Efficient Management of Large Models via Static Analysis

Sorour Jahanbin

Doctor of Philosophy

University of York
Department of Computer Science

November 2023

Abstract

As the size of software and system models grows, scalability issues in the current generation of model management languages (e.g. transformation, validation) and their supporting tooling become more prominent. With the growing popularity of MDE in larger projects, the efficient management and processing of large models have become critical considerations. To address this challenge, execution engines of model management programs need to become more efficient in their use of system resources. Effective resource management is essential not only for minimizing execution costs but also for optimizing resource usage, particularly in scenarios where resources are billed based on usage patterns. This thesis addresses this challenge by presenting an approach to enhance the efficiency of model management programs, which play a pivotal role in querying and manipulating models. This approach focuses on enabling execution engines to load only the necessary parts of models, minimizing the overhead associated with loading unnecessary model elements into memory. Through the utilization of in-advance knowledge obtained from static analysis of model management programs, execution engines can identify, load, and process only the model elements essential for execution. Furthermore, the approach ensures that elements are disposed of from memory when no longer needed, optimizing both memory utilization and processing time. Experimental evaluations demonstrate that our approach empowers model management programs to process larger models faster with a reduced memory footprint compared to current state-of-the-art approaches.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Acknowledgments

Primarily, I want to express my profound gratitude to my supervisors, Professor Dimitris Kolovos and Dr Simos Gerasimou, for their unwavering support, invaluable guidance, and boundless patience during this research journey. Their wisdom, expertise, and mentorship played a pivotal role in shaping the trajectory of my work. Working with Dimitris and the research group was a dream come true, and I consider myself exceptionally fortunate to have had this invaluable experience. I can vividly recall the day I received the acceptance email; tears welled up in my eyes, and I couldn't believe what I was seeing.

I am also thankful to the members of my thesis committee, Dr Nelly Bencomo and Dr Lilian Blot, for their insightful feedback and the time they dedicated to reviewing and evaluating my research. Their expertise and diverse perspectives would greatly enrich the quality of this thesis.

The successful completion of this thesis was made possible by the invaluable support of my dear friend Qurat ul ain. Her encouragement and companionship, particularly during challenging moments such as homesickness due to COVID-19 and PhD-related disappointments, were instrumental in sustaining my motivation. I'd also like to express my gratitude to my friends Farzad, Mahboobeh, and Mohammad for their support and camaraderie. I want to convey my deepest, heartfelt thanks to my master's supervisor, Dr. Bahman Zamani, for the support that instilled in me the confidence and belief that I could pursue a PhD. His guidance and belief in my potential have been a driving force behind my academic journey, and I am profoundly grateful for his influence on this important decision.

I want to express my deepest appreciation to my family for their love and encouragement throughout my whole life, including my challenging journey through the PhD program. My father, Ali, my mother, Soheila, my sister, Samin, and my brother, Amir, have been a constant source of strength, offering support and motivation during these significant years. During the intense final year of my PhD, I am especially grateful for the unwavering presence and support of my partner, Hamid. His encouragement, patience, and unwavering belief in me have been the cornerstone of my progress. His presence has made this challenging journey more manageable, and I deeply appreciate the support.

This research was made possible by the support of the Lowcomote Training Network, which received funding from the European Union's Horizon 2020 Research and Innovation Programme under the Marie Skłodowska-Curie grant agreement no. 813884. Their generous funding was instrumental in the completion of my PhD thesis.

Lastly, I want to thank all the research participants who generously offered their time and insights, contributing to the success of this study, including Dr Kostas Barmpis and Dr Gerson Sunyé, as well as all the members of the Lowcomote project.

Declaration

I declare that this thesis is a presentation of original work, and I am the sole author. This work has not previously been presented for a degree or other qualification at this University or elsewhere. All sources are acknowledged as references. Except where explicitly mentioned, this thesis represents the author's original work. Certain sections of this thesis have been previously published by the author, and these publications are primarily authored by the PhD candidate.

