Optimisation of Model Management Programs Using Automated Program Rewriting

Qurat ul ain Ali

PhD

Department of Computer Science,
University of York

December 2023

Abstract

Over the last few years, the use of Model Driven Engineering (MDE) in industrial applications has been increasing rapidly. For the use of MDE-based applications at a larger scale, they should scale well in terms of the size of the model, execution time, and memory consumption. Adapting the use of MDE for extensive software landscapes, where the underlying models grow large in size poses various scalability challenges. When model management tools run on pay-as-you-go cloud-based resources, inefficiency and limited scalability incur substantial costs. Hence, there is vested interest from vendors of cloud-based MDE solutions for efficient and scalable model management tools.

There are specific high-level languages to develop model management programs tailored for the specific tasks they target. This work aims to improve the performance of certain types of model management programs through static analysis. An approach is proposed for optimising model management tasks, particularly model validation, model-to-model transformation and model comparison over large-scale models. The proposed approach leverages static analysis and automated program rewriting techniques to optimise model management programs over large-scale EMF-based models. This optimisation approach aims to bring efficiency in terms of execution time and memory footprint so that developers can still express model management programs in high-level language and execute these programs efficiently. The program is automatically rewritten to an optimised version (where possible). The optimised program is semantically equivalent to the original program but faster and more efficient to execute. The experiments of this study have shown a significant performance gain in execution time and memory footprint.

Contents

List of Figures

List of Tables

Listings

List of Algorithms

To my beloved family

Papa, Mama, Sania & Hassan

Acknowledgment

First and foremost, I would like to thank my supervisor, Professor Dimitris Kolovos, for his invaluable guidance and encouragement throughout the research process. His expertise and strong work ethic were instrumental in shaping the direction of this study. Professor Dimitris is very kind and always approachable. He would always reply promptly and provide detailed feedback on my work. I would also like to thank my co-supervisor, Dr Konstantinos Barmpis, for his constant support, discussions, and encouragement. Along with the academic guidance, his appreciation and support have helped me stay motivated during this journey. My regards to the TAP accessor, Professor Jim Woodcock, who provided me with assistance and support at various stages of this project. I am also profoundly grateful to the University of York, the Lowcomote project, and the EU for providing the resources and facilities necessary to complete this research successfully. All the ASE members have been very supportive, and the feedback during the seminars benefited me. They helped me improve my presentation skills.

I appreciate and thank Dr Waqar Hussain for motivating me to apply for a PhD and reviewing my documents for the application. My friend Sorour Jahanbin was a significant support during this PhD journey. Our discussions about various topics of life helped me grow not only professionally but personally as well. I honestly appreciate that. Special thanks to Fazeelah for feeding me good Pakistani food. A big thanks to my friends, including Palwasha, Hassan, Summan and Maha, for the wonderful gatherings in York.

Last but certainly not least, a huge thanks to my family, my parents, Mr. Farzand (late) and Mrs. Rukhsana. I am forever grateful for their love and prayers. They have instilled in me a strong work ethic and a love of learning, and I am forever grateful for their tireless efforts in nurturing my intellectual curiosity. My siblings Sania and Hassan have always been there for me, providing emotional support, motivation, and a listening ear whenever needed. Their encouragement and kindness were a constant source of motivation and inspiration. I would not be able to achieve my academic goals without them.

Author Declaration

All the work contained in this thesis represents the original contribution of the author. Chapter 4 and Section 5.3 detail collaborative work performed for integrating this work with other MDE tools and technologies. Parts of the work described in this thesis have been previously published by the author in:

- [**Conference**] Qurat ul ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. 2023. “Towards Efficient Model Comparison using Automated Program Rewriting”. In Proceedings of the 16th ACM SIGPLAN International-Conference on Software Language Engineering (SLE ’23), October23–24, 2023, Cascais, Portugal. https://doi.org/10.1145/3623476.3623519

- [**Conference**] Qurat ul ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. 2022. “Selective Traceability for Rule-Based Model-to-Model Transformations”. In Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2022). Association for Computing Machinery, New York, NY, USA, 98–109. https://doi.org/10.1145/3567512.3567521

- [**Conference**] Q. Ul Ain Ali, D. Kolovos and K. Barmpis, “Identification and Optimisation of Type-Level Model Queries,” 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), Fukuoka, Japan, 2021, pp. 751-760, doi: 10.1109/MODELS-C53483.2021.00121.

- [**Doctoral Symposium**] Q. Ul Ain Ali, “Heterogeneous Model Query Optimisation,” 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), Fukuoka, Japan, 2021, pp. 648-653, doi: 10.1109/MODELS-C53483.2021.00104.

- [**Workshop**] Q. U. A. Ali, B. Horváth, D. Kolovos, K. Barmpis and Á. Horváth, “Towards Scalable Validation of Low-Code System Models: Mapping EVL to VIATRA Patterns,” 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), Fukuoka, Japan, 2021, pp. 83-87, doi: 10.1109/MODELS-C53483.2021.00019.

- [**Workshop**] Qurat ul ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. 2020. “Efficiently querying large-scale heterogeneous models”. In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS ’20). Association for Computing Machinery, New York, NY, USA, Article 73, 1–5. https://doi.org/10.1145/3417990.3420207

Chapter 1

Introduction

This chapter discusses the significance of addressing scalability challenges in the broader Model-Driven Engineering (MDE) context. It states and explains the problem statement and then it lists contributions of the research.

Section 1.1 provides a general context of this research work. Section 1.2 presents the problem that this research work aims to address. Section 1.3 provides a brief description of the contributions of this thesis along with putting the putting the contribution in the wider context of software engineering. Finally, the structure of this thesis is presented in Section 1.4.

1.1 Context

In today's world, with the advancement in software systems, the complexity is also increasing proportionally. Different software methodologies have been proposed to address this complexity challenge. Over time, these methodologies, have been aiming to improve software quality and productivity. Over the past few decades, approaches have been focusing on various features of software quality, but the main aim is to reduce the complexity and raise the abstraction level.

MDE [8, 9] is a software engineering methodology that considers models as first-class artefacts, raising the level of abstraction. Models are the first class citizens in the software development life cycle. It allows software developers to focus on the problem domain (e.g. aircraft engines, banking transaction systems) rather than implementation details (such as programming languages, backend technology etc.). Over the course of time, the use of MDE is becoming an increasingly popular trend in both academia and industrial application [10]. The use of MDE provides benefits not in just making complexity manageable but also it increases productivity [11, 12] by automating processes like code generation and enhances software product quality. In MDE, some artefacts such as working source code, documents etc. are automatically generated from one or more models. Part of the entire code is a result of auto generation but both hand-written and generated code often co-exist in model-based development processes.

Though there is an increased popularity of MDE compared to traditional software engineering methodologies, there are still certain challenges limiting the wider use of MDE in industry such as scalability and collaborative development [13]. These challenges are discussed in detail in Section 2.3. One of the main challenges in the context of this research is the poor scalability [14,15] across various dimensions of MDE tools that do not allow them to scale for large applications having several collaborating users. Scalability becomes a critical issue when it comes to very large and complex models such as those involved in enterprise applications – for example in industries such as automotive and aerospace. In order to utilise the benefits of MDE in these complex and large scale industrial applications the tools and technologies need to be scalable. This can enable developers to develop and manipulate large-scale models in a collaborative manner with other developers.

This research aims to tackle the scalability challenge, producing novel techniques and algorithms for the optimisation of certain model management programs over large-scale models by leveraging static program analysis and automated program rewriting techniques. The work aims to produce a prototype that will design and implement algorithms and techniques on top of an existing model management languages and to demonstrate the performance benefits of the proposed optimisations using experimental evaluation.

1.2 Problem Statement

A typical MDE workflow includes several tasks, including model validation, model-to-model transformations, and model-to-text transformations. The programs to automate these tasks are composed of a common set of queries/expressions (such as iterating over collections of elements in a model) operating over model elements. Queries on models can be specified using general-purpose programming languages such as Java, or using tailored model-management languages such as the Object Constraint Language (OCL) [16] – and its various flavours embedded in model-to-model and model-to-text transformation languages such as Acceleo [17] and ATL [18] – the Epsilon Object Language (EOL) [19] and its task-specific languages that build on top of it, and the VIATRA Query Language (VQL) [20]. The main strength of dedicated model management languages is that they offer built-in abstractions for common tasks (e.g. rule-based decomposition and element resolution in model-to-model transformations, protected regions for mixing generated and hand-written content in model-to-text transformations, constraint dependency management in model validation) which facilitates more concise, maintainable and technology-independent model management programs.

The main shortcoming of these dedicated model management languages compared to general-purpose languages such as Java is performance. While widely-used general-purpose languages are typically compiled and benefit from advanced runtimes offering features such as adaptive optimisation and microarchitecture specific speed-ups, model management languages are predominately interpreted, and therefore their execution speed is substantially lower. This can become a scalability bottleneck as models grow in size and inhibit their applicability to projects that involve large models [14,21]. While executing queries on models containing a large number of model elements, for example in the order of millions, a significant performance cost in terms of execution time is commonly incurred.

In this work, an architecture is proposed for improving the execution speed of a subset of interpreted model management programs namely querying, validation, transformation and comparison over large EMF-based models, written in languages of the Epsilon[1](discussed in detail in Chapter 2) platform, by using static analysis and automated program rewriting techniques. The envisioned approach aims to provide optimisations for several model management tasks. The experiments are carried out on large-scale models containing up to several millions of model elements, to demonstrate the performance benefits compared to the traditional execution of model management programs.

1.3 Contributions

Scalability is one of the key challenges identified in the field of model-driven engineering. In this thesis, the following contributions are presented to optimise the performance of model management programs.

Figure 1.1 places the research area into the wider context of MDE and software engineering in a tree-structured diagram with the contribution areas labelled in yellow. The diagram goes top-down from general to specific concepts, omitting ones not relevant to this work as it becomes more specific. This work aims at optimising model management programs more specifically model querying, model validation, model-to-model transformations and model comparison to tackle the scalability challenge in MDE. The main contributions of this thesis are as follows:

- **Optimisation of model querying and model validation programs** over EMF-based models by pre-computing indices to speed up property-based model element filtering. This is achieved using static analysis and program rewriting. Additionally, this approach has been used in the context of translating EOL expressions to Viatra for providing incremental querying (explained in detail in Chapter 5).

- **Optimisation of model-to-model(M2M) transformation programs** to reduce the size of the transformation trace by leveraging static analysis to translate from a rule-based M2M program to an imperative M2M program (explained in detail in Chapter 6).

[1]https://eclipse.dev/epsilon/

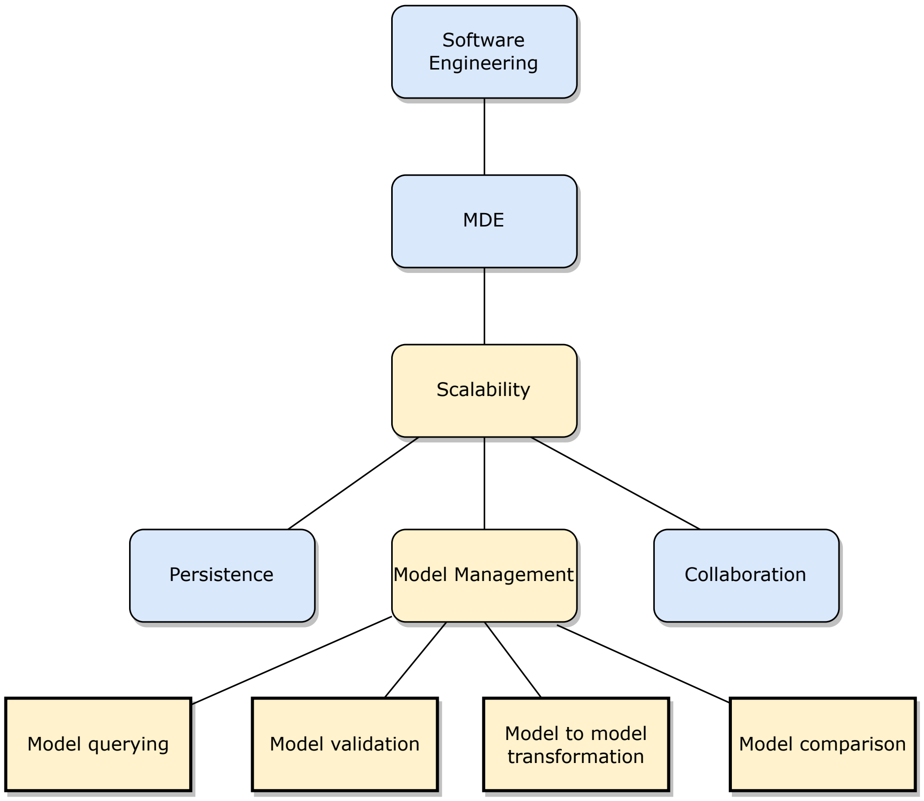


Figure 1.1: A tree structured representation of thesis contribution to the field of Software Engineering

- **Optimisation of model comparison programs** written in the Epsilon Comparison Language (ECL), by pre-filtering and indexing the elements to be matched using static analysis and automatically rewriting the filtered indices and hence reducing the search space (explained in detail in Chapter 7).

1.4 Thesis Structure

Chapter 2 provides an overview of MDE with Section 2.1 introducing models, metamodels and also presenting basic terminology in the MDE world followed by an overview of the current state-of-the-art MDE tools in Section 2.2; Section 2.3 presents the challenges in terms of scalability particularly discussing scalable querying in detail in Section 2.4 and also reviews various widely used optimisations in the field of databases in Section 2.5 .

Chapter 3 states the analysis and hypothesis of this research. It starts with the Section 3.1 presenting the analysis of the literature review presented in Chapter 2 and then the research objectives and questions are presented. Finally the research hypothesis is stated followed by the research scope.

Chapter 4 provides details about the static analyser which is the underpinning tool used for automatic program rewriting. Section 4.1 presents the overall architecture of the static analysis. It provides a detailed description of various components in Section 4.2, 4.4 and 4.5. Section 4.6 discusses the static analysis for EOL, then EVL static analysis is detailed in Section 4.7. ETL and ECL static analysis is described in Sections 4.8 and 4.9 respectively, followed by results of applying the static analysis on Eugenia[2] as a case study in Section 4.10. Finally, the related literature on static analysis is stated in Section 4.11.

Chapter 5 presents the first contribution, discussing the optimisation of queries operating over EMF models. First it presents the motivating example in Section 5.1 and then Section 5.2 describes an approach to efficiently query EMF models by optimising type-level model queries. This is done using static analysis followed by automated custom indices. This approach is extended to provide incremental querying using the EOL to Viatra translation in Section 5.3. Finally it presents a case study used for experiments to evaluate the performance gain by the proposed approach and finally the results obtained in Section 5.2.4.

Chapter 6 describes a novel approach for selectively tracing model to model transformation programs. Starting from a motivating example in Section 6.1, it presents an overall architecture of the proposed approach and then describes all the components step by step in Section 6.2. Finally, Section 6.3 presents the evaluation of the selective traceability approach by presenting the results of experiments evaluating the execution time and memory consumption and concludes with quantitative comparison to the other state-of-the-art tools.

Chapter 7 presents the final major research contribution of this thesis i.e., an approach for optimisation of model comparison programs. After presenting the motivating example in Section 7.1, Section 7.2 discusses the approach for an efficient comparison of two models by automatically rewriting expressions to filter the elements before comparing them. Finally it presents the experimental setup and results in Section 7.3.

Lastly, **Chapter 8** concludes the thesis and provides a summary of all the knowledge gained through this work in Section 8.1 and concludes with identifying the contributions to the field of model-driven software engineering in Section 8.2. Additionally, Section 8.3 pins down possible future directions for the research and suggests areas for further exploration.

[2]https://eclipse.dev/epsilon/doc/eugenia/

Chapter 2

Background

This chapter presents a broader context for this research. It starts with an overview of the model-driven engineering field and then discusses some core terminology. It also provides an overview of various popular MDE tools being used in the industry nowadays, detailing the ones that are used in this research. Then, some areas of active research in MDE are presented. Finally, state-of-the-art model querying approaches and their limitations are presented.

2.1 Model Driven Engineering

MDE is a software engineering approach, where models are the first-class artefacts of the software engineering process.. In general, an MDE methodology consists of the following steps:

- *Concepts:* Concepts build up the methodology and include everything ranging from language artefacts to actors.
- *Notations:* Concepts are represented using notations usually through the introduction of domain languages
- *Process and Rules:* Processes and rules are the tasks that result in the creation of the final product, providing guidelines for their coordination and management, and assurance about the expected quality (such as accuracy, consistency, etc.) of the process or the product.
- *Tools:* Tools are the applications that automate or facilitate various activities of the MDE process and help developers use the concepts represented through the various notations.

According to [22], the Niklaus Wirth programming equation ("Algorithms + Data Structures = Programs") in MDE software engineering process takes the following form

$$Models + Transformations = Software$$

Software in the above equation refers to the code and also the documentation. Models expressed as notations are also referred to as modelling language. Considering this equation, the key components of MDE will be discussed : a) models b) meta-models c) model management tasks. Model management tasks include tasks such as querying, transformation and validation. There are certain MDE tools and languages that allow users to create models and then efficiently manage(query, validate, transform) them. The relevant ones to this work of these tools and languages for model management programs are discussed in this chapter.

2.1.1 Models

A model is an abstract representation [22] defined for a real-world system or a phenomenon for a specific purpose. Models often omit concrete details because they are not meant to capture the entire system or phenomenon. In terms of the systems that model represent, they can be either descriptive or prescriptive.

2.1.1.1 Descriptive Models

Descriptive models are the ones used to describe the phenomena that already exist. It is usually done to explain/understand a natural phenomenon or legacy system. These models are often extracted by observation, experimentation and reverse engineering. A few examples of descriptive models are solar system models, historic stock market analysis models etc., These models are used to understand how these systems work, to formulate general patterns and sometimes to make predictions.

2.1.1.2 Prescriptive Models as Sketches

Prescriptive models on the contrary are used to present an abstract view of a the system that does not exist yet. They define the system that yet needs to be implemented and are used to discuss and verify the system properties without building the actual system. Prescriptive models in software engineering were originally started with simple sketches to guide a software engineer about the properties of software needed to be built. Prescriptive models are usually the result of informal discussions and do not follow a specific set of rules or guidelines. One such example is shown in Figure 2.1. It shows components and different inputs, outputs and dataflow between them which helps developers understand the required system by the customer.

2.1.1.3 Prescriptive Models as Blueprints

Prescriptive models can also be used as a blueprint and are developed using standard modelling languages (e.g. UML). These models capture domain knowledge and design decisions. They require human interpretation and manual implementation while still following a standard structure the modelling language provides. One such example is a class or sequence UML diagram to represent the static

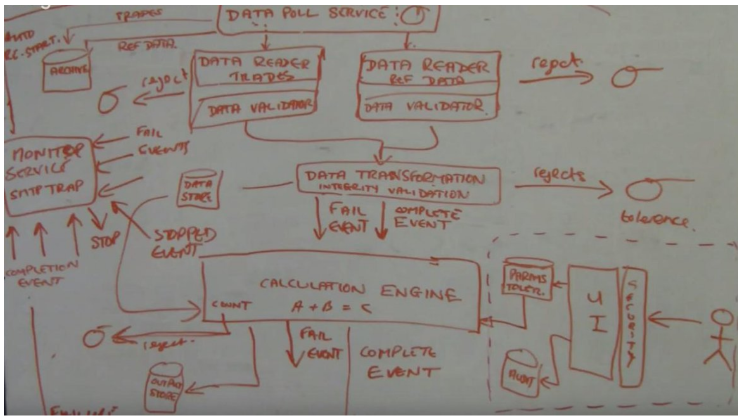


Figure 2.1: An example prescriptive model as a sketch available online

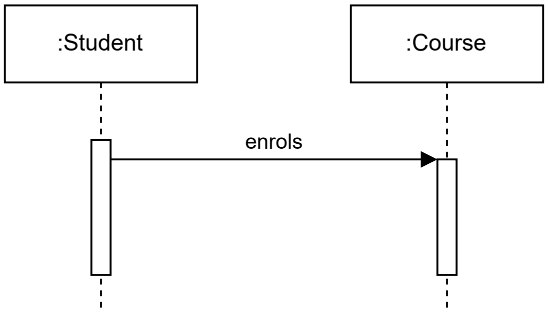


Figure 2.2: An example prescriptive model as a blueprint

and the dynamic behaviour of the software to be developed as shown in Figure 2.2. This diagram presents the sequence diagram showing the enrollment activity where a *Student* enrolls in a *Course*.

2.1.1.4 Prescriptive Models as Programs

Prescriptive models can also be used as programs themselves. They are precise enough to drive automated system implementation/property verification. They are captured in unambiguous (typically domain-specific) languages and machine-processable formats. Also, supported by executors, simulators and code generators. One example is a Simulink model as shown in Figure 2.3. This model takes a sine wave as an input, applies a gain of 2.0 passes it through a saturation block and shows the output on scope. C code can automatically be generated from a Simulink model.

In MDE models are not just sketches or blueprints, models are the primary source that automatically generates other artefacts such as documents

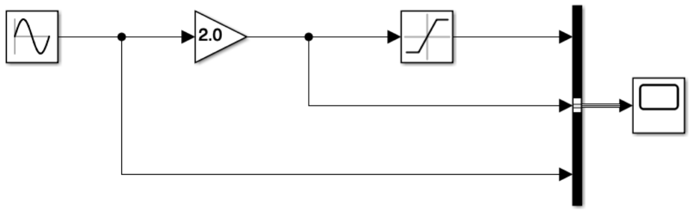


Figure 2.3: An example prescriptive model as a program

and source code. In MDE, domain-specific models are used to automate the development of certain parts of a software system, while the other parts can still be handwritten by the developers as necessary.

Models are represented as instances of another abstract model which is known as a metamodel [22]. Models have a defined abstract representation (structure), metamodel can be defined as another abstract representation by defining properties of elements in a model, their attributes and associations between them. A model is an instance of its metamodel and conforms to its metamodel in the same way as a computer program conforms to the grammar of the programming language that it is written in. Metamodels are used to define DSLs (domain-specific languages-specific languages for a particular domain or context). DSLs are discussed in detail in Section 2.1.3.1. The whole MDE process is described graphically in Figure 2.4, according to which a model conforms to a DSL or a metamodel and then model management programs can generate artefacts such as code, documentation, which along with handwritten code (if required) constitutes the required software.

In MDE, models can be categorised in two categories on the basis of the dimension of the system they represent:

Static models are models that represents static aspects of the systems in terms of architecture and the structure of system.

Dynamic models focus on the behaviour of the system by depicting the sequence of actions and how the different components of the system interact.

Some models can be both static and dynamic, because they model both the dynamic and static aspects of a system. One such example is collaboration diagram that represents the link and nodes(static structure) and also the messages(dynamic behaviour) at the same time.

2.1.2 Metamodeling

Models are defined for different systems and these models are described by yet other models which are known as metamodels. So a model can be considered as an instance of the metamodel. As models are abstractions of a real-world system, metamodel is yet another abstraction defining properties of the model itself. A model needs a set of structural elements and rules that they must follow. These rules can be described either informally (e.g., with natural language statements)

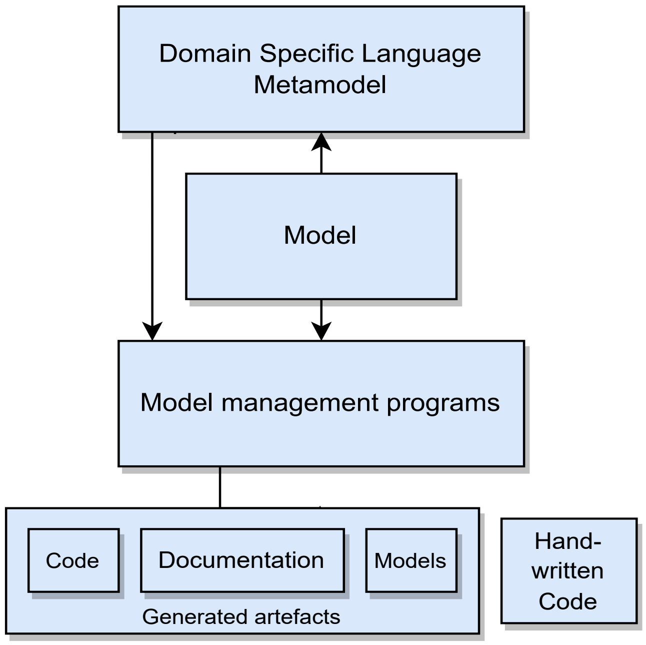


Figure 2.4: An overview of MDE process

or by a structured representation that is called a metamodel (also referred to as modelling language). A simple example of a tree model conforming to tree metamodel is shown in Figure 2.5. In metamodel, *Tree* will have a string label, can have multiple *Trees* as children and can have a parent *Tree*. In model, it can be observed one *Tree* with label t1 having two children Tree t2 and t3. When a model conforms to a metamodel what it means is that all elements of the models are instances of its corresponding metamodel classes. A metamodel can further

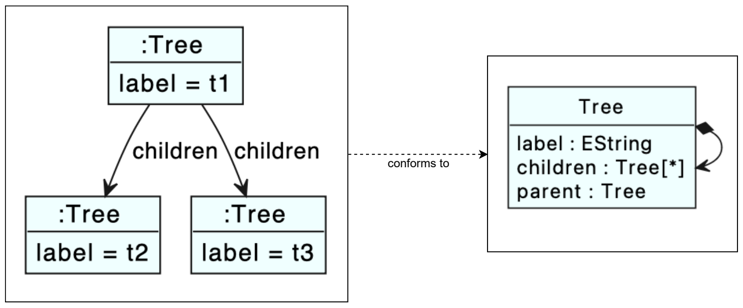


Figure 2.5: Conforms to relationship between model and metamodel

be defined using another subsequent model called a meta-metamodel. In theory one can define an arbitrary number of metamodels but in practice metamodels can be defined on the base of metamodel themselves so it does not makes sense to go beyond this level of abstraction.

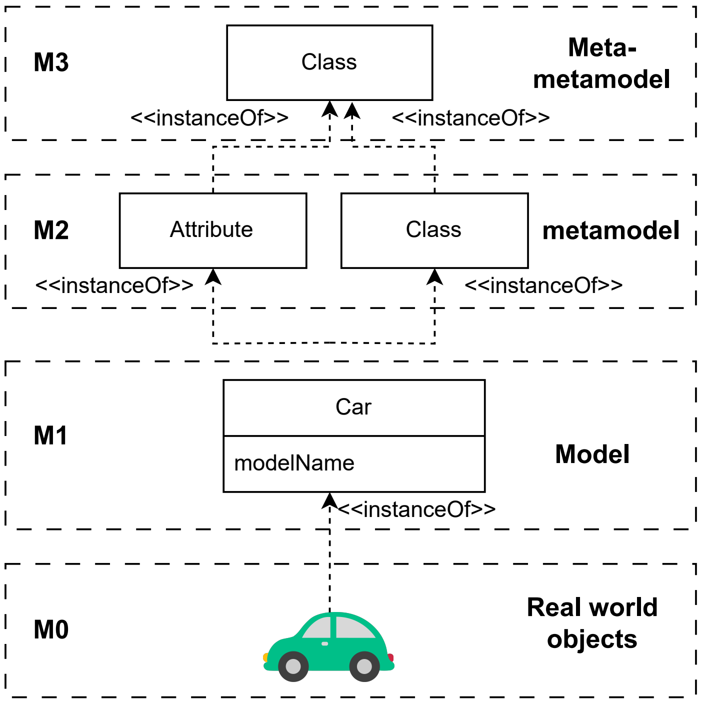


Figure 2.6: Relationship between models, metamodels and meta-metamodels

2.1.3 Modeling Languages

Modeling languages are one of the key components in the MDE world, as they enable modelers specify the models for their respective systems. They can be described in a graphical or a textual representation. These can be broader categorised into:

2.1.3.1 Domain-Specific Language (DSLs)

Domain-Specific Languages refer to modelling languages designed with a specific domain or application in mind. These languages aim to provide specialized syntax, semantics, and constructs that cater to the unique requirements and challenges of a particular field [22].

While DSLs have gained prominence in computer science, they have found extensive usage across diverse domains. In the realm of web development, HTML (Hypertext Markup Language) serves as a prominent DSL that enables the creation of structured web pages and content. By providing a standardized markup syntax, HTML simplifies the process of designing and presenting information on the World Wide Web.

In the field of databases, SQL (Structured Query Language) has emerged as the de facto DSL for interacting with relational database management systems (RDBMS). SQL offers a concise and expressive syntax specifically tailored for querying, modifying, and managing data stored in structured databases. Its domain-specific nature allows developers to efficiently work with databases, retrieve information, and perform complex operations with ease.

2.1.3.2 General Purpose Modeling Languages (GPMLs)

GPMLs on the other hand are the modelling languages that can be used for modelling in a wider domain. They offer a general purpose frameworks for modelling systems from various domains. For instance, there are a large number of off-the-shelf modelling languages. Even though they are broad, they are still scoped to be largely used within one larger software domain, some of them are:

- UML [23] for object-oriented systems
- Simulink by MATLAB [24] mainly used for control systems
- Archimate [25] for enterprise architecture
- BPMN [26] for business modelling.

2.1.4 Model Management

Software engineers develop metamodels and create models conforming to them. In MDE, a common goal is to generate other artefacts from models. Another reason to use MDE is to be able to do analysis and verification of properties of the system. MDE Equation [27] states transformation as the core process. Research in [28] has shown that other tasks such as validation, comparison, merging etc. are of equal importance and are required often for MDE processes. These, in general, are commonly referred to as “model management tasks” [29].

2.2 MDE Technologies

MDE is extensively used in industries where high integrity and reliability software is required [30–33] because software errors can lead to very expensive product recalls,loss of life etc. It is also heavily used in the automotive and aerospace sector. Rolls-Royce [10] is using model-driven engineering to support future generations of control and monitoring system for aerospace applications. Thales is funding a large open-source model-driven systems engineering toolkit (Capella) [34]. AUTOSAR [35] initiative in the automotive industry. The EU project Lowcomote that this research is funded by [36] itself has nine industry partners in its consortium and which are using MDE for developing their products.

MDE tools include:

- IBM Rhapsody [37] - A tool used by system engineers to create embedded systems
- MagicDraw [38] - A software modelling tool with teamwork support
- PTC Integrity Modeller [39]
- ANSYS Scade [40] - A model-driven engineering environment for critical embedded software, which provides requirements management, model-based design, verification, code generation, and interoperability with other development platforms
- JetBrains MPS [41] - A tool by Jetbrains for designing domain-specific languages

There is plenty of tooling available for performing different tasks of MDE process. The ones that are used in the context of this research are detailed. The section below discusses the widely-used modelling framework EMF and then Epsilon will be discussed which is a family of model management languages. Finally, it is concluded with the discussion about VIATRA which is a model querying, validation and framework.

2.2.1 The Eclipse Modeling Framework (EMF)

EMF [1] is the most popular metamodelling architecture. It is an Eclipse-based robust and open-source modelling framework. EMF, being the basis of Eclipse modelling eco-system is widely used in many open-source projects such as ATL [42], VIATRA [43] , Epsilon, Acceleo [17]. EMF provides a modelling language Ecore, which is partially based on the MOF 2.0 standard [44].

All EMF models are built using Ecore language. The top-level object in an Ecore metamodel is an *EPackage* as shown in Figure 2.7, a metamodel is expressed as an *EPackage. EPackage* defines a unique identifier (nsURI) which is used as a unique key identifier for retrieving metamodel from *EPackageRegistry. EPackageRegistry* stores a map of identifiers of metamodels. Each package can contain *EClasses* and *EDataTypes* inherited from *EClassifier. EClasses* have *EStructuralFeatures* which can be either *EAttributes* or *EReferences.*

EMF, by default seralises models in XML[1]-based representation called as XML Metadata Interchange (XMI[2]). This seralisation format allows to save a model in multiple physical files and use lazy references to link them together and thus allowing several optimisations. There are different editors other than

[1]https://www.omg.org/spec/XML/
[2]https://www.omg.org/spec/XMI/

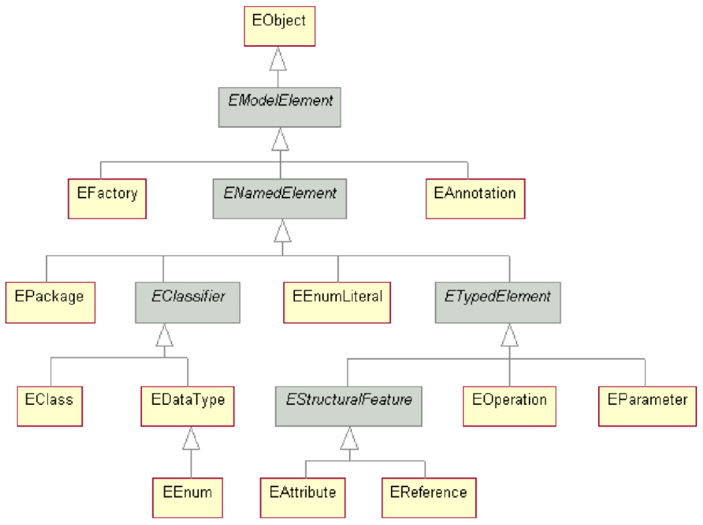


Figure 2.7: Ecore modelling language [1]

default tree-based (hierarchical structure) for writing Ecore models such as Emfatic [45] (textual language) or Sirius diagram based graphical editor for Ecore. Sirius [46] is an Eclipse project which allows one to easily create their own graphical modeling workbench by leveraging the Eclipse Modeling technologies, including EMF and GMF.

2.2.2 Epsilon

The Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon) [47] is an Eclipse-based platform for MDE processes. It includes languages used to facilitate the automation of different model management operations. Epsilon is divided into two main parts a) Task-Specific languages and b) Technology specific drivers through Epsilon Model Connectivity Layer(EMC). Epsilon is a model agnostic technology, while a lot of MDE platforms are often bound to a specific underlying modelling technology. Epsilon seamlessly accesses and manages several underlying model persistence technologies (EMF, MySQL, Spreadsheets etc.). These technologies are supported by Epsilon through several EMC drivers, EMC layer is extensible, any new models defined/persisted in other technology can be implemented and added to the EMC layer as a driver. The architecture of Epsilon is shown in Figure 2.8.

Task-specific part of Epsilon contains several languages each for specific model management tasks. EOL is the base language of Epsilon while other languages are built on the top of EOL. EOL is quite similar to OCL [16] (is one of a well-known and frequently-used declarative language for querying and validation of models) in syntax with additional features such as access to multi-

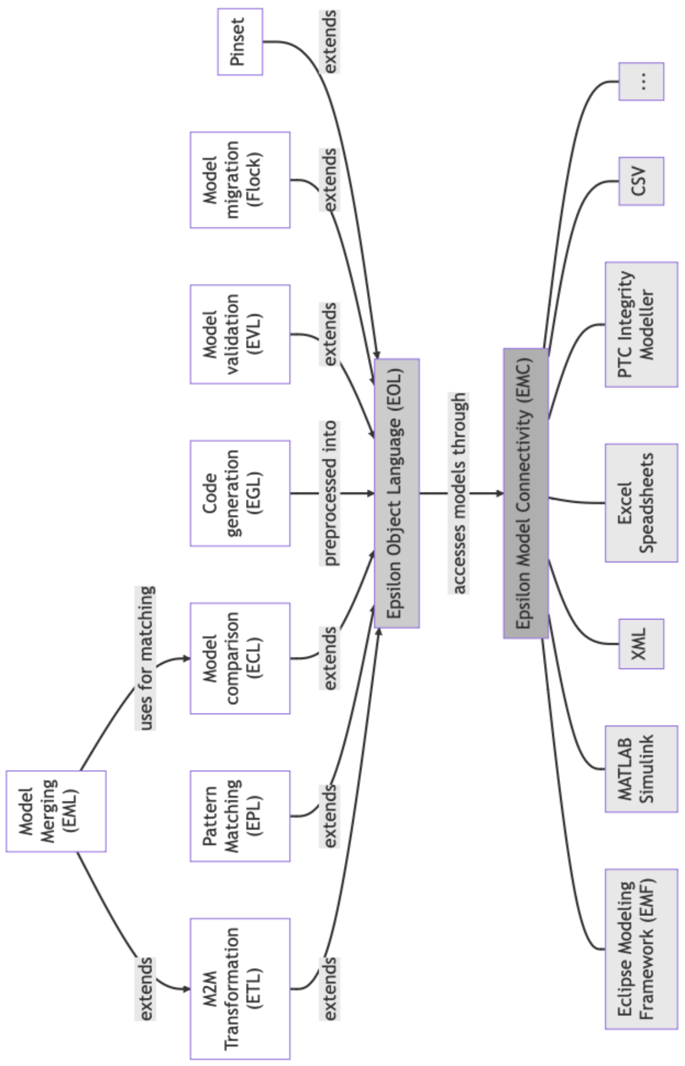


Figure 2.8: An overview of Epsilon architecture

ple models (through EMC). Other languages can be easily added to Epsilon by reusing and extending EOL to remain consistent. Some common model management operations and the corresponding languages in Epsilon are mentioned as follows. A few languages will be discussed in detail later in this section that have been used in this research.

