ModelDeclarationStatement* is used to represent a statement that declares a *model*, and its parameters can be

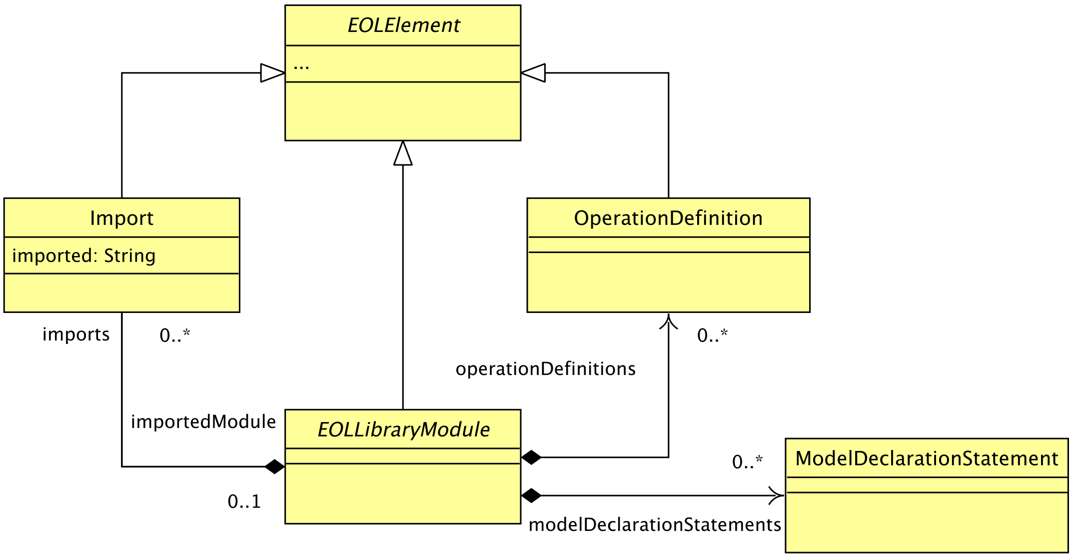


Figure A.3.: The structure of *EOLLibraryModule*

used to specify the location of a *metamodel* and a *model* (the parameters are technology-specific, for example for an EMF model it makes sense to specify an nsURI or the location of the Ecore metamodel; for a database model it makes sense to provide the name of the database and the IP of the server etc.). The details of *ModelDeclarationStatement* are discussed in Section A.3.9;

- A number of *OperationDefinitions*. An *OperationDefinition* is used to denote the concept of an *operation* (or *helper*). The details of *OperationDefinition* are discussed in Section A.4.9.

The structure of *EOLLibraryModule* is shown in Figure A.3. It is worth noting that some details of the classes previously introduced are omitted for visibility purposes.

A.1.2. Import

An *Import* is used to denote the **import** behaviour in EOL. For example, the following statement imports an EOL program named "foo.eol";

```
import "foo.eol";
```

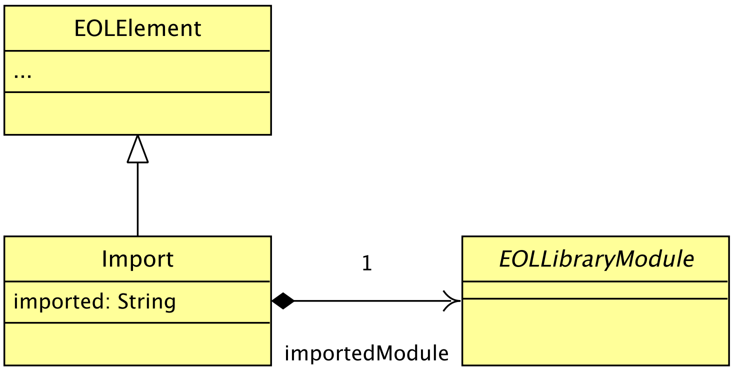


Figure A.4.: The structure of *Import*

The *Import* conceptually contains another *EOLLibraryModule* (the name of the imported module is denoted by the *imported* property), which can be accessed through its *importedModule* reference. The structure of *Import* is displayed in Figure A.4.

A.1.3. EOLModule

EOLProgram is used to denote an EOL program. *EOLProgram* extends *EOLLibraryModule*. In addition, it contains an optional *Block* (discussed in Section A.1.4), which is used to denote a block of *Statements* that are processed when the program is executed (discussed in Section A.3). The structure of *EOLProgram* is shown in Figure A.5.

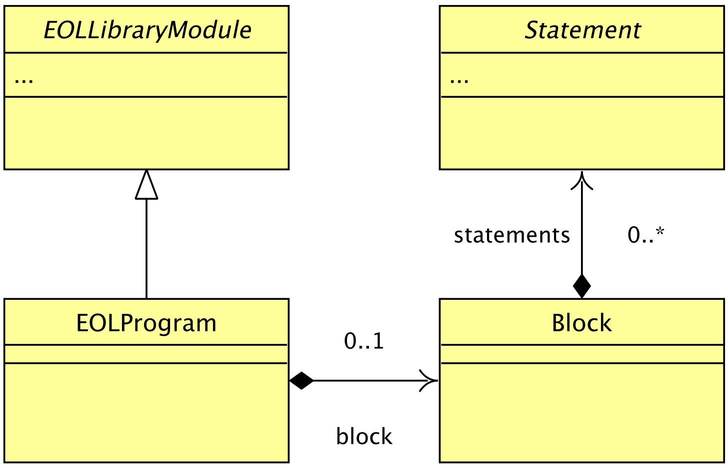


Figure A.5.: The structure of *EOLModule*

A.1.4. Block

Block represents a block of statements in EOL. A *Block* contains a feature named *statements* (of Type *Statement* discussed in Section A.3) which contains a list of *Statements*. A *Block* is normally contained in *EOLLibraryModules* and *OperationDefinitions*. The structure of *Block* is shown in Figure A.6. It is noteworthy that **AnnotationBlock** is a special case for *Block* that contains **AnnotationStatements**, and is discussed in Section A.3.8.

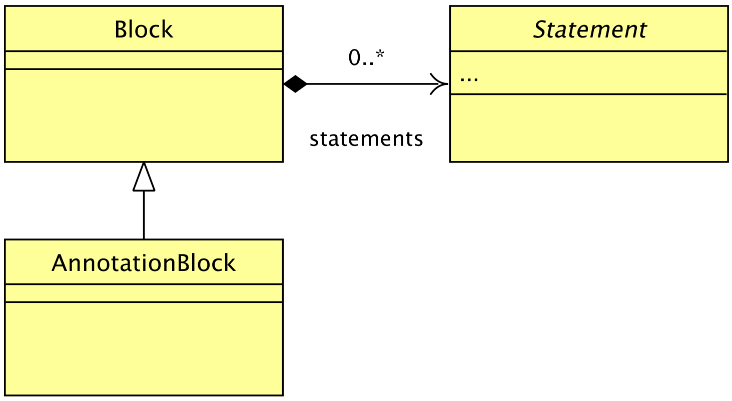


Figure A.6.: The structure of *Block*

A.1.5. ExpressionOrStatementBlock

ExpressionOrStatementBlock is a construct that contains exclusively either an *Expression* (discussed in Section A.2) or a *Block*. *ExpressionOrStatementBlock* is typically used in control flow statements such as if statements:

```
1 var student = Student.all.first;
2 if(student.tutor.isUndefined())
3   (student.first_name + "does not have a tutor").println();
```

The construct in line 3 is an instance of *ExpressionOrStatementBlock*, because control flow statements can omit the curly brackets and give an expression immediately after the if() condition. In line 3, the *ExpressionOrStatementBlock* contains simply an expression.

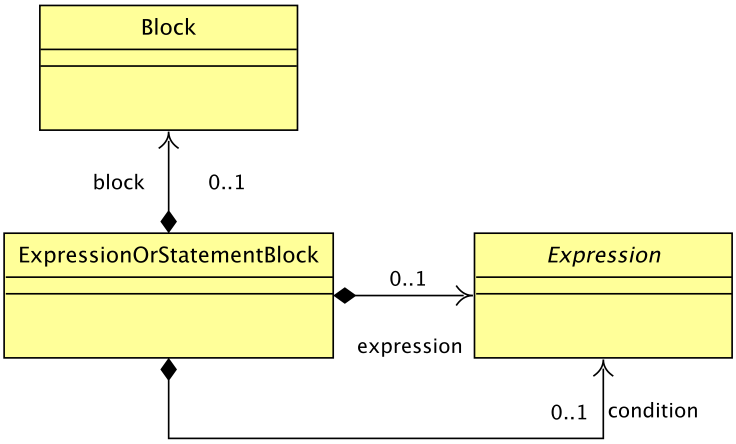


Figure A.7.: The structure of *ExpressionOrStatementBlock*

The structure of *ExpressionOrStatementBlock* is shown in Figure A.7. It contains an *expression* (of type *Expression*) or a *block* (of type *Block*). It also contains a *condition* (of type *Expression*) which is used to represent the condition of the control flow branches.

A.2. Expression

Expression (abstract) is a base class of different types of EOL expressions. *Expression* has a feature named *resolvedType*, which is a **Type**. The *Type* can be one of the types in the EOL type system, which are discussed in Section A.4. The *Expression* and its sub-types are shown in Figure A.8. There is a number of types that extend *Expression*. The sub-types of *Expression* are discussed in detail in this section.

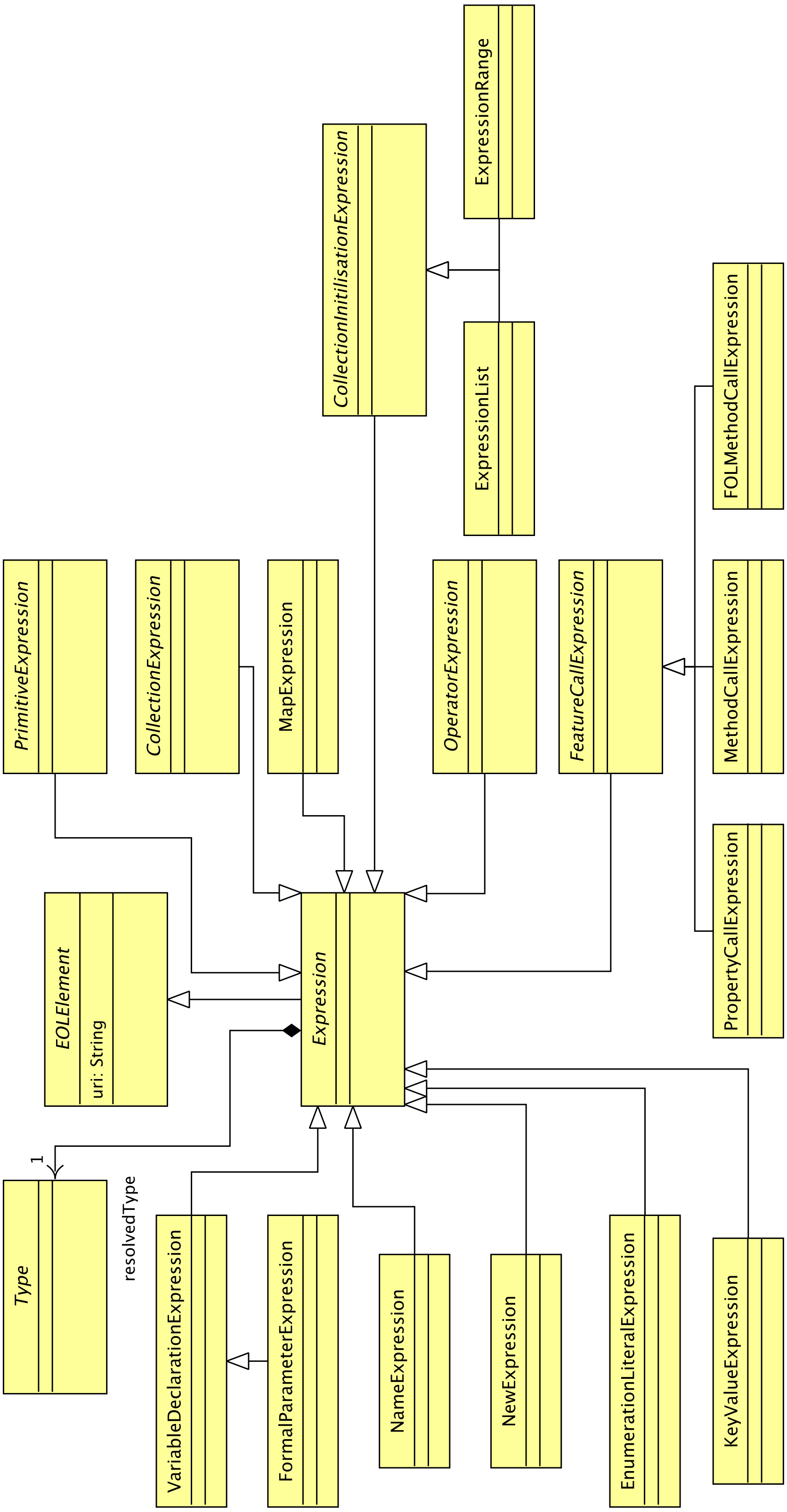


Figure A.8.: The structure of the *Expression* element and its sub-types

A.2.1. PrimitiveExpression

PrimitiveExpression is created to model the primitive literals in EOL. In EOL, there are four primitive types: *Boolean*, *Integer*, *Real* and *String*. The modelling of the *PrimitiveExpression* introduced several conceptual and abstract classes. The *ComparableExpression* (abstract and conceptual) type denotes the primitive types on which compare operators (>, >=, <, <=, = and <>) can be used. The *SummableExpression* (abstract, conceptual) type denotes the primitive types to which the summation operator (+) can be used. **StringExpression**, **RealExpression**, **IntegerExpression** and **BooleanExpression** represent the literals of *String*, *Real*, *Integer* and *Boolean*. *StringExpression*, *RealExpression*, *IntegerExpression* and *BooleanExpression* all have a feature named *value* which holds the value of the literals. The structure of *PrimitiveExpression* and its sub-types is shown in Figure A.9. Thus, for the following EOL program:

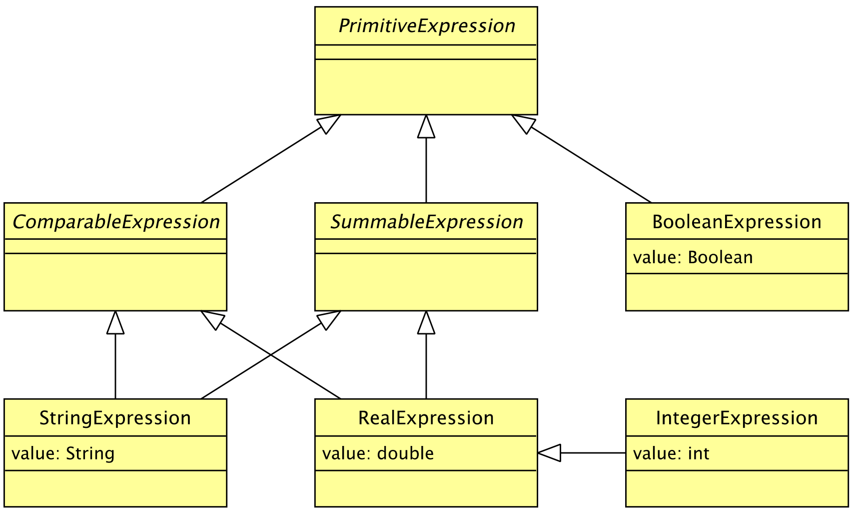


Figure A.9.: The structure of *PrimitiveExpression* and its sub-types

```
var a = "String";
```

The right hand side of = is a *StringExpression*. *PrimitiveExpression* inherits the *resolvedType* feature from *Expression*. This implies that a *PrimitiveExpression* must have a *Type*. In this case, the *StringExpression* (with value "String") has a **StringType** which is discussed in Section A.4.2.

A.2.2. CollectionExpression

CollectionExpression is created to model collection literals in EOL. In EOL, four types of collections are provided. The *Bag* collection represents non-unique, unordered collections; the *Sequence* collection represents non-unique, ordered collections; the *Set* collection represents unique and unordered collections; and the *OrderedSet* represents unique and ordered collections. The structure of *CollectionExpression* is shown in Figure A.10. The abstract types *UniqueCollection* and *OrderedCollection* are types created to categorise collections. Thus, *SetExpression* and *OrderedSetExpression* are kind-of *UniqueCollection*, and *OrderedSetExpression* and *SequenceExpression* are kind-of *OrderedCollection*. *BagExpression* on the other hand is not ordered and not unique.

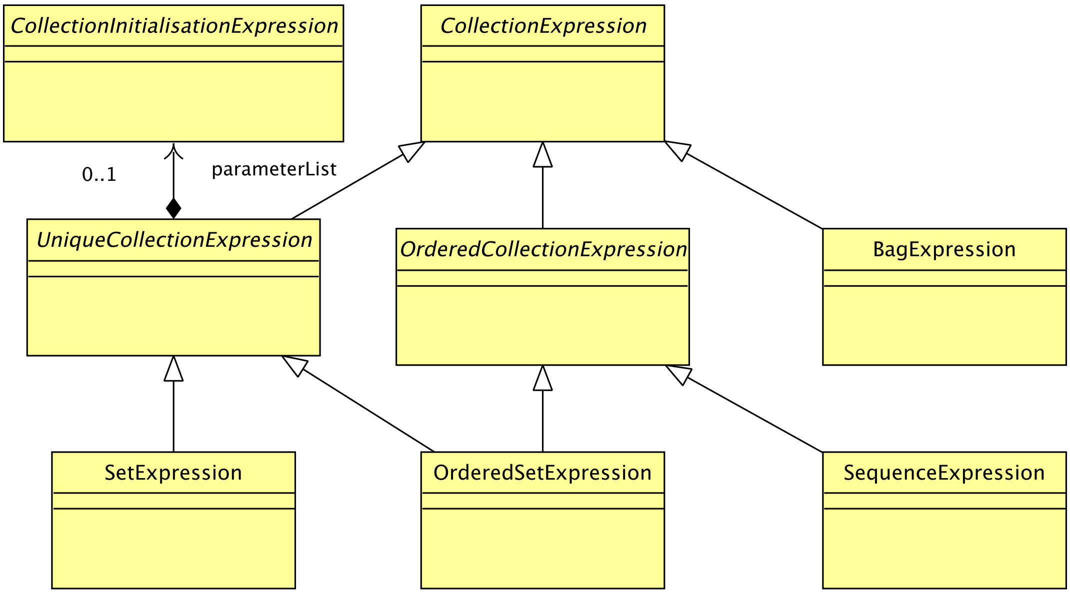


Figure A.10.: The structure of *CollectionExpression* and its sub-types

CollectionExpression inherits the *resolvedType* field from *Expression*. It implies that a *CollectionExpression* must have a type. Consider an example:

```
var a = Sequence(String);
```

The right hand side of the = is a *SequenceExpression*, which comes with a type declaration, which declares that the content type of the sequence should be **StringType**. If no type annotation is provided, the expression is assumed to be of type **AnyType**, and is discussed in Section A.4.1.

CollectionExpression can also be initialised by a **CollectionInitialisationExpression**.

A.2.3. CollectionInitialisationExpression

A *CollectionExpression* may be initialised by a **CollectionInitialisationExpression**. The structure of *CollectionInitialisationExpression* is shown in Figure A.11.

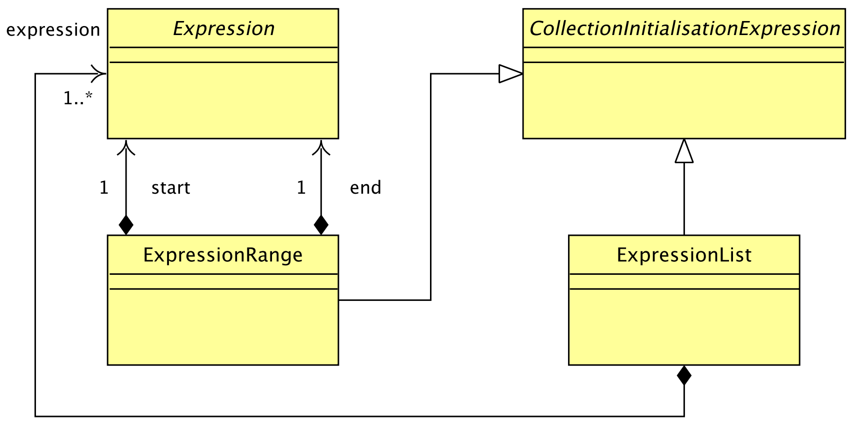


Figure A.11.: The structure of *CollectionInitialisationExpression* and its sub-types

In general, there is a number of ways that a *CollectionExpression* can be initialised. Consider the example below:

```
1 var a = new Sequence;
2 var b = new Sequence(Integer);
3 var c = Sequence{1,2,3,4,5};
4 var d = Sequence{5..10};
```

In line 1, a *Sequence* is created with the *new* keyword, and has an automatic *AnyType* as its content type. In line 2, a *Sequence* is initialised with a type declaration, and its content type is **IntegerType**. In line 3, an explicit expression list is provided to initialise a *Sequence*. Thus, variable *c* is a *Sequence* which contains values {1,2,3,4,5}. Finally, in line 4, an expression range with the .. notation is provided to initialise a *Sequence*.

Thus, variable *d* is a *Sequence* which contains the values {5,6,7,8,9,10}.

Thus, two sub-types of *CollectionInitialisationExpression* are created. **ExpressionList** contains a number of *Expressions* which are used to initialise a *CollectionExpression*. **ExpressionRange** contains an *Expression* named *start* to denote the start of the range, and an *Expression* named *end* to denote the end of the range. It is noteworthy that *ExpressionRange* is only applicable when the *Expressions* involved evaluate to *IntegerExpressions* and only to *CollectionExpressions* with *IntegerType* as their content types.

A.2.4. KeyValueExpression

KeyValueExpression is created to model the key-value pair expressions in EOL. *KeyValueExpression* contains a feature named *key* and a feature named *value* which are both of type *Expression*. *KeyValueExpression* is typically used in **MapExpression**, which is discussed in Section A.2.5. The structure of *KeyValueExpression* is shown in Figure A.12.

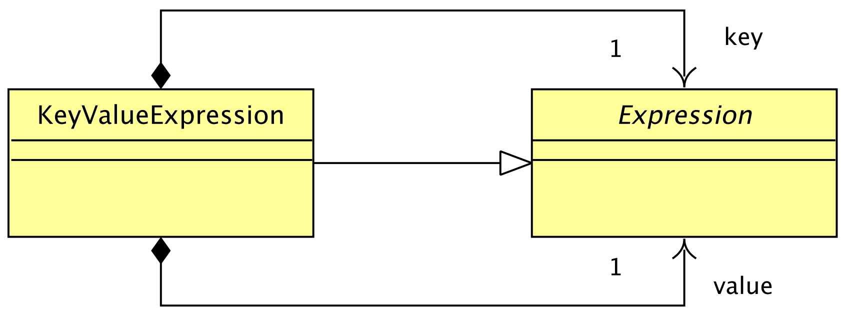


Figure A.12.: The structure of *KeyValueExpression*

A.2.5. MapExpression

EOL provides a way to create a *Map*. The *Map* represents a collection of key-value pairs in which the keys are unique. A *Map* may be initialised in the following forms:

```
var map = new Map;
var map = Map{"John" = "01904-123-456", "Kate" = "01904-987-654"};
```

The *MapExpression* is created to model the *Map* expression in EOL. A *MapExpression* contains an optional collection of *KeyValueExpressions* at initialisation. The structure of *MapExpression* is shown in Figure A.13.

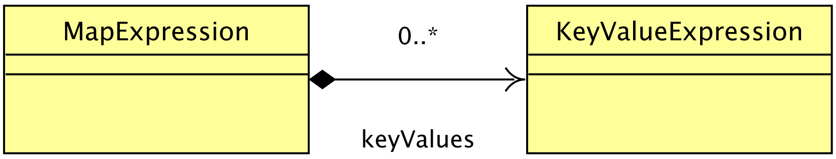


Figure A.13.: The structure of *MapExpression*

A.2.6. NameExpression

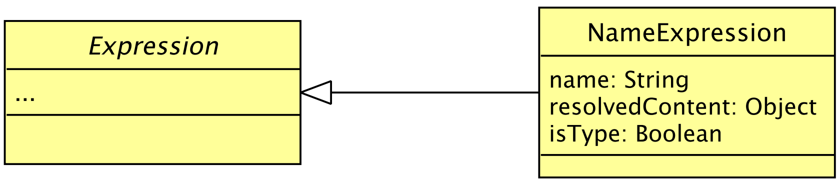


Figure A.14.: The structure of *NameExpression*

In EOL, there are *names* of various kinds. Consider the example:

```
1 var a = 1;
2 a.println();
3 a.isType(Integer);
```

In lines 1 and 2, *a* is an example of *name* in EOL. In line 2, the name of the method call *println* is also a *name*. Parameters are expressions in general but can also be *names* of other expressions. In line 3, *Integer* is an example of a *name*. Type *NameExpression* is created to model *names* in EOL. A *NameExpression* contains a string named *name* which holds the value of the name. In some cases, a *name* can refer to another object; for example, in line 2, *a* refers to the variable declared in line 1. Thus, *NameExpression* contains a feature named *resolvedContent* of type Object. *NameExpression* also contains an attribute *isType*, which is used to denote if the *NameExpression* is a type in EOL.

For example, in line 3, the parameter *Integer* is a name but at the same time it is an EOL type. The structure of *NameExpression* is shown in Figure A.14.

A.2.7. VariableDeclarationExpression

VariableDeclarationExpression is created to model variable declarations in EOL. Consider the example:

```
1 var a = 1;
2 var b: String;
3 var student : new Student;
```

In line 1, a variable named *a* is declared on the left hand side of =. *a* does not have any type declaration, so its type is assumed to be *Any* in EOL. In line 2, a variable named *b* is declared with its type (String). In line 3, a variable named *student* is declared, together with a keyword *new*. This means that a new instance of *Student* is created and assigned to *student*. Thus, *VariableDeclarationExpression* contains a *name* of type *NameExpression* and an attribute *create* to denote if the *new* keyword is used. The *VariableDeclarationExpression* can also have a number of *references* by *NameExpressions*. The structure of *VariableDeclarationExpression* is shown in Figure A.15.

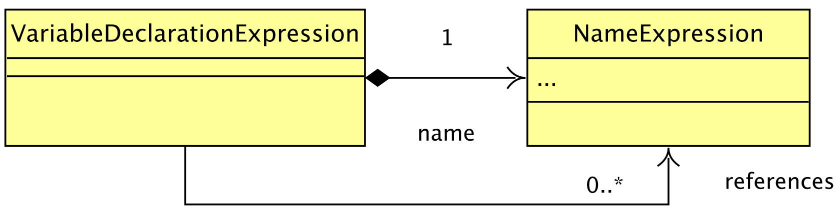


Figure A.15.: The structure of *VariableDeclarationExpression*

A.2.8. FormalParameterExpression

FormalParameterExpression is a sub-type of *VariableDeclarationExpression*, which is used to denote the parameter declarations when an operation is declared. For example:

```
var a = 1; //a is 1
```

```
2 a = a.add(1); //a is now 2
3 operation Integer add(i: Integer): Integer {
4   return self + i;
5 }
```

In line 3, the declared operation takes a single parameter *i* of type *Integer*. This parameter declaration is represented by *FormalParameterExpression*. *FormalParameterExpression* does not contain any additional features compared to *VariableDeclarationExpression*.