- **Conference**
  - *Sorour Jahanbin, Dimitris Kolovos, Simos Gerasimou, and Gerson Sunyé. 2022. Partial Loading of Repository-Based Models through Static Analysis. In Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2022). Association for Computing Machinery, New York, NY, USA, 266–278.*
  - *Sorour Jahanbin, Dimitris Kolovos, Simos Gerasimou. 2023. Towards Memory-Efficient Validation of Large XMI Models. In Proceedings of the 15th System Analysis and Modelling Conference (SAM 2023)*
- **Workshop**
  - *Sorour Jahanbin, Dimitris Kolovos, and Simos Gerasimou. 2020. Intelligent Run-time Partitioning of Low-code System 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 64, 1–5.*
- **Symposium**
  - *Sorour Jahanbin. 2021. Efficient Model Loading through Static Analysis. In Proceedings of the 24th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS '21), Fukuoka, Japan, 2021, pp. 660-665.*

1 Introduction

Technology is crucial in shaping various aspects of human life in the modern era. A key driver of this technological evolution is software, which profoundly influences how individuals interact, work, and devise solutions. Software development is a complex process of instructing computers to perform specific tasks in alignment with human intentions. This involves the adept use of programming languages, frameworks, and tools to translate complex ideas into executable code.

As software projects expand in scope and complexity, ensuring effective communication among developers becomes increasingly challenging. This growing complexity underscores the importance of modelling, a fundamental technique that facilitates the comprehensive design, analysis, and documentation of various aspects of software systems.

Modelling operates at a higher level of abstraction, offering developers a powerful tool to conceptualise the intricate structures and behaviours of software systems. Software models can take the form of visual, textual, mathematical, or other representations. With modelling, developers can create diagrams and representations that provide a clear roadmap for the development process. These models help to ensure that all team members share a common understanding of the system's architecture, functionality, and interactions. They serve as a visual language that bridges the gap between complex technical details and the broader project goals.

This approach to software development simplifies problem-solving, reduces errors, and enhances collaboration. It enables teams to spot potential issues early in the development cycle, saving time and resources that might be wasted on correcting problems later.

In the domain of software development, modern applications can be highly complex. Attempting to create a comprehensive model that accounts for every single detail would not only be impractical but could also hinder the development process by overloading it with unnecessary information. Therefore, developers employ a thoughtful approach, identifying the key components, interactions, and essential functionalities to achieve the project's goals. By concentrating their modelling efforts on these crucial elements, they can create a more focused and useful representation of the system.

This selective modelling strategy not only refines the development process but also enhances the clarity of the model itself. It allows teams to prioritise what truly matters, making it easier to manage, analyse, and communicate about the system. It is akin to zooming in on the critical details of a complex map rather than attempting to absorb the entire landscape at once.

Traditionally, models have held a crucial role in the documentation of systems. However, a paradigm shift known as Model-Driven Engineering (MDE) has emerged in Computer Science. MDE transformed the software development process by making models its central focus. This transition from code-centric approaches to model-centric methodologies introduces a new level of precision and efficiency into software development. Through these models, which comprehensively encapsulate a system's structure and behaviour, MDE greatly enhances the aspects of design, communication, and collaboration within development teams. One of the core concepts in MDE is the use of model transformation programs. These programs take input from one or more models and produce output in the form of models or code. Model transformations enable the automatic or semi-automatic generation of code, documentation, or other artefacts from the models. This automation helps reduce manual errors and improve efficiency.

As software systems grow in complexity, the adoption of MDE becomes increasingly attractive for its ability to handle intricacies through modularisation, clear separation of concerns, and effective visualisation techniques. However, this shift toward MDE, especially in large-scale projects, brings scalability challenges to the forefront. Storing, loading, querying, and transforming extensive models can place considerable pressure on existing tools and practices.