- **Model Querying:** Querying is used to extract information of interest from the model can be done using Epsilon Object Language (EOL) [48]. EOL provides imperative constructs to query models and is inspired by OCL. EOL is explained in more detail in Section 2.2.2.2.

- **Model Validation:** Validation is often used to evaluate certain constraints on models to identify errors and inconsistencies. Models can be validated using Epsilon Validation Language (EVL) [49]. EVL is explained in more detail in Section 2.2.2.3.

- **Model to Model Transformation (M2M):** Model-to-model transformation is a primary task in an MDE workflow and it transforms one or more input model(s) to one or more output model(s). Models can be transformed to other models using Epsilon Transformation Language (ETL) [50]. ETL is explained in more detail in Section 2.2.2.4.

- **Model to Text Generation (M2T):** Code generation is one of the main appeal of MDE. So M2T is used to generate textual artefacts such as code, documentation etc. Code can be generated from models using Epsilon Generation Language (EGL). [51]

- **Model Merging:** Model merging is an activity to combine multiple models into a single coherent one. Models can be merged using Epsilon Merging Language (EML) [52]. Models are merged by first establishing correspondences using ECL which is explained below.

- **Model Comparison:** Model comparison is an activity to identify matching elements in models. Model comparison is usually a prerequisite for activities like model merging and model transformation testing. Models can be compared using Epsilon Comparison Language (ECL) [53]. ECL is explained in more detail in Section 2.2.2.5.

- **Pattern Matching:** Pattern matching is a model management activity to match models conforming to same or different metamodels based on the elements' characteristics and relations. Epsilon Pattern Language (EPL) is used to contribute pattern matching capabilities to Epsilon. [54]

- **Model Migration:** Model migration is the process of updating models as the corresponding metamodel changes or evolves. Models can be migrated to newer versions of models conforming to the new metamodel using Epsilon Flock. [55]

- **Model Generation:** There is often a need to generate models of different sizes and structure that exercise specific features of a program for testing and benchmarking. Models can be generated using Epsilon Model Generation Language (EMG). [56]
- **Unit Testing:** Epsilon also provides a tailored testing framework to test programs written in Epsilon languages, based on EOL and Ant workflow tasks. [57]
- **Dataset Extraction:** Epsilon provides a Pinset language that provides table-like datasets extraction facilities from models. [58]
- **Update Transformation:** Using update transformations, in place modifications are done on the source model itself. Update transformations can be performed using Epsilon Wizard Language(EWL). [59]

Epsilon platform is considered for this research project for a number of reasons:

- Epsilon **supports a number of model persistence formats** and can even be extended to work with unsupported technologies using EMC layer. In this research the only focus is on EMF-based models, but with some extra effort this work can be extended to various other modelling platforms.
- Second, is that all languages of Epsilon extend a common **core language** - EOL. This common core language makes it easier to reuse the optimisations to be used across other languages because other Epsilon languages use common EOL expressions.
- Epsilon is **easily extensible** in various aspects such as adding support for a new modelling technology by extending the EMC layer with a new technology driver or implementing a new model management language on top of EOL.

2.2.2.1 Model Connectivity Layer

Epsilon has its model connectivity layer through which model management programs can access models from different modelling technologies such as EMF, Simulink in a uniform way. EMC provides an *IModel* interface that consists of a number of methods to query and manipulate the model at a higher abstraction level than the underlying persistence technology. A graphical view of different classes of EMC layer is shown in Figure 2.9.

Each driver present in Epsilon implements the *IModel* interface and to add language specific support to load models and retrieve model elements based on the required properties. *IModel* interface specifies properties such as name and alias of the model. *IModel* also provides methods for loading and storing models, type-related methods such as filtering elements of a specific kind and

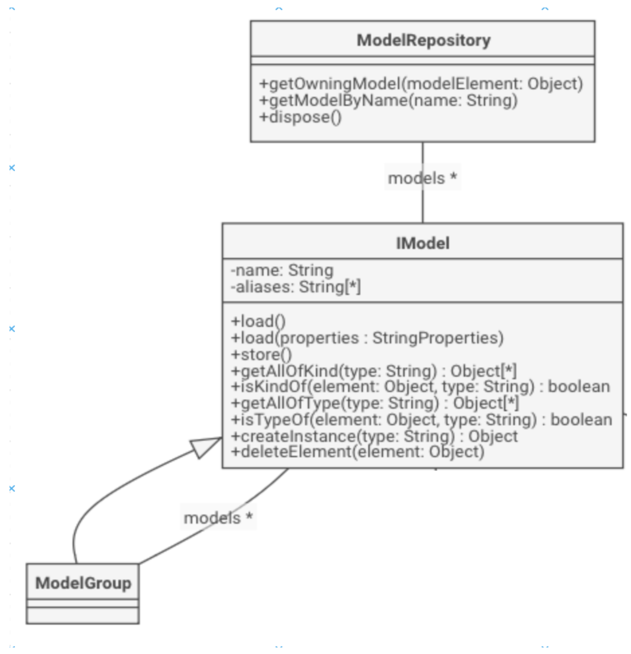

ModelRepository
+getOwningModel(modelElement: Object)
+getModelByName(name: String)
+dispose()
models *
IModel
-name: String
-aliases: String[*]
+load()
+load(properties : StringProperties)
+store()
+getAllOfKind(type: String) : Object[*]
+isKindOf(element: Object, type: String) : boolean
+getAllOfType(type: String) : Object[*]
+isTypeOf(element: Object, type: String) : boolean
+createInstance(type: String) : Object
+deleteElement(element: Object)
models *
ModelGroup

Figure 2.9: Overview of EMC Layer [2]

methods and for creation and modification of model elements. In the context of any model management activity, *ModelRepository* class acts as a container for the models. *ModelGroup* is a group of same alias models. A model group is formed when there are two or more models of the same aliases. This is to perform aggregate operations on the models sharing common alias.

EMC provides a list of drivers to use models of various technologies to be manipulated using Epsilon languages. The most popular ones are EMF, MATLAB/Simulink, Cameo/MagicDraw Systems Modeler, XML/CSV/Excel and Epsilon Hawk.

2.2.2.2 Epsilon Object Language

EOL[3] is the core language of Epsilon and all other languages extend EOL, using the common expressions to access model elements and their properties. EOL is inspired by OCL and has a very similar syntax. EOL can also be used as a general-purpose model management language as, unlike OCL that is side-effect free, it is able to change the underlying models it accesses, if required.

Two example EOL programs is depicted in Listing 2.1. An EOL program can have a main statement block (Line 13) and user defined operations both in the same file and imported files (Lines 4-6). An EOL program can also import other EOL files (Line 12). The *Operation* in file *squareUtil.eol* takes an *Integer*

[3]https://www.eclipse.org/epsilon/doc/eol/

argument and returns an *Integer*. Then in the file *example.eol* the operation *square()* is called with an argument and the resultant *Integer* is printed on the console using a built-in *println()* method.

EOL has a built-in type system which includes Primitive types (String, Integer, Real and Boolean), model element type, collection types (*Bag*(non-unique, unordered collections), *Sequence*(non-unique, ordered collections), *Set*(unique, unordered collections), *OrderedSet*(unique, ordered collections)) and map type. All these type extend *Any* type, which is similar to *OclAny* type. The types in Epsilon are discussed in more detail in Section 4.3.

```
/*-----------------------------------------------------
        file: squareUtil.eol
-----------------------------------------------------*/
operation square(num: Integer) : Integer {
  return num*num;
}

/*-----------------------------------------------------
    file: example.eol
-----------------------------------------------------*/

import "squareUtil.eol";
square(4).println("4*4 = "); // main body

// more user defined operations could be placed here
```
Listing 2.1: An example EOL program

2.2.2.3 Epsilon Validation Language

EVL[4], contributes model validation capabilities to Epsilon. EVL can be used to validate a single model or also for cross validating a number of models. The models, as in other Epsilon languages, can be of various backend technologies. The concrete syntax of EVL is shown in Listing 2.2. It demonstrates the concrete syntax of the context, constraint and fix abstract syntax constructs discussed below.

```
(@lazy)?
context <name> {
  (guard (:expression)|({statementBlock}))?
  (constraint)*
}

((@lazy)?
(constraint|critique) <name> {
  (guard (:expression)|({statementBlock}))?
```

[4]https://www.eclipse.org/epsilon/doc/evl/

```
  (check (:expression)|({statementBlock}))?
  (message (:expression)|({statementBlock}))?
  (fix)*
}

fix {
  (guard (:expression)|({statementBlock}))?
  (title (:expression)|({statementBlock}))
  do {
    statementBlock
  }
}
```

Listing 2.2: Concrete syntax of EVL

An EVL program depicted in Listing 2.3 for validating a movie model conforming to the IMDB metamodel, an excerpt of which is shown in Figure 2.10. IMDB metamodel is a movie database proposed in the Transformation Tool Context (TTC) 2014 [60]. An EVL program can contain a number of constraints(Line 9, 18, 22) . Constraints can be marked as *lazy* or *non-lazy*, a lazy constraint will only be executed when invoked by *satisfies* operation to avoid unnecessary duplication. *satisfies(constraint: String) :Boolean*, *satisfiesAll(constraints: Sequence⟨String⟩)* and *satisfiesOne(constraints: Sequence ⟨String⟩): Boolean* are the built-in operations provided by EVL. These operations can be used in a constraints guard to specify other constraints which need to be satisfied. Constraints are grouped by the context (Line 7, 15) they apply to. An EVL program can also have pre (Line 1-5) and post block (Line 31-35) to be executed before and after the execution of constraints respectively. A constraint can have a guard block to filter which model elements of context, the constraint is applied to. Every constraint has a check block (Line 11) which is the actual condition to be evaluated. Moreover, a constraint can have a fix block that specifies a quick repair if the constraint's check condition is not satisfied.

In the example in Listing 2.3, it validates a model conforming to IMDB metamodel in Figure 2.10. It has *pre* block which computes the number of movies (Line 2), number of actors (Line 3) and the average number of actors per movie (Line 4). These three values are stored in global variables. The first constraint (Line 9-12) checks such movies where the number of actors are more than the average actors have valid actor names. The constraint *IsValid* (Line 18-20) checks if the name of the person using the helper function *isPlain()* returns true. The last constraint *ValidMovieYears* (Line 22-24) validates *Person* (Line 15). This constraint ensures that all movies of an actor are released 3 years after the actors birth year. Finally, *post* block will print the global variable values.

```
1 pre {
2   var numMovies = Movie.all.size();
3   var numActors = Person.all.size();
4   var apm = numActors / numMovies;
```

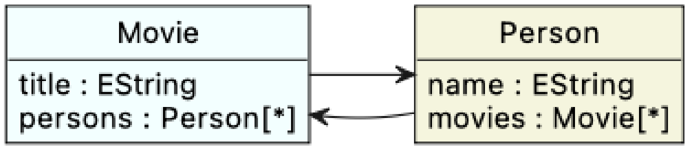


Figure 2.10: An Excerpt of the IMDB metamodel

```
}

context Movie {

constraint ValidActors {
  guard:self.persons.size() > apm
  check:self.persons.forAll(p|p.satisfies("IsValid"))
  }
}

context Person {

@lazy
constraint IsValid {
  check:self.name.isPlain()
}

constraint ValidMovieYears {
  check:self.movies.forAll(m| m.year+1>self.birthYear)
}
}

operation String isPlain() : Boolean {
  return self.matches("[A-Za-z\\s]+");
}

post {
  ("Actors per Movie="+apm).println();
  ("# Movies="+numMovies).println();
  ("# Actors="+numActors).println();
}
```

Listing 2.3: An example EVL program

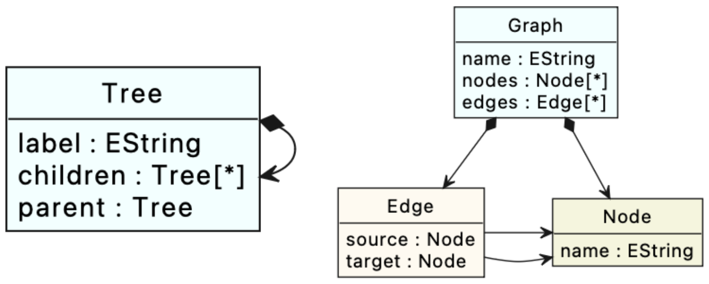


Figure 2.11: Tree and Graph Metamodel

2.2.2.4 Epsilon Transformation Language

ETL[5] is a hybrid rule-based language for model-to-model transformation in Epsilon. A hybrid M2M transformation language has both imperative and declarative constructs. The concrete syntax of ETL is shown in Listing 2.4.

An ETL program (module) takes as input a number of source models and transforms them into a number of target models. An ETL program is depicted in Listing 2.5 for transforming tree model to graph model for which the metamodel is shown in Figure 2.11. The models, as in other Epsilon languages, can be of heterogeneous modelling technologies (e.g., an EMF model can be transformed to a Simulink model or an XML document can be transformed to an Excel spreadsheet). An ETL module can contain a number of transformation rules (Line 5-17), transforming source model elements to one or more target model elements.

```
1 (@abstract)?
2 (@lazy)?
3 (@primary)?
4 rule <name>
5   transform <sourceParameterName>:<sourceParameterType
      >
6   to <targetParameterName>:<targetParameterType>
7     (,<targetParameterName>:<targetParameterType>)*
8   (extends <ruleName> (, <ruleName>*)? {
9 
10   (guard (:expression)|({statementBlock}))?
11 
12   statement+
13 }
14 
15 (pre|post) <name> {
16   statement+
```

[5]https://www.eclipse.org/epsilon/doc/etl/

```
17 }
```

Listing 2.4: Concrete syntax of ETL

An ETL module can optionally have a *pre* (Line 1-3) and a *post*(Line 18-20) block of statements, to be executed before and after the execution of transformation rules respectively. A transformation rule can extend one or more other transformation rules and can be declared as *abstract* or *lazy* through relevant annotations:

- An **abstract rule** must be extended by another transformation rule. Such rules cannot be invoked standalone, they get invoked only when the rule that extends them is invoked.
- A **primary rule** will always get executed automatically.
- A **lazy rule** will get executed only when it is required by another transformation rule.

```
1  pre {
2    // to be executed before the execution of
       constraints
3  }
4
5  rule Tree2Node
6  transform t : Tree!Tree
7  to n : Graph!Node {
8
9    n.label = t.label;
10
11   if (t.parent.isDefined()) {
12     var edge = new Graph!Edge;
13     edge.source = n;
14     edge.target = t.parent.equivalent();
15   }
16 }
17
18 post {
19   // to be executed after the execution of constraints
20 }
```

Listing 2.5: An example ETL program

In a model-to-model transformation, resolving elements created by other transformation rules is quite a common and recurring task. For this resolution ETL provides the *equivalent()/equivalents()* operations. The elements returned by these operations follow the respective order of the rules that have created them. An exception to this occurs when one of the rules is declared as *primary*, in which case its results precede the results of all other rules.

2.2.2.5 Epsilon Comparison Language

The Epsilon Comparison Language (ECL)[6] is a hybrid rule-based dedicated model comparison language, provided by the Epsilon framework. ECL lets developers specify custom comparison algorithms in a rule-based script to identify matching elements between homogeneous and heterogeneous models. The concrete syntax of ECL is shown in Listing 2.6.

An example ECL program is depicted in Listing 2.7. An ECL program contains a number of *MatchRules*(Line 5-12) and optional pre(Line 1-3) and post-block(s) (Line 14-16) executing before and after the rules respectively.

```
1  (@lazy)?
2  (@greedy)?
3  (@abstract)?
4  rule <name>
5    match <leftParameterName>:<leftParameterType>
6    with <rightParameterName>:<rightParameterType>
7      (extends <ruleName>(, <ruleName>)*)? {
8
9    (guard (:expression)|({statementBlock}))?
10   compare (:expression)|({statementBlock})
11   (do {statementBlock})?
12 }
13
14
15 (pre|post) <name> {
16   statement+
17 }
```

Listing 2.6: Concrete syntax of ECL

A *MatchRule* enables developers to create their comparison logic between model elements at a high level of abstraction. *MatchRules* consist of a declared name along with two parameters (left (Line 6) and right (Line 7)) to specify the types of elements they can compare. A *MatchRule* can also optionally extend a number of match rules and can be labelled as *abstract*, *lazy* (Line 12) and/or *greedy* using corresponding annotations.

- An **abstract match rule** cannot be invoked "individually", they are invoked when a rule(s) that extends them is executed.
- A **lazy match rule** will get executed only when it is required by another *MatchRule*, using the *matches* operation.
- A **greedy match rule** is executed for *all* pairs that have a kind-of relationship with the types specified by the left and the right parameters of the *MatchRule*.

[6]https://www.eclipse.org/epsilon/doc/ecl/

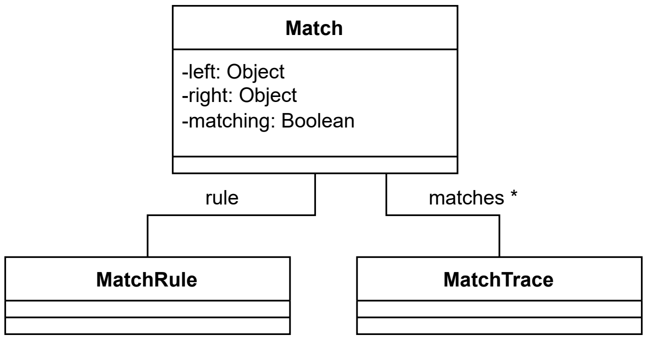


Figure 2.12: Structure of *MatchTrace* of ECL

```
pre {
  // to be executed before the execution of
      constraints
}

rule Movie2Movie
match l : imdb1!Movie
with r : imdb2!Movie {
  compare : l.title= r.title and
  l.persons.matches(r.persons)
}

@lazy
rule Person2Person
match l : imdb1!Person
with r : imdb2!Person {
  compare : l.name = r.name;
}

post {
  // to be executed after the execution of constraints
}
```

Listing 2.7: An example ECL program

The result of an ECL comparison program is a *MatchTrace* after comparing two models. The structure of *MatchTrace* is illustrated in Figure 2.12. *MatchTrace* contains a number of *Match*es. Each match has a reference to the comparing elements from each model, left and right. It also contains a boolean flag indicating the result of matching two elements.

2.2.3 Viatra

A brief introduction to Viatra is presented in this section, as it provides scalable querying over EMF-based models. The proposed query optimisation approach is integrated with Viatra that will be presented in detail in Chapter 5, that is why Viatra is introduced here in this Section. Viatra is an open-source model query, validation and transformation framework supporting the efficient evaluation of model queries on top of EMF models [20]. The core language of Viatra is the Viatra Query Language (VQL), which allows the definition of model queries as incremental graph patterns. A graph pattern is a graph-like structure consisting of conditions (nodes and edges) to be matched against a large instance model. Viatra provides two engines to evaluate the graph patterns on EMF models, local search and incremental. An example graph pattern for quering imdb model conforming to the metamodel shown in Figure 2.10 is depicted in Listing 2.8. Movie is a model class, title is a model feature of the class Movie. The parameter of the pattern is type of *Movie*. Line 2 searches for a movie with title "Spider Man".

```
pattern find(movies: Movie) {
  Movie.title(movies, "Spider Man");
}
```

Listing 2.8: An example VQL program

Consider the pattern is executed in Listing 2.8 over an imdb model named *SmallMovieSet* containing three movies i.e., Super Man, Spider Man, Titanic. *Matcher* provides the API to retrieve the results of the query by the following API calls.

- *hasMatch* - returns true if there is one or more matches. *hasMatch* will return true when Listing 2.8 is evaluated over *SmallMovieSet*.
- *OneArbitraryMatch* - returns one arbitrary matched element. *OneArbitraryMatch* will return *Movie*, Spider Man when Listing 2.8 is evaluated over *SmallMovieSet*.
- *allMatch* - returns all matches. *allMatch* will return Super Man, Spider Man, Titanic when Listing 2.8 is evaluated over *SmallMovieSet*.
- *countMatch* - returns the number of matches. *countMatch* will return 3 when Listing 2.8 is evaluated over *SmallMovieSet*.

2.2.3.1 Local Search Engine

The first execution engine supported by Viatra is the local search engine that employs efficient search plans to compute and collect pattern matches [61]. Both the local search engine and the incremental engine leverage the advantages provided by a base index. This base index serves as a cache, storing the base relations (e.g., all Movies in IMDB model) and objects present in the model, categorized by their types. By organizing and indexing the data in this manner, the search engines gain several benefits [62]. Firstly, the base index enhances the search engines' performance by providing quick access to the relevant data. As the engines execute search operations, they can consult the base index to efficiently navigate through the model and retrieve the required information. The search engines can save time and computational resources by avoiding the need to scan the entire dataset repeatedly.

2.2.3.2 Incremental Engine

The second one is the incremental engine implementing the RETE algorithm [63]. The incremental engine's primary advantage lies in its ability to cache the computed pattern matches. By storing these matches, the engine avoids re-evaluating the entire model each time a search or query is performed. Instead, it can incrementally update the cached matches based on incremental changes in the data. This incremental re-evaluation significantly improves the engine's performance and responsiveness, especially in scenarios where the model is subject to frequent updates or modifications.

2.3 Scalability Challenges in MDE

As MDE is increasingly applied to larger and more complex systems, the efficiency and performance of development, in this case, is now a critical dimension of research. Research suggests that MDE improves productivity [64], but certain challenges still need to be addressed in the area. Although in present days MDE is increasingly used, still there are a lot of significant research challenges [21] in MDE such as traceability, analysis, maintainability and scalability. In this section, the various scalability challenges will be discussed. Scalability in software engineering has a different dimension such as a large number of software developers, a large number of engineering tasks compared to MDE. To achieve scalability in modelling, the goal is to be able to create large models and domain-specific languages, being able to work on large models in a collaborative manner, to query those models efficiently and to store and index them effectively. Thus, scalability challenges in MDE are categorised as follows:

2.3.1 Scalable DSLs

Although MDE tools and technologies have been evolving a lot, still some models do not scale well, specially when it comes to some large-scale models and in particular when combining models from different domains. Therefore, having scalable DSLs means to be able to design and create a domain-specific language for large models in an efficient way. When big models are anticipated, appropriate strategies should be offered to extend the meta-model structure in order to improve the efficiency for certain model queries as shown in [13, 65].

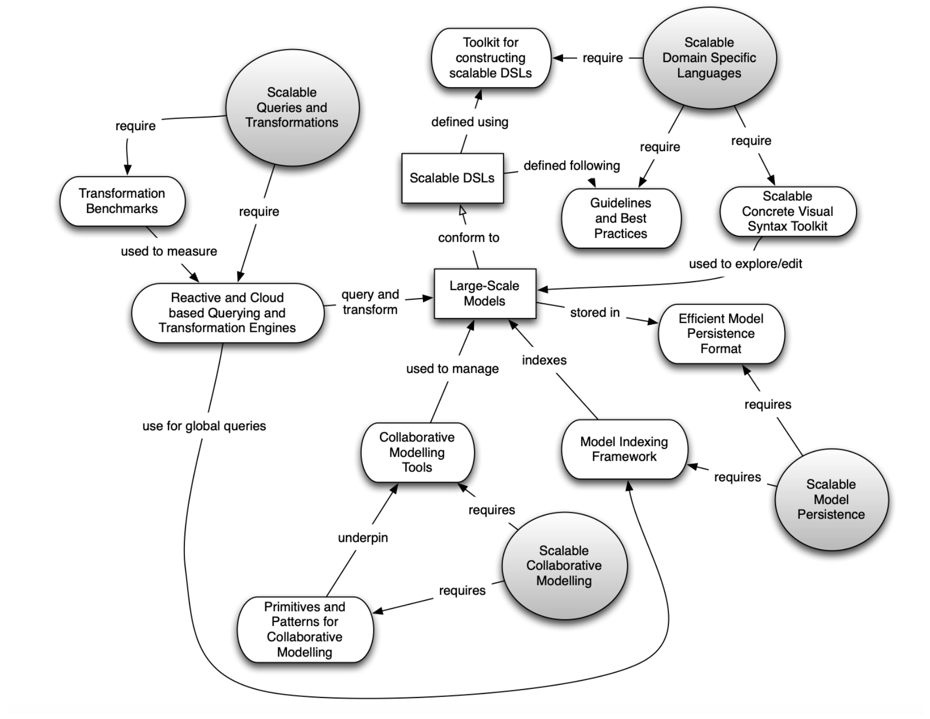

Toolkit for constructing scalable DSLs
require
Scalable Domain Specific Languages
Scalable Queries and Transformations
require
defined using
Scalable DSLs
defined following
require
require
Transformation Benchmarks
require
Guidelines and Best Practices
Scalable Concrete Visual Syntax Toolkit
conform to
used to measure
used to explore/edit
Reactive and Cloud based Querying and Transformation Engines
query and transform
Large-Scale Models
stored in
Efficient Model Persistence Format
used to manage
indexes
requires
use for global queries
Collaborative Modelling Tools
Model Indexing Framework
requires
Scalable Model Persistence
requires
underpin
Scalable Collaborative Modelling
requires
Primitives and Patterns for Collaborative Modelling

Figure 2.13: Challenge of Scalability in MDE [3]

2.3.2 Scalable Persistence

Another goal of scalability in MDE is to be able to store and load large models in an effective way reducing memory footprint. Connected Data Objects (CDO) [66] is a framework to persist EMF-based models in relational databases. CDO implements its own version control system which significantly hinders industrial adoption. CDO does not scale well when it comes to models larger than a few hundred MBs. Other related work such as Mongo EMF [67] and Morsa [68] exercise No SQL database for persisting models. Morsa is able to accomplish scalable model persistence by using on-demand loading facilities that can collect and update model pieces as needed. The most commonly used persistence format is XMI. Often XMI files are very large as XMI includes the verbosity of XML format. To address this issue there are many binary formats which scale very well, in terms of file size.

Models are often required to be version controlled. It is a challenge to query the models stored in large files especially when there are frequent fetches from remote to local repository and back. Storing models in one single file also results in frequent conflicts. Two approaches [69] for this problem are: 1) Storing models in dedicated model repositories to enable model-level version control like CDO [66] and MORSA [68] but this introduces different issues, such as new technology needing to be adopted and pushback from industry to change current practices. 2) Splitting models into many small fragments but this poses issues like global querying for which frameworks like Hawk [5] are used to index and query large models in a more efficient way.

2.3.3 Scalable Querying and Transformation

The goal of scalable querying and transformation is to be able to execute computationally expensive queries and complex transformations in an efficient way. The research directions in scalable querying are i)incrementality - to compute model changes using differencing tools and then propagating those changes incrementally. Incrementality has been implemented in VIATRA by [70] using RETE algorithm. RETE engine focuses on incremental pattern matching for efficient execution of model transformations - ii) Lazy computation - on-demand computations can result in performance improvements to get rid of extra computations. VIATRA [20] does evaluate the matches lazily of connected nodes to reduce unnecessary computation iii) Performance optimisation - Other optimisation [71, 72] such as caching of the results and lazy navigation of the underlying model has also been used as a means to improve the performance of transformations.

2.3.4 Scalable Collaboration

As models grow in size and complexity it is necessary to enable collaboration between multiple users. To enable scalable MDE, it is important to ensure that the collaboration also scales well. This means to enable different modellers and

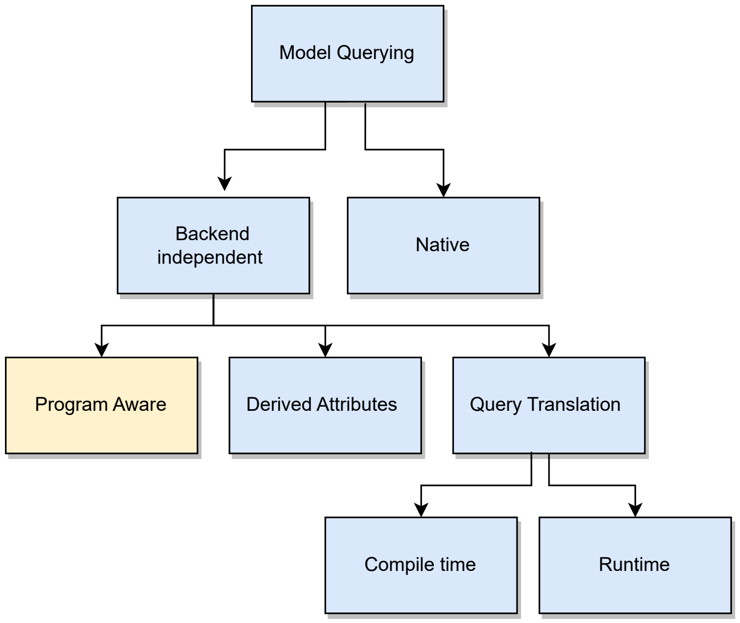


Figure 2.14: Taxonomy of model querying approaches

developers to work on large shared models as a team. In the current era, as the web-based tools are getting more and more popular, there are tools such as Eclipse Che[7] for provisioning workspaces in a cloud-based environment. Using Eclipse Che, for cloud-based modelling application makes it easy to collaborate on models.

2.4 Model Querying

This dissertation deals with tackling the scalable querying challenge. This section will focus on how querying large and complex models can be a challenge and also present state-of-the-art approaches related to model querying. Moreover, it will present a critical analysis of these approaches. Figure 2.14 presents the taxonomy of model querying approaches. There are two principal approaches to query models. i.e. querying models using native query languages of the underlying technology the model is persisted in, and the other approach is querying models using high-level languages. Both will be discussed in detail.

2.4.1 Native Querying

Native querying means querying in the native language of the underlying modeling technology. Native querying being the most straightforward approach, is

[7]https://eclipse.dev/che/

also very efficient as it is specially tailored for the back-end persistence technology. As already mentioned, models can be stored in multiple backend formats e.g. Relational databases, Simulink, Document stores etc. Depending on the model's persistence format native querying can be done in two ways: 1) Native API of the model backend is used to query e.g if a model is stored in a relational database, SQL would be used as a query language and Cypher or Gremlin would be used if it's a NoSQL database. 2) Using direct API calls from programming languages. The most prominent advantage is its efficiency as native query languages have build-in index-based methods, but it also has some drawbacks [73] which are equally important to be considered:

- Query Conciseness: Native queries can sometimes be wordy, difficult to understand
- Query Abstraction Level: Native queries are technology-specific that is if back-end technology is changed, it requires considerable effort to change queries.

Pagan et. al [74] propose a specified efficient query language- MorsaQL (Morsa Query Language) for MORSA repository [75] – a model persistence format for storing large models based on NoSQL database storing large models in NoSQL databases. The design of MorsaQL is based on SQL SELECT – FROM – WHERE schema. SELECT describes the type of resulting element, FROM specifies search scope and WHERE specifies the constraints or condition. Experimentation in [45] has shown better performance compared to plain EMF in terms of efficiency and usability for queries over models stored in Morsa repository.

2.4.2 Back-end Independent Querying

Another common way to query models is the use of high-level languages that abstract over model representations and persistence formats. ATL [18], OCL and EOL (Epsilon Object Language) are some examples of these types of such high-level languages, they make use of intermediate layers (such as OCL pivot metamodel [76], Epsilon model connectivity layer) These languages shield developers from the complexity of underlying technology of model persistence such as SQL, EMF. The OCL pivot metamodel only supports EMF-based models, while EMC supports several model persistence formats (such as Relational Databases, Spreadsheets, Simulink and EMF-based models). It is a driver-based approach, so new technology can easily be integrated by adding a driver that implements IModel Java interface of EMC as discussed in Section 2.2.2.1.

2.4.2.1 Native Query Translation

To execute high-level language queries on different back-ends, they can be translated to their respective native query languages as shown in Figure 2.15. Queries can be translated at compile-time or runtime. As in [77] and [78] have proposed

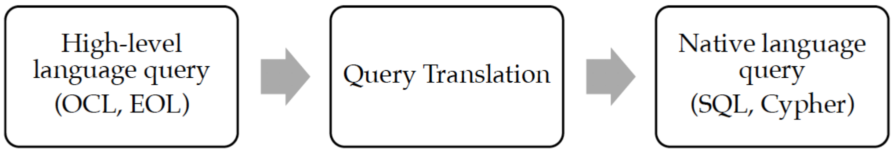

High-level language query (OCL, EOL)
Query Translation
Native language query (SQL, Cypher)

Figure 2.15: Query translation process

approaches for SQL generation from OCL expressions. They both map OCL expressions to SQL queries in one pass. Authors in [79] have proposed an algorithm to map each OCL expression to a single stored procedure. Against each OCL expression, it calls a corresponding stored procedure and creates a table that contains the result of evaluated OCL expression(stored procedure). However, the problem with native query translation is that both high-level and low-level languages do not offer identical capabilities(e.g. EOL supports dynamic dispatch while SQL doesn't) that makes it difficult to have a full translation. Another approach [4] suggests runtime SQL query generation and integrates driver for MySQL database in EMC layer of Epsilon.

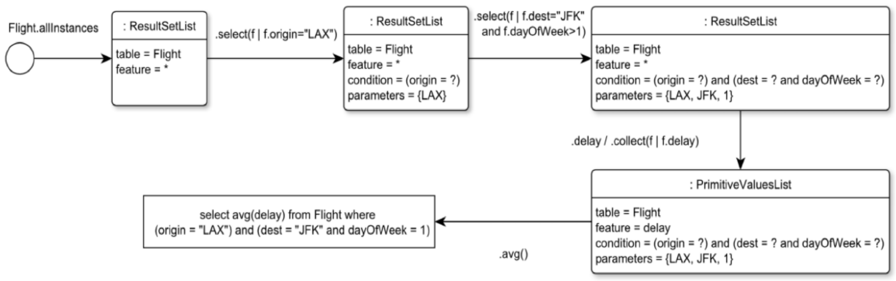
```
Flight.allInstances.select(f|f.origin="LAX")
  .select(f|f.dest = "JFK" and f.dayOfWeek>1)
  .delay.avg();
```

Listing 2.9: EOL program for querying MySQL database

Figure 2.16 describes the whole runtime query generation for the above mentioned EOL query. It presents how an SQL query shown in Listing 2.9 is generated from EOL query step by step. First retrieving all the rows of a table flight. Flight.all instances would be translated to Select * from flight, then select conditions like origin are Los Angeles and Destination is New York on Sunday and the for that records collect delay and compute average. This is how a query written in a high-level language is translated to native persistence level queries, in this case, EOL to SQL.

2.4.2.2 Using Derived Attributes

Consider a library metamodel and model as shown in Figure 2.17. If one needs to select authors that have more than "N" number of book , first on would need to retrieve all authors, then for each author, need to count the corresponding books. Finally, compare that count against N to get the authors having more than N number of books. Counting the number of "books" relationship to each author. This is computationally expensive as it would need to iterate all the instances of books for each author. Hawk [5] is a model indexing solution that can take models written with various technologies and turn them into graph databases, for easier and faster querying. In Hawk, an approach has been introduced to precompute these expensive attributes and cache them in a model index. This



Flight.allInstances
: ResultSetList
table = Flight
feature = *
.select(f | f.origin="LAX")
: ResultSetList
table = Flight
feature = *
condition = (origin = ?)
parameters = {LAX}
.select(f | f.dest="JFK" and f.dayOfWeek>1)
: ResultSetList
table = Flight
feature = *
condition = (origin = ?) and (dest = ? and dayOfWeek = ?)
parameters = {LAX, JFK, 1}
.delay / .collect(f | f.delay)
: PrimitiveValuesList
table = Flight
feature = delay
condition = (origin = ?) and (dest = ? and dayOfWeek = ?)
parameters = {LAX, JFK, 1}
.avg()
select avg(delay) from Flight where (origin = "LAX") and (dest = "JFK" and dayOfWeek = 1)

Figure 2.16: Process flow of query translation from EOL to SQL [4]

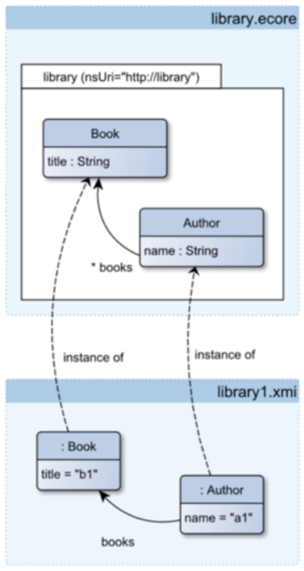


Figure 2.17: Library metamodel and model [5]

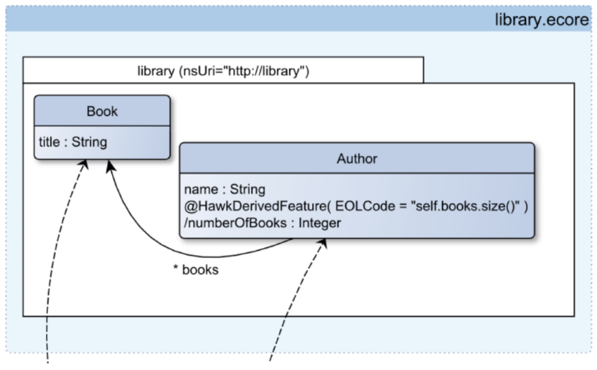

library.ecore
library (nsUri="http://library")
Book
title : String
Author
name : String
@HawkDerivedFeature(EOLCode = "self.books.size()")
/numberOfBooks : Integer
* books

Figure 2.18: Addition of derived attributes [5]

will significantly increase the execution speed of these extensive queries. In this case "number of book," attribute will be precomputed as *self.books.size()*. It will be added and attached to each author as shown in Figure 2.18. After adding a derived attribute, now the new query can be rewritten so that it does not have to iterate through all "books" references of every author, instead, this derived attribute *numberOfBooks* will be used to query authors having more than N number of books. Results in [5] have shown a decrease in execution time by using derived attributes but it has certain shortcomings as well. First, it adds an overhead of computing these derived attributes, which increases the insertion time of models that contain derived attributes. Another consideration is to use these derived attributes only if there is enough certainty that these attributes will be used multiple times in a program so that performance advantage can be gained at the cost of deriving overhead and increase in insertion time.