A.2.9. NewExpression

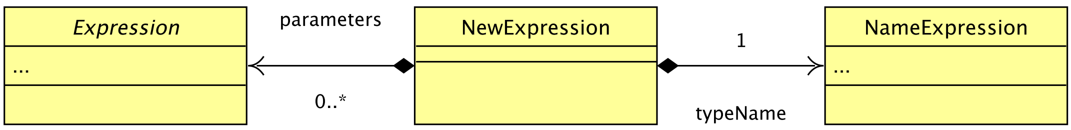


Figure A.16.: The structure of *NewExpression*

In EOL, the *new* keyword is used to create instances of non-primitive types. For example, to create a *Sequence*:

```
var a = new Sequence(Integer);
```

To create a model element type named *Student*:

```
var a = new Student;
```

EOL also supports the creation of native Java objects:

```
var frame = new Native("javax.swing.JFrame");
frame.title = "Opened with EOL";
frame.setBounds(100,100,300,200);
frame.visible = true;
```

The *NewExpression* is created to represent such expressions, and its structure is shown in Figure A.16. *NewExpression* has a *name* (of type *NameExpression*) to denote the name of the type to be instantiated. The parameters of the *new* keyword are represented by the feature named *parameter* (of type *Expression*).

A.2.10. EnumerationLiteralExpression

EOL provides the # operator for accessing enumeration literals. For example:

```
var a = A!B#C;
```

accesses the Literal C in enumeration B in metamodel A. The structure of *EnumerationLiteralExpression* is shown in Figure A.17. *EnumerationLiteralExpression* has three references named *metamodel*, *enumeration* and *literal* (all of type *NameExpression*) to represent the names of the metamodel, the enumeration and the literal.

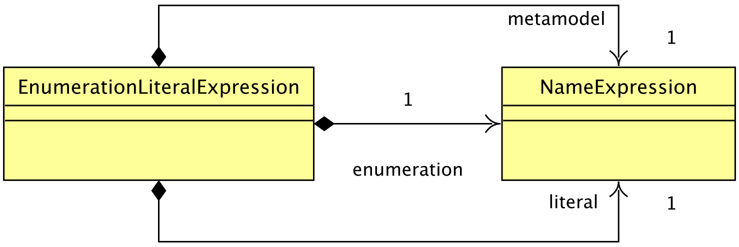


Figure A.17.: The structure of *EnumerationLiteralExpression*

A.2.11. FeatureCallExpression

EOL provides expressions to navigate properties and invoke operations on objects [9]. Such expressions can be collectively summarised as feature call expressions. Consider the example:

```
1 var student = Student.all.println();
2 var tutor = student.tutor;
3 name.println();
4 var firstClassStudents = Student.all.select(s|s.tutor = tutor);
```

In line 2, a property call expression *student.tutor* appears, where *tutor* is the name of the property to be named on *student*. In line 3, a method call expression appears, where *println()* is the name of the method in the EOL standard library to be named. A first-order logical method call expression appears in line 4. The name of the method is *select* and it specifies the condition of *select* using a lambda expression. Property call expression, method call expression and first-order-logic-method call expression can be categorised as feature call expressions [9]. EOL provides two operators to initiate feature call expressions: the . operator and the → operator. The semantics difference is that when the . operator is used, precedence is given to the user-defined operations rather than the standard library operations in case of a name collision. For example:

```
1 "Something".println();
2 operation Any println(): Any {
3   ("Printing: " + self)->println();
4 }
```

In line 2, an operation named *println()* is defined, which collides with the *println()* operation defined in the EOL standard library. To invoke the operation in line 3, the . operator is used in line 1. However, it is noteworthy that in line 3, the → operator is used to call the *println()* operation in the EOL standard library; otherwise, the operation would trigger an infinite recursion.

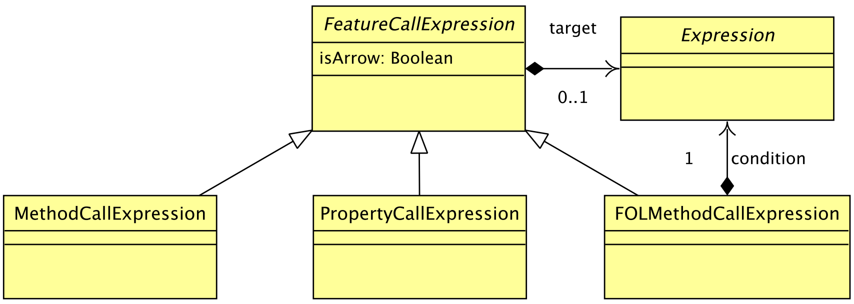


Figure A.18.: The structure of *FeatureCallExpression* and its sub-types

The *FeatureCallExpression* (abstract) is created to represent the concept of feature call expressions of EOL. Its structure is shown in Figure A.18. The *FeatureCallExpression* has an optional *target* (of type *Expression*), which is used to denote which expression initiates the *FeatureCallExpression*. *FeatureCallExpression* also contains an attribute named *isArrow* (of type Boolean), which is used to denote if a *FeatureCallExpression* uses the → operator.

MethodCallExpression

MethodCallExpression is created to represent method calls in EOL. Its structure is shown in Figure A.19. *MethodCallExpression* extends *FeatureCallExpression*, which contains a number of *arguments* (of type *Expression*) and a *method* (of Type *NameExpression*) which is used to refer to the name of the operation definition. A *MethodCallExpression* also has a derived feature named *resolvedOperationDefinition* which is calculated at runtime, so that it points to the operation definition that it calls.

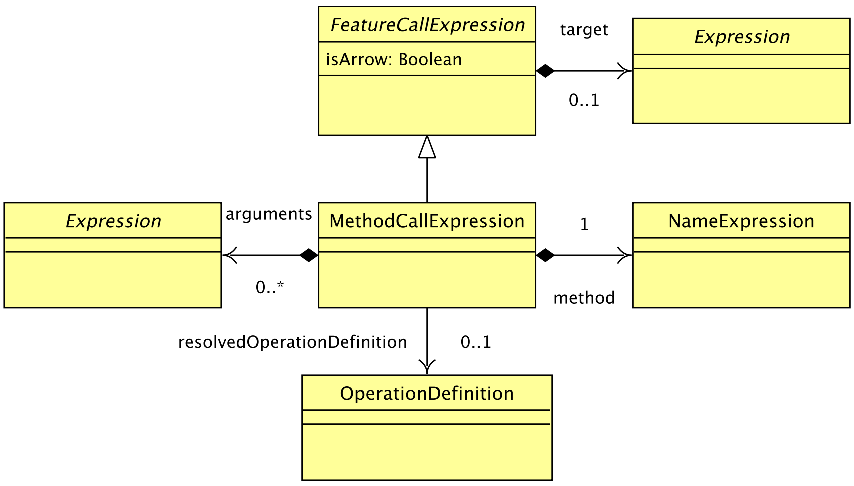


Figure A.19.: The structure of *MethodCallExpression*

FOLMethodCallExpression

FOLMethodCallExpression is created to represent first-order-logic-method calls in EOL. Its structure is shown in Figure A.20. *FOLMethodCallExpression* extends *FeatureCall-Expression*, which contains an *iterator* (of type *FormalParameterExpression*) to denote the iterator of the lambda expression, a number of *conditions* (of type *Expression*) to denote the condition of the lambda expression, a *method* (of type *NameExpression*) to denote the name of the first-order-logic operation, and a *resolvedOperationDefinition* (of

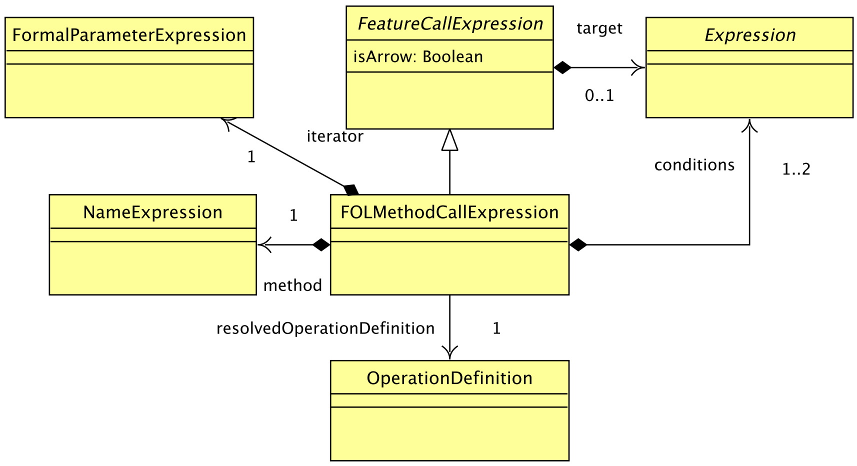


Figure A.20.: The structure of *FOLMethodCallExpression*

type *OperationDefinition*) which is calculated at runtime to refer to the first-order-logic operation (in the standard library) that it calls.

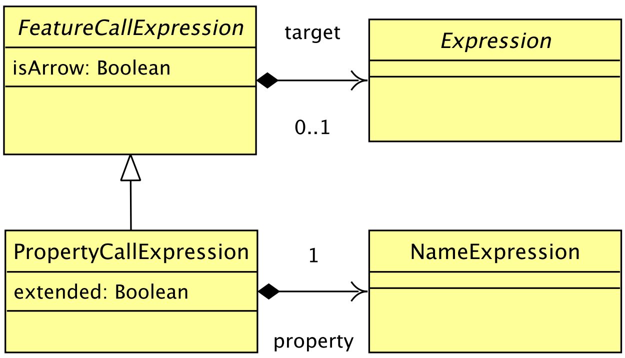


Figure A.21.: The structure of *PropertyCallExpression*

PropertyCallExpression

PropertyCallExpression is created to represent property call expressions in EOL. Its structure is shown in Figure A.21. EOL supports the notion of *extended property*, which temporarily assigns a property to an object which can be retrieved throughout the execution of the EOL program. Consider the example program shown in Listing A.1 which calculates the depth of each Tree element (which does not have a parent node) in a model that conforms to the Tree metamodel from [9] in Figure A.22. In line 10, an extended property, represented by the ∼ operator, named *depth*, is assigned to instances of *Tree*. In line 6, the *depth* property is retrieved. The extended property provides the user of EOL the facility to relate information to individual objects which is not supported by its corresponding meta-type.

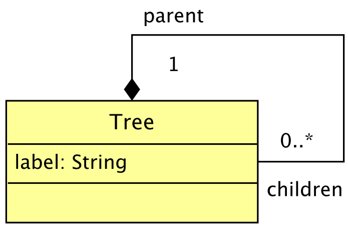


Figure A.22.: The Tree Metamodel from [9]

Therefore, *PropertyCallExpression* has an attribute named *extended* (of type Boolean) to denote if this property call is an extended or a regular property call. *PropertyCallExpression* also has a feature named *property* (of type *NameExpression*) which is used to denote the name of the property.

```
for (n in Tree.allInstances.select(t|not t.parent.isDefined())) {
  n.setDepth(0);
}
for (n in Tree.allInstances) {
  (n.name + " " + n.~depth).println();
}
operation Tree setDepth(depth : Integer) {
  self.~depth = depth;
  for (c in self.children) {
    c.setDepth(depth + 1);
  }
}
```