Given MDE's rising popularity in larger projects, efficiently managing and processing large models has become a critical concern. To address this, model management programs, which are designed to query and manipulate models, are needed to execute efficiently by their execution engine. These programs must be adept at handling the demands of extensive models in terms of both memory utilisation and processing time. Effective resource management becomes crucial, not only to reduce execution costs but also to optimise resource usage, particularly when resources are charged based on usage patterns.

Indeed, the expanding size of models presents a substantial challenge for existing technologies and tools within the field of MDE, such as the Eclipse Modeling Framework (EMF). These tools, while powerful, are now being pushed to their limits due to how they interact with and manipulate models. In simple terms, as projects get larger and more complex, it is becoming harder for existing tools to handle the size of the models efficiently. This challenge calls for finding new ways to work with these large models while making the best use of available resources.

Imagine a program designed to manage models, which involves tasks like loading and processing a model. When it comes to loading a model stored as a file, the program faces a specific challenge.

For file-based models (the models stored in the file(s)), since the model management

program does not know beforehand which parts of the model it will need, it must load the entire model into memory. This means all the model elements will occupy memory space during the whole execution, even when some of them might not be needed at all. This situation leads to unnecessary memory consumption, which is inefficient and wasteful. Moreover, keeping necessary information in memory even after the program no longer requires it makes this memory waste even worse.

Alternatively, suppose the model is stored in a database. In that case, the execution engine of a model management program can retrieve specific parts of the model progressively by issuing queries to the database as needed during execution. There are two approaches to consider in this situation: greedily fetching all features in advance or retrieving them lazily upon demand. However, both of these strategies have their drawbacks.

The first approach involves fetching all attributes and references of a model element (its properties) from the database upfront. While this prioritises faster execution times, it often leads to inefficient memory usage because many of these fetched properties may never be used by the execution engine, resulting in unnecessary memory wastage.

On the other hand, the lazy approach involves fetching properties on-demand, which can save memory since only the required properties are retrieved. However, this strategy often requires multiple roundtrips to the database, as each model element is fetched when needed. While this conserves memory, the frequent roundtrips can negatively impact performance, leading to slower execution times.

This thesis introduces an approach that aims to enable execution engines of model management programs to load only parts of models that are required for their execution and minimise the overhead of loading unnecessary parts in memory. In this work, by using in-advance knowledge about the model management programs provided by static analysis, execution engines are able to identify, load and process model elements of interest only. In addition, instead of keeping elements in memory until the end of execution, the elements will be disposed of from memory when they are no longer needed by the model management program.

Therefore, this thesis proposes three main components for the Epsilon framework (Introduced in Chapter 2), which improve the performance of execution engines. First, we contribute to enhancing a static analyser to enhance our understanding of the program under study. We introduce type inference and type-checking features to Epsilon languages, making the static analyser a fundamental component upon which the subsequent contributions are built. Also, this thesis centres on optimising memory usage by addressing the overhead of loading unnecessary data. While partial loading of file-based models has been explored [51], we direct our efforts toward the partial loading

of large models stored in databases. Our innovative approach relies on insights from the static analysis of model management programs, resulting in significant reductions in both time and memory requirements.

Finally, the final part of our work focuses on enhancing the efficiency of model validation for models serialised in the XMI format (introduced in Chapter 2). We introduce a novel approach that uses static analysis to divide model validation constraints into manageable sub-groups. Combined with existing XMI partial loading capabilities, our approach enables the efficient validation of larger XMI-based models on a single machine, with the potential for further improvements.

1.1 Research Contributions

The main contributions of this research revolve around optimisation and scalability. These have been achieved through two key mechanisms: first, by selectively loading essential information into memory and offering *partial loading* functionality, thereby optimising memory usage within the execution engine. Second, by introducing *partitioning*, the execution engine efficiently manages memory during runtime, retaining information while it is actively referenced by the program and releasing it when it is no longer required. While this section provides a high-level abstract overview of the thesis contributions, in-depth information, including the specific technologies utilised (such as targeted frameworks, model formats, and database types), is extensively discussed in the Background chapter (Chapter 2).