2.4.2.3 Program-Aware Query Optimisation

Another approach for query optimisation is regarded as "program aware". In program-aware query optimisation, before executing the program, optimisation approach is aware of both the "metamodel" and "the query" to be executed. Optimisation approach uses metamodel introspection and compile-time static analysis of query. One such approach has been proposed in [80] to efficiently compute allInstances query. *allInstances()* is an operation which can be invoked on a type to return a set containing all its instances. First, it checks if a program makes multiple calls to *allInstances()* property. Traditionally, *allInstances()* can be a very expensive operation when executed over large models because of iterating through all instances of a model element. If a program makes multiple calls to retrieve *allInstances()* then precomputing and cache all those collections in one go can optimise the program execution. It is based on greedy computation instead of on-demand computation. Naively, if a program calls *allInstances()*, it navigates through an in-memory representation of models. This efficient approach in Wei et al. [36] will increase the initial cost of calculating *allInstances()* in one go, it will also increase the memory footprint, but it ultimately yields efficient results for programs with multiple calls. Greedy caching, in this case, will be inefficient when multiple calls are made to a small number of model element types.

Another optimisation strategy as proposed in [81] suggests how combining three optimisation techniques (parallelisation, lazy evaluation and short-circuiting) can significantly increase the performance of queries over large models.

Mogwai for efficient scalable querying as proposed in [82] translates OCL and ATL expressions to Gremlin scripts- a query language for NoSQL databases. This shifts the computation of queries at the database side and it makes use of the benefits of optimisation strategies of backend technology for large models. To address scalability challenges in MDE, one solution is through the use of distributed systems. In this context, [83] presents an allocation optimisation approach by a combination of heuristics-based 1) reducing the usage of resources by minimising remote traffic and 2) reducing cost by minimising resources for evaluation of queries.

2.5 Query Optimisation in Databases

Query optimisation has been a well-known research area for so long in the field of database and a well researched area in the field of databases. Including a discussion on query optimisation in traditional databases within this research thesis is crucial as it establishes a comprehensive and holistic perspective on the overarching optimisation approaches. This Section presents a summary of how queries are optimised for different kinds of databases. Section 2.5.1 discusses query optimisation in relational databases where the data is stored in the form of tables in rows and columns. Section 2.5.2 will discuss about Document stores, where each record and its associated data is stored within a single document. MongoDB is one of the widely used document store databases.

2.5.1 Relational Databases

Roy et. al. [84] proposed an approach for efficiently querying of databases using multiquery optimisation. It exploits common sub-expressions in multiple queries or even a single query. The main concept used is to make a globally optimal plan, rather than considering local optimisation. For instance, if one considers two queries Q1 and Q2, Q1's locally optimal plans are (R X S) X P and (R X T) X S respectively. Now, consider both the plans there are no common sub-expressions. The key idea is not to use the best local plans but plans that can be globally best. So, one plan (may not be the best plan locally) for Q2 can be (R X S) X T. Now clearly, there is a common subexpression (R X S) which can be evaluated once to optimise the whole query batch. Another optimisation strategy in [7] is that if there is a view in place of a relation in a query. For example, consider Figure 2.19 there is a query Q = Join (R, V) where V is a View, V=Join (S, T). Query Q can be unfolded to Q = Join (R, join (S, T)) and then it can be reordered to optimise accordingly. It is graphically illustrated in Figure 2.19. One optimisation proposed in [7] is to use group by before joins to reduce data. Merging nested subqueries [7] is an alternative approach which

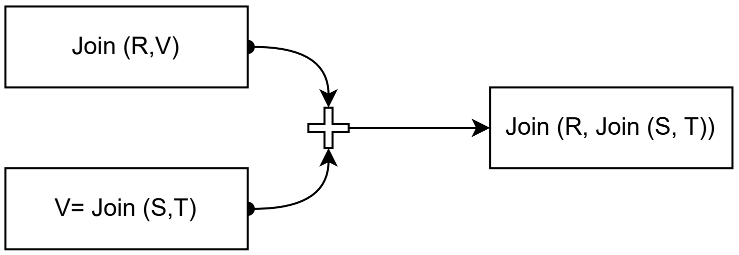


Figure 2.19: Merge nested queries

is described by an example. Considering the query in Listing 2.10 where an employee name is extracted from the employee table whose location is Denver and is a manager.

```sql
SELECT Emp.Name
FROM Emp
WHERE Emp.Dept# IN
SELECT Dept.Dept# FROM Dept
WHERE Dept.Loc ='Denver'
  AND Emp.Emp# = Dept.Mgr
```

Listing 2.10: Nested SQL query [7]

In the above query, all records are traversed from the department before evaluating the condition. It is proposed that these types of nested queries can be merged where before traversing all the records, the condition is evaluated first as shown in Listing 2.11.

```sql
SELECT E.Name
FROM Emp E, Dept D
WHERE E.Dept# = D.Dept#
  AND D.Loc = 'Denver' AND E.Emp#=D.Mgr
```

Listing 2.11: Merged SQL query [7]

2.5.2 Document Stores

NoSQL are non-relational databases and Mongo DB is one of a NoSQL database example. It is a document-oriented database and is increasingly used in the industry [85]. In [6] some optimisation strategies are presented. First is the use of index because a full document scan is much slower. Creating an index at specific columns of a large document makes queries fast. Moreover, creating an index at a field which needs sorting can enhance query execution time where filtration is based on that field. Cases where 80% of the document is retrieved as a result of a query, it is better to use natural sort than using an index. Another optimisation strategy proposed in [6] is the rational use of "and" and "or" in queries. Consider Figure 2.20, suppose three conditions A, B and C. Condition A retrieves 50,000 documents, condition B retrieves 3000 documents while condition C gets 100 documents. According to this, condition A is loose (because it retrieves larger sets of documents) and C becomes the severe condition(because it retrieves smaller sets of documents). If in a query, there is "and" operation severe condition should be evaluated first, because it greatly reduces search space. If there is an "or" in the query, the loose condition should be evaluated first.

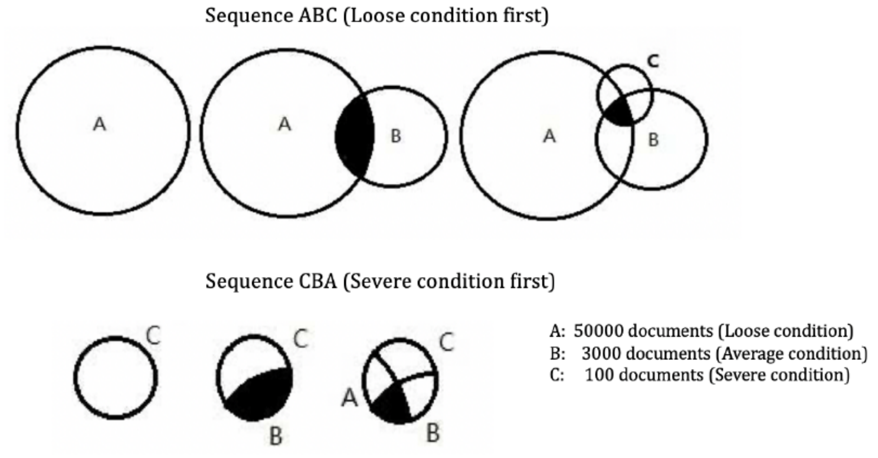

Sequence ABC (Loose condition first)
A
A
B
A
C
B
Sequence CBA (Severe condition first)
C
C
B
C
A
B
A: 50000 documents (Loose condition)
B: 3000 documents (Average condition)
C: 100 documents (Severe condition)

Figure 2.20: Example of Mongo DB query optimisation [6]

2.6 Chapter Summary

This chapter presented a brief overview of the MDE field and how it sits in the wider software engineering context followed by the use of different acronyms in the MDE literature. In **Section 2.1**, it then discusses the basic terminologies such as models, metamodels and model management tasks that will be extensively used in the following chapters of this thesis. It also provides the context to the research topic and then in **Section 2.2**, discusses current state-of-the-art tools and technologies used for different model management activities. Further, in **Section 2.3**, the challenges in MDE are discussed and then in **Section 2.4**,detail discussion of scalable model querying, discussing their different features and capabilities. In **Section 2.5**, it also discusses popular query optimisation in the context of databases field. The next chapter will present the analysis of the literature presented in this chapter and will also state the hypothesis of this research.

Chapter 3

Analysis and Hypothesis

This chapter presents the analysis of the literature review and states the objectives of this research followed by the research questions that needs to be answered to acheive the objectives. It also states the research hypothesis which is then followed by the research scope.

3.1 Analysis

Model-driven engineering (MDE) has been shown to deliver several benefits over traditional software development methodologies such as quality, maintainability and productivity. To continue the broader use of MDE in industrial projects, it is crucial that MDE technologies scale well with larger and more complex applications. A typical MDE workflow includes several tasks, including model validation, model-to-model transformations, and model-to-text transformations. All these tasks have a common set of queries/expressions operating over model elements. As queries grow complex, they can significantly impact performance both in terms of execution time and memory footprint.

Several tailored high-level model management languages such as OCL and EOL enable developers to work on different backend technologies in a uniform way by shielding them from the complexities of different backends. On the contrary, performance with respect to execution time in tailored model management languages programs become one of the major scalability bottlenecks because these languages are typically interpreted.

Considering the analysis of the problem presented above, an approach is proposed for optimising model management programs operating over large-scale EMF-based models. The architecture of the envisioned approach is depicted as a block diagram in Figure 3.1. The primary purpose of this framework is to be able to automatically rewrite expensive queries in an input model management program written in a technology agnostic language i.e., Epsilon to a more efficient form. The rewritten program would be efficient in terms of execution time. The rewriting process will be taking memory footprint into consideration to make a trade-off. Program rewriting is based on information extracted through static analysis. The rewriting and optimisation will vary based on the type of input model management program. After a model management program is parsed, an Abstract Syntax Tree (AST) is generated. This AST may not include any type information attached to its nodes. Before execution, a static analyser component will analyse the program and populate type-related information into the AST, this type-resolved AST is also referred to as an Abstract Syntax Graph. The Abstract Syntax Graph would then be used by the rewriters involved in a program, to rewrite this into an optimised form.

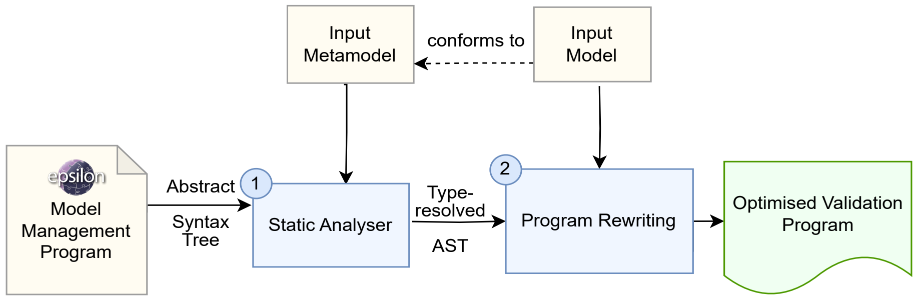


Figure 3.1: Proposed architecture of model management program optimisation

3.2 Research Objectives

The overall research goal for this project is to investigate the applicability of compile-time query optimisation techniques to a wide range of model management programs, and to identify reusable optimisation approaches and patterns. To achieve this goal, an open-source prototype has been developed implementing algorithms that sit on top of existing model management languages to improve performance over EMF-based models. These algorithms are expected to reduce the execution time of complex queries. There are other factors that greatly affect the performance gain such as underlying persistence formats which are not in the scope of this research. A breakdown of the overall research goal into more fine-grained objectives is as follows:

- **RO-1:** Identify the performance challenges involved in executing complex model management programs over large EMF-based models.
- **RO-2:** Identify reusable optimisation approaches and patterns across different model management programs using static analysis of high-level language programs.
- **RO-3:** Propose algorithms for the optimisation of model management programs operating on large-scale EMF-based models.

- **RO-4:** Ensure to preserve the semantics of the original program while producing the optimised version.
- **RO-5:** Evaluate the results of the proposed algorithms in terms of execution time and memory footprint.

3.3 Research Questions

The following research questions have been set out to achieve the above-mentioned research objectives in Section 3.2.

- **RQ-1:** What are the performance challenges in terms of execution time for complex model management programs over large EMF-based models? Answering this research question will help achieve RO-1 i.e. performance challenges for expensive queries and also to help propose a solution for the identified limitations.
- **RQ-2:** What information can be extracted before the execution of a model-management program to facilitate automated rewriting of such programs? Answering RQ-2 will help to achieve RO-2 by implementing static analysis for model management programs and to extract information before execution for optimisation of expensive queries.
- **RQ-3:** Can the information extracted from static analysis help to identify potential rewriting opportunities for model management programs? RO-3 will be achieved by answering this research question and implementing optimisation algorithms and techniques for existing model management languages.
- **RQ-4:** Does the optimised program generated by the rewriting approach produce the same output as the original program? RO-4 i.e., ensuring the preservation of semantics will be achieved by answering this research question.
- **RQ-5:** Does static analysis and automatic program rewriting help in reducing memory footprint and execution time when complex model management programs are executed over large models? RO-5 will be achieved by addressing this research question. This is evaluated over both publicly available and syntethic programs over large models. The proposed approach will be compared to other state-of-the-art tools and engines.

3.4 Research Hypothesis

The hypothesis of this research is stated as follows:

"The execution time of computing expensive queries in ***model validation, model-to-model transformation*** *and* ***model comparison*** *tasks over large*

EMF-based models can be significantly reduced using automated ***program rewriting*** *techniques based on the in-advance information extracted through* ***static analysis****."*

3.5 Scope

This section establishes the scope and boundaries of this research work. As discussed in the chapter above, this work aims at optimisation of model management programs using automated program rewriting. This subsection will narrow this down. Epsilon is used for implementing the techniques developed in this research. The following optimisations are addressed

- Optimisation of type-level model queries (EVL)
- Translating first-order EOL expressions to VIATRA patterns (EVL)
- Selective traceability of model-to-model transformations (ETL) operating over large-scale EMF-based models.
- Efficient model comparison (ECL) programs operating over large-scale EMF-based models

3.6 Chapter Summary

This chapter summarised the research challenges and stated the research hypothesis and scope. In **Section 3.1**, the analysis of the problem domain was presented after discussing the background and literature review in **Chapter 2**. Then **Section 3.2**, described the aim of this research, and then listed the breakdown of objectives and then some research questions to address the above mentioned research objectives in **Section 3.3**. **Section 3.4** states the research hypothesis followed by the proposed architecture for program optimisation. Finally the chapter is concluded by presenting the research scope in **Section 3.5**.

Chapter 4

Static Analysis of Epsilon programs

The role of a static analyser is to analyse source code without executing it. Static analysis will provide in-advance knowledge about any model management program it operates on that can then be used for several purposes such as auto-code completion, detecting compile time errors and improving the quality of the program. In model management context, static analysis can be used for providing capabilities like intelligent model partitioning [?], on-demand model loading [?] and query optimisation [86].

In the context of this research, the information extracted from static analysis is used for optimising the execution of some classes of model management programs. Static analysis work was done in collaboration with Sorour Jahanbin, another PhD student at the University of York. This is a foundational work that could be used in further optimisations. To optimise model management programs, the static analyser will analyse the program written in Epsilon languages and will generate a type-resolved abstract syntax graph. Static analyser will enable identifying accessed model elements and their patterns in the model management program for the purpose of query optimisation. The Static analyser needs to know the structure of the metamodel in order to resolve types of various constructs of the program which implies that in order to perform static analysis on model management programs, the underlying metamodel needs to be accessed. For metamodel introspection, metamodel is accessed using *ModelDeclarationStatement*. *ModelDeclarationStatement* is a statement that can be specified in the Epsilon programs to declare the underlying model and its metamodel. This is discussed in detail in Section 4.2 as shown in Figure 4.1.

4.1 Architecture of Static Analysis

The core language of Epsilon is EOL, which is a general model management language containing imperative constructs. It provides common facilities that

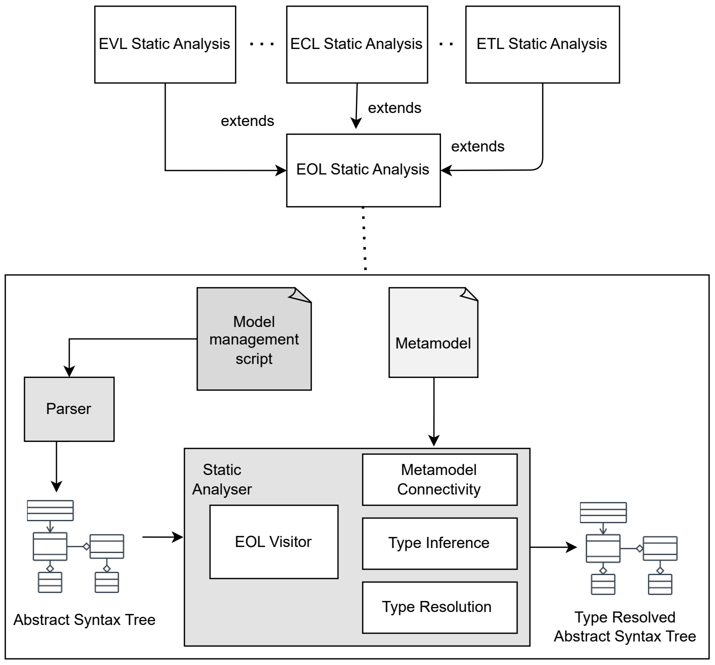


Figure 4.1: Static analysis architecture

are useful for developing task-specific model management languages.

Static analysis of Epsilon started in [87] where the Abstract Syntax Tree (AST) of an EOL program is computed, then resolution algorithms including variable resolution (e.g., resolving identifiers to their definitions) and type resolution (e.g., primitive types and collection types) are applied to derive an Abstract Syntax Graph. Thus, using the Abstract Syntax Graph, the static analyser can extract relevant information (i.e., types and properties accessed by the program).

This work contributes a static analysis facility for EOL, EVL, ETL and ECL, where compile-time errors are produced as a by-product. In first step, a type resolver sets the resolved type of expressions then there is a type checker in order to check type compatibility between types of context, parameter and return types both for user-defined and built-in operations. These core facilities are developed for the core Epsilon language and then language specific support is added for other languages.

4.2 Metamodel Connectivity

When a model management program is to be executed, the model and the metamodel to be loaded are specified in an execution configuration settings. However, structure of the metamodel is required for static analysis which means the metamodel needs to be loaded in order to perform static analysis before the program execution. *ModelDeclarationStatement* is responsible for specifying metamodel configuration in the Epsilon source code. The structure of the *ModelDeclarationStatement* is illustrated below in Figure 4.2.

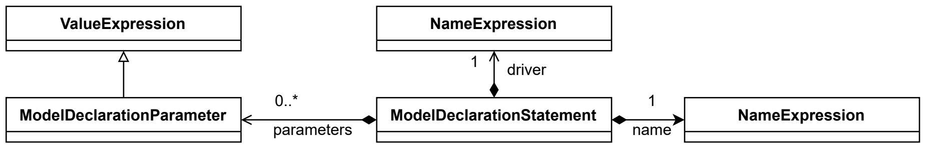


Figure 4.2: *ModelDeclarationStatement* Structure

A *ModelDeclarationStatement* has the following features:

- A *NameExpression* specifying the name of the model to be loaded.
- A *NameExpression* specifying the driver; which means the type of the model that has to be loaded such as EMF and Simulink
- A list of *ModelDeclarationParameters* specified by the key-value pairs. The key represents the name of the parameter while the value represents its corresponding value as shown below

```
nsuri = "http://www.eclipse.org/emf/2002/Ecore"
```

Listing 4.1: Example *ModelParameter*

An example of the usage of *ModelDeclarationStatement* is shown below.

```
1 model flowchat driver EMF{
2   nsuri = "flowchart"
3 };
```

Listing 4.2: Usage of *ModelDeclarationStatement*

This model is named as *flowchart* and then the driver is specified as EMF, which means the metamodel is loaded in the *EPackageRegistry* using the *nsuri* parameter specified in Line 2 above. An *EPackageRegistry* stores a map of identifiers of metamodels.

To access model elements from the above declared model, one needs to use the *ModelName!Type* notation. For instance, to use the *Transition* model element from the model declared in Listing 4.2, it will be used as *flowchart!Transition*.

4.3 Type Heirarchy in Epsilon

The entire type system of Epsilon containing all types along with their subtypes is illustrated in Figure 4.3. These are the built-in types provided by Epsilon. Some additional types are added for the purpose of static analysis and are discussed in Section 4.4.3.

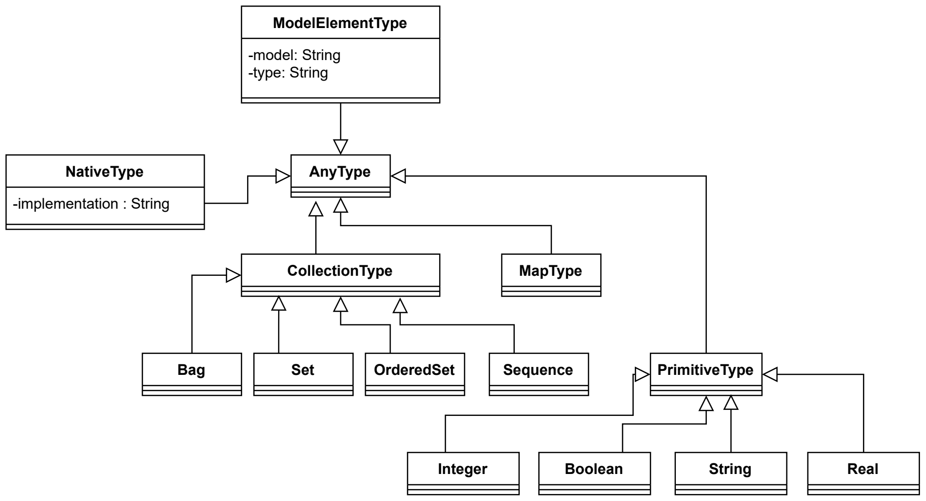


Figure 4.3: Epsilon built-in types heirarchy

4.3.1 *AnyType*

AnyType is the base of all types of Epsilon and is inspired by Object Constraint Language's (OCL) *OclAny*. *AnyType* can hold any type in EOL. Variables in EOL can be declared without a type. If the type of a variable is not explicitly declared, the type is automatically assumed to be *Any*. As in listing below, *foo* would be of type *Any*.

```
var foo;
```

4.3.2 *CollectionType*

Epsilon supports four types of collections:

- *Bag* - holds non-unique, unordered collection. *EolBag* implements the *java.util.Collection* interface.
- *Sequence* - holds non-unique, ordered collection. *EolSequence* implements the *java.util.List* interface.
- *Set* - holds unique and unordered collection. *EolSet* implements the *java.util.Set* interface.
- *OrderedSet* - holds unique and ordered collection. *EolOrderedSet* implements the *java.util.Set* interface.

Below is an example of a declaration of a *Sequence* of *String*s named as *studentNames*.

```
var studentNames: Sequence(String);
```

Epsilon also supports two concurrent collection types *ConcurrentBag* and *ConcurrentSet* since version 2.0. These types are thread-safe variants of the *Bag* and *Set* respectively.

The abstract *Collection* type is the parent type of all collection types. EOL enables logical operations on collections in addition to simple operations. There are additional operations supported by all types of collections, together with any operations declared on the *java.util.Collection* interface.

4.3.3 *MapType*

Epsilon supports the Map type that holds a Set of key-value pairs in which the keys are unique. The Map type implements the *java.util.Map* interface.

```
1 var students = Map{001 = "Alice", 002 = "Bob"};
```

A Map *students* is defined in line 1, with its keys 001, 002 and values “Alice”, “Bob”.

4.3.4 *PrimitiveType*

There are four primitive types supported in Epsilon. All the primitive types provide certain built-in operations specific to each of them[1].

- *Boolean* - hold true/false states
- *Real* - hold read numbers
- *Integer* - hold natural numbers, negatives and extends the Real primitive type
- *String* - hold finite number of characters

The following listing shows a variable *bar* declared as an *Integer*.

```
var bar: Integer;
```

4.3.5 *NativeType*

EOL allows users to create objects from the underlying programming environment using native types. These native types specify implementation properties that identify unique identifiers for corresponding platform types. For example, in listing below written in EOL, users can instantiate and use Java classes using their class identifiers. This approach enables users to overcome EOL’s limitations by leveraging functionalities from the underlying programming language, such as creating a Java window (Swing JFrame) and manipulating its properties.

[1]https://eclipse.dev/epsilon/doc/eol/

```
var frame = new Native("javax.swing.JFrame");
frame.title = "Opened with EOL";
frame.setBounds(100,100,300,200);
frame.visible = true;
```

Listing 4.3: Use of Native Types in Epsilon

4.3.6 *ModelElementType*

ModelElementType is used to represent types defined in the underlying metamodels. EOL uses the ! notation to access model element types. For example:

```
var movie = new imdb!Movie;
```

A model element type is specified by *imdb!Movie* where *imdb* is the name of the model under consideration and *Movie* is the type of the element in the metamodel. *ModelElementType* contains the *modelName* and *elementName*, which is used to identify the model element type.

4.4 Type Resolution

Type resolution is the process of resolving the types of various nodes of the AST of the source code. To resolve types, one option is to make use of either the explicit declared type by the user or an inferred type from the expression or a value, or using a pseudo type (explained later in the section). Type resolution is the primary block in static analysis and is used further in features like type checking.

4.4.1 Type Resolution by Declaration

One way to resolve types of variables is to directly get it from declaration, if the variable is explicitly declared by the developer. For instance, Listing 4.4 specifies a *ModelDeclarationStatement* accessing a model named as *imdb* which is an EMF-based model and is accessed using the specified *nsuri* parameter. It then declares a variable *movie* of explicit type *imdb!Movie* and assigns it the first "Movie" instance from the *imdb* model. The *imdb!Movie* type resolution explicitly specifies that the *Movie* class is part of the *imdb* model.

```
model imdb driver EMF {
  nsuri="https://movies/1.0"
};
var  movie : imdb!Movie = imdb!Movie.all.first();
```

Listing 4.4: An example of type resolution by explicit declaration

4.4.2 Type Resolution by Inference

Another way to resolve types of variables is to infer them by analysing the expressions they are used in. For instance Listing 4.5 specifies a *ModelDeclarationStatement* accessing a model by the keyword *model*, named as *imdb* which is an EMF-based model as explained above. It then declares a variable *highestRated* of explicit type *imdb!Movie* and assigns it the one of the *Movie* instance from the *imdb* model with the *ratings* attribute equal to 10. The iterator variable *m* has no explicitly declared type so using the type inference the static analyser sets the type of *m* as *imdb!Movie*. This is because *selectOne* iterates over *allInstances* of *imdb!Movie*.

```
model imdb driver EMF {
  nsuri="https://movies/1.0"
};
var  highestRated : imdb!Movie = imdb!Movie.all.
    selectOne(m|m.ratings==10);
```

Listing 4.5: An example of type resolution by inference

4.4.3 Type Resolution by PseudoTypes

To enable static analysis of Epsilon programs, it was necessary to introduce a number of pseudo-types to the language. Pseudo types help resolve types of various expressions where the types are changed based on their usage. Pseudo-types are added by taking an inspiration from Object Constraint Language (OCL) [16] for the purpose of static analysis. They are called pseudo-types as they cannot be instantiated. They are just used to determine type of *self* (i.e., context, the object on which the operation is called) in operation signatures. The exact type of these will be determined at the end of static analysis. Pseudo types (*EolSelf*, *EolSelfCollectionType*, *EolSelfExpressionType*, *EolContentType*) were added in EOL to help in static analysis, in order to propagate the inferred types of variables across complex expressions, which would have otherwise been lost.

4.4.3.1 ***EolSelfType***

EolSelfType is a pseudo type used in signature (as shown in Listing 4.6) of operation to propagate the context type as return type. One such example is *println()*. For example, if *"abc".println()* is called, return type would then be *EolPrimitiveType.String* , as self, in this case "abc", is a String.

```
operation Any println() : EolSelf {
  return self;
}
```

Listing 4.6: *EolSelf* Pseduo Type

4.4.3.2 *EolSelfCollectionType*

EolSelfCollectionType is a pseudo type specially for EolCollectionTypes. It is used in operations signature as shown in Listing 4.7.

```
operation Collection<Any> test():
   EolSelfCollectionType {
  return self;
}
```

Listing 4.7: *EolSelfCollectionType* Pseduo Type

If *test()* operation is called on *Sequence⟨Integer⟩*, the return type would also be *Sequence⟨Integer⟩*.

4.4.3.3 *EolSelfContentType*

Whenever this operation is called with any collection type (*Collection*, *Sequence*, *Bag*, *OrderedSet*, *Set*) as context type the return type would be the same type of collection. For example, if this operation is called on *Sequence⟨String⟩*, return type would also be *Sequence⟨String⟩*.

EolSelfContentType is also a pseudotype just for EolCollectionTypes. It is used in operation signature as shown in Listing 4.8.

```
operation Collection<Any> test():EolSelfContentType {
}
```

Listing 4.8: *EolSelfContentType* Pseduo Type

4.4.3.4 *EolSelfExpressionType*

Whenever this operation is called with any collection type (*Collection*, *Sequence*, *Bag*, *OrderedSet*, *Set*) as context type the return type would be the type of content type of the collection. For example, if this operation is called on Sequence⟨String⟩, return type would also be *EolPrimitiveType.String*.

For Bag⟨Integer⟩as context type, return type would be an integer *EolSelfExpressionType* is also a pseudotype just for EolCollectionTypes. It is used in operation signature as shown in Listing 4.9

```
operation Collection<Any> collect(a: Any) :
  Collection<EolSelfExpressionType>{
}
```

Listing 4.9: *EolSelfExpressionType* Pseduo Type

Whenever this operation is called with any collection type (*Collection*,*Sequence*, *Bag*, *OrderedSet*, *Set*) as context type the return type would be the type of content type of the iterator expression. For example, if this operation is called as following, the return type would also be *EolPrimitiveType.Boolean* as shown in Listing 4.10.

```
var b: Sequence = Sequence {1,2,3};
var c = b.collect(f|f<3);
```

Listing 4.10: Use of *EolSelfExpressionType* Pseduo Type

4.5 Type Checking

Type resolution can further help in checking type compatibility which is helpful in producing warnings/errors to provide better programming experience to developers. Type checking features are explained as follows: Consider the movies metamodel. In Figure 4.5 operation *printName()* specifies return type as *String* but it does not have any return statement in the body of the operation. So, a compile-time error would be produced prompting user to add a missing *ReturnStatement*. In second operation *printName()* specifies *String* as return type but in the return statement, it returns an integer value which is incompatible with the return type provided in the operation signature. Figure 4.4 specifies the relation between type compatibility and the corresponding errors or warnings produced. Table 4.1 shows an example of different types and the error or warning produced.

In Figure 4.5, an operation *greetUser()* is called on *Integer* as context. An error is produced in this scenario because the signature of operation the required context type to be *String*.

A simple example is presented in Figure 4.5, *greetUser()* operation requires a *String* context type, but the provided type is *Any* which is the parent type of *String*. So, a warning is produced saying *greetUser()* may not be invoked on *Any*, as it requires *String*.

Provided Type	**Required Type**	**Result**
String	*String*	No error/warning
Sequence	*Collection*	No error/warning
Collection	*Sequence*	Warning
String	*Integer*	Error

Table 4.1: Examples of various types and corresponding errors/warnings

4.5.1 Type Checking in built-in operations

Type checking for user-defined operations is straightforward because the signature and definitions of operations are available to compare them with the provided types in the operation calls. To check the types of built-in operations, a separate EOL file is created containing the signatures of built-in operations. An excerpt of that built-in operations file is shown in Appendix A.

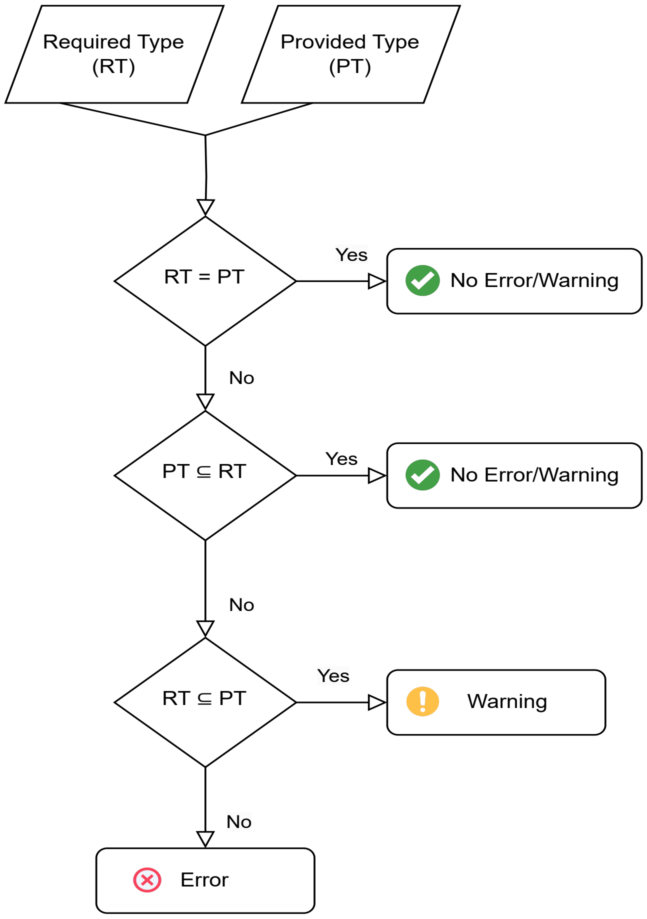
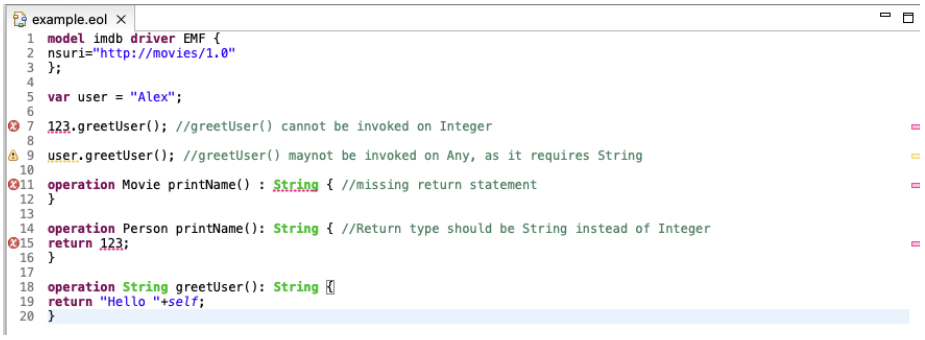


Figure 4.4: Type compatibility and corresonding errors/warnings



```
model imdb driver EMF {
nsuri="http://movies/1.0"
};

var user = "Alex";

123.greetUser(); //greetUser() cannot be invoked on Integer

user.greetUser(); //greetUser() maynot be invoked on Any, as it requires String

operation Movie printName() : String { //missing return statement
}

operation Person printName(): String { //Return type should be String instead of Integer
return 123;
}

operation String greetUser(): String {
return "Hello "+self;
}
```

Figure 4.5: Type compatibility errors - Example

This separate EOL file contains the signatures of all built-in operations provided by Epsilon. This serves as a template to provide type checking facilities for the built-in operations and is just used for static analysis before the execution of the program. When the static analysis is triggered for an Epsilon program, these built-in operation signatures file is loaded so that these operations are added to operations of *EolModule*.