Listing A.1: An example EOL program using extended properties

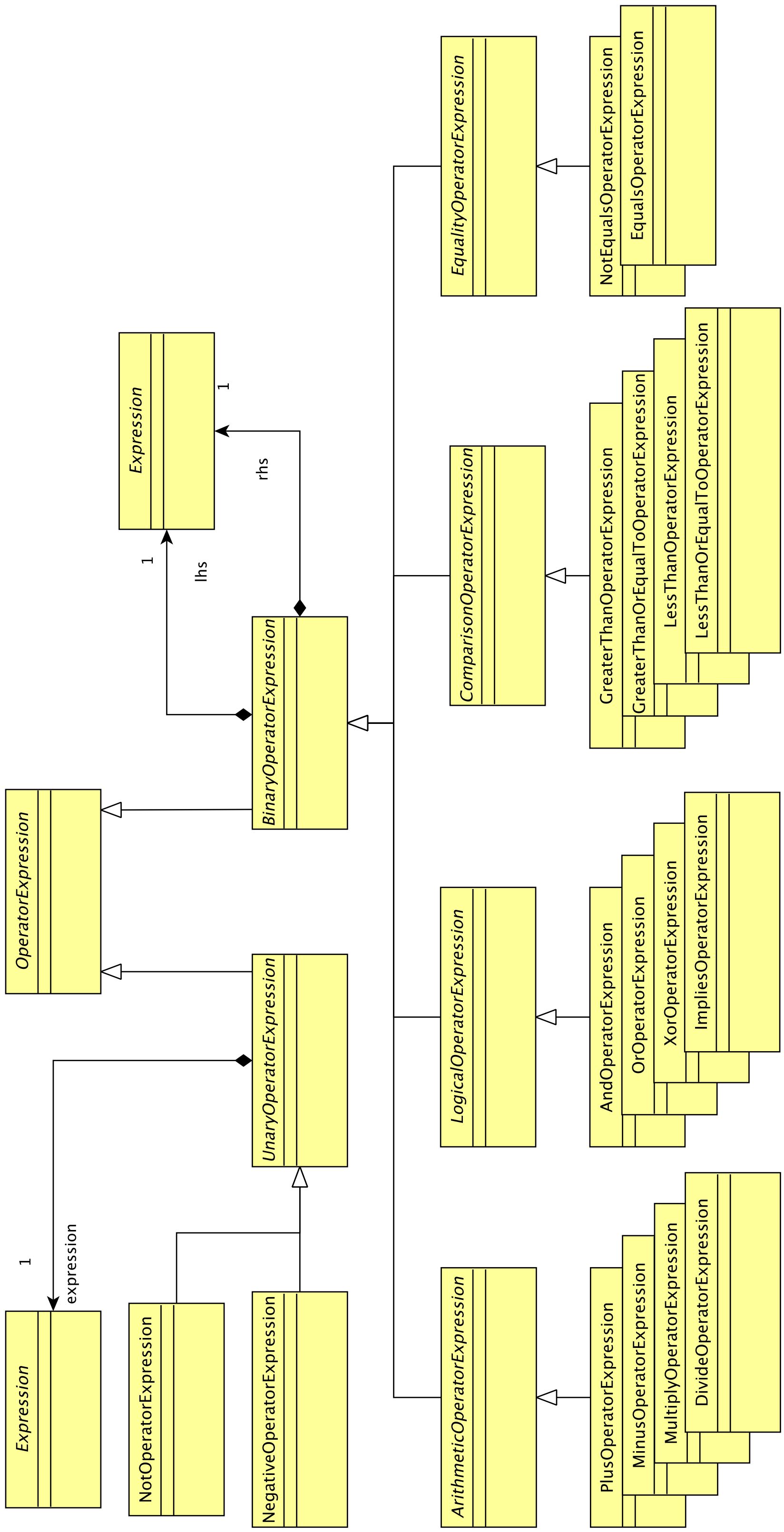


Figure A.23.: The structure of *OperatorExpression* and its sub-types

A.2.12. OperatorExpression

OperatorExpression (abstract) is created to denote operator expressions in EOL. *OperatorExpression* extends *Expression*; thus, it inherits the *resolvedType* property (of type *Type*). The structure of *OperatorExpression* and its sub-types is shown in Figure A.23. *OperatorExpression*s can be categorised into **UnaryOperatorExpression**s (abstract, conceptual) and **BinaryOperatorExpression**s (abstract, conceptual). *UnaryOperatorExpression*s contain a feature named *expression* (of type Expression) to denote the expression used in the operator. There are two kinds of *UnaryOperatorExpression*s in EOL: the **NotOperatorExpression** and the **NegativeOperatorExpression**. Consider the example:

```
var not = not false;
var negative = - (1);
```

The right hand side expression of the = in line 1 is an example of *NotOperatorExpression*, whereas the right hand side expression of the = in line 2 is an example of *NegativeOperatorExpression*.

*BinaryOperatorExpression*s contain a *lhs* (of type *Expression*) and a *rhs* (of type *Expression*) to denote the first and second operand of the binary operator expressions. *BinaryOperatorExpression*s can be further categorised as:

- **ArithmeticOperatorExpression**s (abstract, conceptual), which represent operators for arithmetic computations such as $+$, $-$, $*$ and $/$, represented by **PlusOperatorExpression**; **MinusOperatorExpression**, **MultiplyOperatorExpression** and **DivideOperatorExpression**.

- **LogicalOperatorExpression**s (abstract, conceptual) which include logic operators such as *and*, *xor*, *or* and *implies*, represented by **AndOperatorExpression**, **XorOperatorExpression**, **OrOperatorExpression** and **ImpliesOperatorExpression**.

- **ComparisonOperatorExpression**s (abstract, conceptual) which denote comparison operators such as $>$, $>=$, $<$ and $<=$. These operators are represented by

GreaterThanOperatorExpression, **GreaterThanOrEqualToOperatorExpression**, **LessThanOperatorExpression** and **LessThanOrEqualToOperatorExpression**.

- **EqualityOperatorExpressions** (abstract, conceptual) which are equality operators such as = and <>, represented by **EqualsOperatorExpression** and **NotEqualsOperatorExpression**.

A.3. Statement

Statement (abstract) and its sub-types are created to represent the different types of statements provided by EOL. The structure of *Statement* and its sub-types is shown in Figure A.25.

A.3.1. ExpressionStatement

ExpressionStatement is created to represent the simplest form of statement in EOL. Consider the example:

```
"Hello World".println();
```

where an instance of *ExpressionStatement* appears. *ExpressionStatement* contains an *expression* (of type *Expression*). The structure of *ExpressionStatement* is shown in Figure A.24.

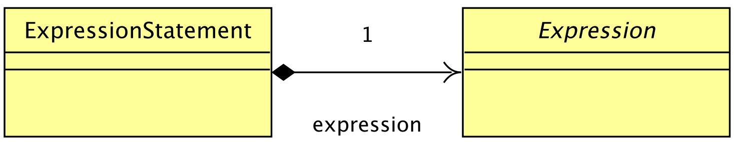


Figure A.24.: The structure of *ExpressionStatement*

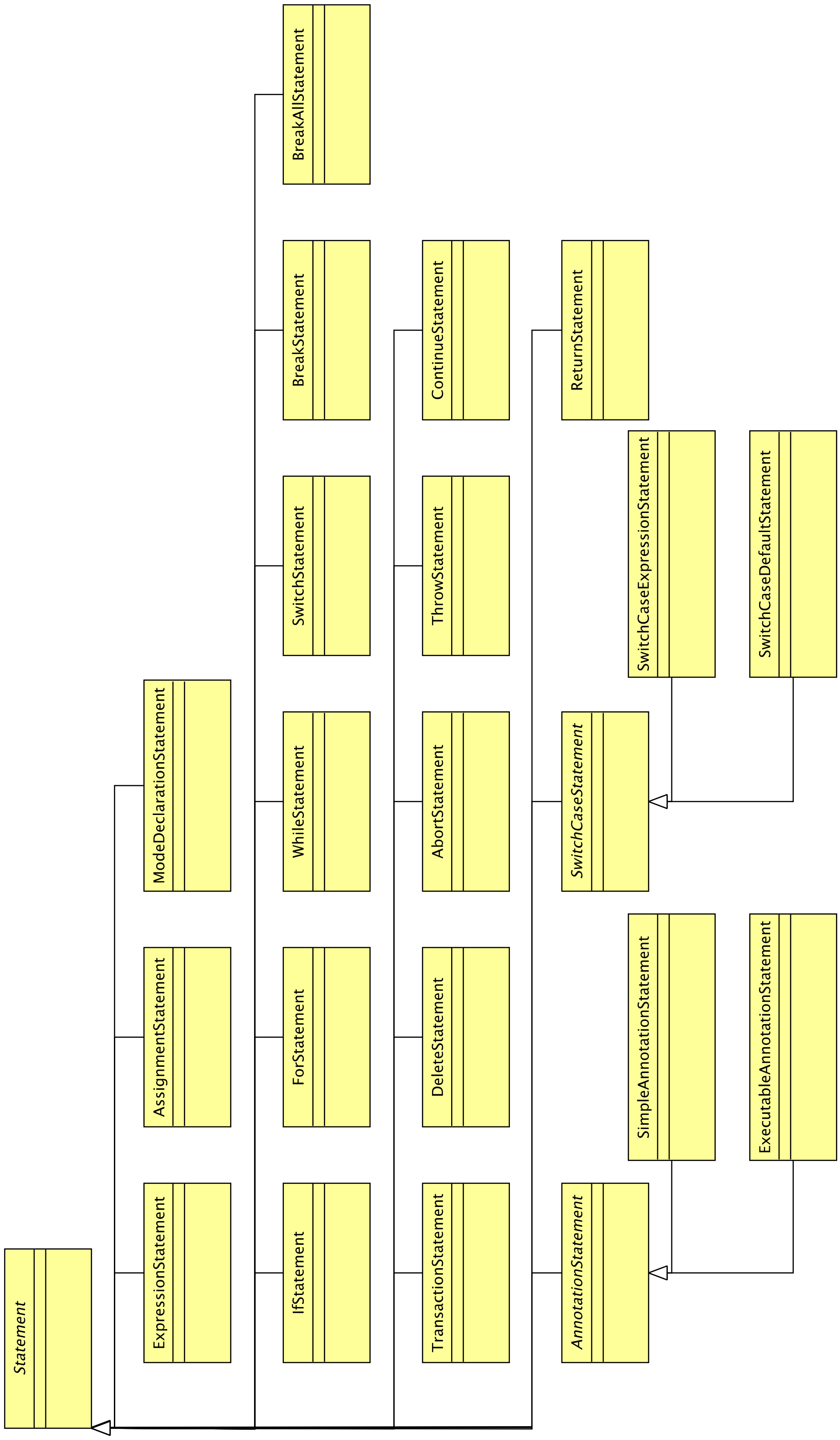


Figure A.25.: The structure of *Statement* and its sub-types

A.3.2. AssignmentStatement

AssignmentStatement is created to represent the assignments in EOL. Consider the example:

```
var a = 1;
```

The left hand side of the = is an instance of *VariableDeclarationExpression*, whilst the right hand side of the = is an instance of *IntegerExpression*. The structure of *AssignmentStatement* is shown in Figure A.26. *AssignmentStatement* contains a *lhs* and a *rhs* (of type *Expression*) to denote the left hand side and the right hand side expressions of the assignment.

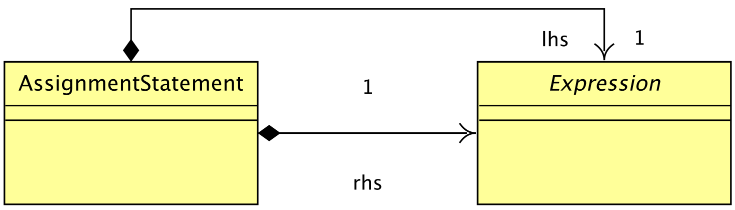


Figure A.26.: The structure of *AssignmentStatement*

A.3.3. ForStatement

ForStatement is created to represent for loops in EOL. Consider the example:

```
for(i in Sequence{1..5})
  i.println();
```

In EOL, for loops contain an iterator and the domain of that iterator (i.e. the values it iterates over). In this instance, the iterator is *i* and its domain is *Sequence*{1..5}. The structure of *ForStatement* is shown in Figure A.27. *ForStatement* contains an iterator (of type *FormalParameterExpression*), a body (of type *ExpressionOrStatementBlock*) and a domain for the iterator (of type *Expression*).

A.3.4. WhileStatement

WhileStatement is created to represent the while loops in EOL. Consider the example:

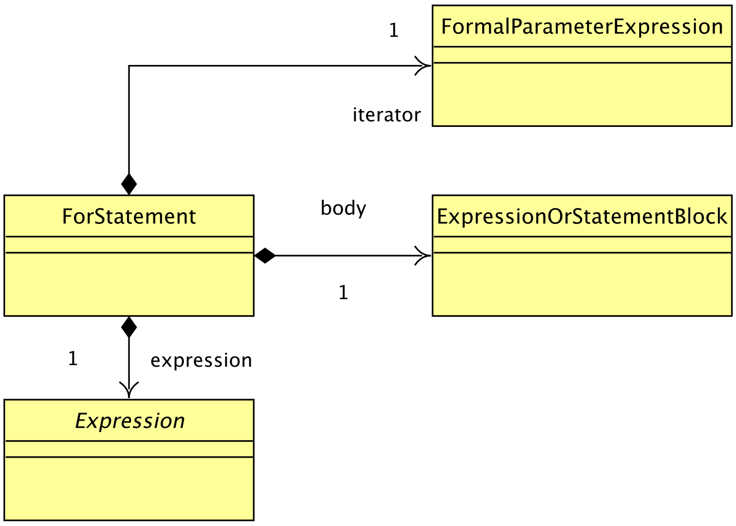


Figure A.27.: The structure of *ForStatement*

```
1 var a = 5;
2 while(a >= 0)
3 {
4   a.println();
5   a = a - 1;
6 }
```

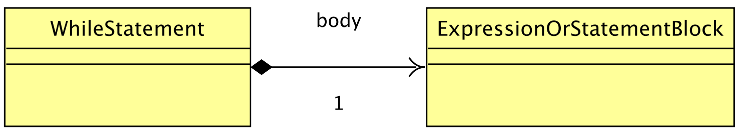


Figure A.28.: The structure of *WhileStatement*

In line 2, a while loop is in place, which contains a condition that evaluates to a boolean value, and a body which may be a block or an expression. The structure of *WhileStatement* is shown in Figure A.28. *WhileStatement* contains a body (of type *ExpressionOrStatementBlock*). It is noteworthy that the condition of the while loop, which evaluates to a boolean value, is represented by the *condition* in the *ExpressionOrStatementBlock*.

A.3.5. IfStatement

IfStatement is created to represent if statements in EOL. Consider the example:

```
var a = Sequence{1..30}.random();
if(a < 10)
{
  "a is less than 10".println();
}
else if(a >= 10 and a < 20)
{
  "a >=10 and a < 20".println();
}
else
{
  "a is greater than or equal to 20".println();
}
```

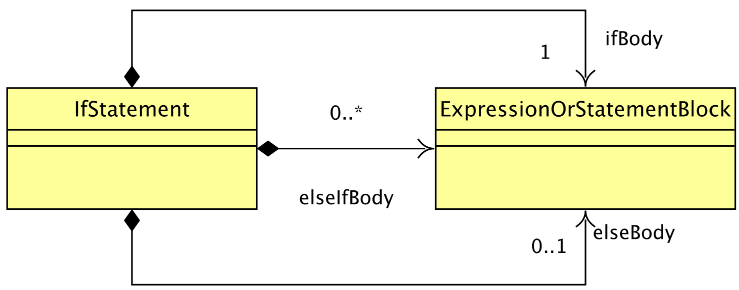


Figure A.29.: The structure of *IfStatement*

An if statement contains multiple branches (if, else-if and else). The structure of *IfStatement* is shown in Figure A.29. *IfStatement* has an *ifBody*, a number of optional *elseIfBody*(-ies) and an optional *elseBody* (all of which are of type *ExpressionOrStatementBlock*). The condition of the branches in the *IfStatement* is represented by the *condition* property in *ExpressionOrStatementBlock*.