In the initial part of this thesis, our primary objective is to enhance the capabilities of the static analyser. To achieve this, we incorporate features like type inference and type checking, which allow us to gain a deeper understanding of the program's behaviour before execution. This not only helps us identify and report compile-time errors to support developers but also lays the foundation for two essential functionalities: *partial loading* and *partitioning* Furthermore, we introduce an algorithm that provides a strategy for applying the static analyser in our research. We demonstrate how to effectively use the valuable information obtained from the static analyser in a format that can be interpreted by the execution engine.

Since partial loading sets the stage for partitioning, we begin with an exploration of partial loading in Chapter 5, followed by a detailed discussion of partitioning in Chapter 6.

Utilising the insights obtained from the static analyser's output, we devised a novel method for selectively loading models from the graph-based database (Neo4j) as one of the contributions of this thesis. Our approach includes the proposal of an algorithm for efficiently storing models in a graph-based database, along with an algorithmic mapping

of model elements to graph nodes and relations. Additionally, we introduced an algorithm to automatically generate queries based on the static analyser's output, initiating the loading process by generating a number of queries. This approach represents a balanced compromise between the traditional lazy loading and resource-intensive greedy loading strategies. Our experimental results show that this approach, which considers the program's analysis, can significantly reduce the time and memory resources required to execute model management programs when dealing with models stored in a database.

In our final contribution, we centred our attention on the partitioning feature. We introduced an algorithm for partitioning model validation programs and loading models selectively based on the data necessary for executing specific parts of the program. To validate the effectiveness of this approach, we implemented a prototype, enabling us to gather results through a comparative analysis with other similar methods.

1.2 Thesis Structure

This thesis is structured as follows:

- **Chapter 2**: This chapter lays the foundations for our research by providing essential background information on the concepts underpinning our study. It also includes a comprehensive review of relevant works in the field.
- **Chapter 3**: This chapter provides an analysis of the problem and outlines the research framework, including the hypothesis, objectives, and scope.
- **Chapter 4**: In this chapter, we delve into the world of static model management analysis, using a practical example to illustrate its principles. This chapter also discusses the enhancements made to the static analyser as a component of this research.
- **Chapter 5**: This chapter represents the first contribution of this thesis. It explains the process of storing a model within a graph database and the retrieval of necessary information from the database. This approach optimises memory usage, contributing significantly to the efficiency of model management.
- **Chapter 6**: This chapter presents the second contribution of this thesis. The key strategy is to organise validation programs into groups of constraints and loading models accordingly. By discarding unneeded information, this method ensures the effective execution of various sections of validation programs.

- **Chapter 7**: This chapter provides an overview of thesis contributions and suggests future research directions.

2 Background

This chapter presents the foundation for this research, including fundamental concepts that are necessary to comprehend the thesis and a review of relevant literature. Section 2.1 introduces key concepts of Model-Driven Engineering, including the notions of *model*, *metamodel*, and abstraction layers. Section 2.2 explores the Eclipse Modeling Framework. Section 2.3 introduces the model management tasks, and Section 2.4 provides an overview of the Epsilon framework, highlighting two languages within this framework (*EOL* and *EVL*). Later in this chapter, Section 2.5 discusses various model persistence formats, while Section 2.6 provides an overview of scalability challenges in MDE and discusses related works focused on achieving scalable MDE solutions. Finally, Section 2.7 summarizes the chapter.

2.1 Model-driven Engineering

Models play an essential role in the software development process. In software projects, gaining a clear understanding of real-world problems is crucial for devising more effective solutions. Consequently, the information models that illustrate the different information elements in a specific business scenario and their interconnections were created [15]. However, this process gives rise to some challenges.