If a user calls a built-in operation, first the operation call is traversed to find the potential matched operations with the same name. Then it looks for an exact match by matching the context type, the number and types of parameters and the return type. The types in the operation call are referred as "provided types" and the ones in the method signature as "required types". Suppose there are two operations with the same name but different parameter types as shown in Listing 4.11. Now, depending on the type of parameter passed in the operation call, the static analyser will match the corresponding operation. For example, in Line 5, the parameter passed is a type of *Movie* model element so the matched operation here will be *operation printName (m : Movie) : String*. While in Line 7, where the parameter is of type *Person*, the matched operation will be *operation printName (p : Person) : String*. Some examples of errors and warnings with respect to type compatibility are shown in Figure 4.4.

```
1  model imdb driver EMF {
2    nsuri = "http://movies/1.0"
3  };
4
5  printName(Movie.all.first());
6
7  printName(Person.all.first());
8
9  operation printName (m : Movie) : String {
10   m.title.println("Movie Title: ");
11 }
12
13 operation printName (p : Person) : String {
14   p.name.println("Actor Name: ");
15 }
```

Listing 4.11: Example of polymorphism in EOL

4.6 EOL Static Analysis

EolStaticAnalyser is the core static analysis facility for all the imperative constructs provided by the core Epsilon language EOL. The static analysis is implemented using the visitor design pattern. An excerpt of EOL Visitor is shown in Listing 4.12. *EOLVisitor* implements the interface *IEolVisitor*. The *IEolVisitor* is extended by the other Epsilon language visitors as shown in Figure 4.6. EOL visitor an excerpt of which is shown in Listing 4.12, traverses each node of the

Abstract Syntax Tree (AST) of EOL program. In each traversal, the types are calculated and added as a resolved type to the AST nodes.

```
public interface IEolVisitor {
  ...
  public void visit(AndOperatorExpression
      andOperatorExpression);
  ...

  public void visit(AssignmentStatement
      assignmentStatement);
  ...

  public void visit(FirstOrderOperationCallExpression
      firstOrderOperationCallExpression);

  public void visit(ForStatement forStatement);

  ...

  public void visit(OperationCallExpression
      operationCallExpression);

  public void visit(VariableDeclaration
      variableDeclaration);

  public void visit(WhileStatement whileStatement);

  ...
}
```

Listing 4.12: An excerpt of the IEolVisitor

The first step is to traverse all the model declarations to load the respective metamodels to resolve and check the types of corresponding model elements. Then, built-in operation signatures (explained in detail in Section 4.5.1) are loaded and added to operations of the EOL program.

```
var firstOperand : Integer = 5;
var secondOperand : Integer = 4;

addResult = firstOperand + secondOperand;
```

Listing 4.13: An example to demonstrate Type Resolution

```
model imdb driver EMF {
  nsuri = "http://movies/1.0"
};
```

```
imdb!Movie.all.select(m|m.ratings == 10);
```

Listing 4.14: An example to demonstrate Type Inference

Table 4.2 shows the resolved types of the various constructs in Listings 4.13 and 4.14.

Expression	Resolved Type
firstOperand	*Integer*
secondOperand	*Integer*
addResult	*Integer*
imdb!Movie.all	*Collection⟨imdb!Movie⟩*
m	*imdb!Movie*
m.ratings	*Real*
imdb!Movie.all.select(m|m.ratings == 10)	*imdb!Movie*

Table 4.2: Resolved Types of various constructs of Listing 4.14 & 4.13

The type resolution either works by checking the type declaration or by the type inference e.g., Line 1 and 2 in the Listing 4.13, the type of *firstOperand* and *secondOperand* is resolved to *Integer* using the declared types. In Line 4, the *addResult*'s type is inferred as *Integer*.

```
public interface IErlVisitor extends IEolVisitor {

  public void visit(Post post);

  public void visit(Pre pre);

}
```

Listing 4.15: IErlVisitor structure

Then *EolStaticAnalyser* is extended to add language specific support. The structure of the various visitors and their hierarchy is shown in Figure 4.6. *IEolVisitor* is the core interface containing visit method signatures to EOL constructs and then there is an *IErlVisitor* (shown in Listing 4.15) containing visit methods to *pre* and *post* blocks. This is because all other languages have these constructs. Further language specific visitors are explained in their respective sections.

4.7 EVL Static Analysis

EvlStaticAnalyser is the static analysis facility provided for validating programs written in EVL. It extends *EolStaticAnalyser* and adds additional support for

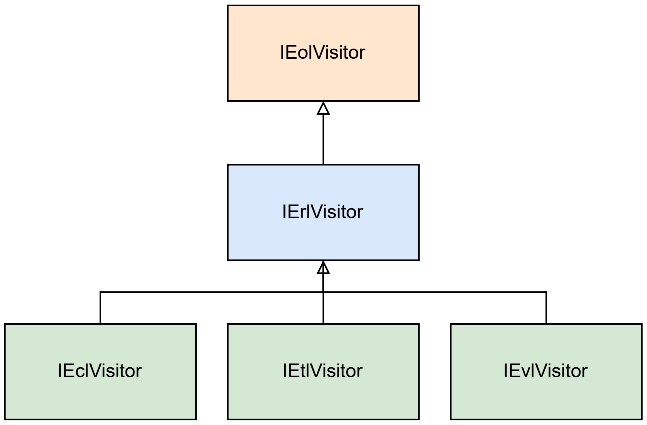


Figure 4.6: Hierarchy of Various Visitors in Epsilon

EVL specific constructs such as Constraint. The structure of *IEvlVisitor* is shown in Listing 4.16. Additionally the sequence in which EVL static analyser works is depicted in Figure 4.7.

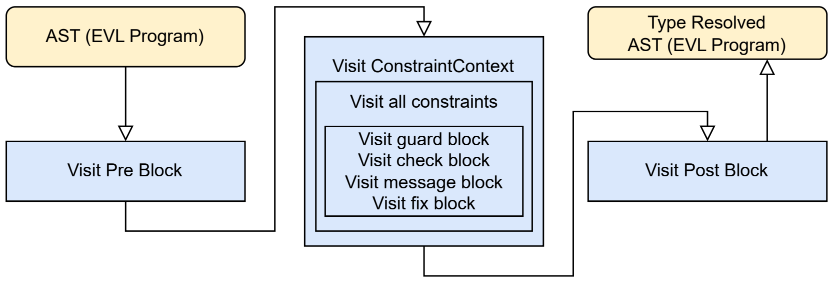


Figure 4.7: Flow of *EvlStaticAnalysis*

```
public interface IEvlVisitor extends IErlVisitor {

  public void visit(ConstraintContext
     constraintContext);

  public void visit(Constraint constraint);

  public void visit(Fix fix);

}
```

Listing 4.16: *IEvlVisitor* structure

Consider an example EVL program as shown in Listing 4.17 where a constraint *NameNotNull* is checking if a movie name is not null.

```
model imdb driver EMF {
  nsuri= "http://movies/1.0"
};

pre {
  //to be executed before the execution of constraints
}

context Movie {
  constraint NameNotNull {
    check : self.title!=null
  }
}

post {
  // to be executed after the execution of constraints
}
```

Listing 4.17: An example EVL program

Expression	**Resolved Type**
self	*imdb!Movie*
self.title	*String*
self.title!=null	*Boolean*

Table 4.3: Resolved Types of various constructs of Listing 4.17

Type resolution of elements in the Listing 4.17 is shown in Table 4.3.

4.8 ETL Static Analysis

EtlStaticAnalyser is the static analysis facility provided for model to model transformation programs written in ETL. It extends *EolStaticAnalyser* and adds additional supports for ETL specific construct i.e., *TransformationRule*. The structure of *IEtlVisitor* is shown in Listing 4.18. Additionally the sequence in which ETL static analyser works is depicted in Figure 4.8.

```
public interface IEtlVisitor extends IErlVisitor{
  public void visit(TransformationRule
      transformationRule);
}
```

Listing 4.18: *IEtlVisitor* structure

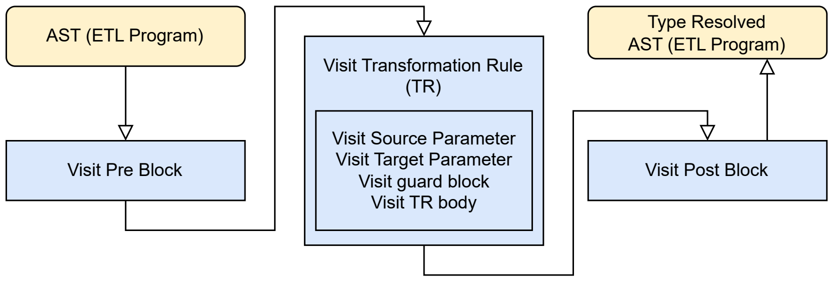


Figure 4.8: Flow of *EtlStaticAnalysis*

Consider an ETL transformation as shown in the Listing 4.19. It transforms a *Tree* to Graph conforming to metamodels as shown in Figure 2.11.

```
model Tree driver EMF {
  nsuri= "Tree"
};

model Graph driver EMF {
  nsuri= "Graph"
};

rule Tree2Node
transform t : Tree!Tree
to n : Graph!Node {

  n.label = t.label;

  if (t.parent.isDefined()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target = t.parent.equivalent();
  }
}
```

Listing 4.19: An example ETL program

4.9 ECL Static Analysis

EclStaticAnalyser is the static analysis facility provided for comparison programs written in ECL. It extends *EolStaticAnalyser* and adds additional supports for ECL specific construct i.e., *MatchRule*. The structure of *IEclVisitor* is shown in Listing 4.20. Additionally the sequence in which ECL static analyser works is depicted in Figure 4.9.

Line number	Expression	Resolved Type
10	*t*	*Tree!Tree*
11	*n*	*Graph!Node*
13	*n.label*	*String*
13	*t.label*	*String*
15	*t.parent.isDefined()*	*Boolean*
16	*edge*	*Graph!Edge*
17	*edge.source*	*Graph!Node*
18	*edge.target*	*Graph!Node*

Table 4.4: Resolved Types of various constructs of Listing 4.19

```
public interface IEclVisitor extends IErlVisitor {
  public void visit(MatchRule matcRule);
}
```

Listing 4.20: *IEclVisitor* structure

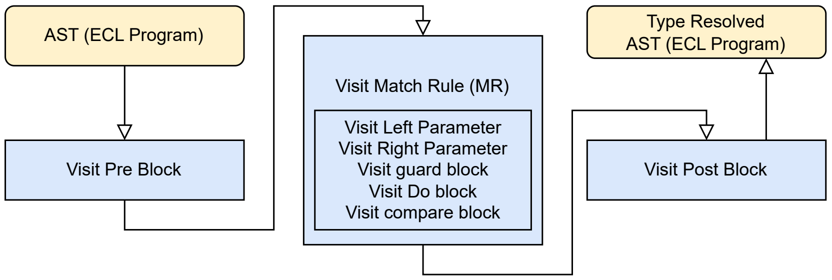


Figure 4.9: Flow of *EclStaticAnalysis*

Consider an example ECL program as shown in Listing 4.21, where two models conforming to Tree metamodel (shown in Figure 2.11) are compared. Two *Trees* are matched checking if their labels are the same and the parent and the children of the Trees are the same.

```
model T1 driver EMF {
  nsuri= "Tree"
};

model T2 driver EMF {
  nsuri= "Tree"
};
```

```
 8
 9 rule Tree2Tree
10 match l : T1!Tree
11 with r : T2!Tree {
12
13   compare : l.label = r.label and
14   l.parent.matches(r.parent) and
15   l.children.matches(r.children)
16 }
```

Listing 4.21: An example ECL program

Type-resolved in the above listing is shown in Table 4.5.

Line number	**Expression**	**Resolved Type**
10	*l*	*T1!Tree*
11	*r*	*T2!Tree*
13	*l.label*	*String*
13	*r.label*	*String*
14	*l.parent*	*T1!Tree*
14	*r.parent*	*T2!Tree*
15	*l.children*	*T1!Tree*
15	*r.children*	*T2!Tree*

Table 4.5: Resolved Types of various constructs of Listing 4.21

4.10 Eugenia - Test case

As static analysis is a foundational tool that this research is based on, it is crucial to check the correctness of the static analyser. JUnit test cases are used to test the correctness of various constructs of the Epsilon programs using static analysis. The features of the static analyser are evaluated on Eugenia [88]. Eugenia is a well-known tool built using the Epsilon platform implemented in EOL to generate GMF files from an annotated Ecore metamodel. Eugenia consists of an extensive EOL program, having 1212 lines of code, which transforms annotated metamodels to 4 different models required by the GMF framework in order to generate a graphical editor. Eugenia automatically generates *.gmfgraph*, *.gmfmap* and *.gmftool* required by GMF from a single annotated Ecore[2] metamodel.

The following features were evaluated:

[2]https://wiki.eclipse.org/Ecore

- An error to be produced if the required type in a method or assignment is not the same as the provided type.
- A warning to be produced if the provided type is a super type of the required type.
- No error or warning if the provided type is a subtype of the required type or same as the required type.

In Eugenia, warnings were observed where parent type of required type is provided. A screenshot of static analysis on *Ecore2GMF.eol* (a transformation from Eugenia) is shown in Appendix A in Figure A.1 and Figure A.2. This case study helps to evaluate the correctness of components of static analyser such as type resolution and type checking.

4.11 Related Work

AnATLyzer [89] is a tool for static analysis of ATL model transformations. Basically, it is an IDE that provides type checking, quick fixes and problem explanations. AnATLyzer focuses on three main points: 1) It checks that the source metamodel is correctly typed with respect to the transformation. 2) It ensures that the model generated through transformation conforms to the target metamodel. 3) It identifies any conflicting or missing rules. This static analyser is limited to ATL model transformations only.

In [90], Born et al. extended Henshin, a rule-based model transformation language adapting graph transformation concepts and being based on the Eclipse Modeling Framework (EMF). This extension computes all potential conflicts and dependencies of a set of rules and reports them in form of critical pairs. Each critical pair consists of the respective pair of rules, the kind of potential conflict or dependency found, and a minimal instance model illustrating the conflict or dependency.

Another tool in [91], provides a static analysis facility for graph transformations. This work is based on Constraint Satisfaction Programming (CSP). It also presents a type checker for Viatra2 framework. As this type checker is based on CSP, it is not possible to find all the errors in a single run using static analysis.

The static analysis of OCL is presented in [92], a pseudo-type OCLSelf, is introduced to infer the type of context for few operations such as:

- OclAny::oclAsSet() – returns Set⟨Self⟩
- OclAny::oclType() : Class⟨OclSelf⟩

Willink [93] introduced safe navigation operators in OCL. This operator solves the problem of declaring non-null objects and null-free collections. It enables OCL navigation to be fully checked for null safety.

These static analysers tools mentioned in this section have motivated the development of static analysis facility for Epsilon languages. The purpose of static analysis for this research was to use it as a foundation tool for enabling query optimisation but static analysis can be used to provide features like auto-code completion.

4.12 Chapter Summary

This chapter started with defining the static analysis and what its role in MDE followed by its usage in the context of this research. In **Section 4.1**, the architecture of static analysis of Epsilon languages is presented giving a brief overview of the whole process going from AST to type-resolved AST. The first step metamodel connectivity is explained by detailing the *ModelDeclarationStatement* with an example to show how an underlying model is accessed without executing the program in **Section 4.2**,. Then, in **Section 4.3**, the hierarchy of built-in types in Epsilon are illustrated by giving a brief overview of *AnyType*, *CollectionType*, *MapType*, *PrimitiveType*, *NativeType* and *ModelElementType*. Then the next step type resolution is described and also how types are resolved by explicit declaration, inference and pseudo types. *EolSelfType*, *EolSelfCollectionType*, *EolSelfExpressionType*, *EolSelfContentType* are explained in detail with examples.

Section 4.5 explains how the resolved types as explained in **Section 4.4** are used in the type checking process. This section also explains different scenarios where an error or warning would be produced. Additionally, how types are checked in builitin operations is also explained.

Finally, Epsilon languages static analysis are presented as a whole along with the corresponding examples starting with EOL static analyser in **Section 4.6**, EVL static analyser in **Section 4.7**, ETL static analyser in **Section 4.8** followed by ECL static analyser in **Section 4.9**. This is followed by a test case - Eugenia and the results of static analyser on Eugenia in **Section 4.10**. Finally state-of-the-art static analysis tools are presented in **Section 4.11**.

This chapter provides the foundation of the static analyser, which will be heavily used in other chapters as a pre-processing step before the optimisation.

Chapter 5

Optimistation of Queries over EMF Models

Model querying is an essential part of automated model management activities, such as model-to-model, model-to-text transformation and model validation. Queries on models can be specified using general-purpose programming languages such as Java or using tailored model-management languages such as Object Constraint Language (OCL) – and its various flavours embedded in model-to-model and model-to-text transformation languages such as Acceleo and ATL – the Epsilon Object Language (EOL) and the task-specific languages that build on top of it, and the VIATRA Query Language (VQL). The main strength of dedicated model management languages is that they offer built-in abstractions for common tasks (e.g. rule-based decomposition and element resolution in model-to-model transformation, protected regions for mixing generated and hand-written content in model-to-text transformation, constraint dependency management in model validation) which facilitate more concise, maintainable and technology-independent model management programs.

The work presented in this chapter has been published titled as *"Identification and Optimisation of Type-Level Model Queries"* and *"Towards Scalable Validation of Low-Code System Models: Mapping EVL to VIATRA Patterns"* in 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)

In this chapter, an architecture is introduced for improving the execution speed of interpreted model management programs written in languages of the Epsilon platform, using static analysis and program rewriting techniques. Then an application of this architecture is demonstrated for detecting repeated queries on all instances of types in EMF-based models and for speeding-up their execution through the construction of relevant indices. The proposed optimisation technique is evaluated on large models that have been reverse-engineered from Java code and a set of existing constraints, and performance improvements of up to 99.56% have been observed.

Secondly, an approach is presented for automatically mapping expressions in Epsilon validation programs to VIATRA graph patterns to make the validation of large-scale low-code system models scalable by leveraging the incremental execution engine of VIATRA. Finally, the performance of the proposed approach is evaluated on large Java models of the Eclipse source code. The results show performance speed-up up to 1481x compared to the sequential execution in Epsilon. This work has been presented as workshop paper [94].

5.1 Motivating Example

Consider a scenario where one queries a model for validating a UML model conforming to the UML2 [95] EMF-based metamodel. Suppose one wishes to check that:

- The names of all classes in the model are unique
- All class methods are called in at least one sequence diagram

The relevant subset of the UML2 metamodel and implementations of the two constraints (using the Epsilon Validation Language) are illustrated in Figure 5.1 and in Listing 5.1 respectively. These constraints can be written in any similarly expressive model management language. In Listing 5.1, the *UniqueName* constraint checks that for every *Class* in the model, its name attribute is unique (lines 9-14). Similarly, *IsCalledInSequenceDiagram* constraint checks that every *Operation* is called in a sequence diagram at least once (lines 16-21).

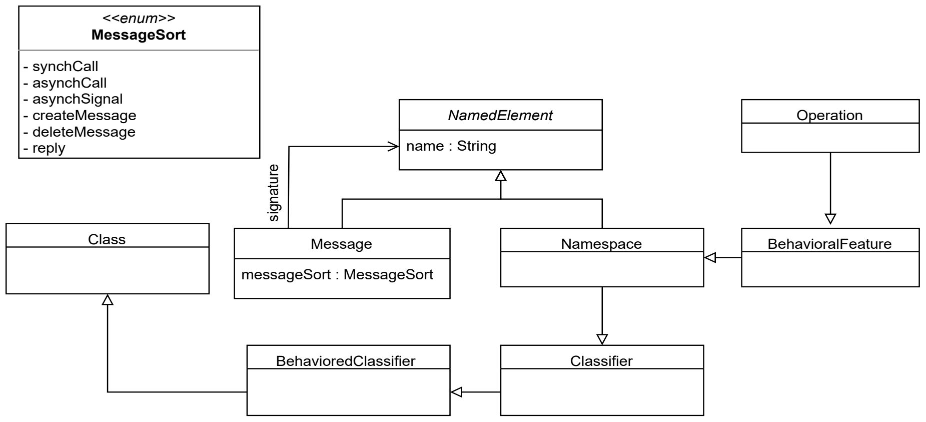


Figure 5.1: An excerpt of the UML2 metamodel

```
model UML driver EMF {
  nsuri =  "http://www.eclipse.org
    /uml2/5.0.0/UML"
};
```

```
5
6  pre {
7  }
8
9  context Class {
10   constraint UniqueName {
11     check: not Class.all.exists
12       (c|c.name = self.name and self != c)
13     }
14 }
15
16 context Operation {
17   constraint IsCalledInSequenceDiagram {
18     check: Message.all.exists
19       (m | m.signature = self)
20     }
21 }
```

Listing 5.1: Example EVL validation constraints before optimisation

Epsilon languages use *all* as an alias for *allInstances()*. *allInstances()* is an operation which can be invoked on a type to return a set containing all its instances. In Listing 5.1, *Class.all* in line 10 and *Message.all* in line 17 retrieve all instances of *Class* and *Message* anywhere in the model respectively. Evaluating these constraints over a UML model containing a large number of classes and operations would be computationally expensive. More specifically, the complexity of *UniqueName* constraint is *O(N*N)* if the number of Classes is considered to be *N*, and the complexity of evaluating *IsCalledInSequenceDiagram* over *M* operations and *P* messages would be *O(M*P)*. Reverse reference navigation is a recurring issue in model management programs [96] when working with EMF models. For example in UML model, navigating from *Operation* to *Message*. A common workaround to reduce complexity in such occasions is to define opposite references (e.g one could define an opposite reference from NamedElement to Message) however this pollutes the metamodel and in the case of standardized (immutable) meta models (e.g. such as the UML2 metamodel used in this example) adding opposite references is not an option. Moreover, one has to either anticipate the needs of future model management programs when constructing the metamodel or to naively add opposites for all references in the metamodel.

```
model UML driver EMF {
  nsuri = "http://www.eclipse.org
    /uml2/5.0.0/UML"
};

pre {
  UML.createIndex("Class", "name");
  UML.createIndex("Message", "signature");
```

```
 9 }
10
11 context Class {
12   constraint UniqueName {
13       check: not UML.findByIndex
14       ("Class", "name", self.name)
15       .select(c|c.self != c).size() > 0
16     }
17 }
18
19 context Operation {
20   constraint IsCalledInSequenceDiagram {
21     check: UML.findByIndex("Message",
22       "signature", self).size() > 0
23     }
24 }
```

Listing 5.2: Example EVL validation constraints after optimisation

To speed up this type of model validation, one optimisation strategy is to programmatically create in-memory indices and then use them for look-ups. Existing languages such as Acceleo offer different facilities for this e.g., search for *eInverse* [97]. Another approach for OCL is shown in [96]. This can significantly reduce the complexity compared to the naive iteration through all instances of the relevant model element types. Such an optimised validation program is depicted in Listing 5.2. These constraints are semantically equivalent to the ones in Listing 5.1 but are much faster to execute. In Line 7 of Listing 5.2, an index is constructed which maps names to lists of classes with their name attribute, rather than naively iterating through all the instances of *Class*. Similarly, in Line 8, an index is constructed which maps names to lists of messages with their signature attribute, rather than naively iterating through all the instances of *Message*. Then in constraints, these constructed in-memory indices (Lines 13-15, 21-22) are searched instead. As in-memory indices can be stored as hashmaps, finding UML classes by names and similarly finding messages by signature, the computation cost would be that of a hash function. Considering the complexity of hash functions being *O(1)*, the overall complexity of both the constraints would be reduced to *O(N)* and *O(M)*, respectively.

This chapter provides an approach for detecting optimisation opportunities such as the ones shown in the above example and then automatically rewriting relevant model management programs accordingly. This research focuses on investigating how such optimisations can be performed behind the scenes, using static analysis and automated program rewriting so that developers can express model management programs in a naive form (as in Listing 5.1) and benefit from index-based optimisation (as in Listing 5.2) as seamlessly as possible.

5.2 Optimisation of Type Level Model Queries

This section discusses the proposed query optimisation architecture in detail, an overview of which is illustrated in Figure 5.2. This approach takes a model management program as input and passes it through a static analyser component to compute an abstract syntax graph. The abstract syntax graph (type-resolved abstract syntax tree) is input to the query optimiser block, which outputs the rewritten optimised program to be executed.

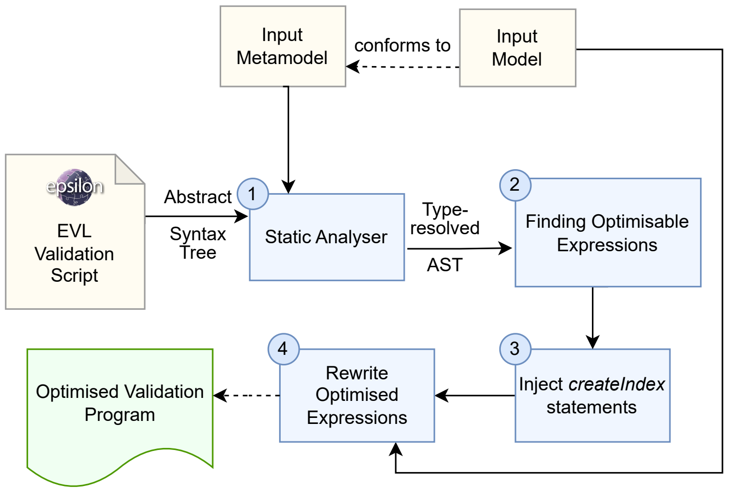


Figure 5.2: Architecture of the Query Optimisation Approach

5.2.1 Static Analysis

Static analysis is the first step of the proposed query optimisation approach. An overview of the process of static analysis is detailed in Chapter 4. Beyond the typical activity of checking the program for type-related errors and warnings, static analysis is useful for extracting information useful for program optimisation. This information mainly includes type information of elements and control flow of the program.

Consider Listing 5.1 to see how type resolution works. *Class* in line 8 would be resolved to the respective model element type of the UML model. In line 11, *c* and *self* are variables and are inferred to be of type *Class*, as *Class.all* is a collection of Classes. Hence, *c.name* and *self.name* would be resolved to be of String type. Overall the resolved type of the expression in the *check* part of the constraint *UniqueName* in Line 9-12 is boolean. Similarly for the second constraint *IsCalledInSequenceDiagram*, *UML!Operation* in line 15 would be resolved to the *Operation* model element type. *Message.all* in line 17 returns a collection of *Messages* and therefore the type of *m* is inferred as *Message*. Similarly, in line 18, the types of *self* and *m.signature* are resolved to *Operation*.

5.2.2 Finding Optimisable Queries

The second step of the approach is to find potential opportunities for speeding up queries using indices. The query optimiser operates over programs that consume EMF models. It is worth noting that the indexing approach is currently only for EMF models but can be extended to cover other types of models (e.g. Simulink) as well as it is possible to look up elements by feature values using built-in indices maintained by the modelling tool. The first step of the process is to find potential indices by visiting the entire program. The static analyser detects where the user is retrieving all instances of a type, filtered by a specific property or attribute, then only for such properties will indices be created. This approach works by detecting expressions in the form of *Class.all.operation(...)* to optimise. It currently supports filtering operations like *select*, *selectOne* and *exists*, while it can be extended to support other first-order operations as well, as discussed in the Further Work section in Chapter 8. As all Epsilon languages are built on top of EOL, certain expressions in an Epsilon program may be executed just once such as Line 9 in Listing 5.6. For such expressions the overhead of the computation of indices would not pay off if that index is to only be used once. To tackle this issue, one needs to find the expressions that are likely to be executed multiple times in a script. Algorithm 1 is used to carry out call graph analysis and identify such expressions. *optimiseBlock* method is a recursive method and calls *optimiseStatement* for every atomic statement. Finally *optimiseStatement* method checks if the statement is optimisable or not and then added to a list of potential indices.

The condition expression in the detected first-order expressions, which can be executed multiple times, can have logical operators. The condition expression abstract syntax graph is decomposed into each logical operand and then indexed separately based on the type of logical operator. Indexing for “and” and “or” logical operator conditions for expression in Listing 5.3 is shown in Listing 5.4.

```
UML!Class.all.select(c|c.name = "ClassA" or c.
    visibility = "public");

UML!Class.all.select(c|c.name = "ClassA" and c.
    visibility = "public");

UML!Class.all.select(c|c.name = "ClassA" and c.name =
    "ClassB");

UML!Class.all.select(c|isPublic(c) and c.name = "
    ClassA");

UML!Class.all.select(c|c.visibility = "Public" and not
     c.returnType = null);
```

Listing 5.3: Example logical operator expressions

```
UML.findByIndex("Class","name","ClassA").includingAll(
    UML.findByIndex("Class","visibility","public"));

UML.findByIndex("Class", "name","ClassA").select(c|c.
    visibility ="public");

UML.findByIndex("Class", "name","ClassA").includingAll
    (UML.findByIndex("Class","name","ClassB");

UML!Class.all.select(c|isPublic(c) and c.name = "
    ClassA");

UML.findByIndex("Class", "visibility","Public").select
    (c|not c.returnType = null)
```

Listing 5.4: Rewritten logical operator expressions

A call graph is a control flow graph representing the operation calls from the program's entry point(s) and within each operation. The call graph's vertices (nodes) represent the operations, starting from the program's entry point(s) and then the hierarchy of how other operations are called. Each edge *(x,y)* indicates that operation x calls operation y. The edge labels capture if an operation is called from a loop or not. If an operation is called from a *for* or a *while* loop or from a first-order operation call (e.g. *select*, *collect*, *reject*, *exists*). Since in Epsilon programs context type and parameter type polymorphism is supported, it can be challenging to understand which operation would be called at runtime. This is handled through type resolution and inference achieved through static analysis, which has been explained in detail in Section 5.2.1. The type-resolved AST is then used to identify the most exact match operation for every operation call. Consider the following example:

```
var a : Class = Class.all.first();
a.printName();

operation Message printName() {
  return self.name.println("Message Name:");;
}

operation Class printName() {
  return self.name.println("Class Name:");
}

operation Any printName(){
  return self.name.println("Name:");
}
```

Listing 5.5: Context type polymorphism example

There are three operations with same name *printName()* but the context type is different for each operation. In line 2, the second *printName()* operation will be called.

The type-resolved AST is passed to a call graph generator component, which generates the input program's call graph. An example of a call graph generated from Listing 5.6 is shown in Figure 5.3. Call graphs for EOL programs can be visualised on the fly using Graphviz[1] through Picto [98].

```
model UML driver EMF {
  nsuri = "http://www.eclipse.org/uml2/5.0.0/UML"
};

printMessagesofReplySort();
getClassByName("A").println("Class A:");

operation getClassByName(name: String) {
  Class.all.select(a|a.name = name);
}

operation printMessagesofReplySort() {
  for(n in Message.all.select(m|m.messageSort =
      MessageSort#reply)) {
    getMessageByName(n.name).println();
  }
}

operation getMessageByName(name: String) {
  Message.all.select(a|a.name = name);
}
```

Listing 5.6: Example of call graph program input

For EVL programs, the call graph generator considers the expressions in the *check*, *guard* and *message* block of constraints as being called from a loop. This is because in EVL constraints are often evaluated over a Context (instances of a model element) and hence such expressions are to be considered as candidates for potential indices. Also, the operation calls from within constraints are considered as being made from a loop.

5.2.3 Query Rewriting

After collecting potential indices i.e., class-feature pairs, by analysing the input program in the first phase, the final phase is to rewrite the program. The program is traversed again to find expressions which can leverage the created in-memory indices. This rewriting is performed behind the scenes: it does not alter the behaviour of the original program nor is it visible to the user (unless

[1]https://www.graphviz.org

Algorithm 1 Algorithm for Finding Potential Indices

```
 1: let model = current model rewriter (separate rewriters for every model)
 2: let inLoop = false
 3: let allOperations = all, allInstances
 4: let optimisableOperations = select, exists
 5: let callGraph = call graph of the input program
        optimiseBlock(main statement block)
 6: procedure optimiseBlock(StatementBlock)
 7:     for all statement s in StatementBlock do
 8:         if s is a ForStatement or WhileStatement then
 9:             inLoop = true optimiseBlock(body of s)
10:         else if
          then
11:             visit every DOM element recursively
12:         else
        optimiseStatement(s)
13:         end if
14:     end for
15: end procedure
16: procedure optimiseStatement(Statement)
17:     if s is an OperationCallExpression then
18:         repeat
         optimiseStatement(target of s)
19:         until targetExpression is instance of NameExpression
20:         for all Parameters of s do
21:             repeat
         optimiseStatement(parameterExpression)
22:             until parameterExpression is instance of NameExpression
23:         end for
24:     end if
25:     if s is an FirstOrderOperationCallExpression then
26:         if target of s is a PropertyCallExpression or OperationCallExpression
    then
27:             if allOperations contains name of target then
28:                 if optimisableOperations contains operationName of s then
29:                     if target of propertyCallExpression is owned by model
    then
30:                         if inLoop then
31:                             add to Potential Indices
32:                         end if
33:                     end if
34:                 end if
35:             end if
36:         end if
37:     end if
38: end procedure
```

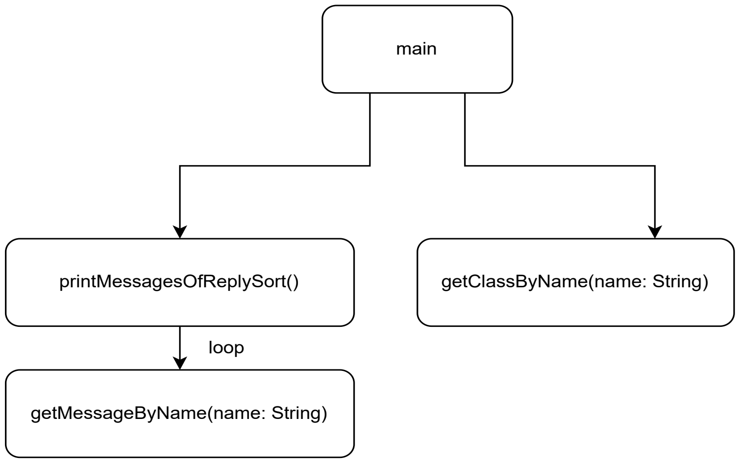

main
printMessagesOfReplySort()
getClassByName(name: String)
loop
getMessageByName(name: String)

Figure 5.3: Generated Call Graph (from Listing 5.6)

```
39: for all op in getDeclaredOperations do
40:     if path p from main to op exists then
41:         if p contains an edge labelled as loop then
42:             inLoop= true
43:         end if
        optimiseBlock(body of op)
44:     end if
45: end for
```

they wish to see it in which case there is a dedicated Eclipse view for this, detailed below). Rewriting includes two main tasks: i) Injecting *createIndex* statements for creating in-memory indices, ii) Rewriting the relevant expressions to *findByIndex* statements, where these indices are used. The respective syntax of *createIndex()* and *findByIndex()* statements is showcased in Listings 5.7 and 5.8.

```
ModelName.createIndex("ModelElement", "property");
```

Listing 5.7: Syntax of *createIndex()* Statement

```
ModelName.findByIndex("ModelElement", "property",
  "value");
```

Listing 5.8: Syntax of *findByIndex()* Statement

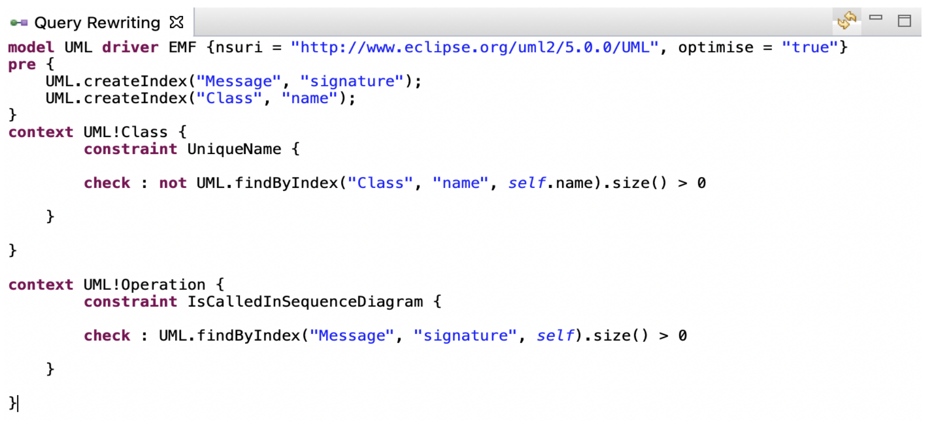
Calls to *createIndex* statements are injected at the beginning of an EOL program, for creating in-memory indices. The target expression *ModelName* is the name of the model for which an index is to be created. *ModelElement* is the metaclass, while *property* is the name of the feature based on which *allInstances* are filtered. For an EVL program, these statements are injected into a *pre* block, which contains EOL statements to be executed before evaluating the constraints themselves.

Next, *findByIndex* statements are injected in the AST of the EOL/EVL program, for searching model element instances through their respective indices, replacing the naive iteration code that would have otherwise been executed. The target expression *ModelName* is the name of the model in which *ModelElement* belongs. *Property* is the index that should be traversed, and the *value* represents the value of the property that needs to be searched. When rewriting the AST to *findByIndex* statements, any expressions that can make use of the available indices are rewritten, even if those expressions are not detected to be executed multiple times by call graph analysis. This is done to reuse the established in-memory indices in the entire program, for reducing the program execution time. Consider an example scenario for such a case as shown below:

```
1 var c2= Class.all.select(c|c.name = "c2"
2 and c.visibility = "private");
3 while (condition) {
4   var c2= Class.all.selectOne(c|c.name = "c2"); }
```

Since an index *class.name* will be created due to line 4, it will be used to rewrite the statement in line 1 to take avdantage of the re-writing.