A.3.6. SwitchStatement

SwitchStatement is created to represent switch statements in EOL. In particular, a switch statement in EOL is in the form of the following:

```
var i = Sequence{1..4}.random();
switch(i) {
  case 1: i.println();
  case 2: i.println();
  case 3: i.println();
  default : "default".println();
}
```

The structure of *SwitchStatement* is shown in Figure A.30. *SwitchStatement* contains an *expression* (of type *Expression*) to represent the expression to switch. *SwitchStatement* contains a number of *cases* (of type **SwitchCaseExpressionStatement**) and a *default* (of type **SwitchCaseDefaultStatement**). *SwitchCaseExpressionStatement* and *SwitchCaseDefaultStatement* are sub-types of *SwitchCaseStatement* (abstract), which contains a feature named *body* (of type *ExpressionOrStatementBlock*).

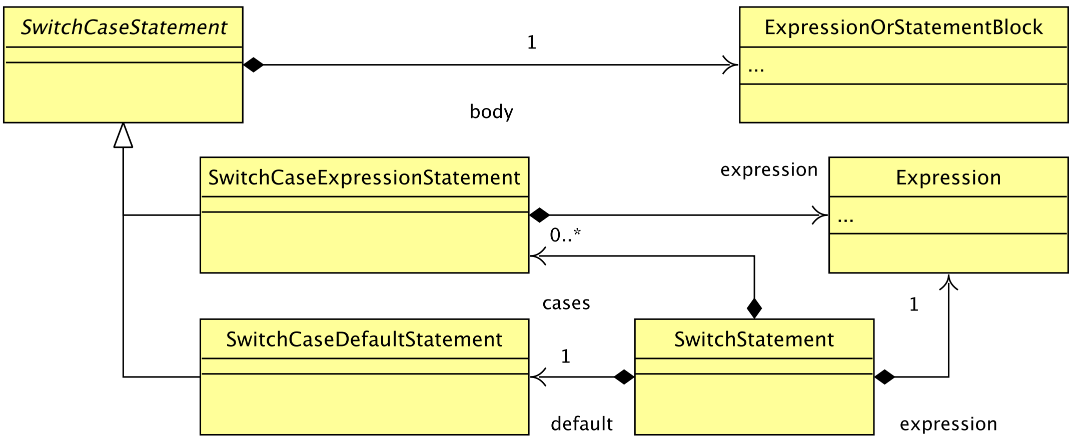


Figure A.30.: The structure of *SwitchStatement*

A.3.7. ContinueStatement, BreakStatement and BreakAllStatement

ContinueStatement, *BreakStatement* and *BreakAllStatement* are control flow statements that are typically used in loops in EOL. Consider the example:

```
1 for(i in Sequence{1..3}) {
2   if(i == 1)
3     continue;
4   for(j in Sequence{1..4}) {
5     if(j = 2) {break;}
6     if(j = 3) {breakAll;}
7     (i + "," + j).println();
8   }
9 }
```

Lines 3, 5 and 6 illustrate examples of continue, break and breakAll statements. The structures of *ContinueStatement*, *BreakStatement* and *BreakAllStatement* are shown in Figure A.25.

A.3.8. AnnotationStatement

In EOL, an *operation* may be preceded by an annotation block [9]. In EOL, annotation blocks have two purposes: simple annotations are used to declare a name and several strings [9], and executable annotations are used to describe pre/post conditions of an operation. The concrete syntax of simple annotations is:

```
@name value(,value)*
```

An example of simple annotations is:

```
@colours red, blue, green
```

A number of pre and post executable annotations can be attached to EOL operations to specify the pre- and post-conditions of the operation. When an operation is invoked, before its body is evaluated, the expressions of the pre- annotations are evaluated. If all of them return true, the body of the operation is processed; otherwise, an error is raised. Similarly, once the body of the operation has been executed, the expressions of the post- annotations of the operation are executed to ensure that the operation has had the desired effects. Pre- and post- annotations can access all the variables in the parent scope, as well as the parameters of the operation and the object on which the operation is invoked (through the *self* variable). Moreover, in post annotations, the returned value of the operation is accessible through the built-in *_result* variable.

```
1 1.add(2);
2 1.add(-1);
3
4 $pre i>0
5 $post _result>self
6 operation Integer add(i: Integer) : Integer {
7   return self + i;
8 }
```

Listing A.2: pre- and post- conditions of an operation definition

For example, the program in Listing A.2 demonstrates an example of pre- and post-conditions. In line 4, the pre- condition specifies that i should be greater than 0. The post- condition specifies that the *_result* should be greater than *self*. Thus, the statement in line 1 will pass and the statement in line 2 will throw an error.

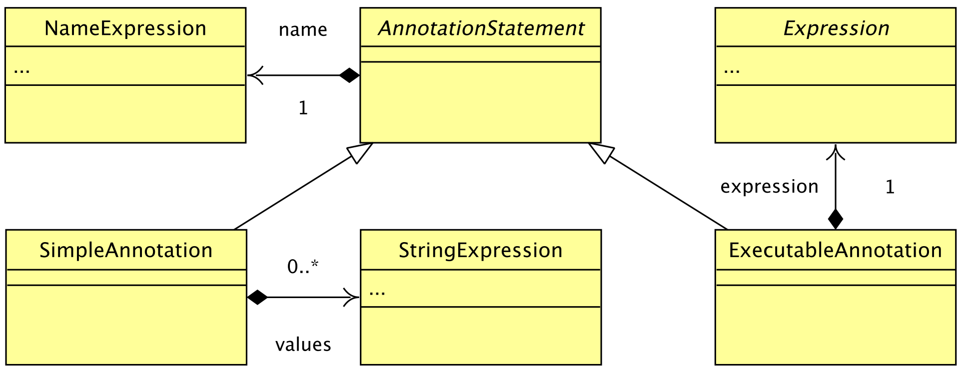


Figure A.31.: The structure of *AnnotationStatement*

AnnotationStatement (abstract) and its sub-types are created to represent annotations in EOL. The structure of *AnnotationStatement* and its sub-types are shown in Figure A.31. *AnnotationStatement* contains a *name* (of type *NameExpression*). **ExecutableAnnotationStatement** inherits *AnnotationStatement* and contains an *expression* (of type *Expression*), where **SimpleAnnotationStatement** inherits *AnnotationStatement* and contains *values* (of type *StringExpression*).

A.3.9. ModelDeclarationStatement

ModelDeclarationStatement is not currently supported by EOL at runtime, but is essential for the purpose of static analysis. At the moment, the EOL run configuration is responsible for specifying the locations of *model*s and *metamodel*s managed by an EOL program via Eclipse Epsilon's GUI. However, in order to achieve static analysis, there needs to be a way to specify such information in EOL source code rather than in the configuration screen. Thus, the *ModelDeclarationStatement* is created. The structure of *ModelDeclarationStatement* is displayed in Figure A.32.

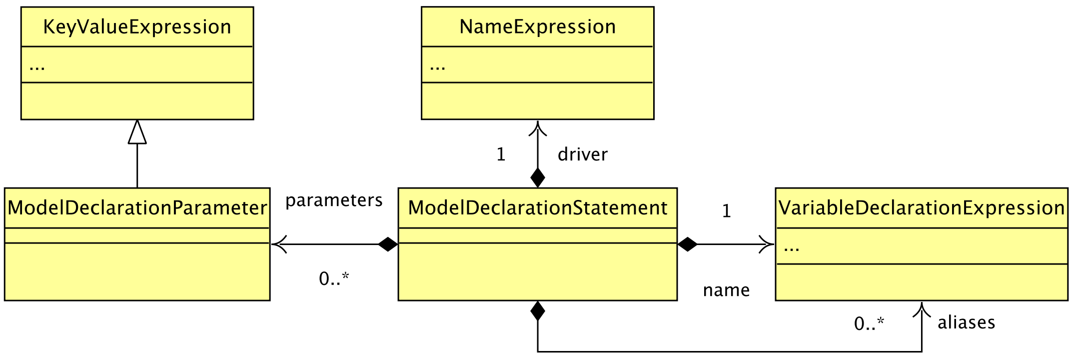


Figure A.32.: The structure of *ModelDeclarationStatement*

ModelDeclarationStatement is a sub-type of *Statement* and contains the following features:

- A mandatory name, represented by *VariableDeclarationExpression*;
- A number of aliases, represented by *VariableDeclarationExpression*;
- A mandatory driver, represented by *NameExpression*;

- A number of parameters, represented by **ModelDeclarationParameter**. A *ModelDeclarationParameter* is a sub-type of **KeyValueExpression**. A *KeyValueExpression* is used to represent expressions such as:

  ```
  a = "1", b = "2";
  ```

  KeyValueExpression has a *key* and a *value*, both of which are *Expressions*.

An example of the concrete syntax of *ModelDeclarationStatement* is shown below:

```
model Ecore alias e driver EMF {nsuri =
             "http://www.eclipse.org/emf/2002/Ecore"};
```

In this example, the name of the statement is *Ecore*. It also has an alias named *e*, and specifies that the *metamodel* should be loaded using *EMF*. It then gives the *nsuri* of the *metamodel* in the EPackage registry using a key-value pair.

A.3.10. ReturnStatement

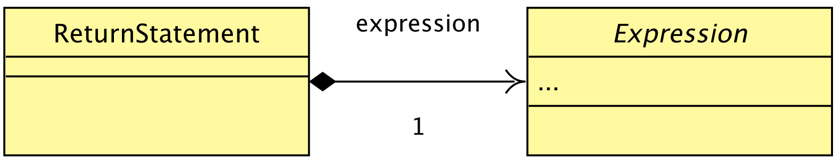


Figure A.33.: The structure of *ReturnStatement*

ReturnStatement is created to represent the return keyword in EOL. The *return* keyword is typically used in operation definitions to direct the control flow to the caller to the operation definition. For example:

```
var a = 10;
a.add(10).println();
operation Integer add(i:Integer) {
  return self+i;
}
```

The *return* keyword in line 4 denotes that the operation *add()* in line 3 should return the value of *self+i*. The structure of *ReturnStatement* is shown in Figure A.33.

A.3.11. ThrowStatement

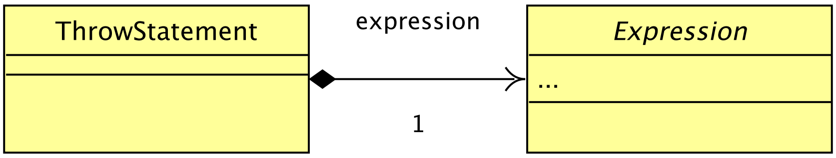


Figure A.34.: The structure of *ThrowStatement*

EOL provides the *throw* statement for throwing a value as an *EOLUserException* Java exception:

```
throw 42;
throw "Error!"
```

ThrowStatement is created to represent the *throw* keyword in EOL. The structure of *ThrowStatement* is shown in Figure A.34.

A.3.12. DeleteStatement

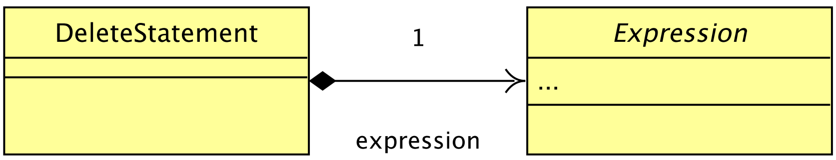


Figure A.35.: The structure of *DeleteStatement*

EOL provides the *delete* keyword to delete a model element from a model. Consider the example:

```
var student = Student.all.random();
if(student.first_name.isUndefined() and
     student.last_name.isUndefined()) {
```

```
  delete student;
}
```

The program picks a random *Student* and checks if the first and last names of the student are defined. If not, it deletes the student from the model. *DeleteStatement* is created to represent the *delete* keyword in EOL. Its structure is shown in Figure A.35.