Developers in the Great Corporate Data Modeling Fiasco [15] of the 1980s and 1990s attempted to capture all the information used in an organization, but modelling the whole of complex systems that contain all of the information and the relation between them, made the modelling activity very complicated [15]. In addition, keeping the models up to date was impossible because of the large number of models and their sizes. Hence, the solution that they considered for this problem was modelling the part of the software that is more important and ignoring the details. So, the definition of models can be considered as a *model* is “an overview of fact which is shared between members of teams to reach the same understanding from a concept and make the software development process easier” [34]. In fact, models never show the entire reality.

Models are a part of the documentation and design phases in traditional software engineering. However, they are considered the central artefacts in Model-Driven Software Engineering (MDE). MDE is a methodology that elevates models to first-class artefacts of the software development process to enhance productivity [28, 26], maintainability, consistency, and traceability [39]. The models are used for generating code in a (semi)automatic way by programs, which are called model transformation programs. Model transformation programs are known as the heart and soul of MDE [43].

Whereas models are defined as a level of the abstract of a phenomenon in the

real world, a metamodel is at a higher abstraction level, representing the modelling language [8]. It specifies the elements of models and the way they can relate to each other. In other words, models are instances of their metamodels, and every model should conform to its metamodel.

Figure 2.1 shows the hierarchy of model abstraction. A real world object such as a book is shown in the M0 layer. The object is represented by a model as it is shown at M1. In this example, a book is described by its title attribute. This model conforms to a metamodel in M2 which outlines the concepts used at M1 (Class, Attribute and Instance). Finally, at level M3, the meta-metamodel is shown, which establishes the concepts utilized at M2. In this case, the entire set simplifies into a single *Class* concept, as illustrated in the example. M3 represents the highest level required for metamodeling in this context, as it would always include only the Class concept.

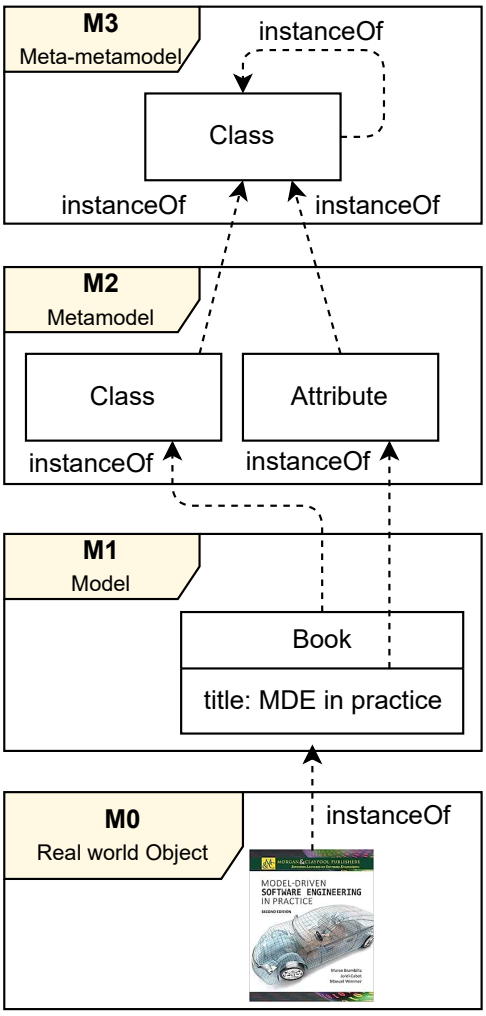


Figure 2.1: Abstraction layers (Inspired from [8])

2.2 Eclipse Modeling Framework

There are some tools which are used in MDE for different purposes. MagicDraw [36] is a visual modelling tool for business analysis, software analysis, and programming. IBM Rhapsody [25] is a group of products for modelling and design activities which enables the user to manage the complexity of system development, and MATLAB Simulink [44] is another tool that is a MATLAB-based graphical programming environment for modelling, simulating and analysing dynamical systems.