Rewriting is performed behind-the-scenes, before the execution of the program. The original lines and column coordinates of ASTs are maintained, so that if exceptions occur at runtime, they are reported at the correct location in the original program. If the user wishes to visualise this automated program rewriting, a query rewriting view is implemented as shown in Figure 5.4, which displays the rewritten program of the EOL or EVL file in the currently active editor.



```
model UML driver EMF {nsuri = "http://www.eclipse.org/uml2/5.0.0/UML", optimise = "true"}
pre {
    UML.createIndex("Message", "signature");
    UML.createIndex("Class", "name");
}
context UML!Class {
        constraint UniqueName {

        check : not UML.findByIndex("Class", "name", self.name).size() > 0

    }

}

context UML!Operation {
        constraint IsCalledInSequenceDiagram {

        check : UML.findByIndex("Message", "signature", self).size() > 0

    }

}
```

Figure 5.4: Screenshot of the Query Rewriting View

5.2.4 Evaluation

This section presents the experimental setup used for evaluating the static analysis based query optimisation approach, explains the methodology employed and discusses the results obtained. It concludes by presenting the limitations and threats to the validity of the obtained results.

Table 5.1: Specifications of java models used for benchmarking

ID	Model Name	No of Model Elements	Size in MBs
1	eclipseModel-0.1	100,126	24.5
2	eclipseModel-0.2	200,224	50.8
3	eclipseModel-0.5	500,510	131.8
4	eclipseModel-1.0	1,000,658	258.3
5	eclipseModel-1.5	1,500,304	410.3
6	eclipseModel-2.0	2,000,329	555.7
7	eclipseModel-2.5	2,500,194	698.2
8	eclipseModel-3.0	3,000,159	948.5
9	eclipseModel-3.5	3,500,107	1080.0
10	eclipseModel-4.0	4,000,426	1110.0
11	eclipseModel-all	4,357,774	1210.0

The execution-time performance of the proposed approach to optimise EVL programs over large-scale EMF models has been evaluated. Since Epsilon supports parallel execution [99] for EVL, the proposed approach is compared with the parallel mode of EVL execution. The first experiment evaluates the constraints using EVL without optimisation with parallel mode enabled. The second evaluates the use of the proposed rewriting strategy (with an extension of EMF EMC driver with two additional *createIndex* and *findByIndex* methods discussed in Section 5.2.3) also in the parallel mode of EVL. In the rest of the section, the first approach is referred to as EVL– since it executes the EVL programs in a naive parallel mode, while the second one is referred to as EVL-QR – since it makes use of the query rewriting strategy, on the top of the EVL engine in parallel mode. Then comparison of the results are presented as speedups compared to OCL. For OCL evaluation, the same *Java_findBugs* in OCL is rewritten and the execution time is reported.

Constraints and Models: For evaluating the query optimisation approach, the validation constraints that were used were introduced in [99] and are based on the Findbugs [100] project, a static analysis tool that reports a large number of "code smells" in Java code. The EVL script (*Java_findBugs*) consists of 31 constraints over 17 contexts, and 11 operations. *Java_findBugs*

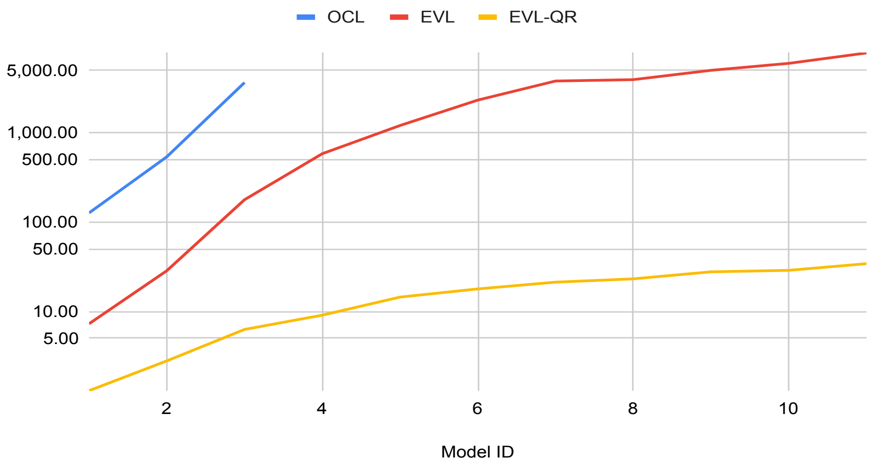


Figure 5.5: Comparison of OCL, EVL and EVL QR

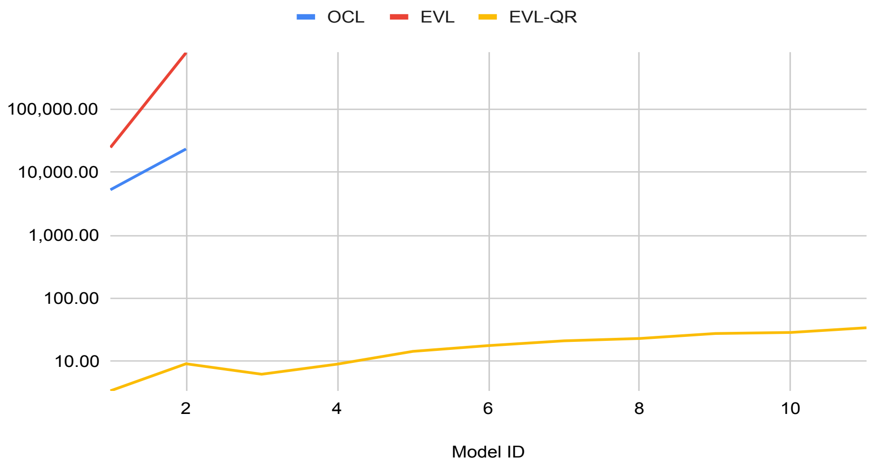


Figure 5.6: Comparison of OCL, EVL and EVL QR without eOpposites

script is executed over a set of large models reverse-engineered from the Java source code of Eclipse projects [101] using MoDisco [102]. The Modisco Java metamodel was opted, as it is both complex enough and relatively familiar to Java programmers. Also, such reverse engineered models are commonly used to evaluate the scalability of MDE tools [103,104]. The models that were used vary from approximately 100k to over 4 million elements, as illustrated in Table 5.1.

Correctness: The program is rewritten by the query rewriter, so it is essential to check that this rewritten program is semantically equivalent to the original input program. Correctness of the results has been verified through automated JUnit tests, ensuring that query results are the same in EVL and in EVL-QR. For that, several test EOL and EVL scripts were executed which were mined from GitHub and compared the outputs of both programs. No differences were found in outputs given by the input (original) and the rewritten programs. For the main test case *Java_findBugs*, the number of unsatisfied constraints were matched for OCL,EVL and EVL-QR. After running the correctness tests, the semantic equivalence of the rewritten programs is ensured and hence of the query rewriter logic used in this approach.

Machine Specification: The evaluation experiments were performed on a machine with the following specifications: MacBookPro @ 2.8 GHz Quad-Core Intel Core i7, 16 GBs of RAM, Mac operating system BigSur version 11.1, and Java 15 on JDK 15.0.2 with JVM MaxHeapSize 4GBs.

5.2.4.1 Results

The computation time taken for the static analysis and query rewriting processes has been measured to assess the overhead they incur. Then, the execution time of the program itself is recorded, as this approach does not interact with model loading and thus has no effect on model loading times. The script is executed using Epsilon in a standalone manner and the execution time is measured using Epsilon's profiling capabilities. The measured program execution times are reported in milliseconds in Table 5.2 and Table 5.3.

Static analysis and query rewriting work at the metamodel level and do not require any information from models themselves. Static analysis and program rewriting took less than 50ms for all the experiments, and therefore the overhead incurred can be seen as negligible, with respect to the overall execution times observed for these experiments. Also, this computation time is independent of model size, due to the fact that the whole process of query optimisation only uses metamodel introspection. Time for static analysis and query rewriting depends on two major factors: the size of the program under consideration and the size of the underlying model's metamodel. To investigate the most computationally expensive constraints, the distribution of overall execution times of the validation program will be measured. This program will be divided into two parts and reported execution times for the first constraint as *FindBugs_First* and then execution times of the remainder of the constraints as *FindBugs_Rest* in Table 5.4. Out of 31 constraints in the *Java_FindBugs* script, the first constraint named *allImportsAreUsed* is the most expensive one, as illustrated in

Table 5.2: Execution time in seconds

Model ID	OCL	EVL	EVL-QR	vs EVL	vs OCL
1	126.9	7.29	1.3	5.6	97.61
2	538.7	28.6	2.8	10.21	192.39
3	3649.5	178.3	6.3	28.46	579.28
4	TO	583.5	9.1	64.1	-
5	TO	1206.7	14.5	83.22	-
6	TO	2322.6	17.9	129.75	-
7	TO	3797.4	21.3	178.28	-
8	TO	3934.9	23.2	169.60	-
9	TO	5003.6	27.8	179.98	-
10	TO	5987.3	28.9	207.17	-
11	TO	7826.9	34.3	228.18	-

Table 5.3: Execution time in seconds (without eOpposites)

Model ID	OCL	EVL	EVL-QR	vs EVL	vs OCL
1	5256	24579	3.44	7145	1527.9
2	23263	777997	9.22	84564	2522.9
3	TO	TO	6.3	-	-
4	TO	TO	9.1	-	-
5	TO	TO	14.5	-	-
6	TO	TO	17.9	-	-
7	TO	TO	21.3	-	-
8	TO	TO	23.2	-	-
9	TO	TO	27.8	-	-
10	TO	TO	28.9	-	-
11	TO	TO	34.3	-	-

Table 5.4. *allImportsAreUsed* being very demanding, takes 99% of the execution time and it contains an expression that is optimisable using the proposed approach. Due to this, in the case of EVL Query rewriting significant improvement in performance can be seen, by just creating one index.

Table 5.4: Distribution of Execution Time in FindBugs script

Model ID	**FindBugs_First**	**FindBugs_Rest**	**FindBugs_All**
1	5,94	1,351	7,29
2	26,40	2,243	28,64
3	175,11	4,265	179,38
4	576,13	7,451	583,59
5	1 196,81	9,930	1 206,74
6	2 310,47	12,181	2 322,65
7	3 780,41	17,020	3 797,44
8	3 916,61	18,370	3 934,98
9	4 985,02	18,665	5 003,69
10	5 966,36	21,003	5 987,37
11	7 798,84	28,066	7 826,90
Average %	99.54	0.46	100

Observing the comparison graph shown in Figure 5.5, it can be seen that EVL with query rewriting is substantially more performant than EVL. In a naive EVL execution, as the model size grows, the execution time increases non-linearly, in this case from about 7 seconds to 130 minutes for models with 100k elements to 4.35M elements, respectively (a three order of magnitude increase, for models of around one order of magnitude in variance). In comparison with EVL, EVL-QR speeds up the validation by 5.6x for the smallest model and 228.18x for the largest model. While in comparison with OCL, EVL-QR speeds up the validation by 97.6x for the smallest model and upto 579.2x and even more for larger models where OCL timed out. This gives confidence that the proposed query rewriting approach is scalable and efficient for very large models. Overall, these results illustrate that automated query rewriting can have performance benefits both for small and large models. The results also suggest that the larger the model size, the more the performance gain is in terms of execution time. This can be explained by the fact that in smaller models, the overhead of creating indices is proportionally larger, than for larger models.

Experiments were performed on the same validation constraints after removing the opposite references from the Java metamodel. The reason for removing opposites is to create more room for optimisations as sometimes adding opposite references is not an option, such as for standardised metamodels. The original Java metamodel has 173 references in total out of which 48 have opposite references. The opposites were removed on the following criteria: if a reference is containment, remove its opposite reference. if a reference is non-containment, then remove one of the pairs of opposites based on alphabetical order. In total, 23 opposite references were removed, which leaves 150 references in the modified metamodel. Model migration was then carried out to update the original models to conform to the new metamodel without opposites using Flock [55]. EVL validation constraints and OCL constraints were also updated to not make use of the removed opposite references. For example, *VariableDeclaration* class had an opposite reference to *SingleVariableAccess*. The *variableIsUsed* constraint is written originally as:

```
  context VariableDeclaration {
    constraint variableIsUsed {
      check: self.usageInVariableAccess.notEmpty()
  }
}
```

After removing the opposite reference the constraint is changed and rewritten as:

```
  context Java!VariableDeclaration {
    constraint variableIsUsed {
      check: Java!SingleVariableAccess.all
      .select(sva|sva.variable=self).notEmpty()
  }
}
```

This experiment has a goal to measure the performance of query optimisation when having opposites (to speed up certain classes of queries) is not possible. The graph shown in Figure 5.6 illustrates the comparison between EVL and EVL-QR with no opposite references in the model. Validation with EVL is so computationally expensive in this case, that it timed out(TO) even for models with around 500K elements. Utilising EVL-QR, shows a performance gain of over 84564x in comparison with EVL for the experiments that were completed. EVL-QR provides a performance gain of over 2522x while comparing with OCL. When there are opposite references, there is still more room for creating in-memory indices and thus reducing the execution time overall, as some queries may have to keep navigating through the entire model to find matching elements (that could have otherwise been navigable through an opposite reference).

5.2.4.2 Threats to validity

This experiment uses one metamodel and one set of increasingly large models conforming to it. While both the models and metamodel were not specifically targeted for any other reasons other than availability and ease of understanding (as well as offering model sizes that are both large enough and not synthesized), it is possible that they play a large role in determining the results obtained. The proposed query optimisation approach can benefit from experiments performed on more diverse models with a broader range of sizes and more complex constraints, both for investigating semantic equivalence and performance. Creating in-memory indices naturally has an added overhead in the execution time, which is handled by call graph analysis at the program and metamodel level. Another possible threat to the validity of these experiments, is the addition of possibly substantial overheads when evaluating large enough programs or metamodels. For example, if a constraint is evaluated over a context with one or very few elements then indexing attributes from the respective check block can incur additional overhead. It is observed that for large enough models, whereby this approach offers the most benefits, this is very unlikely to be the case. Also, to ensure more accurate static analysis and thus enable efficient program rewriting, it is recommended to use a more strict coding style and explicitly declare types, and avoiding *Any* type as much as possible for accurate type resolution. Finally, it is worth noting that the model management program used for this benchmarking is limited to read-only operations. Since EOL offers model manipulation it would be worth investigating programs that change the model, to ensure there are no unforeseen consequences of the proposed approach there.

5.3 Incremental Querying using Viatra

In this section, a method is proposed to improve the performance (primarily by reducing the execution time) of model validations by mapping OCL-like expressions embedded in Epsilon validation constraints to graph patterns. A prototype is implemented demonstrating the proposed solution by mapping Epsilon Validation Language (EVL) to VIATRA graph patterns. The implementation of the aforementioned tool is open-source and available on GitHub [105]. Moreover, the performance of the solution is measured on validation rules from the Findbugs validation suite [106]. The main goal of the proposed approach is to reuse the static analysis by exploring the query translation optimisation to improve performance reducing the execution time. This approach is illustrated in Figure 5.7.

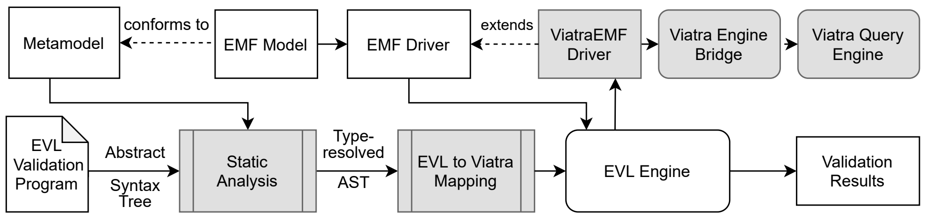


Figure 5.7: EVL to VIATRA mapping architecture

5.3.1 EVL to VIATRA Mapping

```
1  model Java driver ViatraEMF {
2    nsuri = "http://www.eclipse.org/MoDisco/
3      Java/0.2.incubation/java"
4  };
5
6  context Java!ImportDeclaration {
7    constraint allImportsAreUsed {
8      check: Java!NamedElement.all.exists(
9        ne|ne.originalCompilationUnit =
10       self.originalCompilationUnit and
11       ne.usagesInImports = self)
12   }
13 }
14
15 context Java!VariableDeclaration {
16   constraint variableIsUsed {
17     check: Java!SingleVariableAccess.all
18       .exists(sva|sva.variable = self)
19   }
20 }
21
22 context Java!CatchClause {
23   constraint exceptionIsUsed {
24     check: Java!SingleVariableAccess.all
25       .exists(sva|sva.variable =
26       self.exception)
27   }
28 }
```

Listing 5.9: Example EVL script before optimisation

The first step is the static analysis which is explained in Chapter 4. After the static analysis, a type-resolved AST is extracted. In the second step, the type-resolved AST is traversed sequentially and identify expressions that can be optimised. These expressions are in the form of first-order operations operating over *allInstances* of a type. In a check block of a constraint, each expression is evaluated over all elements of the context, thus first-order operations can be computationally expensive for large models. Therefore, these operations translate to VIATRA patterns as follows.

First, the namespace URI (NsUri) of an EPackage as specified in the program is extracted from the model declaration statement (lines 2-3 in Listing 5.9) and mapped to import statement in VQL (lines 1-2 in Listing 5.10). After that, the body of the first-order operations are translated to graph patterns. Names of the patterns should be unique, due to VQL naming conventions. Therefore, they are generated based on the model name concatenated with a sequence number, see line 4 in Listing 5.10.

```
1 import "http://www.eclipse.org/MoDisco/
2   Java/0.2.incubation/java"
3
4 pattern Java2(sva: SingleVariableAccess,
5   self0: VariableDeclaration){
6   SingleVariableAccess.variable(
7     sva, self0);
8 }
```

Listing 5.10: variableIsUsed's check block translated to a VQL pattern

In the operation body, operator expressions are translated consisting of property call expressions to graph patterns in VIATRA. Property call expressions define navigations in EOL, e.g., *sva.variable* in line 18 of Listing 5.9 reads the *variable* field of the *sva* object. In operator expressions, the name of the first operand is the name of the property on which the navigation should happen. The operator defines the comparison basis, and the value of the second operand is the value to be compared against. It can be either a literal value or a reference to the single-valued result of another EOL expression. In the latter case, the value is received as an additional pattern parameter. Model navigations are represented by declarative graph patterns in VQL, where the starting node is the type of the model element, and the edge is the property used in the property call expression. If the operator is an inequality operator, then an additional `check` constraint is generated in VQL whose body contains the inequality comparison. As an example, let's consider the EOL expression *sva.variable = self* in line 18 of Listing 5.9, that is translated to *SingleVariableAccess.variable(sva, self0)* in lines 6-7 of Listing 5.10 with *self0* being an additional pattern parameter in VQL (line 5 of Listing 5.10). In the proposed prototype implementation [105], only conjunctions of operator expressions are supported.

```
Java!NamedElement.all.select(u|u.name="main")
Java!NamedElement.all.selectOne(u|u.name="main")
Java!NamedElement.all.exists(u|u.name="main")
Java!NamedElement.all.one(u|u.name="main")
Java!NamedElement.all.none(u|u.name="main")
Java!NamedElement.all.count(u|u.name="main")
Java!NamedElement.all.nMatch(u|u.name="main",2)
Java!NamedElement.all.atLeastNMatch(u|u.name="main",1)
Java!NamedElement.all.atMostNMatch(u|u.name="main",1)
```

Listing 5.11: EOL first-order expressions-I

```
pattern Java1(namedElement: NamedElement){
  NamedElement.name(namedElement, "main");
}
```

Listing 5.12: Rewritten VIATRA first-order expressions-I

Table 5.5: Memory use of the engines (in MB)

Line number	**Matcher API call**
1	allMatches
2	oneArbitraryMatch
3	hasMatch
4	countMatches == 1
5	countMatches == 0
6	countMatches
7	countMatches == 2
8	countMatches ⟩= 1
9	countMatches ⟨= 1

```
Java!NamedElement.all.reject(u|u.name="main")
Java!NamedElement.all.forAll(u|u.name="main")
```

Listing 5.13: EOL first-order expressions-II

```
pattern Java1(namedElement: NamedElement) {
  neg find Java1internal(namedElement);
}

pattern Java1internal(namedElement: NamedElement) {
  NamedElement.name(namedElement, "main");
}
```

Listing 5.14: Rewritten VIATRA first-order expressions-II

Table 5.6: Memory use of the engines (in MB)

Line number	**Matcher API call**
1	allMatches
2	hasMatch == false

The operator expression in lines 17-18 of Listing 5.9 checks if the *variable* field of the Single Variable Access object is the Variable Declaration object (*self*) that is received as parameter. The property call expression is the navigation from *sva.variable*

As shown in Listing 5.12, most first-order operations (Listing 5.11) are translated to only one VQL pattern. However, in some cases, such as in the *reject* and *forAll* operations (Listing 5.13), there is a need to generate an additional pattern (Listing 5.14). In those cases, the matches of the negated expression are found, therefore the main pattern contains a negative invocation of the second pattern (`neg find`). Besides, the type of the first-order operation defines the method to be called on the Matcher API of VIATRA with some additional parameters used for comparing the result, e.g., countMatches == 0.

Finally, the body of the first-order operation is replaced by an operation call expression, encapsulating a Run Viatra Call Parameters object that contains: (*i*) the generated VQL patterns, (*i*) the name of the main VQL pattern that is used to collect the pattern matches, (*iii*) the name and the (*iv*) parameter of the method to be called on VIATRA's Matcher API, and (*v*) the EOL expressions representing the extra parameters of the patterns. The values of these expressions will be bound at runtime to the corresponding parameters of the patterns.

5.3.1.1 Collecting Validation Results

After the translation of optimizable EOL expressions to VQL patterns, the EVL engine iterates through the EVL program and evaluates the expressions. If it finds a translated EOL expression, then the *runViatra* method of the ViatraEMF driver is called, which calls the VIATRA Engine Bridge that prepares query specifications from the textual VQL patterns, obtains a matcher for the main specification and invokes the corresponding method with the appropriate parameters on the matcher. Finally, the found matches are returned to the EVL engine, which combines them with the matches from the unoptimized expressions and returns the validation results.

5.3.2 Evaluation

In order to measure the query execution time and memory use of the proposed approach three validation constraints were adopted (Listing 5.9) from the Findbugs validation suite [106] and evaluated them on the Java MoDisco EMF model of the Eclipse source code [106]. The incremental (RETE) and local search (LS) engines of VIATRA were compared with the sequential EVL engine. In the query evaluation phase, the models are already loaded in memory, and the EOL expressions are already translated to VQL. The query rewriting took 9 ms for all queries. The measurements were conducted on a machine with Windows 10, Intel i7-9750H CPU @ 2.60GHz, 32 GB RAM, Java HotSpot^{TM} 64-Bit Server VM 13.0.1+9 (with 16 GB max heap size).

Table 5.7: Execution time of the engines (in min:sec.milliseconds)

Model size	Query engine			
	RETE	LS with base index	LS without base index	Sequential EVL
100K	0.937	1.346	03:25.720	03:25.985
200K	1.766	2.488	14:43.912	15:55.617
500K	4.315	6.056	93:00.186	106:30.364

Table 5.8: Memory use of the engines (in MB)

Model size	Query engine			
	RETE	LS with base index	LS without base index	Sequential EVL
100K	94	92	88	49
200K	141	133	126	80
500K	284	268	243	183

5.3.2.1 Analysis of the Results

As Table 5.7 shows, the RETE engine provides the shortest execution time, due to the incremental caching of pattern matches which speeds-up the pattern evaluation process on large models. The local search engine with base index gives similar results, due to the caching of the base relations and objects in the model. The LS engine without base index and the sequential EVL engine were several magnitudes slower compared to the previous engines. The largest speed-up is 1481x between the RETE and the sequential EVL engine, in the case of models with 500k elements. The evaluation of the validation constraints varies with the model sizes. Due to the caching backend of the RETE engine, the repeated evaluation of the validation constraints on each model element results in a faster execution time. Similar to RETE, the local search (LS) engine with base index finishes second, due to caching of the base relations and model types. LS engine without base index and sequential EVL have to perform complete model traversals each time, therefore they complete slower.

Comparing the memory use of the engines in Table 5.8, it can be observed that the RETE engine consumes the most memory, while sequential EVL the least. Interestingly, the local search engine without base index consumes almost the same amount of memory as the engine with base index. This is because the base index is initialized in both cases, but in the first case the engine does not retrieve any object from the index.

5.3.2.2 Threats to Validity

Internal validity: The results reported in this chapter are computed on programs containing first-order operation calls on all instances of model elements. If there are no such optimisable operations and queries in the program, then the execution time is the same as in sequential EVL.

External validity: Opposed to the local search engine of VIATRA, the sequential Epsilon engine does not consider opposite edges in the metamodel when creating the search plan. Therefore, to have comparable results, the Java metamodel without these edges is used. Otherwise the local search engine would have performed similar to the RETE engine in both cases, due to the simplicity of the patterns.

5.4 Related Work

This section summarises existing work within the scope of this chapter, it discusses model query optimisation strategies. In Hawk [107], a derived attributes approach includes pre-computing certain expensive features and caching them in the model index. Results have shown a decrease in execution time by using derived attributes, but it has certain shortcomings as well. Firstly, it adds an overhead of computing these derived attributes, which increases the model insertion time containing derived attributes, as well as the overhead of updating the values of these features when the model changes. However, these attributes are defined by the user and to the best of our knowledge, there is no automatic detection of optimisation opportunities through static analysis such as the one proposed in this chapter. Another approach presented in [108] is to execute calls to *allInstances()* queries efficiently. This approach is based on greedy computation instead of on-demand computation. It checks if the program makes multiple calls to *allInstances()*, then precomputed all collections and caches them in one pass. The approach just works on *allInstances()* calls.

In [109], the authors present how combining three optimisation techniques (parallelisation, lazy evaluation, and short-circuiting) can significantly increase the performance of queries over large models. It requires the use of the parallel variant of EOL, which can be automated through static analysis, as is the case in the proposed approach. Parallelisation proposed in [109] is used as a comparison baseline for the proposed approach. In [110], a tool called Mogwai is proposed for efficient and scalable querying. Mogwai maps OCL and ATL expressions to Gremlin scripts – a query language for NoSQL databases. This leverages the optimisations implemented by the underlying database technology.

Now the use of query translation approaches will be discussed for optimization purposes. A solution for efficient querying large-scale databases is presented in [111], where OCL queries are translated to SQL at runtime. Heidenreich et al. proposed an approach for translating from OCL to multiple query languages like SQL and XQuery using model-to-text transformations [112]. These solutions work on relational database backends. Sanchez et al. proposed an approach for translating OCL queries to MATLAB commands for efficiently querying large Simulink models [113]. Bergmann et al. presented a mapping strategy from OCL to graph-based patterns [114]. In their approach, they map a subset of OCL expressions to EMF-IncQuery graph patterns.

The novelty of the EVL to VIATRA approach proposed in this chapter is adding partial incrementality by just translating a part of EVL program detected through static analysis. Only the expensive expressions are translated to corresponding VIATRA patterns. This is achieved as a trade-off between execution time and memory consumption.

5.5 Chapter Summary

This chapter presented the first two contributions of this research. First, it presented the optimisation of type level model queries by precomputing the custom indices that are used multiple times in the program. A motivation example was presented in **Section 5.1**, and then in **Section 5.2**, the proposed approach was explained step by step followed by the results and the analysis. Secondly, the second contribution which was optimisation of EOL expressions to Viatra to leverage the incrementality features provided by Viatra. The proposed approach was explained step by step followed by the results and the analysis in **Section 5.3**. This is followed by the internal and external threats to validity. Finally in **Section 5.4**, the state of the art in program rewriting is presented and also state of the art related to query translation.

Chapter 6

Selective Traceability of Model-to-Model Transformation Programs

This work has been published as a conference paper titled *"Selective Traceability for Rule-Based Model-to-Model Transformations"* 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22). In this chapter, a novel approach is proposed that leverages the benefits of static analysis and automated program rewriting to speed up and reduce the memory footprint of model-to-model transformation programs. In this study the Epsilon Transformation Language (ETL) is targeted, however, the proposed approach applies to any rule-based interpreted M2M language that supports imperative constructs. A map-like data structure is used to cache the results of imperative operations generated by rules as a transformation trace.

Model-to-model (M2M) transformation is one of the key activities used in a typical MDE workflow. It is essentially used to map one or more input model(s) to one or more output model(s). Various M2M languages like ETL [115] and ATL [42] provide tailored support for automating this task, but they can face scalability issues when it comes to transforming larger models [116]. The proposed approach involves statically analysing the M2M transformation, extracting type information of its various constructs and also extracting dependency information between the transformation rules as a dependency graph. Using the information extracted from the static analyser, a rule-based M2M program is then rewritten into an imperative M2M program, where the transformation rules are converted to operations. Moreover, exploiting the dependency graph allows reducing the global transformation trace into a selective trace, lowering its memory footprint. A key novelty of the proposed optimisation approach is that **it does not sacrifice any of the expressiveness** of the M2M language in contrast to e.g. [117], which only supports a subset of ATL. This is because the proposed approach performs in-place rewriting of rules and calls to *equivalent/equivalents()*, effectively desugaring a rule-based ETL transformation into an imperative form (still in ETL due to the potential presence of *pre/post* blocks). All other constructs of the transformation (e.g. method calls, property call expressions, user-defined operations, instantiation of native types) remain untouched and are executed using the standard ETL interpreter.

Using the proposed approach, performance gains up to 39% in terms of execution time and up to 59% in terms of memory consumption have been achieved in the evaluation experiments.

6.1 Motivating Example

Consider the example of a partial (for conciseness) *OO2DB* transformation. It describes the transformation of a model conforming to an object-oriented schema metamodel, as shown in Figure 6.1, into a model conforming to a relational database metamodel as shown in Figure 6.2. This transformation has been adapted from [118] and an excerpt is shown in Listing 6.1. The transformation contains four transformation rules:

- *Class2Table* to transform all the Classes in the object-oriented model to Tables in the database model;
- *SingleValuedAttribute2Column* to transform single-valued Attributes to Columns in the database model;
- *MultiValuedAttribute2Table* to transform multi-valued Attributes to Tables and foreign key Columns in the database model;
- *Reference2ForeignKey* to transform References in the object-oriented model to foreign key Columns in the database model.

ETL transformation keeps a transformation trace which contains tracelinks between source to target element. The size of the trace of the transformation (as shown in Listing 6.1), which relates source to target elements in the current implementation of the ETL execution engine will be *O+M+N*, if its is evaluated it over a source model containing *O* classes, *M* number attributes and *N* references.

```
model  Source driver EMF {
  nsuri="oo"
};

model  Target driver EMF {
  nsuri="db"
};

pre {
  var db : new Target!Database;
```

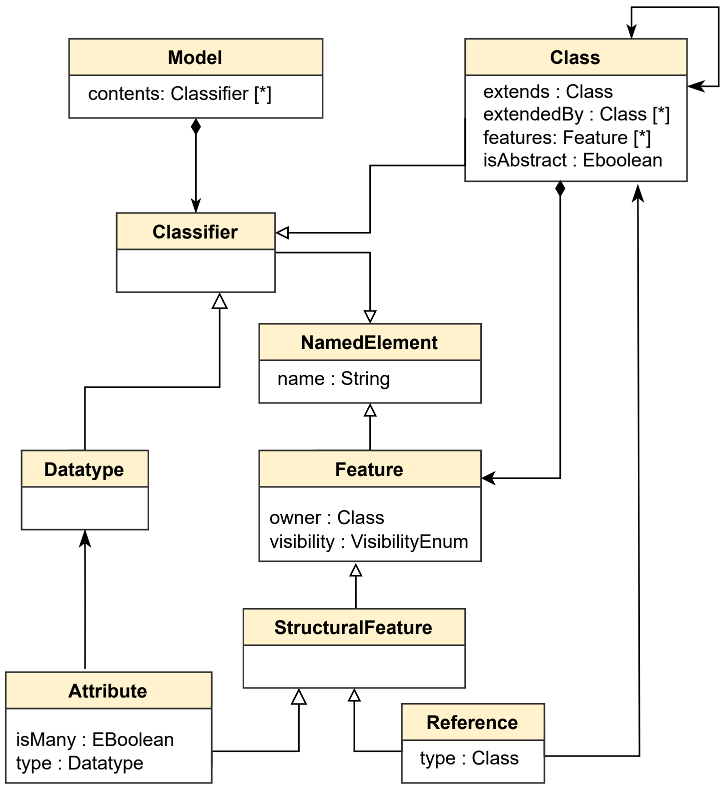


Figure 6.1: Object-oriented Metamodel

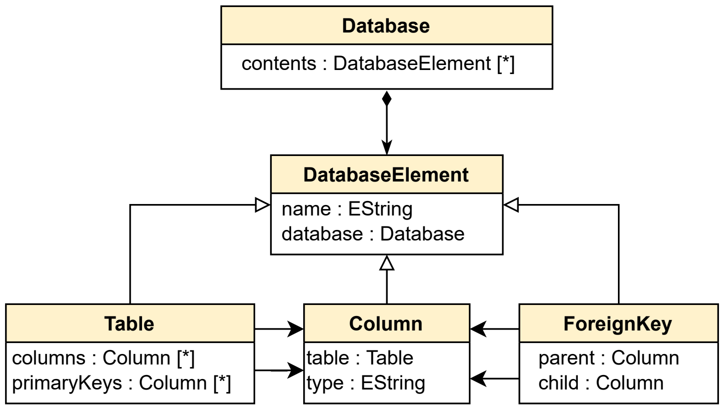


Figure 6.2: Database Schema Metamodel

```
}
rule Class2Table
transform c : Source!Class
to t : Target!Table{
  if (c.`extends`.isDefined()){
    var parentTable : Target!Table;
    parentTable =c.`extends`.equivalent();
  }
}

rule SingleValuedAttribute2Column
transform a : Source!Attribute
to c : Target!Column {
  guard : not a.isMany
  c.table = a.owner.equivalent();
}

rule MultiValuedAttribute2Table
transform a : Source!Attribute
to t : Target!Table,
fkCol : Target!Column {

  guard : a.isMany
  fkCol.table = a.owner.equivalent();
}

rule Reference2ForeignKey
transform r : Source!Reference
to fkCol : Target!Column {

  fkCol.table = r.type.equivalent();

}
```

Listing 6.1: Object-oriented 2 Database Transformation

However, at a closer look, in this *OO2DB* transformation, only trace links created by the rule *Class2Table* are needed by the other rules. The remaining trace links, created by the three other rules (*SingleValuedAttribute2Column*, *MultiValuedAttribute2Table* & *Reference2ForeignKey*) are not used anywhere in the transformation and therefore establishing and keeping them in memory is wasteful. This would reduce the size of the transformation trace to O.

The first contribution is an approach for reducing the memory footprint of the transformation trace by selectively tracing only pairs of source-target elements that may be needed elsewhere in the transformation.

This is achieved by computing a dependency graph between rules through static analysis and storing only the traces of a rule that are later needed by another rule. To resolve *equivalent()* operations (as in Lines 17, 25, 34 & 41 in Listing 6.1), the ETL engine normally triggers a lookup on a global transformation trace, ignoring the fact that the resulting target objects can only have been produced by specific rule(s), in this case the *Class2Table* rule. Hence, one possible optimisation is to benefit from static analysis, discovering which specific rule would provide the resulting target object instead.

The second contribution of this work is an approach for rewriting (desugaring) transformation programs in an imperative form, where rules are turned into operations, and calls to *equivalent/s()* are replaced with calls to appropriate transformation operations determined through static analysis.

6.2 Proposed Approach

In this section, the proposed approach is discussed for the efficient execution of rule-based model transformation programs using static analysis and automatic program rewriting. This approach is illustrated in Figure 6.3.

```
model  Source driver EMF {
  nsuri="oo"
};

model  Target driver EMF {
  nsuri="db"
};

pre {
  var db : new Target!Database;
  var cache_rule_Class2Table : Map;

for (c : Source!Class in Source!Class.all) {
      c.rule_Class2Table();
}

for (a : Source!Attribute in Source!Attribute.all) {
      a.rule_SingleValuedAttribute2Column();
}
}

operation Source!Class rule_Class2Table():Target!Table {
  if(cache_rule_Class2Table.containsKey(self))
    return cache_rule_Class2Table.get(self);
  var t : Target!Table = new Target!Table;
  t.name = self.name;
  t.database = db;
  if (c.`extends`.isDefined()){
    var parentTable : Target!Table;
    parentTable =c.`extends`.rule_Class2Table();
  }
  cache_rule_Class2Table.put(self, t);
  return t;
}

operation guardSingleValuedAttribute2Column(a : Source
    !Attribute) : Boolean {
  return not a.isMany;
}

operation Source!Attribute
    rule_SingleValuedAttribute2Column() : Collection {
  if (guardSingleValuedAttribute2Column(a)) {
    var c : Target!Column = new Target!Column;
    c.name = self.name;
    c.table = self.owner.rule_Class2Table();
    return Collection{c};
  }
}
```

Listing 6.2: Rewritten excerpt of the *OO2DB* Transformation

The proposed approach contains four main components, with a source metamodel, a source model, a target metamodel and a transformation being its inputs. The *Static Analyser*① component is given the model-to-model transformation and the source and the target metamodel(s), extracting the type information and yielding a type-resolved abstract syntax tree (AST). Then, using the *Dependency Graph Generator*②, the dependencies are extracted between the different transformation rules in the transformation: the dependency graph uses the type-resolved AST to populate these dependencies. Following this, there is a *Selective Tracer* to selectively create hash map caches to store the result of source target key-value pairs generated by corresponding operations and use them as a minimal transformation trace. In the next step, this dependency graph is passed to the *Rewriter*③, where the transformation is rewritten, i.e., the transformation rules are converted to the corresponding operations to optimise the resolution of elements by other rules. Finally, the rewritten optimised transformation is passed to the *ETL Engine*④ for execution. ETL Engine is the default engine already provided by Epsilon. The default engine is extended to provide the resolution of operation calls to their corresponding user-defined methods mapping provided by the static analyser.