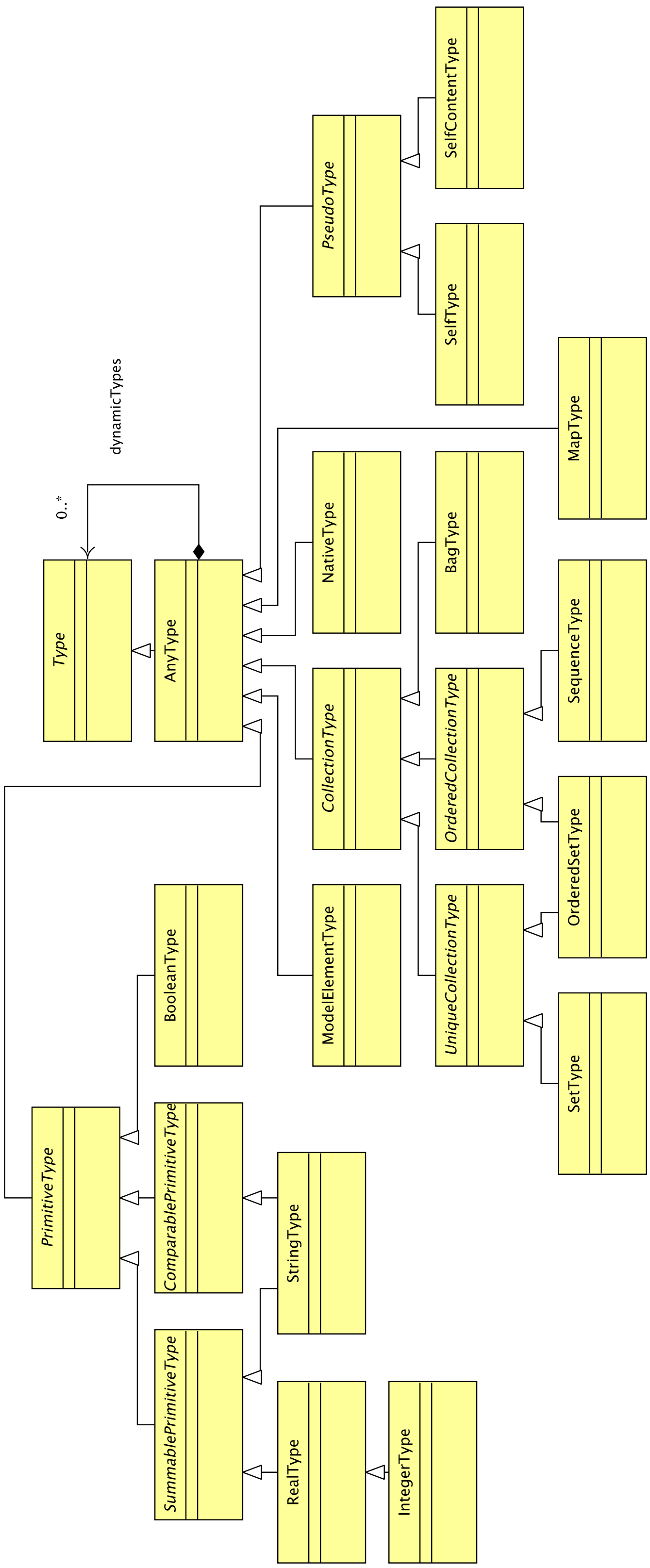


Figure A.36.: The structure of Type and its sub-types

A.4. Type

Type (abstract) and its sub-classes are created to represent the type system of EOL. In addition, a number of conceptual types are also introduced for the purpose of static analysis. The structure of *Type* and its sub-types is shown in Figure A.36.

A.4.1. AnyType

The *Any* type is a wildcard type in EOL's type system. The *Any* type in EOL originates from the *OclAny* type of the Object Constraint Language (OCL) [41]. A variable of type *Any* is able to hold any value of any type in EOL. In EOL, when a variable declaration is made, the type of the variable can be left unspecified:

```
1 var a;
2 var b: String;
```

In EOL, if no type declaration is provided in a variable declaration, it is assumed to be of type *Any*. Thus, variable *a* in line 1 is of type *Any*. *AnyType* is created to represent the *Any* type in EOL. *AnyType* contains the following features:

- *dynamicTypes* (of type *Type*). This property is used to hold all the possible types that an expression can hold;
- *declared*, of type *Boolean*, is used to denote if the *AnyType* is specifically declared by the developer.

The structure of *AnyType* is shown in Figure A.37

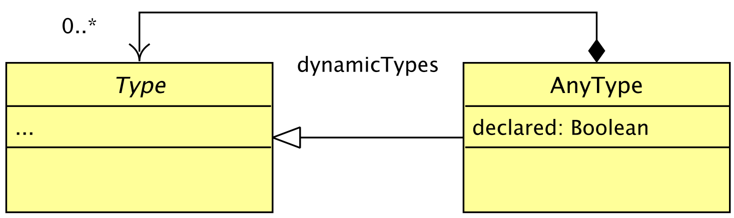


Figure A.37.: The structure of *AnyType*

A.4.2. PrimitiveType

PrimitiveType (abstract) is created to model primitive types in EOL. There are four primitive types in EOL: *Boolean*, *Integer*, *Real* and *String*. **BooleanType**, **IntegerType**, **RealType** and **StringType** are created to represent these types.

A number of conceptual types are also created to categorise the primitive types. **ComparablePrimitiveType** (abstract, conceptual) is created to represent the primitive types that are applicable to comparable operators (<, <=, > and >=). **SummablePrimitiveType** (abstract, conceptual) is created to represent the primitive types that are applicable to the summation operator (+).

A.4.3. CollectionType

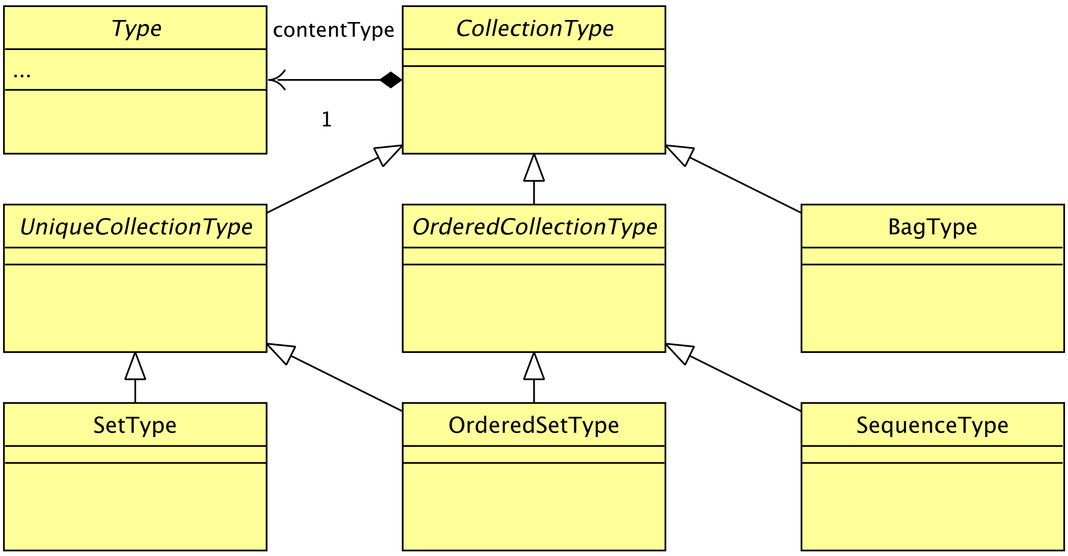


Figure A.38.: The structure of *CollectionType*

CollectionType (abstract) is created to model collection types in EOL. There are four collection types in EOL: *Bag*, *Set*, *OrderedSet* and *Sequence*. **BagType**, **SetType**, **OrderedSetType** and **SequenceType** are created to represent these types. *CollectionType* also has a *contentType* which denotes the type of the contents in a collection. For example:

```
var a: Sequence(String);
```

The line above declares a *Sequence* named *a*. Note that the content type (String) of the *Sequence* is also declared. Thus, *a* is a *Sequence* that can hold *String* values.

A number of conceptual types are also created to categorise the collection types. **UniqueCollectionType** (abstract, conceptual) is created to represent collection types in which contents are unique. **OrderedCollectionType** (abstract, conceptual) is created to represent the collection types in which contents are ordered.

The structure of *CollectionType* and its sub-types are shown in Figure A.38.

A.4.4. MapType

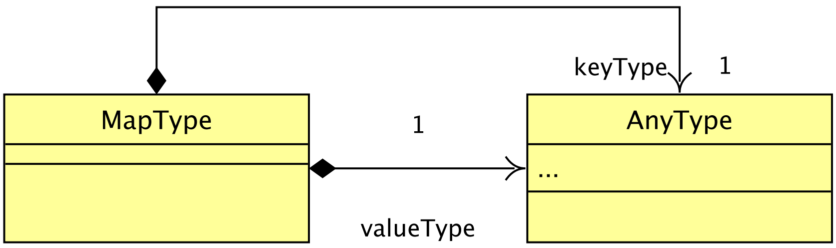


Figure A.39.: The structure of *MapType*

EOL supports the creation of *Map*. Consider the example:

```
var map = Map{1 = "Hello", 2 = "World"};
```

A *Map* is defined in line 1, with its keys {1,2} and values {“Hello”, “World”}. *MapType* is created to represent the type of a *map*. It is noteworthy that EOL does not provide concrete syntax for declaring the types of keys and values of *Maps*; thus, they are both considered to be of *Any* type.

MapType is created to represent the type of a *Map*. Its structure is shown in Figure A.39. *MapType* has a *keyType* and *valueType* which are bounded by *AnyType*.

A.4.5. ModelElementType

ModelElementType is created to represent types defined in metamodels. EOL adopts the syntax ! to access model element types. For example:

```
var student = new University!Student;
```

a model element type is specified by *University!Student* where *University* is the name of the model of interest and *Student* is the name of the element type in its metamodel. The structure of *ModelElementType* is shown in Figure A.40. *ModelElementType* contains the *modelName* and *elementName*, which is used to identify the model element type. It contains a *modelType* (of type Object) which is used to directly point to the model element type when it is resolved.

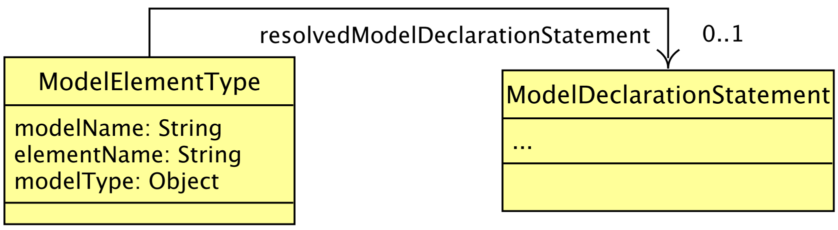


Figure A.40.: The structure of *ModelElementType*

A.4.6. ModelType

EOL allows the user to query the owning model of an object using the *owningModel()* operation:

```
var students = Student.all;
var newStudent = new Student;
students.add(newStudent);
var randomStudent = students.random();
randomStudent.owningModel().println();
```

The return type of *owningModel* is a *Model* type. *ModelType* is created to represent this type. The structure of *ModelType* is shown in Figure A.41. *ModelType* contains a *resolvedIMetamodel* (of type Object) which points directly to the corresponding *IMetamodel* driver, which is calculated at runtime. *ModelType* also has a reference to *ModelDeclarationStatement* named *resolvedModelDeclarationStatement*, which is calculated at runtime.

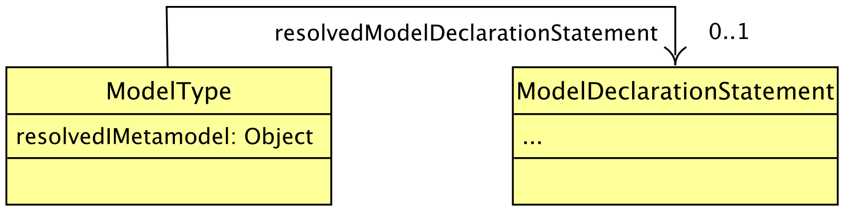


Figure A.41.: The structure of *ModelType*

A.4.7. NativeType

EOL enables users to create objects of the underlying programming environment (Java for example) by using *native* types:

```
var file = new Native("java.io.File")("myFile.txt");
```

NativeType is created to represent types of the underlying implementation platform (i.e. Java in the case of EOL's current implementation). The structure of *NativeType* is shown in Figure A.36. *NativeType* contains a *typeName* which is used to denote the name of the native type, and a *type* (of Type Object) which will be derived from and refer to the native type at runtime.

A.4.8. PseudoType

PseudoType (abstract) is created to help with the modelling of the EOL standard library. A detailed discussion of *PseudoTypes* is provided in Section 7.4.1.

A.4.9. OperationDefinition

OperationDefinition is used to represent an *operation/helper* definition in EOL. In EOL, operation definition is typically in the form as in the example below:

```
1 var a = 1; //a is 1
2 a = a.add(); //a is now 2
3 operation Integer add(): Integer {
4   return self + i;
5 }
```

Listing A.3: An example EOL program with variable declaration and operation definition

In the example, an *operation* is defined in line 3. The keyword *operation* is used to denote that an operation is being defined, the name of the operation is *add()* and the operation takes no parameters. Each *operation* has a *context* type, which is (optionally) declared after the keyword *operation.* The *context* type is used to denote to which types of objects the operation is applicable. In the example, the *operation add* is applicable to instances of *Integer*. The *operation* also has a *return* type, in this case, an *Integer*. An operation also has a special keyword named *self* which is used to fetch the caller of this operation. Thus, when line 2 is executed and control is transferred to the operation in line 3, the *self* is actually an alias of the variable *a*.