Since all these tools operate with models, it is essential to establish a common structure for these artefacts to ensure smooth integration. Additionally, having standardized facilities and Application Programming Interfaces (APIs) for interacting with models becomes crucial for enhancing efficiency and reducing complexities in the development process. A modelling framework provides precisely these functionalities, serving as a unifying platform for various tools to work with models at a higher level of abstraction.

The Eclipse Modeling Framework (EMF) [21] is an example of such a modelling framework. It is widely adopted in the field of Model-Driven Engineering (MDE) and plays a vital role in the development of various software tools and applications. This open-source ecosystem is purpose-built for domain-specific language engineering, editor development, model validation, transformation, and beyond. Using the framework's efficient reflective API, developers can interact with EMF objects and generically manipulate them.

The core of the Eclipse Modeling Framework is a modelling framework and code generation facility for building tools and other applications based on a structured data model. Key to this framework is Ecore, a metamodeling language that enables precise definition and specification of domain-specific models.

EMF also includes persistence support with default XMI serialization. XMI (XML Metadata Interchange) is an Object Management Group standard for model persistence. XMI is a common format for importing or exporting models in UML[1] [48] tools, as well as, it is a native format for model persistence in EMF. XMI is a text-based file format that employs XML to describe the structure, relations, attributes, and additional metadata of model elements.

Figure 2.2 shows an abstract view of Ecore's component hierarchy which is inspired by Java. The main components that we need to know in this work are listed below:

- **EObject** is the root element of ecore which is implemented by all model elements in EMF instances.

[1]The OMG's Unified Modeling Language help to specify, visualize, and document models of software systems, including their structure and design, in a way that meets all of these requirements

- **EClassifier** is a generic concept representing all types of model elements. The model element can be *EClass* or *EDataType*.
- **EClass** represents a class in the model and can have structural features (attributes and references). *EAttribute*s define the data properties of the class, while *EReference*s define the associations or relationships between different classes.
- **EDatatype** reperesents data type that can be the type of *EAttribute*, *EReference* or type of any other features and elements. Examples of EDataTypes include primitive types (e.g., int, boolean) and user-defined data types (e.g., enumerations).

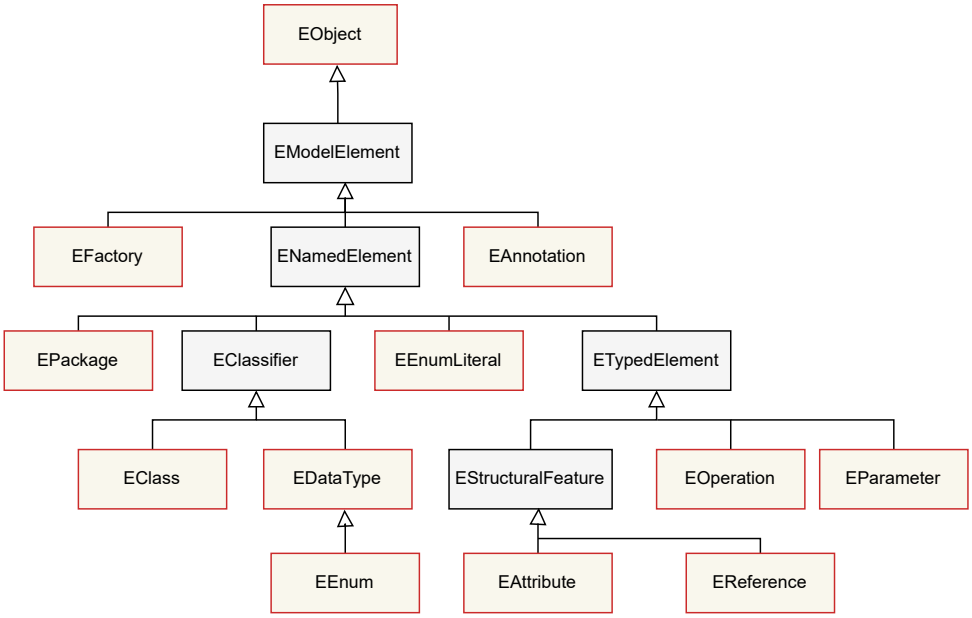


Figure 2.2: Ecore's components hierarchy [16]