6.2.1 Static Analysis

In the first step of this approach, the ETL transformation program is parsed into an Abstract Syntax Tree (AST). The static analyser yields a type-resolved

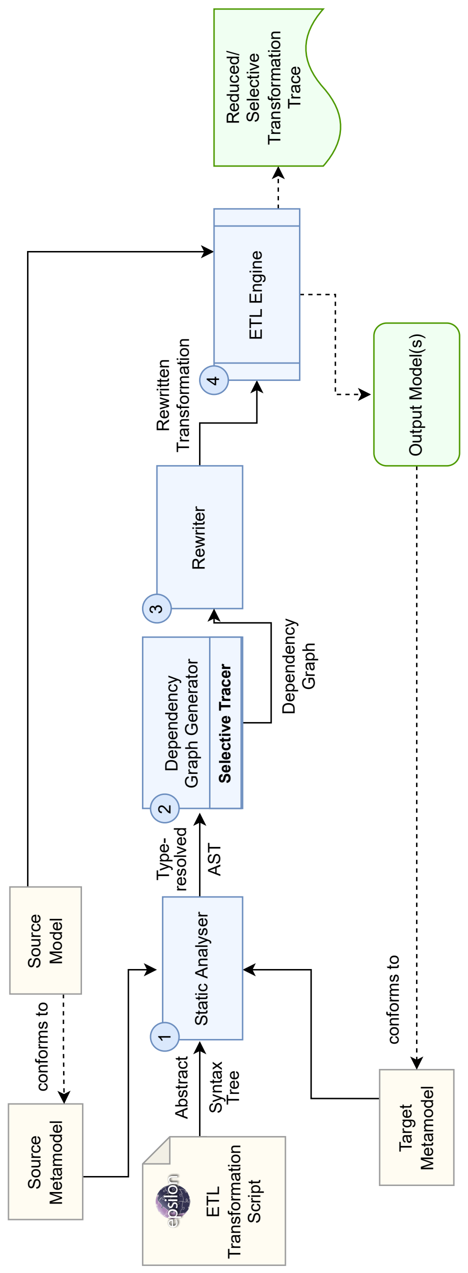


Figure 6.3: An overview of the proposed approach

AST (an AST augmented with the computed types of expressions), using metamodel introspection and type inference, as shown in Table 6.1 for the example of Listing 6.1. Epsilon programs define configuration details of the models they access using model declaration statements (Lines 1-3 & Lines 5-7 in Listing 6.1) which are then used by the static analyser to retrieve the available types and typed properties in each model. This static analyser extends the EOL one by including support for analysing expressions inside transformation rules, their source and target parameters, and for pre and post blocks.

Table 6.1: Resolved types of various constructs in Listing 6.1

Line#	**Expression**	**Resolved Type**
10	db	Target!Database
13	c	Source!Class
14	t	Target!Table
22	a	Source!Attribute
23	c	Target!Column
24	not a.isMany	Boolean
25	c.table	Target!Table
25	a.owner	Source!Class
29	a	Source!Attribute
30	t	Target!Table
31	fkCol	Target!Column
33	a.isMany	Boolean
34	fkCol.table	Target!Table
34	a.owner	Source!Class
38	r	Source!Reference
39	fkCol	Target!Column
41	fkCol.table	Target!Table
41	r.type	Source!Class

6.2.2 Dependency Graph

In an ETL transformation, resolving target elements that have been (or can be) transformed from source elements by other rules is a frequent task in the body of

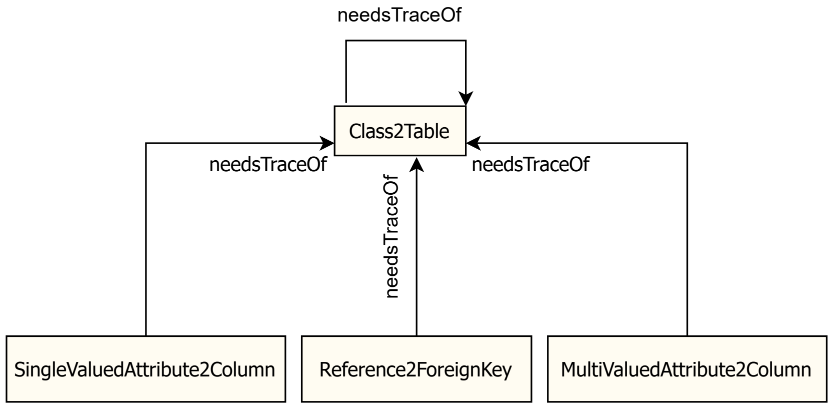


Figure 6.4: Dependency Graph of Listing 6.1

a transformation rule. This creates dependencies between these rules, which can be extracted from a type-resolved AST. In the body of a transformation rule say *TRx*, if there is an *equivalent(s)* statement that uses the elements transformed by another transformation rule say *TRy* as depicted in Line 9 of Algorithm 2, i.e., *TRx* is dependent on *TRy*. So, such dependencies are extracted as shown in Figure 6.4, using static analysis, describing which transformation rule is dependent on which other transformation rules for its execution (Line 10). The process of extracting such a dependency graph as shown in Algorithm 2. For example, in Line 18 of Listing 6.1, there is an *equivalent()* operation. The target expression of equivalent is *a.owner*, the type of which is resolved to *Source!Class* as shown in Table 6.1. Then a rule whose source parameter is of the same type or a compatible type (super type) will be searched, which in this case is rule *Class2Table*. Hence an edge is created between *SingleValuedAttribute2Column* and *Class2Table*.

6.2.3 Selective Traceability

While the resolution of elements using ETL's *equivalent/equivalents* operations is explained in Section 2.2.2.4, how these equivalent statements are actually executed is defined by the Epsilon execution engine. The proposed approach was to completely replace calls to *equivalent/s()* with calls to operations produced by the corresponding transformation rules. In the running example, the original transformation in Line 17 of Listing 6.1, calls an equivalent operation, while the optimised rewritten program in Listing 6.2 calls the corresponding operation *rule_Class2Table*. Using the dependency graph, in-memory caches were created (*HashMaps*) as shown in Lines 10 & 11 of Listing 6.2, which marks them as traceable. Hence, the operation cache also serves as a selective trace for the resolution elements(as anything not in these caches is simply not traced). The cache is populated with the corresponding target element along with source

Algorithm 2 Algorithm for Extracting Dependency Graph from Transformation Program

```
1: procedure extractDependencyGraph($a$)
2:     Let DG = Dependency graph
3:     Let a = Transformation program
4:     for each rule in a do
5:         add rule as a vertex in DG
6:     end for
7:     for all rule in $a$.rules do
8:         for all element =elements in body of rule do
9:             if element is an OperationCallExpression then
10:                if element.name = "equivalent" or "equivalents" then
11:                    type = resolvedType of element.target
12:                    for all r in $a$.rules do
13:                        if r is not abstract & (source parameter of r's type = type or is supertype of type) then
14:                            add r to rules
15:                        end if
16:                    end for
17:                    create an edge(s) in DG from rule to rules
18:                    replace element with corresponding operation call of rule.
19:                end if
20:            end if
21:        end for
22:    end for
23: end procedure
```

elements as a key as shown in Line 33 of Listing 6.2. If the cache contains a source element as key then the target elements are retrieved from the cache in the body of the operation as shown in Lines 28-30 of Listing 6.2.

6.2.4 Transformation Rewriting

After extracting the rule dependency graph, the rule-based transformation program is rewritten into an imperative form, as shown in Listing 6.2, where all rules are mapped to operations. The detailed process is presented in Algorithm 3. All rules are mapped to operations with the body of the rule mapped to the body of the operation, as in Lines 17-20 for rule *Class2Table* and in Lines 32-39 for rule *SingleValuedAttribute2Column* (depicted in Lines 3-26 of Algorithm 3). If a rule extends other transformation rules, those rules are called in the body of the operation, by setting the source parameter of the rule as a context to the operation (Line 17). The target parameters of a rule are instantiated in the body of the corresponding operation (Line 15), and then returned from the operation (Line 18). If the target parameter is multi-valued, then the resulting values are returned in a *Collection*. If a transformation rule has a guard block (Line 13), the guard block is also mapped to a corresponding operation (Line 4), with the same body (Line 6). The source parameter of the rule is also passed as a parameter (Line 7) of the corresponding operation of the guard block. Table 6.2 illustrates how the expressions are executed in regular ETL and how they are rewritten using the proposed approach. *single* represents a single source model element while *collection* represents a collection of model elements. If there exists more than one matched rule the results of all the matched rules are combined and executed depending on the equivalent/equivalents call. During the rewriting process, the behaviour of these calls is preserved using operation calls as shown in Table 6.3.

Table 6.2: *equivalent/equivalents()* operations in ETL

Original input expression	**Resolution in ETL**
single.equivalent()	Return only the first element of the target elements of matched rules
single.equivalents()	Return targets of all the matched rules
collection.equivalent()	Return a flattened collection of targets of all the matched rules
collection.equivalents()	Return targets of all the matched rules for all collection elements

Secondly, all these converted operations are added to the rewritten ETL transformation (Line 24). Then, the dependency graph (detailed in Section 6.2.2)

Algorithm 3 Algorithm for ETL Program Rewriting

```
 1: procedure REWRITE($\boldsymbol{a}$)
Require: $\boldsymbol{DG}$ = Dependency graph
 2:     Let $\boldsymbol{a}$= Transformation program
 3:     for all rule in $a$ rules do
 4:         Map guard block of rule to an operation op_gd
 5:         op_gd.name = operation guard_ruleName
 6:         Body of op_gd ← body of guard block
 7:         Param of op_gd ← Source parameter of rule
 8:         Add op_gd to the ETL module
 9:         Map rule to an operation op_rule
10:         op_rule.name = operation rule_ruleName
11:         Body of op_rule ← body of rule
12:         context of op_rule ← Source parameter of rule
13:         if guardBlock exists then
14:             Call op_gd as an if statement
15:         end if
16:         Instantiate target element(s)
17:         Add above as statement(s) to the body of op_rule
18:         Call super rules of rule
19:         Return target element(s) as a Collection
20:         Add above as a return statement in op_rule
21:         Set the type of target element of rule as a return type of op_rule
22:         If multiple targets set return type as Collection
23:         if rule is traceable according to DG then
24:             declare a HashMap (cache_ruleName) variable in the pre block
25:             add target elements for the corresponding source element in the
    cache_ruleName
26:             add an if statement to search in cache_ruleName if a key with
    source element exists
27:         end if
28:         Add op_rule to the ETL module
29:     end for
30:     for all rule in transformation rules do
31:         if rule is not lazy or abstract then
32:             Iterate through all instances of the rule's source parameter type
33:             Call the corresponding operations of rule in for loop
34:             Set the iterating variable as a context of operation
35:             Add the for statements in the pre block
36:         end if
37:     end for
38:     Clear all transformation rules from the ETL module
39: end procedure
```

Table 6.3: Rewriting of *equivalent/equivalents()* operations

Original input expression	**Corresponding rewritten expression**
single.equivalent()	single.matched_rule().first()
single.equivalents()	single.matched_rule()
collection.equivalent()	collection.collect(x—x.matched_rule()).flatten()
collection.equivalents()	collection.collect(x—x.matched_rule())

is analysed, to see if a rule needs to be traced, in which case a cache is created for the respective operation (Line 11).

Finally, mapped operations are called in the *pre* block of the ETL transformation (Line 30). All the operations corresponding to non-lazy, non-abstract rules are called in for loops by iterating through all instances of the source parameter of the rule (Line 27-28), setting it as a context to the operation (Line 29). Lazy rules are called by the other rules. At the end of this process, all the original transformation rules are removed from the ETL transformation (Line 31), as equivalent constructs are already being called as operations in the *pre* block.

6.2.5 ETL Engine

The rewritten transformation program is executed using a modified version of the ETL engine. This program is semantically equivalent to the original transformation but converted to imperative code, converting rules to operation calls, as discussed above. The ETL engine is the same engine used by the naive ETL, with just one modification in resolving operation calls. Usually operation calls are resolved using Java's reflection API, which can be computationally expensive. Static analysis, as described in Section 6.2.1, other than resolving types, also exposes which operation call expressions in the program are mapped to which corresponding user defined operations. Hence, this information can be used for providing the exact matched operation, to avoid having to search for it as the program is being executed. This optimisation is not specific to ETL transformations so it can be leveraged by any Epsilon language using user-defined operations.

6.3 Evaluation

This section presents the experimental setup used for evaluating the optimisation of model-to-model transformation programs based on static analysis, explains the methodology used and analyses the results. Finally, it discusses the limitations and possible threats to the validity of the obtained results.

6.3.1 Experimental Setup

The proposed approach is evaluated against the default ETL execution engine to measure its benefits in terms of execution time and memory footprint. The first contribution, selective traceability, is expected to substantially improve the memory consumption, by reducing the trace size, while the second one i.e., rewriting, is similarly expected to reduce execution time. The evaluation is divided into two parts. First, a comparison is performed of the proposed approach with the default ETL engine. Second, the proposed optimisation approach is compared with other state-of-the-art languages for M2M transformation.

The experiment referred to as "ETL" evaluates running the transformation using standard ETL without any optimisations. The "Optimised ETL" one uses the proposed optimisation/rewriting strategy described in Section 6.2. The results of the proposed approach is compared with two other widely-used model-to-model transformation languages, ATL and YAMTL [119], to position this work in the broader context of M2M languages. For the ATL and YAMTL evaluation, the same transformation is rewritten in ATL and YAMTL and then the execution time and memory footprint are reported.

6.3.2 Case Study and Models

OO2DB transformation is used as presented in Section 6.1 for evaluating the proposed approach. The *OO2DB* transformation is executed over a set of *OO* models of increasing sizes, as shown in Table 6.4. These synthetic models conforming to the *OO* metamodel are created using an EOL program which can be found online[1].

6.3.3 Correctness

The transformation is rewritten to an efficient form behind the scenes, so it is crucial to ensure that the rewritten transformation is semantically equivalent to the original input program. To gain confidence that the rewritten program is correct, the generated output model(s) should be the same as the ones generated by the original transformation. Using EMFCompare [120], the output models for ETL and Optimised ETL are checked to assess if they are the same. For ensuring broad coverage of the tests, seven test ETL scripts mined from GitHub were executed and for all cases no differences were found in the outputs given by

[1]https://github.com/quratulain-ali/runtime-eclipse.git

Table 6.4: Sizes of the Object Oriented models used for benchmarking

ID	Model Name	# of model elements
1	OO_10K	140,006
2	OO_15K	210,006
3	OO_20K	280,006
4	OO_25K	350,006

the original and the rewritten programs. For this test case, the object-oriented to relational database transformation, the generated output models are matched for ATL, ETL, Optimised ETL and YAMTL. After executing these equivalence tests, the semantic equivalence of the rewritten transformation is ensured and hence of the optimised and selective traceability used in this approach.

6.3.4 Machine Specification

The experiments were conducted on a machine with the following specifications: MacBookPro @ 2.8 GHz Quad-Core Intel Core i7, 16 GBs of RAM, macOS Big Sur version 11.1 with JVM 11 MaxHeapSize 4GBs.

6.3.5 Internal Evaluation

Table 6.5: Execution time of naive and optimised ETL, in ms

Model size	**Execution engine**	
	ETL	Optimised ETL
10K	5,899	3,519
15K	9,623	6,423
20K	16,040	10,171
25K	21,836	14,641

The results of the execution time of the naive ETL vs Optimised ETL are reported in Table 6.5 and visualised in Figure 6.5.

It can be observed that overall ETL's execution time is significantly improved in 'Optimised ETL' version. This is because of the optimisations provided

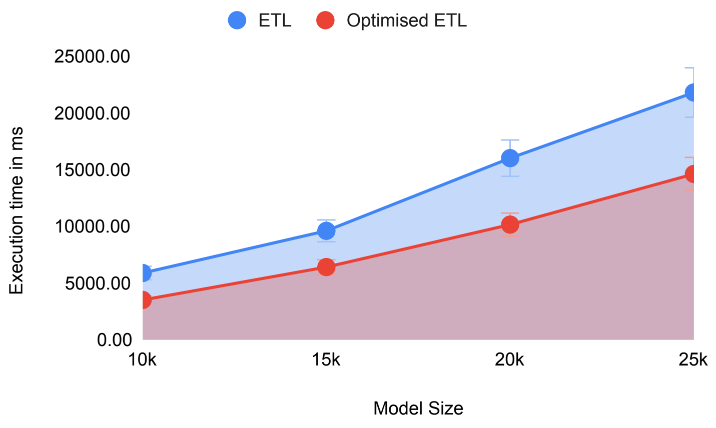


Figure 6.5: Execution time comparison of Optimised ETL with ETL

by static analysis and program rewriting to avoid operation call resolutions at runtime and also because of efficient resolution of equivalents before the execution.

As the proposed optimisation approach relies on extracting information (such as static analysis, extracting of dependency graph), it is necessary to compute the incurred overhead of these processes. Static analysis took an average of 50ms, extracting the dependency graph took an average of 35ms, while optimisation and rewriting took 2ms on average for all the experiments. It is worth noting that the size of the models does not affect the time needed to extract this information, as all these steps are performed at the metamodel level, before executing the program itself.

The memory used for the naive ETL and optimised ETL can be seen in Table 6.6, where it can be observed that the Optimised ETL consumes less memory compared to ETL as shown in Figure 6.6 due to the reduced (selective) transformation trace provided by the selective traceability mechanism that is discussed in this chapter.

6.3.6 External Evaluation

In the MDE ecosystem task specific languages are typically interpreted but compiled languages can also be used to perform the same tasks and can be faster compared to the interpreted ones. They are more verbose and less amenable to static analysis. This section discusses other M2M languages used in the MDE community: ATL, YAMTL and the A2L compiler. The proposed approach is compared with ATL and YAMTL because ATL is a widely used interpreted language, YAMTL is a compiled language that builds on top of Xtend and is a state-of-the-art language for large-scale model transformations. A2L is a compiler developed for the ATL language that compiles ATL transformations

Table 6.6: Memory consumption of naive and optimised ETL, in MBs

Model size	**Execution engine**	
	ETL	Optimised ETL
10K	83	30
15K	128	51
20K	175	69
25K	216	85

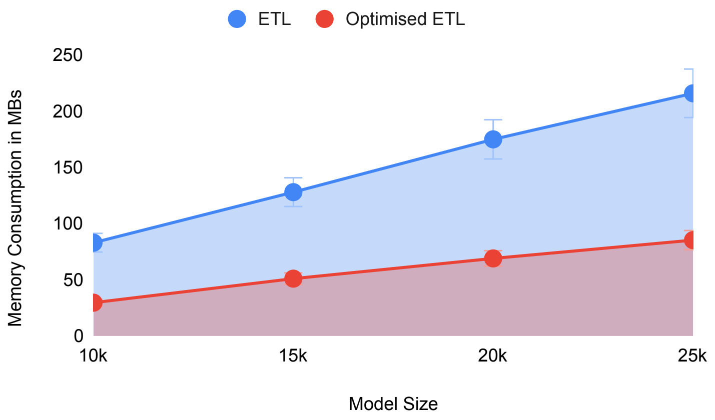


Figure 6.6: Memory consumption comparison of Optimised ETL with ETL

into Java code.

In Tables 6.7 and 6.8, the results of execution time and memory consumption of the proposed approach are presented in comparison with ATL & YAMTL, respectively (depicted in Figure 6.7 and 6.8). It can be clearly seen that YAMTL executes faster than the others, because YAMTL is compiled to Java, while ETL and ATL are interpreted.

On the other hand, YAMTL consumes the most memory, while ATL consumes the least. The excessive memory consumption of YAMTL is explained by the fact it supports incremental execution and hence consumes more memory due to the caching required. The results of the experiments compared to A2L are not presented as it was attempted to compile the *OO2DB* case study that was used in the experiments above, but due to certain limitations of A2L (which are discussed below), compilation failed.

Table 6.7: Execution time of various transformation languages, in ms

Model size	**Transformation Language**		
	Optimised ETL	YAMTL	ATL
10K	3,519	1,318	10,355
15K	6,423	1,806	15,202
20K	10,171	2,631	84,726
25K	14,641	3,200	103,596

While A2L provides a considerable speed up by generating efficient Java code from ATL transformations, still there are certain limitations such as 1) Rule Inheritance 2) Global variables 3) Reflective operations. The proposed approach does not limit or change any of the language semantics of an ETL transformation, thus supporting the above-mentioned features. Moreover, the *OO2DB* case study could not be compiled using A2L due to use of a global variable (Line 2 of Listing 6.1). So, in cases like this, using the approach achieves a significant speedup, without having to alter the original transformation.

6.3.7 Threats to Validity

This experiment uses two metamodels: *OO* and *DB* and a set of increasingly large synthetic models conforming to the source *OO* metamodel. Both the metamodels and the transformation were not specifically targeted but were chosen for two reasons. The first was using a well-known transformation, predating this work, as well as the metamodels exercising all core features of Ecore like inheritance, attributes, containment and non-containment references. The second was the generality and ease of understanding of both, as they are generic

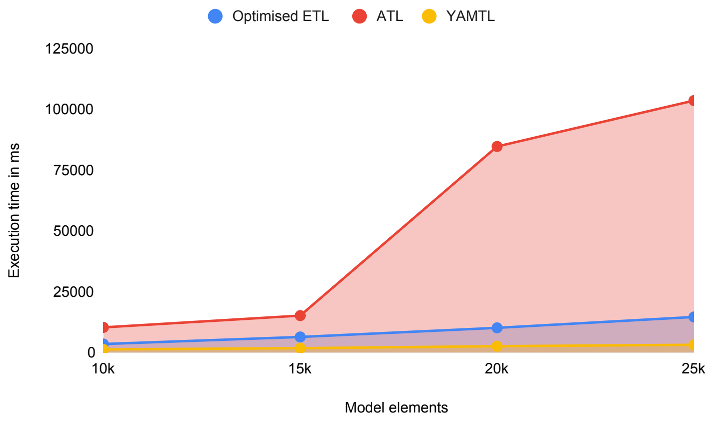


Figure 6.7: Execution time comparison of Optimised ETL with ATL and YAMTL

Table 6.8: Memory consumption of various transformation languages, in MBs

Model size	**Transformation Language**		
	Optimised ETL	YAMTL	ATL
10K	30	131	23
15K	51	200	32
20K	69	267	44
25K	85	337	54

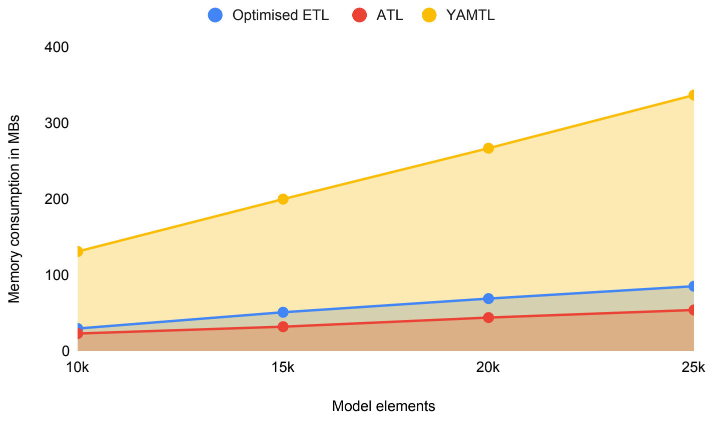


Figure 6.8: Memory consumption comparison of Optimised ETL with ATL and YAMTL

enough to understand and hence demonstrate the novel work presented in this chapter. Nevertheless, they may play a significant role in determining the results obtained. Hence, it cannot be claimed that the results obtained are generalisable for every type of transformation and model. The proposed transformation optimisation approach can benefit from experiments performed on more diverse models with a broader range of sizes and more complex transformations, both for investigating semantic equivalence and performance gains.

Another possible threat to the validity of these results is the addition of possibly substantial overheads of this approach when evaluating large enough programs or meta-models: for example, if the selective trace ends up being almost equal in size to the entire transformation trace (e.g. due to a fully connected dependency graph).

Finally, as the optimisation approach leverages the benefit of information extracted through static analysis, it is crucial to have an accurate static analysis of the transformation. To ensure more complete static analysis information and thus enable efficient program rewriting, it is recommended to use a more strict coding style, explicitly declaring types where possible, and avoiding *Any* type unless necessary, for more accurate type resolution.

6.4 Related Work

This section summarises the state-of-the-art within the scope of this chapter; it discusses model transformation optimisation strategies used for improving the performance of the engines executing such transformations.

In [90], Born et al. extend Henshin, a rule-based model transformation language, adapting graph transformation concepts based on EMF. This extension computes all potential conflicts and dependencies for a set of rules and reports them in the form of critical pairs. Each critical pair consists of the respective pair of rules, the kind of potential conflict or dependency found, and a minimal instance model illustrating the conflict or dependency.

In [91], Ujhelyi introduces a static analysis facility for graph transformations. This work uses Constraint Satisfaction Programming (CSP) to provide a type checker for the Viatra2 framework. This type checker is based on CSP, and is not guaranteed to find all the errors in a single run using static analysis.

In [121], the A2L compiler is introduced for parallel execution of ATL transformations. It uses static analysis through ATLyzer (discussed above in Chapter 4), to generate efficient code at the transformation level. A2L was discussed earlier in Section 6.3.6.

Gremlin-ATL is another approach presented in [122]. It is a model-to-model transformation framework that translates ATL transformations into Gremlin, a query language supported by several NoSQL databases.

Another model-to-model transformation language, YAMTL, was introduced in [119]. YAMTL provides an efficient engine to transform EMF-based models with transformations defined in the internal DSL of Xtend. Support for incremental transformations was also added in [123] using the forward change propagation mechanism. Several approaches and languages are available for incremental model-to-model transformations, such as the Tefkat tool, by Hearnden et al. in [124]. Here, changes to the source models are directly mapped to their effects on transformation execution, allowing modifications to target models to be computed efficiently.

To summarise, the novelty of the approach presented in this chapter is that it takes the benefit of static analysis to reduce the transformation trace while not sacrificing the language (ETL) expressiveness by compiling it down to a general-purpose programming language such as Java.

6.5 Chapter Summary

This chapter presented the second contribution of this research, which is to optimise model-to-model transformations written in ETL by introducing a selective traceability facility. ETL keeps a transformation trace of all the source-target pairs. An approach is presented that is used to optimise programs written in ETL. The proposed approach resolves the types of various constructs using static analysis and then creates a dependency graph between the transformation rules. Based on this dependency graph, the rule-based transformation program is rewritten to an imperative program that only maintains a selective trace. The evaluation experiments have demonstrated that the proposed approach can deliver significant performance benefits both in terms of execution time and memory footprint compared to the default ETL execution engine, particularly where larger models are involved.

Chapter 7

Optimisation of Model Comparison Programs

This work has been published as a conference paper titled *"Towards Efficient Model Comparison using Automated Program Rewriting"* 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23).

Model comparison is usually a prerequisite to various other key model management activities such as model differencing, model versioning, etc. It involves establishing matches/correspondences between elements of two models. There are different ways to compare models, such as traditional text-based comparison, comparison based on unique identifiers, model-to-model (M2M) transformation to compare models as in [125], or to use a dedicated comparison language, such as the Epsilon Comparison Language (ECL), which supports specifying matching criteria. Such model comparison can be computationally very expensive because element of the first model needs to be traversed and compared to a corresponding element of the second model as specified in the comparison program, which does not scale well.

In this chapter, an efficient model comparison approach is introduced based on static program analysis and automated program rewriting. A prototype implementation of the proposed approach has been developed that can rewrite ECL programs, which operate on models with Ecore-based metamodels. According to the current ECL engine, all elements of one type are compared against the elements of their matching type based on the provided comparison logic by the developer. Using program analysis, elements to be compared are pre-filtered, indexed them and then compared the pre-filtered instances rather than all instances. These pre-filtered elements are automatically embedded into the original ECL program, using program rewriting, and executed using the traditional ECL engine. Also, it is ensured that all the rules that are needed for the execution of a particular rule have already been executed, by reordering the rules, to avoid the extra overhead of searching the appropriate rule to invoke. The output of an ECL program is a match trace that contains all the established results of matches between elements of two models. The proposed approach yields a reduced match trace, by omitting any unsuccessful matches, whenever it was possible to identify these beforehand through program analysis. The match trace contains a number of matches, each match contains the two objects that were matched and a boolean to indicate if the match was successful or not.

The proposed approach has shown performance gains up to 95% in terms of execution time in the experiments that have been conducted.

The rest of the chapter is structured as follows: Section 7.1 presents a running example. Section 7.2 presents the overview of the proposed comparison optimisation approach and then discusses each component step-by-step. Experiments, case studies and the obtained results are presented and analysed in Section 7.3. Section 7.4 discusses the relevant state-of-the-art in the field of model comparison optimisation and static analysis.

7.1 Motivating Example

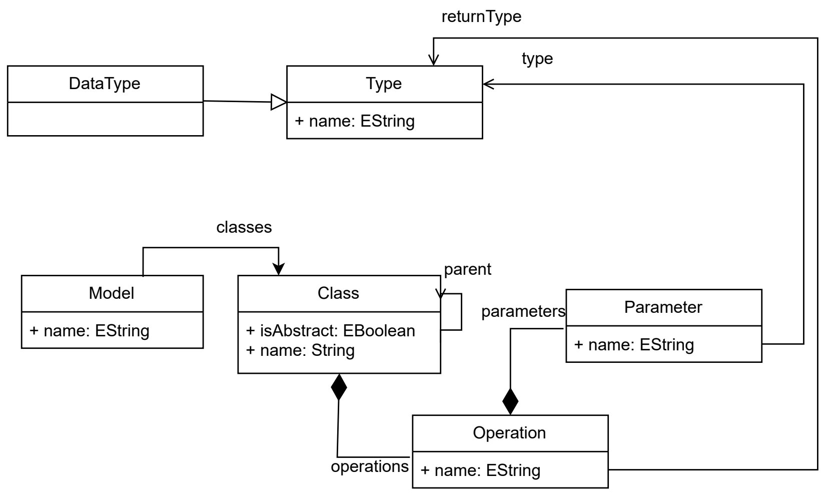


Figure 7.1: An excerpt of the Class Diagram metamodel

In this section, as a running example, class diagrams are compared with sequence diagrams. Figure 7.1 is a class diagram that illustrates the metamodel of a class diagram-like language. The class diagram shows a structural view of the system containing the classes, their attributes and their operations.

Then a metamodel of a sequence diagram is considered, an excerpt of which is shown in Figure 7.2. Sequence diagrams show the interaction between objects of a system - its intended behaviour.

Now, as the sequence diagram depicts the interaction between objects and the class diagram represents the classes and their features, correspondences can

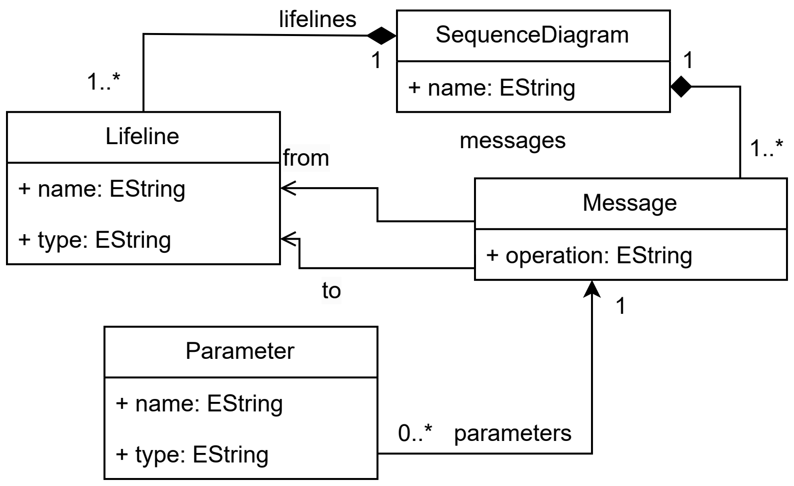


Figure 7.2: An excerpt of the Sequence Diagram metamodel

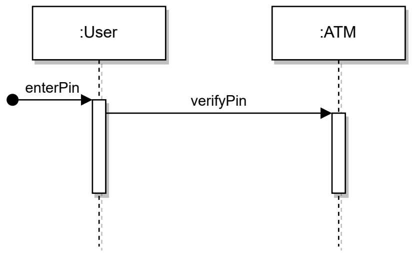


Figure 7.3: Sequence Diagram of ATM

be established between the two, which can be used for downstream activities such as validation, model merging, etc.

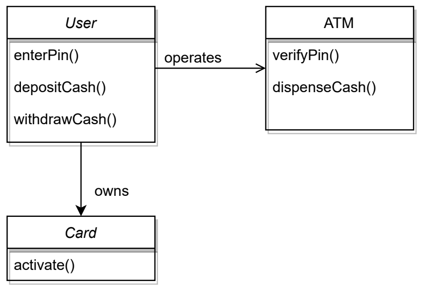


Figure 7.4: Class Diagram of ATM

A custom comparison algorithm written in ECL is shown in Listing 7.1. For this comparison, there is following basic criteria:

- A lifeline matches a class when the type of the lifeline is the same as the name of the class in class diagram.
- A message matches an operation when the operation of the message is the same as the name of the operation. Also, the class corresponding to the "to" lifeline of the message or one of its supertypes should contain the operation.
- The parameters of the message need to be matched with the parameters of the operation.

```
model Left driver EMF{
  nsuri = "sd"
};

model Right driver EMF{
  nsuri = "cd"
};

rule Lifeline2Class
  match l : Left!Lifeline
  with r : Right!Class {
    compare : l.type = r.name
}

rule Message2Operation
  match l : Left!Message
  with r : Right!Operation {

```

```
19     compare : l.`operation` = r.name
20     and (l.`to`.matches(r.class) or l.`to`.matches(r.
           class.superTypes)) and l.parameters.matches(r.
           parameters)
21 }
22
23 rule Param2Param
24   match l: Left!Parameter
25   with r: Right!Parameter {
26
27     compare : l.name = r.name and l.type = r.type.name
28 }
29
30 operation String matchOperation(others : Collection<
       Right!Operation>) : Boolean {
31     return others.exists(o|o.name = self);
32 }
```

Listing 7.1: Example ECL script before optimisation

In this ECL program, there are three match rules: *Lifeline2Class* (Line 9-13), which compares the type of lifeline to the class name (Line 12), *Message2Operation*, which compares the operation of *Message* with *Operation* name (Line 20). *Message2Operation* also compares whether the operation's owner class is the same as the *to* (Lifeline) of the *Message*. Finally, *Param2Param* compares the name and type of the parameters of both models (Line 27).

As an example, consider matching a class diagram of an ATM system as shown in Figure 7.4 with its corresponding sequence diagram as shown in Figure 7.3. If the ECL program is executed (Listing 7.1 over these two models it would produce the match trace shown in Table 7.1. As it can be seen, it returns all matches of each element with its corresponding type and a boolean indicating if the element was matched or not.

```
model Left driver EMF {
  nsuri = "sd"
};

model Right driver EMF {
  nsuri = "cd"
};

pre {
  var Lifeline2ClassMap = Right!Class.all.mapBy(param|
      param.name);
  var Message2OperationMap = Right!Operation.all.mapBy
      (param|param.name);
  var Param2ParamMap = Right!Parameter.all.mapBy(param
      |param.name);
```

Table 7.1: Match trace produced from the execution of Listing 7.1 on the models in Figure 7.3 and Figure 7.4

S #	Left	Right	Matching
1	Lifeline (qa: User)	Class (User)	True
2	Lifeline (qa: User)	Class (ATM)	False
3	Lifeline (qa: User)	Class (Card)	False
4	Lifeline (hsbc: ATM)	Class (User)	False
5	Lifeline (hsbc: ATM)	Class (ATM)	True
6	Lifeline (hsbc: ATM)	Class (Card)	False
7	Message (enterPin)	Operation (verifyPin)	False
8	Message (enterPin)	Operation (dispenseCash)	False
9	Message (enterPin)	Operation (enterPin)	True
10	Message (enterPin)	Operation (depositCash)	False
11	Message (enterPin)	Operation (withdrawCash)	False
12	Message (enterPin)	Operation (activate)	False
13	Message (verifyPin)	Operation (verifyPin)	True
14	Message (verifyPin)	Operation (dispenseCash)	False
15	Message (verifyPin)	Operation (enterPin)	False
16	Message (verifyPin)	Operation (depositCash)	False
17	Message (verifyPin)	Operation (withdrawCash)	False
18	Message (verifyPin)	Operation (activate)	False

Table 7.2: Match trace produced from the execution of Listing 7.2 on the models in Figure 7.3 and Figure 7.4

S #	Left	Right	Matching
1	Lifeline (qa: User)	Class (User)	True
2	Lifeline (hsbc: ATM)	Class (ATM)	True
3	Message (enterPin)	Operation (enterPin)	True
4	Message (verifyPin)	Operation (verifyPin)	True

```
}

rule Lifeline2Class
match l : Left!Lifeline
with r : Right!Class
from : Lifeline2ClassMap.get(l.type) ?: Sequence{}{
  compare : true
}

rule Param2Param
match l : Left!Parameter
with r : Right!Parameter
from : Param2ParamMap.get(l.name) ?: Sequence{}{
  compare : true and l.type = r.type.name
}

rule Message2Operation
match l : Left!Message
with r : Right!Operation
from : Message2OperationMap.get(l.‘operation‘) ?:
    Sequence{} {
  compare : true and (l.‘to‘.matches(r.class) or l.‘to
      ‘.matches(r.class.superTypes)) and l.parameters.
      matches(r.parameters)
}

operation String matchOperation(others : Collection(
    Right!Operation)) : Boolean {
  return others.exists(o : Right!Operation|o.name =
      self);
}
```

Listing 7.2: Example ECL script after optimisation

The default execution engine of ECL will compare each instance of the left parameter (i.e., *Lifeline*) to all the instances of the right parameter (i.e., *Class*). The complexity of this rule here would be $O(M\times N)$, if there are M number of *Lifeline*s and N number of *Class*es. Using program analysis, it is a possibility to index the instances by analysing these compare blocks as shown in Listing 7.2. Considering example sequence and class diagrams in Figure 7.3 and Figure 7.4, as there are 2 Lifelines and 3 Classes so there will be 6 matches for the rule *Lifeline2Class*. In the rule *Lifeline2Class*, the *Class* instances can be filtered only keeping ones where the name of the class is equal to the *type* of the *Lifeline*. These indices can be pre-computed once, and then used as required. This could reduce the complexity to $O(M)$, considering the complexity of the hash function to be $O(1)$. Again considering the example models, if Listing 7.2 is executed, the resultant match trace would be the same as shown in Table 7.2. There are only two matches for the same *Lifeline2Class* rule. Hence, the idea of this work is to analyse the ECL matching program and to automatically replace it with an efficiently rewritten program, to reduce the complexity of (some of) the comparisons. This is done by getting rid of the false matches as much as possible. This way, the program would not waste time nor memory in computing them.