The structure of *OperationDefinition* is shown in Figure A.42. An *OperationDefinition* typically comprises:

- An optional *contextType* (of type *Type*). If no *contextType* is declared, it is assumed that the *contextType* is *AnyType*;
- An optional *returnType* (of type *Type*). If no *returnType* is declared, it is assumed that the *returnType* is *AnyType*;
- A *name* of type *NameExpression*;
- A number of *parameters* of type *FormalParameterExpression*;
- A *body* of type *Block*;
- An optional *annotationBlock* of type *AnnotationBlock*;
- A special variable *self* of type *VariableDeclarationExpression*;

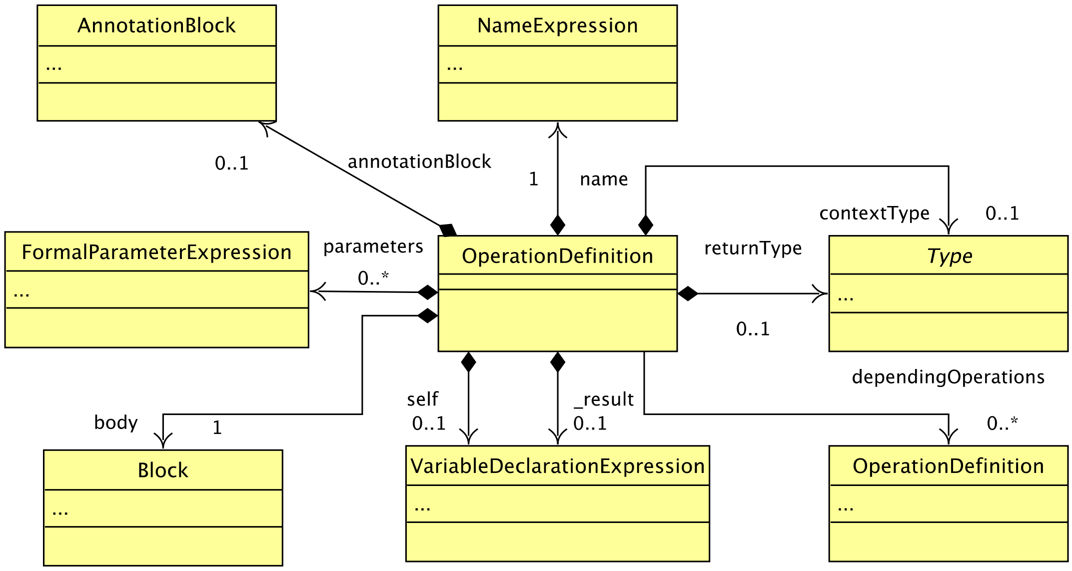


Figure A.42.: The structure of *OperationDefinition*

- A special variable *_result*, accessible from executable annotations, of type *VariableDeclarationExpression*;
- A collection of references to other *OperationDefinitions* named *dependingOperationDefinitions*, which are used to construct method call graphs, which will be used later in this thesis.

B. Metamodels Involved in the OO2DB Transformation

```
@namespace(uri="OO", prefix="OO")
package OO;
class Model extends Package { }

abstract class PackageableElement extends NamedElement {
  ref Package#contents ~package;
}

abstract class AnnotatedElement {
  val Annotation[*] annotations;
}

class Annotation {
  attr String key;
  attr String value;
}

abstract class NamedElement extends AnnotatedElement {
  attr String name;
}
```

```
class Package extends PackageableElement {
  val PackageableElement[*]#~package contents;
}

abstract class ~Classifier extends PackageableElement { }

class ExternalClass extends ~Class { }

class ~Class extends ~Classifier {
  ref ~Class#extendedBy ~extends;
  ref ~Class[*]#~extends extendedBy;
  val Feature[*]#owner features;
  attr Boolean isAbstract;
}

class Datatype extends ~Classifier { }

abstract class Feature extends NamedElement {
  ref ~Class#features owner;
  ref ~Classifier type;
  attr VisibilityEnum visibility;
}

abstract class StructuralFeature extends Feature {
  attr Boolean isMany;
}

class Operation extends Feature {
  val Parameter[*]#owner parameters;
}
```

```
class Parameter extends NamedElement {
  ref ~Classifier type;
  ref Operation#parameters owner;
}

class Reference extends StructuralFeature { }

class Attribute extends StructuralFeature { }

enum VisibilityEnum {
  public = 1;
  private = 2;
}
```

Listing B.1: The Emfatic Specification of the OO Metamodel

```
@namespace(uri="DB", prefix="DB")
package DB;
abstract class NamedElement {
    attr String name;
}

class Database {
   val DatabaseElement[*]#database contents;
}

abstract class DatabaseElement extends NamedElement {
   ref Database#contents database;
}
```

```
class Table extends DatabaseElement {
  val Column[*]#table columns;
  ref Column[*] primaryKeys;
}

class Column extends DatabaseElement {
  ref Table#columns table;
  attr String type;
}

class ForeignKey extends DatabaseElement {
  ref Column parent;
  ref Column child;
  attr Boolean isMany;
}
```

Listing B.2: The Emfatic Specification of the DB Metamodel

C. List of Acronyms

A

ANTLR: ANother Tool for Language Recognition

AMW: Atlas Model Weaving

AST: Abstract Syntax Tree

AST2EOL: AST to EOL model transformation

AST2ETL: AST to ETL model transformation

AST2EVL: AST to EVL model transformation

ATL: Atlast Transformation Language

C

CSV: Comma Separated Values

D

DSL: Domain Specific Language

E

ECL: Epsilon Comparison Language

EGL: Epsilon Generation Language

EMC: Epsilon Model Connectivity

EMF: Eclipse Modeling Framework

EMFATIC: A textual syntax for EMF Ecore (meta-)models.

EML: Epsilon Merging Language

EOL: Epsilon Object Language

EPL: Epsilon Pattern Language

Epsilon: Extensible Platform of Integrated Languages for Model Management

ESAMC: Epsilon Static Analysis Model Connectivity

ETL: Epsilon Transformation Language

EVL: Epsilon Validation Language

EWL: Epsilon Wizard Language

G

GME: Generic Modelling Environment

M

M2T: Model to Text transformation

M2M: Model to Model transformation

MDE: Model Driven Engineering

MDR: Meta Data Repository

MOF: Meta Object Facility

O

OCL: Object Constraint Language

OMG: Object Management Group

Q

QVT: Queries/Views/Transformations

R

RDBMS: Relational Database Management Systems

S

SPPD: Sub-Optimal Performance Pattern Detection

SAX: Simple API for XML

T

T2M: Text to Model Transformation

U

UML: Unified Modelling Language

X

XMI: XML Model Interchange

XML: Extensible Markup Language

XSLT: Extensible Stylesheet Language Transformation

12. Bibliography

[1] Object Management Group. Unified Modelling Language 2.1.2 Specification. `http://www.omg.org/spec/UML/2.1.2/`. Accessed 01-01-2016.

[2] Jean Bézivin. Model Driven Engineering: An Emerging Technical Space. In *Generative and Transformational Techniques in Software Engineering*, pages 36–64. Springer, 2006.

[3] Frdric Jouault and Ivan Kurtev. Transforming models with atl. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer Berlin Heidelberg, 2006.

[4] Alexander Königs. Model Transformation with Triple Graph Grammars. In *Proceedings of Model Transformations in Practice Workshop at MoDELs Conference 2005*, pages 166–182, 2005.

[5] Dániel Varró, András Balogh, and András Pataricza. The VIATRA2 Transformation Framework: Model Transformation by Graph Transformation. In *Eclipse Modeling Symposium*. Citeseer, 2006.

[6] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[7] J. Sanchez Cuadrado, E. Guerra, and J. de Lara. Uncovering Errors in ATL Model Transformations Using Static Analysis and Constraint Solving. In *IEEE 25th International Symposium on Software Reliability Engineering (ISSRE), 2014*, pages 34–44, Nov 2014.

[8] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, 98, Part 1:80–99, 2015. Fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modelling with Eclipse (ACME2012).

[9] Dimitrios Kolovos, Louis Rose, Richard Paige, and A Garcia-Dominguez. *The Epsilon Book*. Eclipse, 2010.

[10] Martin Matula. NetBeans Metadata Repository. *NetBeans Community*, 2003.

[11] Emfatic Language Reference. `http://www.eclipse.org/epsilon/doc/articles/emfatic/`. Accessed: 01-01-2016.

[12] Moritz Eysholdt and Heiko Behrens. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 307–309, New York, NY, USA, 2010. ACM.

[13] Andrew Watson. A Brief History of MDA. *Upgrade, the European Journal for the Informatics Professional*, 9(2):7–11, 2008.

[14] Ari Jaaksi. Developing Mobile Browsers in a Product Line. *IEEE software*, 19(4):73–80, 2002.

[15] Juha Kärnä, Juha-Pekka Tolvanen, and Steven Kelly. Evaluating the Use of Domain-Specific Modeling in Practice. In *Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling*, 2009.

[16] Parastoo Mohagheghi, MiguelA. Fernandez, JuanA. Martell, Mathias Fritzsche, and Wasif Gilani. MDE Adoption in Industry: Challenges and Success Criteria. In MichelR.V. Chaudron, editor, *Models in Software Engineering*, volume 5421 of *Lecture Notes in Computer Science*, pages 54–59. Springer Berlin Heidelberg, 2009.

[17] Paul Baker, Shiou Loh, and Frank Weil. Model-Driven Engineering in a Large Industrial Context Motorola Case Study. In Lionel Briand and Clay Williams, editors, *Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 476–491. Springer Berlin Heidelberg, 2005.

[18] Dimitrios S. Kolovos, Louis M. Rose, Nicholas Matragkas, Richard F. Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan De Lara, István Ráth, Dániel Varró, Massimo Tisi, and Jordi Cabot. A Research Roadmap Towards Achieving Scalability in Model Driven Engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 2:1–2:10, New York, NY, USA, 2013. ACM.

[19] Louis M Rose. *Structures and Processes for Managing Model-Metamodel Co-evolution*. PhD thesis, University of York, 2011.

[20] IEEE Standard Classification for Software Anomalies. *IEEE Std 1044-1993*, 1994.

[21] Benjamin Livshits. *Improving software security with precise static and runtime analysis*. PhD thesis, Stanford University, 2006.

[22] Michael I Schwartzbach. Lecture notes on static analysis. *Basic Research in Computer Science, University of Aarhus, Denmark*, 2008.

[23] J McQuillan and J Power. White-box Coverage Criteria for Model Transformations. In *Proceedings of the First International Workshop on Model Transformation with ATL*, pages 63–77, 2009.

[24] J.-M. Mottu, S. Sen, M. Tisi, and J. Cabot. Static Analysis of Model Transformations for Effective Test Generation. In *IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE), 2012*, pages 291–300, Nov 2012.

[25] Dresden OCL. `http://www.dresden-ocl.org/index.php/DresdenOCL`. Accessed: 01-01-2016.

[26] Eclipse OCL. `http://www.eclipse.org/modeling/mdt/?project=ocl`. Accessed: 01-01-2016.

[27] Dimitrios S. Kolovos, Richard F. Paige, and Fiona AC Polack. Scalability: The holy grail of model driven engineering. In *ChaMDE 2008 Workshop Proceedings: International Workshop on Challenges in Model-Driven Software Engineering*, pages 10–14, 2008.

[28] István Ráth, Gergely Varró, and Dániel Varró. Change-Driven Model Transformations. In *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 342–356. Springer Berlin Heidelberg, 2009.

[29] Frédéric Jouault, Jean Bézivin, and Mikaël" Barbero. Towards An Advanced Model-Driven Engineering Toolbox. *Innovations in Systems and Software Engineering*, 5(1):5–12, 2009.

[30] The Connected Data Objects Model Repository (CDO) Project. `http://www.eclipse.org/cdo/`. Accessed: 01-01-2016.

[31] Javier Espinazo Pagán, Jesúss Sánchez Cuadrado, and Jesús García Molina. Morsa: A Scalable Approach for Persisting and Accessing Large Models. In *Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 77–92. Springer Berlin Heidelberg, 2011.

[32] Bryan Hunt. MongoEMF. `http://github.com/BryanHunt/mongo-emf/wiki`. Accessed: 01-01-2016.

[33] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[34] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[35] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical Assessment of MDE in Industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, New York, NY, USA, 2011. ACM.

[36] Dimitriois Kolovos. *An Extensible Platform for Specification of Integrated Languages for Model Management*. PhD thesis, University of York, 2008.

[37] Richard F. Paige, Phillip J. Brooke, and Jonathan S. Ostroff. Metamodel-based model conformance and multiview consistency checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(3), July 2007.

[38] Object Management Group. XML Metadata Interchange (XMI) Specification. `http://www.istr.unican.es/pyemofuc/PyEmofUCFiles/XMI_formal-14-04-04.pdf`. Accessed: 01-01-2016.

[39] Object Management Group. Object constraint language. `http://www.omg.org/spec/OCL/`. Accessed: 01-01-2016.

[40] José Álvarez, Andy Evans, and Paul Sammut. Mml and the metamodel architecture. In *WTUML: Workshop on Transformation in UML*. Citeseer, 2001.

[41] Jordi Cabot and Martin Gogolla. Object Constraint Language (OCL): A Definitive Guide. In *Formal Methods for Model-Driven Engineering*, volume 7320 of *Lecture Notes in Computer Science*, pages 58–90. Springer Berlin Heidelberg, 2012.

[42] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

[43] Stephen A White. Introduction to bpmn. `https://www.ibm.com/developerworks/community/files/basic/anonymous/api/library/7624eb5a-089a-41bf-9b71-b3c33739e18d/document/e908d328-7b50-40e3-8107-70af4e6bb48f/media`. Accessed: 01-01-2016.

[44] The Open Group. ArchiMate 2.1 Specification. `http://pubs.opengroup.org/architecture/archimate2-doc/`. Accessed: 01-01-2016.

[45] M. Faugere, T. Bourbeau, R. De Simone, and S. Gerard. MARTE: Also an UML Profile for Modeling AADL Applications. In *12th IEEE International Conference on Engineering Complex Computer Systems, 2007.*, pages 359–364, July 2007.

[46] Jeff Gray, Jules White, and Aniruddha Gokhale. Model-driven Engineering: Raising the Abstraction Level Through Domain-specific Modeling. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 1:1–1:2, New York, NY, USA, 2010. ACM.

[47] Object Management Group. Meta Object Facility. `http://www.omg.org/mof/`. Accessed: 01-01-2016.

[48] Microsoft. Microsoft Domain Specific Languages. `https://msdn.microsoft.com/en-us/library/dd361847.aspx`. Accessed: 01-01-2016.

[49] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The Generic Modeling Environment. In *Workshop on Intelligent Signal Processing*, volume 17, pages 1–6, 2001.