7.2 Proposed Approach

In this section, the proposed approach is presented in detail, the overview of which is illustrated in Figure 7.5. The idea is to optimise ECL matching programs automatically using program analysis. The developer writes the comparison algorithm in ECL to compare two models, say left and right. The expected outcome is a match trace resulting from computing the compare block of each match rule. The match trace contains a number of matches, each match contains the two objects that were matched and a boolean to indicate if the match was successful or not. So using the proposed optimisation approach, a match trace is generated, which is a reduced or pre-filtered version, containing a significantly smaller number of unsuccessful matches. The search space is hence reduced, making the comparison faster. This is because all instances of left parameter are not compared to all instances of right parameter (which is done in existing ECL execution), rather instances of the left parameter are compared to pre-filtered/pre-indexed instances of the right parameter.

The first step in the proposed approach is the static analyser, a block used to populate the Abstract Syntax Tree (AST) of the ECL matching program, with the respective type information. This type-resolved AST is then used for two purposes: i) For the dependency graph extractor, a block that extracts the dependencies between different match rules of an ECL program by analysing compare blocks and *matches()* operations. Dependency here means that if a rule *MRx* invokes another rule *MRy* then the rule *MRx* would be dependent on *MRy*. The dependency graph is then used by the rule scheduler to efficiently reorder the execution of rules. This is to ensure that if a rule invokes another rule like in Line 20 of Listing 7.1, rule *Message2Operation* is dependent on rule

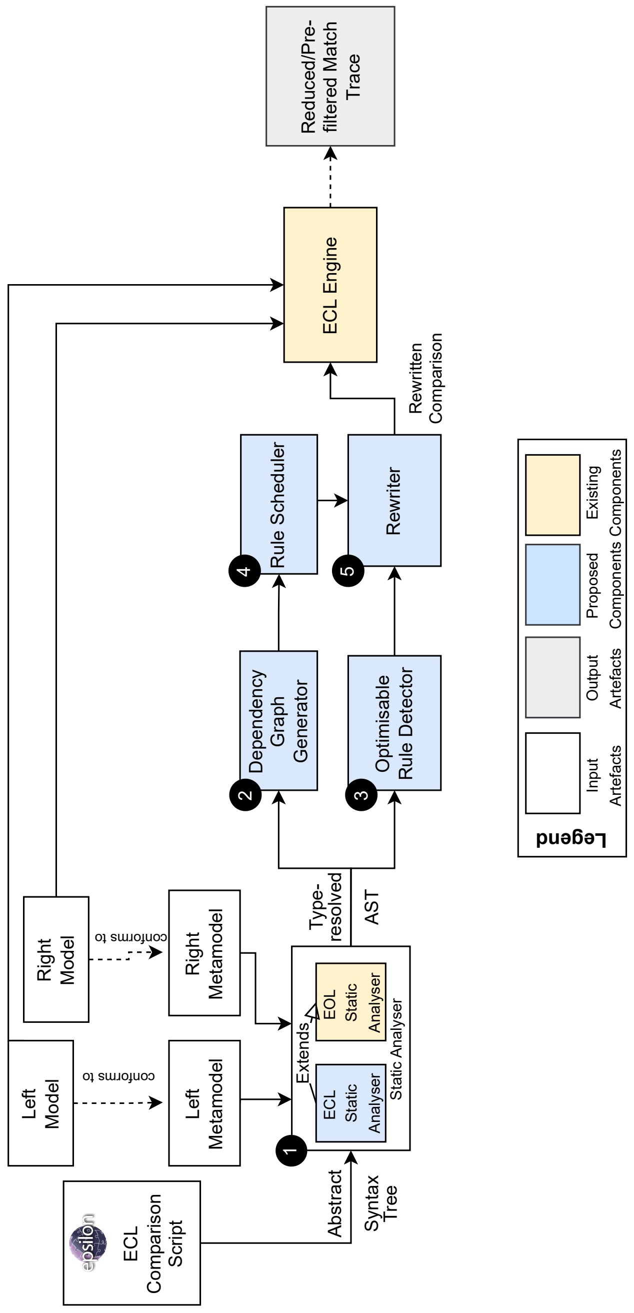


Figure 7.5: An overview of the proposed approach

Lifeline2Class and *Param2Param*. Both the rules on which *Message2Operation* is dependent should be executed before *Message2Operation*. ii) For the optimisable rule detector, a block for program analysis to identify the match rules which can be optimised based on the expressions in compare block. Here, optimisable rules mean the rules which are matching two elements on the basis of a specific property and can be indexed. So, this step identifies optimisable match rules along with the specific property name. Finally, the rewriter block will replace the original program with a rewritten optimised program along with the new order of the match rules. This optimised comparison program will then be executed by the existing ECL engine. The resultant match trace would be a subset of the trace that would have been produced by the original comparison program. This subset trace would exclude the matches which would not satisfy the domain (an EOL expression to narrow the search space), while including all positive matches.

7.2.1 Static Analysis

Static analysis is the first step of the proposed approach workflow. It analyses the ECL program's abstract syntax tree (AST) and computes the types of all expressions in it. This type information is extracted using metamodel introspection, type resolution and type inference. *ModelDeclarationStatement*s in Lines (1-3 and 5-7) in Listing 7.1 actually access the metamodel structure and help retrieve the types and their hierarchy available in the metamodel. To statically analyse ECL programs, the already available EOL static analyser [1] is extended by adding language specific support (e.g. analysing *MatchRule*s, compare blocks etc.). The resolved types of various constructs in Listing 7.1 are shown in Table 7.3. The outcome of the static analyser block is a type-resolved AST, which is just the input AST with its nodes populated with their respective types.

7.2.2 Dependency Graph

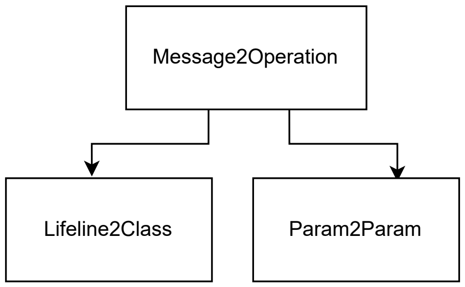


Figure 7.6: Dependency graph of Listing 7.1

[1] https://github.com/epsilonlabs/static-analysis

Table 7.3: Resolved types of various constructs in Listing 7.1

Line#	**Expression**	**Resolved Type**
13	l	Left!Lifeline
14	r	Right!Class
15	l.type	String
15	r.name	String
19	l	Left!Message
20	r	Right!Operation
21	l.operation	String
21	r.operations	Collection<Right!Operation>
21	r.superTypes.operations	Collection<Right!Operation>
29	l.parameters	Collection<Left!Paramter>
29	r.parameters	Collection<Right!Paramter>
33	l	Left!Parameter
34	r	Right!Parameter
36	l.name	String
36	r.type.name	String

ECL provides a built-in operation *matches(right :Any)* for model elements and collections. When invoked, the *matches()* operation returns the cached result, if the elements have already been matched. Otherwise, it finds the rules comparing the same two elements and then returns its results. Due to these rule invocations, rules can be dependent on one another. These dependencies can be extracted with the help of the type-resolved AST, as done for model-to-model transformations in [126]. To construct a dependency graph, a vertex is created for each *MatchRule* declared in the ECL program. If a rule *MRx* has a statement in its compare block that calls a *matches()* operation which invokes another rule, say *MRy*. The resolution of which rule is invoked by the *matches()* operation is done by finding the rule where the type of the left and right parameters of rule is the same as the type of the target and parameter expressions of the *matches()* operation. Then, an edge is created from the vertex corresponding to *MRx*, to the vertex corresponding to *MRy*. If there are multiple rules invoked by the *matches()* operation, multiple edges were created from *MRx*. For example, as in rule *Message2Operation* Line 19 of Listing 7.1 there is a call to the *matches()* operation with *l.to* (resolved type: *Lifeline*) as the target expression and *r.class* (resolved type: *Class*). This means that this *matches* operation call will invoke a rule which is matching *Lifeline* with *Class* i.e., rule *Lifeline2Class*. So, an edge is created from *Message2Operation* to *Lifeline2Class* as shown in Figure 7.6. The reason for extracting the dependency graph is to reorder the rules in a way that if *MRx* is invoked by a rule *MRy* then *MRy* is scheduled before *MRx*. This rescheduling helps achieve a performance gain, because it can reduce the number of attempts needed to find the appropriate rules to invoke. Any rule invocation using a *matches()* operation can use the cached results in the match trace, if the rules have been reordered properly. An edge is not created when a rule invokes itself, as it does not affect the reordering for which a dependency graph is extracted. However, ECL provides a mechanism to avoid an infinite loop, in case of a cyclic invocation of a rule i.e., two rules implicitly invoking each other. ECL maintains a temporary trace along with the primary trace. In a primary trace the matching value is added after the execution of compare block, while the matching value is set to true in the temporary trace before the execution of the compare block. In case of another attempt to match elements from already invoked rules, these rules would not be re-invoked. Finally, the temporary trace is reset when a top-level rule returns.

7.2.3 Identifying Optimisable *MatchRules*

This is the third step of the approach that takes in a type-resolved AST as an input with the aim to identify the rules which can be optimised. Optimisable rules are the rules which are comparing the elements of the two models based on a specific property. This is done by traversing the compare block of each *MatchRule* and finding expressions where two elements are compared on the basis of a specific property or attribute. In this case, the elements can be indexed based on that property. The process for identifying such optimisable rules is specified in Algorithm 4. The algorithm traverses a set of Match rules (Line 2) and their compare block (Line 3). Then, in a compare block all DOM elements are traversed to identify cases where a *PropertyCallExpression*(Line 4) is used within an *EqualsOperatorExpression* (Line 6) and it records the relevant Match rules and properties in a HashMap for later use in indexing (Line 13). With one exception, if there is a logical operator between equals expression, just the index is recorded if it is an and operator (Line 10). For instance, in Listing 7.1 Line 12 the rule *Lifeline2Class* is comparing the *Lifeline* from Sequence diagram to *Class* from Class diagram on the basis of the property *name*, in this class. So the Algorithm 4, would return the Hashmap containing rule *Lifeline2Class* with the respective property "*name*".

Algorithm 4 Algorithm for Identifying Optimisable Rules

```
 1: Let $op$ = HashMap<rule, NameExpression>
 2: for all Matchrules $rule$ do
 3:     Visit all DOM elements ($elem$) of compare block of $rule$
 4:     if $elem$ instanceof PropertyCallExpression and !($op$.contain($rule$)) then
 5:         $parent \leftarrow elem$.parent
 6:         if $parent$ instanceof EqualsOperatorExpression then
 7:             $parent \leftarrow parent$.parent
 8:             if $parent$ instanceof OperatorExpression then
 9:                 if $parent$ instanceof AndOperatorExpression then
10:                     op $\leftarrow$ $rule$ and elem.$NameExpression$
11:                 end if
12:             else
13:                 op $\leftarrow$ $rule$ and elem.$NameExpression$
14:             end if
15:         end if
16:     end if
17: end for
```

7.2.4 Program Rewriting

The final step is the rewriting phase illustrated in Algorithm 5, now that all the program analysis is in place. As discussed in the previous step, the optimisable rules are identified say *MR1, MR2.., MRn* along with the specific properties say *p1, p2.., pn* on the basis of which the elements are compared in the compare block. All instances of the right parameter of the identified rule *MRn* are indexed on the basis of the respective property *pn*. This is done using a built-in method called *mapBy* (Line 10 in Listing 7.2), which returns a map containing the results of the parameter expression as keys and the respective items of the target collection as values. The *mapBy* operation is called with all instances of the identified rule's right parameter and assigned to a newly declared variable (Line 4-10 of Algorithm 5). The naming convention of these variables is the rule name concatenated with the string "Map". So, a Map for the rule *Lifeline2Class* will be called as *Lifeline2ClassMap* (Line 11 of Algorithm 5). These variable statements are then added to the *pre* block of the ECL program (Line 13 of Algorithm 5). The *Pre* block is a set of EOL statements that are executed before the execution of match rules in ECL. This can be seen in Listing 7.2 (Line 10-12).

The next step is to utilise these pre-computed hashmaps (indices). For this, the facility of specifying domains was added to ECL. Each parameter in an ECL rule can define a domain, which is an EOL expression that yields a set of model elements, allowing the developers to narrow down the search space. Two types of domains are supported in ECL. Static domains which are computed once for one match rule and are independent of bindings of the other parameter of the *MatchRule*. Static domains are denoted by the "*in*" keyword) and dynamic domains which are recomputed every time the other parameter value is changed. Dynamic domains are dependent on the other parameter values and are denoted by the "*from*" keyword. So these hashmap variables that are added in the pre block were used, as a dynamic domain for the right parameter of the corresponding *MatchRule*. For instance in Line 18 of Listing 7.2, the value from the corresponding hashmap i.e., *Lifeline2ClassMap* was retrieved using the left parameter's compared property (identified in the previous step) as a key.

Hashmaps return null values if they do not contain the mapping for a particular key, so to cater for possible null pointer exceptions, a safe navigation operator was used. EOL already supports a safe navigation operator *?.*, for making the null checks more concise without crashing the program. The use of the safe navigation operator is shown in Line 18 of Listing 7.2, where an empty *Sequence* is returned if the *get()* operation returns a null value. `var result = a?.someProperty?.anotherProperty;`

If *a* is not null, *someProperty* would be assigned to *result*, otherwise, *anotherProperty* would be assigned.

The last bit in the rewriting phase is to rewrite the order of the rules as described in the Rule Scheduler step. The reordering is done on the basis of the dependency graph as in Figure 7.6, so that the independent rules can be executed first and then the ones dependent on them. Now, instead of the ECL engine executing the original program written by the developer, as listed in Listing 7.1, the automatically rewritten program as in Listing 7.2 will be executed.

7.3 Evaluation

In this section, first the experimental setup is presented, including the case study and the models used for the benchmarks, and the results of the conducted experiments were presented. Finally the section is concluded by analysing and then stating any threats to the validity of the presented results.

Algorithm 5 Algorithm for ECL Program Rewriting

```
1: Let $op$ = HashMap of rules with the corresponding properties as in Algorithm 4
2: $DG$ = Dependency Graph
3: for all $rule$ in $op$ do
4:     Construct property call expression ($pce$)
5:     target $\leftarrow$ type of right parmeter of $rule$
6:     property $\leftarrow$ all
7:     Construct operation call expression ($oce$
8:     target $\leftarrow$ $pce$
9:     operation $\leftarrow$ mapBy
10:    expression $\leftarrow$ $op$.get($rule$)
11:    declare variable ($v$) with name $rule$.getName()+"Map"
12:    $v \leftarrow oce$
13:    add $v$ to $pre$ block
14:    add domain block with expression $v$.get(leftParameter.property)
15: end for
16: reorder rules according to topological order of $DG$
```

7.3.1 Experimental Setup

To evaluate the efficient comparison approach proposed, the execution time of the original ECL programs were measured using the existing ECL engine with the rewritten ECL programs (also using the existing ECL engine). Since Epsilon already supports parallel execution of ECL programs, all these experiments were conducted with the parallel execution mode. First, the execution time was measured for running the comparison program with the existing ECL engine (without any optimisations) in parallel mode and this is referred to as ECL in all the results tables and graphs. Second, the proposed approach was used to automatically rewrite the ECL program (as described in Section 7.2) and execute the rewritten program using the existing ECL engine in parallel mode. This is referred to as *Optimised ECL* in the results tables and graphs.

7.3.1.1 Case Study & Models

Table 7.4: Sizes of the models used for benchmarking

	No of model elements					
ID	**OO**	**DB**	**OO+DB**	**Seq**	**Class**	**Class+Seq**
1	287	184	471	305	356	661
2	357	229	586	417	356	773
3	427	274	701	417	469	886
4	497	319	816	342	356	698
5	567	364	931	342	356	698
6	637	409	1046	305	469	774
7	707	454	1161	342	469	811

For evaluating the proposed approach, two case studies were used: one is the class and sequence diagram comparison as shown in Listing 7.1, the second is the comparison of object oriented (OO) models with database (DB) models. The class and sequence diagram models of different sizes conforming to these metamodels were used which are publicly available on GitHub [127]. OO & DB are the synthetic models generated in [126]. The number of elements of different models are mentioned in Table 7.4. The point to note is that the sizes of the models that were used were not very large but the comparison of these models still becomes computationally very expensive. Hence, a notable performance gain can be observed in these models.

```
rule Class2Table
match l : OO!Class
with r : DB!Table{

  compare : l.name = r.name
}

rule Attribute2Column
match l : OO!Attribute
with r : DB!Column
{
  compare : l.name = r.name and l.owner.matches(r.
     table)
}
```

Listing 7.3: ECL comparison program for OO-DB models

To compare OO models with DB ones, a simple comparison algorithm (depicted in Listing 7.3) was used to establish matches between tables and classes, when their names are equal. In the second rule, attributes with columns are compared on the basis of the property *name*, and also on whether they belong to same class and table respectively. This example is quite simple but this is used as a case study to show the substantial performance benefits observed even for simpler matching programs, with increasing model sizes. The comparison program in Listing 7.3 would be optimised and rewritten as represented in Listing 7.4. Line 5 in Listing 7.3 informs that the caching has to be done by name attribute as shown in Line 2 of Listing 7.4.

```
1  pre {
2    var Class2TableMap = DB!Table.all.mapBy(param|param.
         name);
3    var Attribute2ColumnMap = DB!Column.all.mapBy(param|
         param.name);
4  }
5
6  rule Class2Table
7  match l : OO!Class
8  with r : DB!Table
9  from : Class2TableMap.get(l.name) ?: Sequence{} {
10   compare : true
11 }
12
13 rule Attribute2Column
14 match l : OO!Attribute
15 with r : DB!Column
16 from : Attribute2ColumnMap.get(l.name) ?: Sequence{} {
17   compare : true and l.owner.matches(r.table)
18 }
```

Listing 7.4: ECL rewritten program for OO-DB models

7.3.1.2 Correctness

As the approach is based on automatic rewriting of the program, it is crucial that the rewritten program preserves the semantics of the original program. To ensure this, equivalence testing is used to compare the match trace for both the original and the rewritten programs. Several comparison programs are used mined from GitHub to compare models both conforming to same and different metamodels and then compared their output match traces. Mostly, comparison programs available on GitHub were comparing models from the same modelling language. It was verified that the number of successful matches in both the optimised and the unoptimised version remained the same, as shown in Table 7.5 and 7.6. While the number of successful matches is the same, one can observe the difference in number of unsuccessful matches in Table 7.5 and 7.6. This is because of the successful pre-filtering/pre-indexing in the proposed approach. Some of the instances are filtered which, using the static program analysis, can be categorised as unsuccessful, before actually running the comparison algorithms.

Table 7.5: Match Trace of Class and Sequence Diagram Comparison

ID	ECL (All)	Optimised (All)	ECL (S)	Optimised(S)
1	29400	92	38	38
2	29400	92	38	38
3	33700	78	0	0
4	33284	54	0	0
5	33284	54	0	0
6	33700	78	0	0
7	38100	208	72	72

Table 7.6: Match Trace of OO and DB Comparison

ID	ECL (All)	Optimised (All)	ECL (S)	Optimised(S)
1	18060	4120	60	60
2	28200	6400	75	75
3	40590	9180	90	90
4	55230	12460	105	105
5	72120	16240	120	120
6	91260	20520	135	135
7	112650	25300	150	150

7.3.1.3 Machine Specification

The set of evaluation experiments presented in this chapter were performed on a MacBookPro @ M2 Core i7, 24 GBs of RAM, Mac operating system Ventura version 13.0, and Java 17 on JDK 17.0.6 with JVM MaxHeapSize 6GBs.

7.3.2 Results

In this section, the results from the conducted experiments are presented. Table 7.7 presents the execution time in milliseconds for the OO and the DB model comparison, and the Class and Sequence Diagram model comparison respectively. The results can also be visualised for the OO & DB comparison in Figure 7.7 and the Class and Sequence diagram in Figure 7.8.

Table 7.7: Execution time of existing ECL and optimised ECL, in ms

	OO - DB		**CL - SEQ**	
ID	**ECL**	**Optimised**	**ECL**	**Optimised**
1	1962	535	3287	196
2	3488	781	3109	194
3	6745	1238	3894	205
4	14051	1735	4046	188
5	22044	1924	4286	287
6	31611	3705	4342	199
7	52159	4520	5050	250

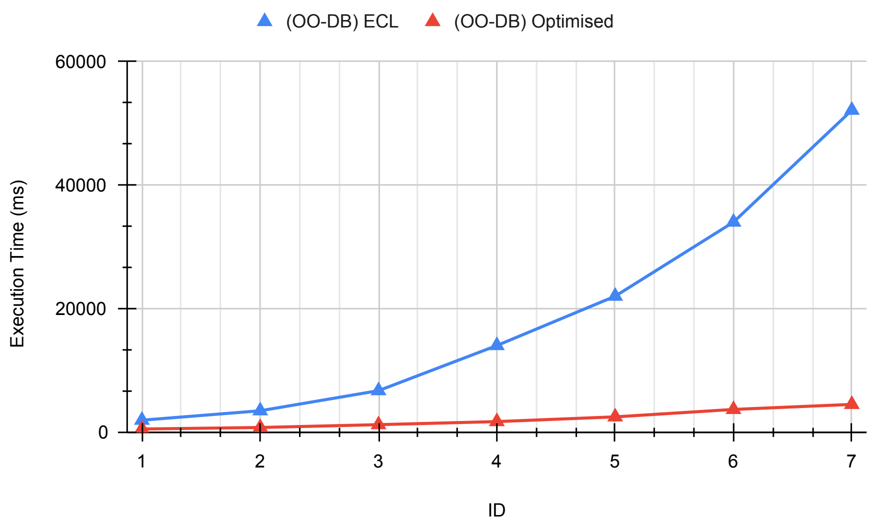


Figure 7.7: Comparison of Execution time in OO DB Comparison

As seen in Table 7.4, the OO and the DB models are of increasing sizes, while this is not the case with the Class and Sequence Diagram models. Keeping these sizes of models in mind, a continuous rise in performance gain can be seen as the model size increases (Figure 7.7). While in Figure 7.8, almost a constant performance gain compared to the existing ECL engine can be seen. This suggests that the performance benefits are proportional to model size.

This performance gain is achieved by reducing the search space needed for matching. It can be observed in the match traces produced for both case studies in the Table 7.5 and Table 7.6 that the number of unsuccessful matches were

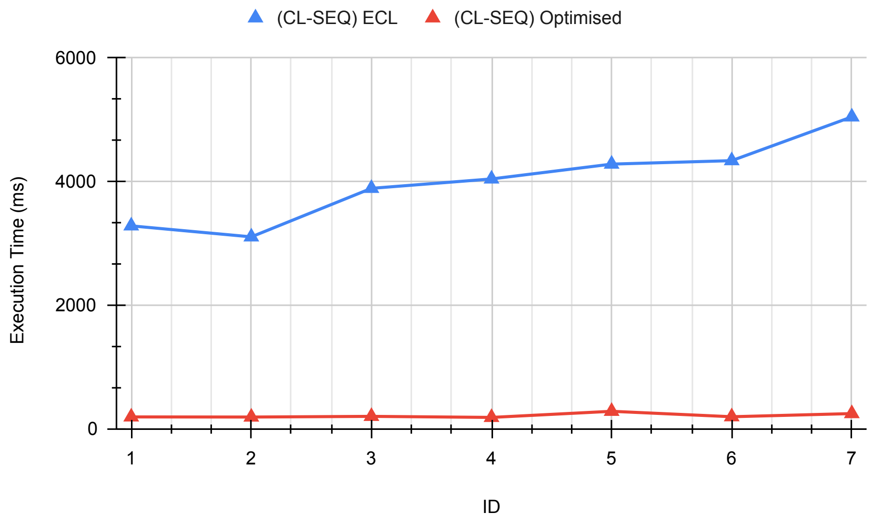


Figure 7.8: Comparison of Execution time in Class Sequence Diagram Comparison

significantly reduced in the proposed approach.

Another important factor to note here is that this approach might not bring performance benefits when comparing very small models, because precomputing the indices (as mentioned in the program rewriting section) has an overhead, which pays off for larger models, a much clearer improvement is expected to be seen in performance when it comes to larger models.

7.3.3 Threats to Validity

A primary threat to the validity of the results presented here, is that the measured performance may be particular to the models that were created for the tests, to the kind of model, or to the comparison programs that were proposed. A key challenge identified in MDE research is a lack of publicly accessible real-world models [128]. Although both synthetic and publicly available models are used, this can still affect the measured performance benefits. To further generalise the results, it is required to perform experiments with different models and comparison programs as well as with different modeling technologies such as Simulink and CDO to demonstrate the scalability of the proposed approach, especially for larger models.

As the rewriting is based on static analysis, it is recommended to explicitly state the types of the constructs wherever possible, to allow accurate type resolution and enable automated rule optimisation (as described in Section 7.2).

7.4 Related Work

Model comparison deals with finding similarities and differences between elements of different models. This comparison can be done on the basis of structure, semantics and metrics [129]. In the context of this chapter the state-of-the-art work that involves structural model comparison has been discussed.

It has been demonstrated in [130] that conventional text-based comparison and differencing techniques are insufficient for model comparison due to the structured nature of models.

Model-to-model transformations have been shown to be used for comparing models as in [125]. M2M languages are not tailored for model comparison task and hence generally are very verbose if used in the context of model comparison. M2M languages do not have tailored constructs for model comparison activities.

Change-based model comparison was presented in [131] where the comparison is done only for the model elements that have been changed since the previous version which is quite efficient compared to state-based comparison.

EMF Compare [132] & EMF Diff Merge [133] are two tools available to compare and then merge two models. EMF Compare uses built-in heuristics for model element references and attribute values while a tailored language like ECL lets one write custom matching rules for different model elements.

There are other comparison approaches as shown in [134] that compares different UML models but the approach is only limited to models conforming to a single metamodel. Additionally as mentioned in [53], most similarity based approaches such as SiDiff [130] and DSMDiff [135], have limited support when it comes to heterogeneous models which are supported by ECL, where one can specify complex matching algorithms for models conforming to multiple metamodels.

7.5 Chapter Summary

In this chapter, an approach is presented in **Section 7.2** for efficiently comparing models using programs written in a rule-based model comparison language followed by an example in **Section 7.1**. This efficient comparison approach incorporates an automatic rewriting facility to speed up model comparison (both homogeneous and heterogeneous) based on static analysis. The rewriting automatically extracts dynamic domains to provide pre-filtering of model elements before actually comparing them. Decreasing the cost of switching between comparison rules, allows to execute independent rules before those dependent on them, optimising the comparison process. In **Section 7.3**, it is shown through experiments that the proposed approach significantly reduces execution time when compared to the default ECL execution engine, resulting in performance gains. Finally related work is presented in **Section 7.4**.

Chapter 8

Conclusion

This chapter lists the contributions of this thesis to the field of model driven engineering, summarises its findings, and states potential directions for future work for this research work. The hypothesis of this research was as follows:

"The execution time of computing expensive queries in ***model validation****,* ***model-to-model transformation*** *and* ***model comparison*** *tasks over large EMF-based models can be significantly reduced using automated* ***program rewriting*** *techniques based on the in-advance information extracted through* ***static analysis****."*

A breakdown of the overall research goal into more fine-grained objectives is as follows:

- **RO-1: Identify the performance challenges involved in executing complex model management programs over large EMF-based models.**

  This research objective was achieved by reviewing the existing literature and the state-of-the-art tools. The challenges identified are detailed in Chapter 2.

- **RO-2: Identify reusable optimisation approaches and patterns across different model management programs using static analysis of high-level language programs.**

  This research objective was achieved by contributing static analysis facilities to Epsilon languages. Static analysis helps extract things such as type information, call graph and dependency graph and is described in detail in Chapter 4.

- **RO-3: Propose algorithms for the optimisation of model management programs operating on large scale EMF-based models.**

  This was achieved by contributing approaches for optimising model validation (Section 5.2), model-to-model transformation (Section 6.2) and model comparison programs (Section 7.2).

- **RO-4: Ensure to preserve the semantics of the original program while producing the optimised version.**

  This was achieved by performing equivalence testing for all the rewritten optimised programs. Automated JUnit tests were developed to check the semantic equivalence of the rewritten program with the original input program.

- **RO-5: Evaluate the results of the proposed algorithms in terms of execution time and memory footprint.**

  To achieve this research objective, experiments were conducted to evaluate the proposed approaches and are explained in detail in Sections 5.2.4, 6.3 and 7.3.

The following research questions have been set out to achieve the abovementioned research objectives in Section 3.2.

- **RQ-1: What are the performance challenges in terms of execution time for complex model management programs over large EMF-based models.**

  The challenges identified are listed in Chapter 2.

- **RQ-2: What information can be extracted before the execution of a model-management program to facilitate automated rewriting of such programs?**

  Static analysis helps extract information such as type resolution, dependency graph and call graph.

- **RQ-3: Can the information extracted from static analysis help to identify potential rewriting opportunities for model management programs?**

  Yes, it can be seen in proposed approaches how type information has helped to rewrite the optimised program. In Chapter 5 call graph generated using static analysis helped in optimising the program also in Chapters 6 and 7 dependency graph have been used to optimise the programs.

- **RQ-4: Does the optimised program generated by the rewriting approach produce the same output as the original program?**

  Yes, this has been confirmed using the automated JUnit testing to ensure the semantic equivalence of the rewritten programs.

- **RQ-5: Do static analysis and automatic program rewriting help in reducing memory footprint and execution time when complex model management programs are executed over large models?**

  Yes, it can be clearly observed in the experiments that the proposed optimisation approaches produce significant performance gains.

The proposed approaches and the results of the experiments conducted have confirmed the research hypothesis "The execution time of computing expensive queries in **model validation**, **model-to-model transformation** and **model comparison** tasks over large EMF-based models can be significantly reduced using automated **program rewriting** techniques based on the in-advance information extracted through **static analysis**."

8.1 Summary

Model querying and model validation programs: First, an approach and a prototype was presented for optimisation of type-level model queries (i.e. queries on *allInstances()*) built on top of Epsilon's EOL and EVL languages. The proposed approach detects expressions of interest using static call graph analysis and then augments the input program with index-building statements and replaces calls to said expressions with equivalent expressions that make use of the pre-computed indices. Experimental evaluation has demonstrated that the proposed approach can deliver significant performance benefits compared to the default EOL and EVL engine, particularly where larger models are involved.

Secondly, an architecture for automatically mapping certain expensive expressions from an EVL validation program to VQL patterns has been presented. This mapping of EVL expressions to VQL patterns leverages static type information extracted by the static analyser from the EMF models. The translated VQL patterns executed by the RETE engine outperform the sequential execution of EVL validation. It is discussed that this partial translation can help model validation scale well for large-scale models.

Selective traceability: Thirdly, an approach has been presented that is used to optimise programs written in rule-based M2M transformation languages. The proposed approach resolves the types of various constructs using static analysis and then creates a dependency graph between the transformation rules. Based on this dependency graph, the rule-based transformation program is rewritten into an imperative program that only maintains a selective trace. The evaluation experiments have demonstrated that the proposed approach can deliver significant performance benefits both in terms of execution time and memory footprint compared to the default ETL execution engine, mainly where larger models are involved.

Model comparison programs:Finally, an approach has been presented for efficiently comparing models using programs written in a rule-based model comparison language (ECL). This efficient comparison approach incorporates an automatic rewriting facility to speed up model comparison (both homogeneous and heterogeneous) based on static analysis. The rewriting automatically extracts dynamic domains to provide pre-filtering of model elements before actually comparing them. Additionally, static analysis helps reorder the rules based on the dependencies identified between these match rules by creating a dependency graph. This enables to execute dependency-free rules before those dependent on them, optimising the comparison process by reducing the cost of jumping between comparison rules. Through experiments, it is demonstrated that the proposed approach significantly improves execution time compared to the default ECL execution engine, providing substantial performance benefits.

8.2 Thesis Contributions

The contributions of this thesis are as follows:

- **Optimisation of model querying and model validation programs** over EMF-based models by creating automated custom indices. This is achieved using static analysis and program rewriting. Additionally, an approach has been contributed for translating EOL expressions to VIATRA to support incremental querying.
- **Optimisation of model-to-model (M2M) transformation programs** to reduce the size of the transformation trace by leveraging static analysis to translate from a rule-based M2M program to an imperative M2M program.
- **Optimisation of model comparison programs** written in Epsilon Comparison Language (ECL), by pre-filtering and indexing the elements to be matched using static analysis and automatically rewriting the dynamic domains (reducing the search space). ECLs grammar is extended to provide support for both static and dynamic domains. Static domain is computed once for each *MatchRule* while the dynamic domain is computed each time when the left model element is updated/

8.3 Future Work

One potential future work direction is to use the program optimisation strategies proposed in this work is to execute on top of partial loading. Partial loading helps reduce the loading time of the model and only load the necessary parts of the model while program optimisation on the top would further reduce the execution time. This would result in an overall decrease in end to end time for a model management operation. Other specific future work for different contributions are detailed below:

Model querying and model validation programs: Directions for future work include extending the proposed approach and prototype to support additional model management languages (e.g. for M2M/M2T transformation), additional modelling technologies (e.g. Simulink models i.e. translating from EOL to native MATLAB commands) and for detecting further language and modelling technology-specific optimisation opportunities.

The EVL to VIATRA work can be extended in further iterations to cover the mapping of more complex expressions, e.g., navigating multi-valued references. Besides rewriting queries to a different language, another way to improve the performance is to use the parallel EVL engine [106] or to cache all instances of every type in the model. Comparing the performance of these approaches with the RETE engine is also an interesting future direction.

Selective traceability: Directions for future work on selective traceability include conducting experiments to evaluate the proposed approach with other modelling technologies (e.g. Simulink models, repository-based models). Also, providing a disposal facility for the transformation trace can offer further memory footprint reductions. Moreover, using program analysis to deduct additional optimisation opportunities at the expression level is an interesting direction for potentially further improving the performance of such model-to-model transformations.

Model comparison programs: In future iterations, the proposed model comparison optimisation approach can be potentially used to provide correspondence between models from heterogeneous modelling technologies. For instance, it can facilitate the comparison between Simulink models and EMF models. Moreover, this automatic domain rewriting facility can be integrated with other rule-based languages such as Epsilon's pattern matching language (EPL).

Appendices

Appendix A

```
@builtin
operation String firstToLowerCase () : String {
}

@firstorder @builtin
operation Collection<Any> sortBy(a : Any) : Collection
    <Any> {
}

@firstorder @builtin
operation Collection<Any> collect(a: Any) : Collection
   <EolSelfExpressionType> {
}

@firstorder @builtin
operation Collection<Any> exists(a: Any) : Collection<
   EolSelfExpressionType> {
}

@builtin
operation String toCharSequence() : List<String> {
}

@builtin
operation String pad(a: Integer, b: String, c: Boolean
   ) : String {
}

@builtin
operation String isSubstringOf (a: String) : Boolean {
}

@builtin
```

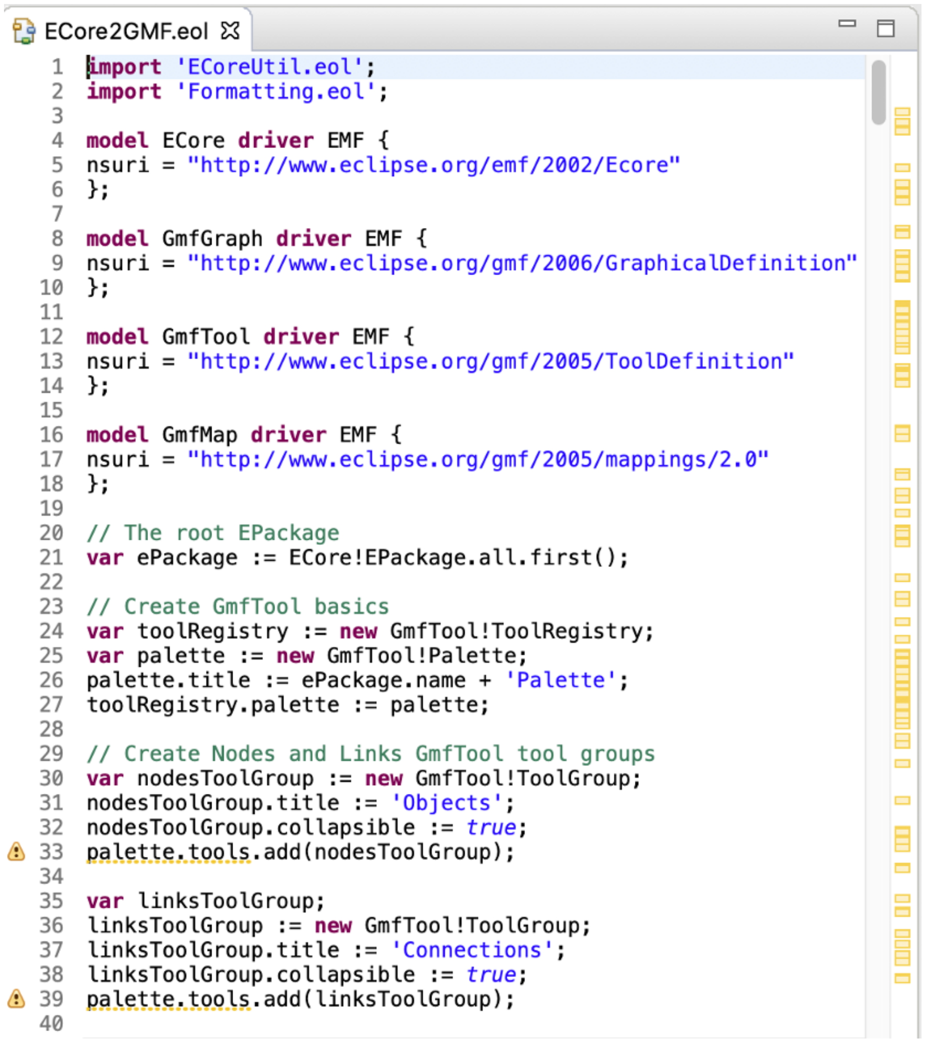
```
operation String startsWith (a: String) : Boolean {
}

@builtin
operation String store(where: String) {
}

@builtin
operation String firstToUpperCase() :String {
}

@builtin
operation String ftuc() : String {
}

@builtin
operation String trim() : String {
}

@builtin
operation String ftlc () : String {
}

@builtin
operation String characterAt(index: Integer): String {
}

@builtin
operation String escapeXml() : String {
}

@builtin
operation String setTarget(target: Any) {
}

@builtin
operation String toEnum() : Any {
}

@builtin
operation String contributesTo(target : Any) : Boolean
    {
}

@builtin
operation Integer mod(other: Integer) : Integer {
}

@builtin
operation Integer toHex() : String {
}

@builtin
operation Integer toBinary() : String {
}

@builtin
operation Integer iota (i: Integer, step: Integer) :
    Sequence <Integer> {
}

@builtin
operation Integer to (end: Integer) : Sequence<Integer
    > {
}

@builtin
operation Any asDate (format: String) : Date {
}

@builtin
operation Any asUnicode () : String {
}

@builtin
operation Any asBoolean () : Boolean{
}

@builtin
operation Any asLong () : Integer {
}

@builtin
operation Any asFloat () : Real {
}

@builtin
operation Any asDouble () : Real {
}

@builtin
operation Any asReal () : Real {
}

@builtin
operation Any isReal () : Boolean {
}

@builtin
operation Any isInteger () : Boolean {
}

@builtin
operation Any asInteger () : Integer {
}

@builtin
operation Any format () : String {
}

@builtin
operation Any errln (prefix: Any, suffix: Any) :
    EolSelf {
}

@builtin
operation Any errln (prefix: Any) : EolSelf {
}

@builtin
operation Any errln () : EolSelf {
}

@builtin
operation Any err (prefix: Any, suffix: Any) : EolSelf
     {
}

@builtin
operation Any err (prefix: Any) : EolSelf {
}

@builtin
operation Any err () : EolSelf {
}
@builtin
operation Any print (prefix: Any, suffix: Any) :
    EolSelf {
}
@builtin
operation Any print (prefix: Any) : EolSelf {
}

@builtin
operation Any print () : EolSelf {
}

@builtin
operation Any println (prefix: Any, suffix: Any) :
    EolSelf {
}

@builtin
operation Any println (prefix: Any) : EolSelf {
}

@builtin
operation Any println () : EolSelf {
}

@builtin
operation Any ifUndefined (alternative: Any) : Any {
}

@builtin
operation Any isUndefined () : Boolean {
}

@firstorder
operation String split(ab: String) : Collection<String
    > {
}

@builtin
operation Collection<Any> testCol () : EolSelf {
}

@builtin
operation Collection<Any> testContent () :
    EolSelfContentType {
}

@builtin
operation Collection<Any> first() : EolSelfContentType
```

```
    {
}

@builtin
operation Collection<Any> isEmpty() : Boolean {
}
@builtin
operation Collection<Any> notEmpty() : Boolean {
}

@builtin
operation Any isDefined() : Boolean {
}

@builtin
operation Any add(item : Any) : Boolean {
}

@builtin
operation Collection<Any> addAll(col : Collection) :
    Boolean {
}

@builtin
operation Collection<Any> clone() : Collection <Any> {
}

@builtin
operation Collection<Any> size() : Integer {
}

@builtin
operation Any asSequence() : Sequence {
}

@builtin
operation Collection<Any> asSequence() : Sequence {
}

@builtin
operation Collection<Any> at(index : Integer) : Any {
}

@builtin
operation Any isKindOf(type : Any) : Boolean {
}
```

```

@builtin
operation Any allOfKind() : Collection <Any> {
}

@builtin
operation Any allInstances() : Collection <EolSelf> {
}

@builtin
operation Any allOfType() : Collection <EolSelf> {
}

@builtin
operation Collection<Any> includes(item : Any) :
    Boolean {
}

@builtin
operation Collection<Any> excludes(item : Any) :
    Boolean {
}

@builtin
operation Any isTypeOf(type : Any) : Boolean {
}

@builtin
operation Collection<Any> get(key : Any) :
    EolSelfContentType {
}

@builtin
operation Collection<Any> testSelfCollection() :
    EolSelfCollectionType{
}

@builtin
operation Any asSet(): Set {
}

@builtin
operation Any satisfies(invariant :String) : Boolean {
}

@builtin
```

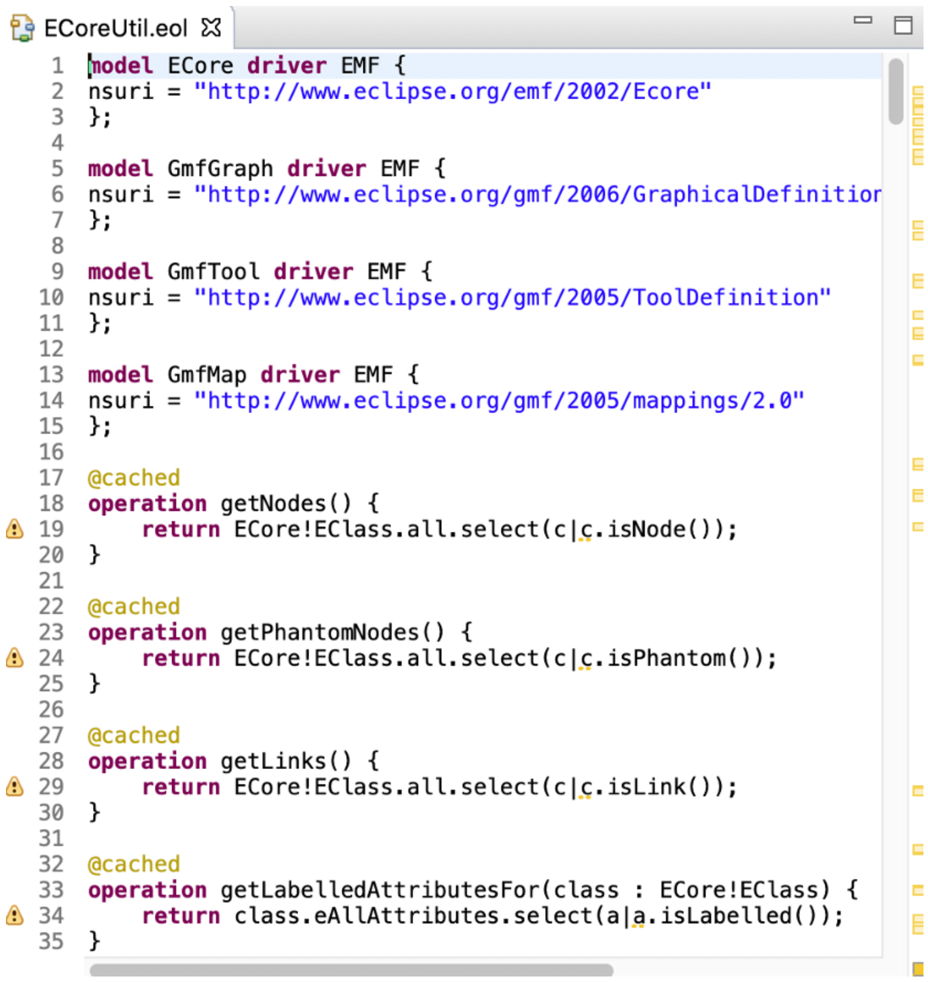
```
operation String substring(startIndex : Integer,
    endIndex : Integer) : String {
}

@builtin
operation String substring(startIndex : Integer) :
    String {
}

@builtin
operation String toLowerCase():String {
}

@builtin
operation String toUpperCase():String {
}

@builtin
operation Collection<Any> excludesAll(col : Collection
    <Any>) :Boolean {
}

@builtin
operation Collection<Any> flatten() :Collection<Any> {
}

@builtin
operation Any contains(param : Any) :Boolean {
}

@builtin
operation Any equivalent() :Any {
}

@builtin
operation Any matches(param: Any) :Boolean {
}

@builtin
operation String compareTo(str : String) :Integer{
}

@builtin
operation Collection<Any> excludingAll(a: Collection<
    Any>) :Collection<EolSelfContentType> {
}
```

```

@builtin
operation Any asString() :String {
}

@builtin
operation Any toString() :String {
}

@builtin
operation String indexOf(a: String) :Integer {
}

@builtin
operation Object type() :Object {
}
```
Listing A.1: An excerpt of builtin operation file


ECore2GMF.eol

```
import 'ECoreUtil.eol';
import 'Formatting.eol';

model ECore driver EMF {
nsuri = "http://www.eclipse.org/emf/2002/Ecore"
};

model GmfGraph driver EMF {
nsuri = "http://www.eclipse.org/gmf/2006/GraphicalDefinition"
};

model GmfTool driver EMF {
nsuri = "http://www.eclipse.org/gmf/2005/ToolDefinition"
};

model GmfMap driver EMF {
nsuri = "http://www.eclipse.org/gmf/2005/mappings/2.0"
};

// The root EPackage
var ePackage := ECore!EPackage.all.first();

// Create GmfTool basics
var toolRegistry := new GmfTool!ToolRegistry;
var palette := new GmfTool!Palette;
palette.title := ePackage.name + 'Palette';
toolRegistry.palette := palette;

// Create Nodes and Links GmfTool tool groups
var nodesToolGroup := new GmfTool!ToolGroup;
nodesToolGroup.title := 'Objects';
nodesToolGroup.collapsible := true;
palette.tools.add(nodesToolGroup);

var linksToolGroup;
linksToolGroup := new GmfTool!ToolGroup;
linksToolGroup.title := 'Connections';
linksToolGroup.collapsible := true;
palette.tools.add(linksToolGroup);

```

Figure A.1: Static analysis on Ecore2GMF.eol



```
model ECore driver EMF {
nsuri = "http://www.eclipse.org/emf/2002/Ecore"
};

model GmfGraph driver EMF {
nsuri = "http://www.eclipse.org/gmf/2006/GraphicalDefinitior
};

model GmfTool driver EMF {
nsuri = "http://www.eclipse.org/gmf/2005/ToolDefinition"
};

model GmfMap driver EMF {
nsuri = "http://www.eclipse.org/gmf/2005/mappings/2.0"
};

@cached
operation getNodes() {
    return ECore!EClass.all.select(c|c.isNode());
}

@cached
operation getPhantomNodes() {
    return ECore!EClass.all.select(c|c.isPhantom());
}

@cached
operation getLinks() {
    return ECore!EClass.all.select(c|c.isLink());
}

@cached
operation getLabelledAttributesFor(class : ECore!EClass) {
    return class.eAllAttributes.select(a|a.isLabelled());
}
```

Figure A.2: Static analysis on EcoreUtil.eol

Bibliography

[1] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework.* Pearson Education, 2008.

[2] "Epsilon Model Connectivity Layer," https://www.eclipse.org/epsilon/doc/emc/, 2022, [Online; accessed 29-April-2022].

[3] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi *et al.*, "A research roadmap towards achieving scalability in model driven engineering," in *Proceedings of the Workshop on Scalability in Model Driven Engineering*, 2013, pp. 1–10.

[4] D. S. Kolovos, R. Wei, and K. Barmpis, "An approach for efficient querying of large relational datasets with ocl-based languages," in *XM 2013-Extreme Modeling Workshop*, vol. 48, 2013.

[5] K. Barmpis and D. S. Kolovos, "Towards scalable querying of large-scale models," in *European Conference on Modelling Foundations and Applications.* Springer, 2014, pp. 35–50.

[6] Y. Zhao, "Research on mongodb design and query optimization in vehicle management information system," in *Applied Mechanics and Materials*, vol. 246. Trans Tech Publ, 2013, pp. 418–422.

[7] S. Chaudhuri, "An overview of query optimization in relational systems," *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems - PODS 98*, 1998.

[8] J. Bézivin, "Model driven engineering: An emerging technical space," in *International Summer School on Generative and Transformational Techniques in Software Engineering.* Springer, 2005, pp. 36–64.

[9] D. C. Schmidt, "Model-driven engineering," *COMPUTER-IEEE COMPUTER SOCIETY-*, vol. 39, no. 2, p. 25, 2006.

[10] J. Cooper, A. De la Vega, R. Paige, D. Kolovos, M. Bennett, C. Brown, B. S. Piña, and H. H. Rodriguez, "Model-based development of engine control systems: Experiences and lessons learnt," in *2021 ACM/IEEE*

24th International Conference on Model Driven Engineering Languages and Systems (MODELS), 2021, pp. 308–319.

[11] A. Jaaksi, "Developing mobile browsers in a product line," *IEEE software*, vol. 19, no. 4, pp. 73–80, 2002.

[12] J. Kärnä, J.-P. Tolvanen, and S. Kelly, "Evaluating the use of domain-specific modeling in practice," in *Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling*, 2009.

[13] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The grand challenge of scalability for model driven engineering," in *International Conference on Model Driven Engineering Languages and Systems.* Springer, 2008, pp. 48–53.

[14] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi *et al.*, "A research roadmap towards achieving scalability in model driven engineering," in *Proceedings of the Workshop on Scalability in Model Driven Engineering*, 2013, pp. 1–10.

[15] J. Rymer, C. Richardson, C. Mines, D. Jones, D. Whittaker, J. Miller, and I. McPherson, "Low-code platforms deliver customer-facing apps fast, but will they scale up," *Forrester Research.*

[16] "Object Constraint Language," https://www.omg.org/spec/OCL/2.4/About-OCL/, 2022, [Online; accessed 29-April-2022].

[17] Accelo. [Online]. Available: https://www.eclipse.org/acceleo/

[18] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, "Atl: a qvt-like transformation language," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006, pp. 719–720.

[19] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The epsilon object language (eol)," in *European conference on model driven architecture-foundations and applications.* Springer, 2006, pp. 128–142.

[20] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi, "Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework," *Software and Systems Modeling*, vol. 15, no. 3, pp. 609–629, 2016.

[21] A. Bucchiarone, J. Cabot, R. F. Paige, and A. Pierantonio, "Grand challenges in model-driven engineering: an analysis of the state of the research," *Software and Systems Modeling*, pp. 1–9, 2020.

[22] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis lectures on software engineering*, vol. 3, no. 1, pp. 1–207, 2017.

[23] G. Booch, *The unified modeling language user guide.* Pearson Education India, 2005.

[24] C. B. Moler, "Introduction to MATLAB / SIMULINK," in *Numerical Computing with MATLAB, Revised Reprint.* Philadelphia: SIAM, 2008, ch. 1, pp. 1–51. [Online]. Available: http://epubs.siam.org/doi/book/10.1137/1.9780898717952

[25] A. Josey *et al.*, *ArchiMate® 2.1–A Pocket Guide.* Van Haren, 2013.

[26] T. Allweyer, *BPMN 2.0: introduction to the standard for business process modeling.* BoD–Books on Demand, 2016.

[27] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis lectures on software engineering*, vol. 3, no. 1, pp. 1–207, 2017.

[28] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE software*, vol. 20, no. 5, pp. 42–45, 2003.

[29] D. Kolovos, "An extensible platform for specification of integrated languages for model management," Ph.D. dissertation, Citeseer, 2008.

[30] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of mde in industry," in *Proceedings of the 33rd international conference on software engineering*, 2011, pp. 471–480.

[31] M. Zeller, D. Ratiu, and K. Höfig, "Towards the adoption of model-based engineering for the development of safety-critical systems in industrial practice," in *International Conference on Computer Safety, Reliability, and Security.* Springer, 2016, pp. 322–333.

[32] O. Philipp, K. Stefan, and S. Eric, "Model-based resource analysis and synthesis of service-oriented automotive software architectures," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS).* IEEE, 2019, pp. 128–138.

[33] W. Haberl, M. Herrmannsdoerfer, S. Kugele, M. Tautschnig, and M. Wechs, "Seamless model-driven development put into practice," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation.* Springer, 2010, pp. 18–32.

[34] P. Roques, "Mbse with the arcadia method and the capella tool," 2016.

[35] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange, "Autosar–a worldwide standard is on the road," in *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, vol. 62, 2009, p. 5.

[36] M. Tisi, J.-M. Mottu, D. Kolovos, J. de Lara, E. Guerra, D. Di Ruscio, A. Pierantonio, and M. Wimmer, “Lowcomote: Training the next generation of experts in scalable low-code engineering platforms,” 2019.

[37] Ibm raphsody. [Online]. Available: https://www.ibm.com/products/systems-design-rhapsody

[38] Magic draw. [Online]. Available: https://www.nomagic.com/products/magicdraw.

[39] Ptc integrite modeler. [Online]. Available: https://www.ptc.com/en/products/plm/plm-products/integritymodeler-what-is-new.

[40] Ansys scade. [Online]. Available: https://www.ansys.com/products/embedded-software/ansys-scadesuite

[41] Jetbrains mps. [Online]. Available: https://www.jetbrains.com/mps/.

[42] Atlas Transformation Language. Last Accessed on 2021, Aug 31. [Online]. Available: https://www.eclipse.org/atl/

[43] Viatra. [Online]. Available: https://www.eclipse.org/viatra/

[44] Object management group. meta object facility. [Online]. Available: http://www.omg.org/mof/

[45] Emfatic language reference. [Online]. Available: http://www.eclipse.org/epsilon/doc/articles/emfatic/

[46] “Sirius,” https://www.obeosoft.com/en/products/eclipse-sirius.

[47] D. Kolovos, L. Rose, A. Garcia-Dominguez, and R. Paige, *The Epsilon Book. Eclipse*, 2010.

[48] R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos, and F. A. Polack, “The design of a conceptual framework and technical infrastructure for model management language engineering,” in *2009 14th IEEE International Conference on Engineering of Complex Computer Systems.* IEEE, 2009, pp. 162–171.

[49] D. S. Kolovos, R. F. Paige, and F. A. Polack, “On the evolution of ocl for capturing structural constraints in modelling languages,” in *Rigorous Methods for Software Construction and Analysis.* Springer, 2009, pp. 204–218.

[50] ——, “The epsilon transformation language,” in *International Conference on Theory and Practice of Model Transformations.* Springer, 2008, pp. 46–60.

[51] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. Polack, “The epsilon generation language,” in *European Conference on Model Driven Architecture-Foundations and Applications.* Springer, 2008, pp. 1–16.

[52] D. S. Kolovos, R. F. Paige, and F. A. Polack, "Merging models with the epsilon merging language (eml)," in *International Conference on Model Driven Engineering Languages and Systems.* Springer, 2006, pp. 215–229.

[53] D. S. Kolovos, "Establishing correspondences between models with the epsilon comparison language," in *Model Driven Architecture - Foundations and Applications*, R. F. Paige, A. Hartman, and A. Rensink, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 146–157.

[54] D. S. Kolovos and R. F. Paige, "The epsilon pattern language," in *2017 IEEE/ACM 9th International Workshop on Modelling in Software Engineering (MiSE).* IEEE, 2017, pp. 54–60.

[55] L. M. Rose, D. S. Kolovos, R. F. Paige, F. A. Polack, and S. Poulding, "Epsilon flock: A model migration language," *Softw. Syst. Model.*, vol. 13, no. 2, p. 735–755, May 2014.

[56] S. Popoola, D. S. Kolovos, and H. H. Rodriguez, "Emg: A domain-specific transformation language for synthetic model generation," in *Theory and Practice of Model Transformations: 9th International Conference, ICMT 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-5, 2016, Proceedings 9.* Springer, 2016, pp. 36–51.

[57] A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige, and I. Medina-Bulo, "Eunit: a unit testing framework for model management tasks," in *Model Driven Engineering Languages and Systems: 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings 14.* Springer, 2011, pp. 395–409.

[58] A. De La Vega, P. Sánchez, and D. Kolovos, "Pinset: A dsl for extracting datasets from models for data mining-based quality analysis," in *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC).* IEEE, 2018, pp. 83–91.

[59] D. S. Kolovos, R. F. Paige, F. A. Polac, and L. M. Rose, "Update transformations in the small with the epsilon wizard language," *Journal of Object Technology*, vol. 6, no. 9, pp. 53–69, Oct. 2007, tOOLS EUROPE 2007 — Objects, Models, Components, Patterns. [Online]. Available: http://www.jot.fm/contents/issue_2007_10/paper3.html

[60] P. Inostroza and T. van der Storm, "The ttc 2014 movie database case: Rascal solution," vol. 1305, 01 2014.

[61] M. Búr, Z. Ujhelyi, Á. Horváth, and D. Varró, "Local search-based pattern matching features in EMF-IncQuery," in *Proc. of the 8th International Conference on Graph Transformations.*, ser. Lecture Notes in Computer Science, vol. 9151. Springer, 2015, pp. 275–282.

[62] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró, "EMF-IncQuery: An integrated development environment for live model queries," *Science of Computer Programming*, vol. 98, pp. 80–99, 2015.

[63] C. Forgy, "Rete: A fast algorithm for the many patterns/many objects match problem," *Artif. Intell.*, vol. 19, no. 1, pp. 17–37, 1982.

[64] P. Mohagheghi and V. Dehlen, "Where is the proof?-a review of experiences from applying mde in industry," in *European Conference on Model Driven Architecture-Foundations and Applications.* Springer, 2008, pp. 432–443.

[65] M. Barbero, F. Jouault, and J. Bézivin, "Model driven management of complex systems: Implementing the macroscope's vision," in *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ecbs 2008)*, 2008, pp. 277–286.

[66] "The connected data objects model repository (CDO) project," http://eclipse.org/cdo, 2023, [Online; accessed 26-May-2023].

[67] "Bryan Hunt. Mongo-EMF, howpublished = "https://github.com/BryanHunt/mongo-emf/.", year = 2023, note = "[online; accessed 26-may-2023]"."

[68] J. E. Pagán, J. S. Cuadrado, and J. G. Molina, "Morsa: A scalable approach for persisting and accessing large models," in *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 77–92.

[69] D. Kolovos, R. Paige, and F. Polack, "The grand challenge of scalability for model driven engineering," in *International Conference on Model Driven Engineering Languages and Systems (MODELS2008)*, ser. Lecture Notes in Computer Science, vol. 5421. Springer, Berlin, Heidelberg, 04 2008, pp. 48–53.

[70] I. Ráth, G. Varró, and D. Varró, "Change-driven model transformations," in *Model Driven Engineering Languages and Systems*, A. Schürr and B. Selic, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 342–356.

[71] F. Jouault, J. Bézivin, and M. Barbero, "Towards an advanced model-driven engineering toolbox," *ISSE*, vol. 5, pp. 5–12, 03 2009.

[72] R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. Szalkowski, "Grgen: A fast spo-based graph rewriting tool," in *Graph Transformations*, A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 383–397.

[73] K. Barmpis and D. S. Kolovos, "Evaluation of contemporary graph databases for efficient persistence of large-scale models." *Journal of Object Technology*, vol. 13, no. 3, pp. 3–1, 2014.

[74] J. E. Pagán and J. G. Molina, "Querying large models efficiently," *Information and Software Technology*, vol. 56, no. 6, pp. 586–622, 2014.

[75] J. E. Pagán, J. S. Cuadrado, and J. G. Molina, "Morsa: A scalable approach for persisting and accessing large models," in *International Conference on Model Driven Engineering Languages and Systems.* Springer, 2011, pp. 77–92.

[76] E. D. Willink, "Aligning ocl with uml," *Electronic Communications of the EASST*, vol. 44, 2011.

[77] U. Konrad and L. Iskhakova, *Innovation Information Technologies: Theory and Practice.* Forschungszentrum Dresden-Rossendorf, 2010.

[78] M. Egea, C. Dania, and M. Clavel, "Mysql4ocl: A stored procedure-based mysql code generator for ocl," *Electronic Communications of the EASST*, vol. 36, 2010.

[79] M. Egea and C. Dania, "Sql-pl4ocl: an automatic code generator from ocl to sql procedural language," *Software & Systems Modeling*, vol. 18, no. 1, pp. 769–791, 2019.

[80] R. Wei and D. S. Kolovos, "An efficient computation strategy for allinstances ()." in *BigMDE@ STAF*, 2015, pp. 32–41.

[81] S. Madani, D. Kolovos, and R. F. Paige, "Towards optimisation of model queries: a parallel execution approach," *Journal of Object Technology*, vol. 18, no. 2, 2019.

[82] G. Daniel, G. Sunyé, and J. Cabot, "Scalable queries and model transformations with the mogwai tool," in *International Conference on Theory and Practice of Model Transformations.* Springer, 2018, pp. 175–183.

[83] J. Makai, G. Szárnyas, I. Z. Ráth, D. Varró, and Á. Horváth, "Optimization of incremental queries in the cloud," 2015.

[84] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and extensible algorithms for multi query optimization," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000, pp. 249–260.

[85] "MongoDB Atlas for Industries," https://www.mongodb.com/industries.

[86] Q. u. a. Ali, D. Kolovos, and K. Barmpis, "Efficiently querying large-scale heterogeneous models," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and*

Systems: Companion Proceedings, ser. MODELS '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3417990.3420207

[87] R. Wei and D. Kolovos, "Automated analysis, validation and suboptimal code detection in model management programs," in *CEUR Workshop Proceedings*, vol. 1206, 01 2014, pp. 48–57.

[88] L. M. Rose, D. S. Kolovos, and R. F. Paige, "Eugenia live: a flexible graphical modelling tool," in *Proceedings of the 2012 Extreme Modeling Workshop*, 2012, pp. 15–20.

[89] J. S. Cuadrado, E. Guerra, and J. de Lara, "Anatlyzer: An advanced ide for atl model transformations," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 85–88.

[90] K. Born, T. Arendt, F. Heß, and G. Taentzer, "Analyzing conflicts and dependencies of rule-based transformations in henshin," in *Fundamental Approaches to Software Engineering*, A. Egyed and I. Schaefer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 165–168.

[91] Z. Ujhelyi, "Static analysis of model transformations," Master's thesis, Budapest University of Technology and Economics, May 2009.

[92] E. D. Willink, "Modeling the OCL standard library," *ECEASST*, vol. 44, 2011.

[93] ——, "Safe navigation in OCL," in *Proceedings of the 15th International Workshop on OCL and Textual Modeling co-located with 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, Canada, September 28, 2015*, vol. 1512, 2015, pp. 81–88.

[94] Q. U. A. Ali, B. Horváth, D. Kolovos, K. Barmpis, and A. Horváth, "Towards scalable validation of low-code system models: Mapping evl to viatra patterns," in *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2021, pp. 83–87.

[95] UML2. Last Accessed on 2021, Aug 31. [Online]. Available: https://www.eclipse.org/modeling/mdt/?project=uml2

[96] M. Hanysz, T. Hoppe, A. Uhl, A. Seibel, H. Giese, P. Berger, and S. Hildebrandt, "Navigating across non-navigable ecore references via ocl," *Electronic Communications of the EASST*, vol. 36, 2010.

[97] Acceleo. Last Accessed on 2021, Aug 31. [Online]. Available: https://www.eclipse.org/acceleo/

[98] D. Kolovos, A. de la Vega, and J. Cooper, “Efficient generation of graphical model views via lazy model-to-text transformation,” in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 12–23.

[99] S. Madani, D. S. Kolovos, and R. F. Paige, “Parallel model validation with epsilon,” in *European Conference on Modelling Foundations and Applications.* Springer, 2018, pp. 115–131.

[100] Find Bugs Project. Last Accessed on 2021, Aug 31. [Online]. Available: https://findbugs.sourceforge.net/bugDescriptions.html

[101] LinTra. Last Accessed on 2021, Aug 31. [Online]. Available: https://www.atenea.lcc.uma.es/projects/LinTra.html

[102] H. Brunelière, J. Cabot, G. Dupé, and F. Madiot, “Modisco: A model driven reverse engineering framework,” *Information and Software Technology*, vol. 56, no. 8, pp. 1012–1032, 2014.

[103] K. Barmpis and D. Kolovos, “Hawk: Towards a scalable model indexing architecture,” in *Proceedings of the Workshop on Scalability in Model Driven Engineering*, 2013, pp. 1–9.

[104] D. Kolovos, L. Rose, R. Paige, E. Guerra, J. Cuadrado, J. De Lara, I. Ráth, D. Varró, G. Sunyé, and M. Tisi, “Mondo: scalable modelling and model management on the cloud,” in *STAF2015 Project Showcase*, 2015.

[105] “Epsilon Validation Language to Viatra Patterns,” https://github.com/lowcomote/evl-viatra-prototype.git.

[106] S. Madani, D. S. Kolovos, and R. F. Paige, “Parallel model validation with Epsilon,” in *Proc. of the 14th European Conference on Modelling Foundations and Applications, ECMFA@STAF 2018*, ser. Lecture Notes in Computer Science, vol. 10890. Springer, 2018, pp. 115–131.

[107] K. Barmpis and D. S. Kolovos, “Towards scalable querying of large-scale models,” in *Modelling Foundations and Applications*, J. Cabot and J. Rubin, Eds. Cham: Springer International Publishing, 2014, pp. 35–50.

[108] R. Wei and D. S. Kolovos, “An efficient computation strategy for allinstances ().” in *BigMDE@ STAF*, 2015, pp. 32–41.

[109] S. Madani and D. K. R. F. Paige, “Towards optimisation of model queries: A parallel execution approach,” *Journal of Object Technology*, vol. 18, no. 2, pp. 3:1–21, 2019, the 15th European Conference on Modelling Foundations and Applications.

[110] G. Daniel, G. Sunyé, and J. Cabot, "Scalable queries and model transformations with the mogwai tool," in *International Conference on Theory and Practice of Model Transformations.* Springer, 2018, pp. 175–183.

[111] D. S. Kolovos, R. Wei, and K. Barmpis, "An approach for efficient querying of large relational datasets with OCL based languages," in *In Proc. of the Workshop on Extreme Modeling co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems, MODELS 2013*, ser. CEUR Workshop Proceedings, vol. 1089. CEUR-WS.org, 2013, pp. 46–54.

[112] F. Heidenreich, C. Wende, and B. Demuth, "A framework for generating query language code from OCL invariants," *Electronic Communications of the EASST*, vol. 9, 2007.

[113] B. Sanchez, A. Zolotas, H. Hoyos Rodriguez, D. Kolovos, and R. Paige, "On-the-fly translation and execution of OCL-like queries on simulink models," in *Proc. of the 22nd International Conference on Model Driven Engineering Languages and Systems, MODELS*, 2019, pp. 205–215.

[114] G. Bergmann, "Translating OCL to graph patterns," in *Proc. of the 17th International Conference on Model-Driven Engineering Languages and Systems, MODELS 2014*, ser. Lecture Notes in Computer Science, vol. 8767. Springer, 2014, pp. 670–686.

[115] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The epsilon transformation language," in *International Conference on Theory and Practice of Model Transformations.* Springer, 2008, pp. 46–60.

[116] D. Strüber, T. Kehrer, T. Arendt, C. Pietsch, and D. Reuling, "Scalability of model transformations: Position paper and benchmark set." in *BigMDE@ STAF*, 2016, pp. 21–30.

[117] "A2L GitHub repository," https://github.com/anatlyzer/a2l.git, 2022, [Online; accessed 29-April-2022].

[118] M. Lawley, K. Duddy, A. Gerber, and K. Raymond, "Language features for re-use and maintainability of mda transformations," in *Workshop on Best Practices for Model-Driven Software Development*, 2004.

[119] "Yet Another Model Transformation Language," https://yamtl.github.io, 2022, [Online; accessed 29-April-2022].

[120] "EMF Compare," https://www.eclipse.org/emf/compare/, 2022, [Online; accessed 29-April-2022].

[121] J. Sanchez Cuadrado, L. Burgueno, M. Wimmer, and A. Vallecillo, "Efficient execution of atl model transformations using static analysis and parallelism," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[122] G. Daniel, F. Jouault, G. Sunyé, and J. Cabot, "Gremlin-atl: a scalable model transformation framework," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 462–472.

[123] A. Boronat, "Incremental execution of rule-based model transformation," *International Journal on Software Tools for Technology Transfer*, vol. 23, no. 3, pp. 289–311, 2021.

[124] D. Hearnden, M. Lawley, and K. Raymond, "Incremental model transformation for the evolution of model-driven systems," in *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*, ser. MoDELS'06. Berlin, Heidelberg: Springer-Verlag, 2006, p. 321–335. [Online]. Available: https://doi.org/10.1007/11880240_23

[125] M. D. Del Fabro and P. Valduriez, "Semi-automatic model integration using matching transformations and weaving models," in *Proceedings of the 2007 ACM Symposium on Applied Computing*, ser. SAC '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 963–970. [Online]. Available: https://doi.org/10.1145/1244002.1244215

[126] Q. u. a. Ali, D. Kolovos, and K. Barmpis, "Selective traceability for rule-based model-to-model transformations," in *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 98–109. [Online]. Available: https://doi.org/10.1145/3567512.3567521

[127] F. Khorram, M. Taromirad, and R. Ramsin, "Sega4biz: Model-driven framework for developing serious games for business processes."

[128] J. A. H. López and J. S. Cuadrado, "Towards the characterization of realistic model generators using graph neural networks," in *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2021, pp. 58–69.

[129] L. Gonçales, K. Farias, M. Scholl, T. Oliveira, and M. Veronez, "Model comparison: a systematic mapping study," 07 2015.

[130] C. Treude, S. Berlik, S. Wenzel, and U. Kelter, "Difference computation of large models," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 295–304.

[131] A. Yohannis, R. H. Rodriguez, F. Polack, and D. Kolovos, "Towards efficient comparison of change-based models," *Journal of Object Technology*, vol. 18, no. 2, pp. 7:1–21, Jul. 2019, the 15th European Conference on Modelling Foundations and Applications. [Online]. Available: http://www.jot.fm/contents/issue_2019_02/article7.html

[132] "Eclipse EMF Compare," https://projects.eclipse.org/projects/modeling.emfcompare, 2023, [Online; accessed 10-April-2023].

[133] "EMF DiffMerge," https://wiki.eclipse.org/EMF_DiffMerge, 2023, [Online; accessed 10-April-2023].

[134] D. Ohst, M. Welle, and U. Kelter, "Differences between versions of uml diagrams," *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 5, p. 227–236, sep 2003. [Online]. Available: https://doi.org/10.1145/949952.940102

[135] Y. Lin, J. Gray, and F. Jouault, "Dsmdiff: a differentiation tool for domain-specific models," *European Journal of Information Systems*, vol. 16, no. 4, pp. 349–361, 2007.