[50] Shane Sendall and Wojtek Kozaczynski. Model Transformation the Heart and Soul of Model-Driven Software Development. Technical report, 2003.

[51] Anneke G Kleppe, Jos B Warmer, and Wim Bast. *MDA Explained: the Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003.

[52] DimitriosS. Kolovos, RichardF. Paige, and FionaA.C. Polack. Merging Models with the Epsilon Merging Language (EML). In *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 215–229. Springer Berlin Heidelberg, 2006.

[53] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005) Graph and Model Transformation 2005.

[54] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.

[55] Jean-Rémy Falleri, Marianne Huchard, and Clementine Nebut. Towards a Traceability Framework for Model Transformations in Kermeta. In *ECMDA-TW'06: ECMDA Traceability Workshop*, pages 31–40, Bilbao (Spain), July 2006. Sintef ICT, Norway.

[56] D.H. Akehurst. Transformations Based on Relations. In *ECOOP 2004 Workshop Reader, Workshop on Model Driven Development (WMDD 2004)*, 2004.

[57] Edward D Willink. UMLX: A Graphical Transformation Language for MDA. In *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications*, pages 13–24, 2003.

[58] Edward Willink, Horacio Hoyos, and Dimitris Kolovos. Yet Another Three QVT Languages. In *Theory and Practice of Model Transformations*, volume 7909 of *Lecture Notes in Computer Science*, pages 58–59. Springer Berlin Heidelberg, 2013.

[59] Octavian Patrascoiu and Peter Rodgers. Embedding OCL expressions in YATL. *Proc. OCL and Model Driven Engineering wokshop, UML04*, 2004.

[60] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Transforming Models with ATL. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer Berlin Heidelberg, 2006.

[61] DimitriosS. Kolovos, RichardF. Paige, and FionaA.C. Polack. The Epsilon Transformation Language. In *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg, 2008.

[62] Dimitrios S Kolovos, Richard F Paige, Fiona AC Polack, and Louis M Rose. Update transformations in the small with the epsilon wizard language. *Journal of Object Technology (JOT)*, 6(9):53–69, 2003.

[63] Rubino Geiß and Moritz Kroll. GrGen.NET: A Fast, Expressive, and General Purpose Graph Rewrite Tool. In *Applications of Graph Transformations with*

Industrial Relevance, volume 5088 of *Lecture Notes in Computer Science*, pages 568–569. Springer Berlin Heidelberg, 2008.

[64] Gabriele Taentzer. AGG: AGraph Transformation Environment for Modeling and Validation of Software. In *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer Berlin Heidelberg, 2004.

[65] beo group. Acceleo. `http://www.eclipse.org/acceleo/`. Accessed: 01-01-2016.

[66] Benjamin Klatt. Xpand: A Closer Look At the Model2text Transformation Language. *Language*, 10(16):2008, 2007.

[67] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. The Epsilon Generation Language. In *Model Driven Architecture Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2008.

[68] Terence Parr. *The definitive ANTLR reference: building domain-specific languages*. Pragmatic Bookshelf Raleigh, 2007.

[69] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Model-Based Language Engineering with EMFText. In *Generative and Transformational Techniques in Software Engineering IV*, volume 7680 of *Lecture Notes in Computer Science*, pages 322–345. Springer Berlin Heidelberg, 2013.

[70] P. Klint, T. van der Storm, and J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM '09. Ninth IEEE International Working Conference on*, pages 168–177, Sept 2009.

[71] Lennart CL Kats, Karl T Kalleberg, and Eelco Visser. Domain-Specific Languages for Composable Editor Plugins. *Electronic Notes in Theoretical Computer Science*, 253(7):149–163, 2010. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

[72] Christian Lange, MRV Chaudron, Johan Muskens, LJ Somers, and HM Dortmans. An Empirical Investigation in Quantifying Inconsistency and Incompleteness of UML Designs. In *Workshop Consistency Problems in UML-based Software Development II*, pages 26–34, 2003.

[73] DimitriosS. Kolovos, RichardF. Paige, and FionaA.C. Polack. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In *Rigorous Methods for Software Construction and Analysis*, volume 5115 of *Lecture Notes in Computer Science*, pages 204–218. Springer Berlin Heidelberg, 2009.

[74] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelsteiin, and Ernst Ellmer. Flexible Consistency Checking. *ACM Trans. Softw. Eng. Methodol.*, 12(1):28–63, January 2003.

[75] Jean Louis Sourrouille and Guy Caplat. A Pragmatic View About Consistency Checking of UML Models. In *Proceedings of the Workshop on Consistency Problems in UML-Based Software Development*, pages 43–50, 2003.

[76] Raf Haesen and Monique Snoeck. Implementing consistency management techniques for conceptual modeling. *Proc. 3rd Wordshop on Consistency Problems in UML-Based Software Development*, 2004.

[77] Dimitrios S. Kolovos. Establishing Correspondences between Models with the Epsilon Comparison Language. In *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 146–157. Springer Berlin Heidelberg, 2009.

[78] Maik Schmidt and Tilman Gloetzner. Constructing difference tools for models using the sidiff framework. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion ’08, pages 947–948, New York, NY, USA, 2008. ACM.

[79] EMF Compare. `https://www.eclipse.org/emf/compare/`. Accessed 01-01-2016.

[80] Jean Bézivin, Frédéric Jouault, and David Touzet. An Introduction to the Atlas Model Management Architecture. *Rapport de recherche*, (05.01):10–49, 2005.

[81] Amine Benelallam, Abel Gmez, Gerson Suny, Massimo Tisi, and David Launay. Neo4EMF, A Scalable Persistence Layer for EMF Models. In *Modelling Foundations and Applications*, volume 8569 of *Lecture Notes in Computer Science*, pages 230–241. Springer International Publishing, 2014.

[82] Pramod J Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education, 2012.

[83] Graphical modellin project. `http://www.eclipse.org/modeling/gmp/`. Accessed 01-01-2016.

[84] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 173–174, New York, NY, USA, 2010. ACM.

[85] Andreza Vieira and Franklin Ramalho. A Static Analyzer for Model Transformations. In *Third International Workshop on Model Transformations with ATL*. Citeseer, 2011.

[86] DimitriosS. Kolovos, RichardF. Paige, and FionaA.C. Polack. The Grand Challenge of Scalability for Model Driven Engineering. In MichelR.V. Chaudron, editor, *Models in Software Engineering*, volume 5421 of *Lecture Notes in Computer Science*, pages 48–53. Springer Berlin Heidelberg, 2009.

[87] Konstantinos Barmpis and Dimitris Kolovos. Hawk: Towards a Scalable Model Indexing Architecture. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 6:1–6:9, New York, NY, USA, 2013. ACM.

[88] Frdric Besson, David Cachera, Thomas Jensen, and David Pichardie. Certified Static Analysis by Abstract Interpretation. In *Foundations of Security Analysis*

and Design V, volume 5705 of *Lecture Notes in Computer Science*, pages 223–257. Springer Berlin Heidelberg, 2009.

[89] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP95 Object-Oriented Programming, 9th European Conference, arhus, Denmark, August 711, 1995*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101. Springer Berlin Heidelberg, 1995.

[90] Gertrude Neuman Levine. Defining Defects, Errors, and Service Degradations. *ACM SIGSOFT Software Engineering Notes*, 34(2):1–14, 2009.

[91] Yichen Xie. *Static Detection of Software Errors.* PhD thesis, Citeseer, 2006.

[92] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

[93] Lu Luo. Software Testing Techniques. *Institute for software research international Carnegie mellon university Pittsburgh, PA*, 15232(1-19):19, 2001.

[94] Michael D Ernst. Static and Dynamic Analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27. Citeseer, 2003.

[95] Julien HENRY. *Static Analysis by Abstract Interpretation and Decision Procedures.* PhD thesis, Université de Grenoble, 2014.

[96] Arnaud J. Venet and Michael R. Lowry. Static Analysis for Software Assurance: Soundness, Scalability and Adaptiveness. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 393–396, New York, NY, USA, 2010. ACM.

[97] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[98] CodePeer. `http://www.adacore.com/codepeer`. Accessed 01-01-2016.

[99] Thomas Hofer. *Evaluating Static Source Code Analysis Tools*. PhD thesis, Citeseer, 2010.

[100] Brian A Davey and Hilary A Priestley. *Introduction to Lattices and Order*. Cambridge university press, 2002.

[101] Alfred Tarski et al. A Lattice-theoretical Fixpoint Theorem and Its Applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[102] Wolfgang Wögerer. A survey of static program analysis techniques. *Vienna University of Technology*, 2005.

[103] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[104] William R Bush, Jonathan D Pincus, and David J Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software-Practice and Experience*, 30(7):775–802, 2000.

[105] Stephen C Johnson. *Lint, a C Program Checker*. Citeseer, 1977.

[106] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A Tool for Using Specifications to Check Code. *ACM SIGSOFT Software Engineering Notes*, 19(5):87–96, 1994.

[107] David Hovemeyer and William Pugh. Finding Bugs Is Easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[108] Eclipse JDT API Specification. `http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Foverview-summary.html`. Accessed: 28/10/2015.

[109] Birgit Demuth. The Dresden OCL Toolkit and Its Role in Information Systems Development. In *Proc. of the 13th International Conference on Information Systems Development (ISD2004)*, volume 7, 2004.

[110] S. Mitra and Kee Sup Kim. XPAND: an efficient test stimulus compression technique. *Computers, IEEE Transactions on*, 55(2):163–173, Feb 2006.

[111] Jonathan Musset, Étienne Juliot, Stéphane Lacrampe, William Piers, Cédric Brun, Laurent Goubet, Yvan Lussaud, and Freddy Allilaire. Acceleo User Guide. `http://www.acceleo.org/doc/obeo/en/acceleo-2.6-user-guide.pdf`. Accessed 01-01-2016.

[112] DimitriosS. Kolovos, LouisM. Rose, James Williams, Nicholas Matragkas, and RichardF. Paige. A Lightweight Approach for Managing XML Documents with MDE Languages. In *Modelling Foundations and Applications*, volume 7349 of *Lecture Notes in Computer Science*, pages 118–132. Springer Berlin Heidelberg, 2012.

[113] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 18–33. Springer Berlin Heidelberg, 2009.

[114] Edward Willink, Jordi Cabot, Robert Clarisó, Martin Gogolla, Burkhart Wolff, Tiziana Margaria, Julia Padberg, and Gabriele Taentzer. Modeling the OCL Standard Library. *ECEASST*, 44, 2011.

[115] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *Static Analysis*, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255. Springer Berlin Heidelberg, 2009.

[116] Eclipse OCL User Guide. `http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FUsersGuide.html`. Accessed 01-01-2016.

[117] Edward Willink, Jordi Cabot, Robert Clarisó, Martin Gogolla, and Burkhart Wolff. Aligning OCL with UML. In *OCL 2011 Workshop*. Citeseer, 2011.

[118] Uml2. `https://eclipse.org/modeling/mdt/?project=uml2`. Accessed 01-01-2016.

[119] Object Management Group. MOF 2.0 Query/Views/Transformations RFP. `http://www.omg.org/cgi-bin/doc?ad/2002-4-10`. Accessed 01-01-2016.

[120] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS:: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 249–254, New York, NY, USA, 2006. ACM.

[121] Atl user manual. `http://www.eclipse.org/atl/documentation/old/ATL_User_Manual[v0.7].pdf`. Accessed 01-01-2016.

[122] Sagar Sen, Naouel Moha, Benoit Baudry, and Jean-Marc Jézéquel. Meta-model Pruning. In *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 32–46. Springer Berlin Heidelberg, 2009.

[123] Object Management Group. MOF Model To Text Transformation Language (MOFM2T), 1.0. `http://www.omg.org/spec/MOFM2T/1.0/`. Accessed 01-01-2016.

[124] Charles L Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[125] Carlos A. González, Fabian Büttner, Robert Clarisó, and Jordi Cabot. EMFtoCSP: A Tool for the Lightweight Verification of EMF Models. In *Proceedings of the First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches*, FormSERA '12, pages 44–50, Piscataway, NJ, USA, 2012. IEEE Press.

[126] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A Challenging Model Transformation. In *Model Driven Engineering Languages and Systems: 10th International Conference, MoDELS 2007, Nashville,*

USA, September 30 - October 5, 2007. Proceedings, pages 436–450. Springer Berlin Heidelberg, 2007.

[127] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. Extensive Validation of OCL Models by Integrating SAT Solving into USE. In *Objects, Models, Components, Patterns: 49th International Conference, TOOLS 2011, Zurich, Switzerland, June 28-30, 2011. Proceedings*, pages 290–306. Springer Berlin Heidelberg, 2011.

[128] Emina Torlak and Daniel Jackson. Kodkod: A Relational Model Finder. In *Tools and Algorithms for the Construction and Analysis of Systems: 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007. Proceedings*, pages 632–647. Springer Berlin Heidelberg, 2007.

[129] DimitriosS. Kolovos, RichardF. Paige, and FionaA.C. Polack. The Epsilon Object Language (EOL). In *Model Driven Architecture Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer Berlin Heidelberg, 2006.

[130] S Skiena. Dijkstra's algorithm. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley*, pages 225–227, 1990.

[131] Zest - eclipse. `https://www.eclipse.org/gef/zest/`. Accesse 01-01-2016.

[132] Marcel Van Amstel, Steven Bosems, Ivan Kurtev, and Luís Ferreira Pires. Performance in Model Transformations: Experiments with ATL and QVT. In Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations*, volume 6707 of *Lecture Notes in Computer Science*, pages 198–212. Springer Berlin Heidelberg, 2011.

[133] Markus Scheidgen. Reference Representation Techniques for Large Models. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 5.1–5.9, New York, NY, USA, 2013. ACM.