2.3 Model Management

As discussed, models are the main artefacts in MDE which are used in the development process. *Model management* refers to a set of activities and techniques that involve creating, modifying, analyzing, and manipulating models to achieve specific goals during the software development process. Model management operations are defined at metamodel level of abstraction and are applied on models [42].

2.3.1 Model Transformation

Model transformation involves defining transformation rules or mappings that specify how elements in the source model should be transformed into elements in the target model.

The target model can be in the same abstraction level (model to model transformation) or a different abstraction level (model to code transformation). A model to model transformation program takes one or more models as input (source model) and produces one or more models as output (target model). It is useful for converting models from one modelling language to another.

Transformation can be performed automatically using transformation scripts that are written in transformation languages. There are three kinds of transformation languages.

- **Declarative Model Transformation Languages:** They are programming languages in which (ideally) a program specifies what is to be done rather than how to do it. The emphasis is on providing a high-level description of the problem domain, making the code more abstract and concise. QVT (Query/View/Transformation)[2] [38] is an example of declarative languages.

- **Imperative Model Transformation Languages:** This type of language programming is where the programmer explicitly specifies the sequence of operations or steps the computer must follow to achieve a particular outcome. In imperative languages, the focus is on describing the detailed algorithm and control flow the computer should execute to perform a task. Examples of imperative languages include ETL (Epsilon Transformation Language) [31] and Henshin [23].

- **Hybrid Model Transformation Languages:** Hybrid languages provide users with the advantage of higher-level constructs found in declarative languages while retaining the flexibility inherent in the imperative style. ATL (Atlas Transformation Language)[3] [27] is considered as a hybrid language.

2.3.2 Model Validation

In MDE, model validation is the process of checking and ensuring that models adhere to specific constraints, rules, or requirements. Model validation aims to identify potential inconsistencies or errors in the model, ensuring that it accurately represents the intended system or domain. Model validation also helps to maintain the models' quality and correctness, as they serve as the basis for generating code or other artefacts.

Metamodeling languages typically support only basic modelling constraints, such as cardinality constraints that define the possible relationships between different modelling elements. As a result, they may not capture all the complexities required in a complete modelling language definition. For example, an instance of a model, which conforms

[2]A family of model transformation languages standardized by the Object Management Group (OMG)

[3]A language provided by the Eclipse Modeling Framework (EMF) for defining model-to-model transformations

to a metamodel, cannot have its attribute values checked to ensure they fall within a specific range using the metamodel structure alone.

To overcome this limitation, the model validation languages are often used in conjunction with metamodels. Object Constraint Language (OCL) [41] is a validation language which provides a set of textual rules that models conforming to a specific metamodel must adhere to. These rules are referred to as well-formedness rules, as they define the set of valid, well-formed models that can be specified using that particular modelling language.

By using OCL, additional constraints and rules beyond what the metamodel can express directly can be imposed on the models, allowing for more precise and comprehensive specifications of the modelling language's requirements. Epsilon Validation Language (EVL) [33] is another validation language which is discussed in Section 2.4.2.

2.4 Epsilon

Epsilon [17] provides a collection of languages and tools designed to automate typical model-based software engineering tasks. As it is shown in Figure 2.3, at the core of Epsilon is the Epsilon Object Language (EOL) [30], a scripting language with the model querying capabilities of OCL (Object Constraint Language). Building upon EOL, Epsilon offers a range of task-specific languages that work seamlessly together (as shown in Figure 2.3, most languages extend EOL). These languages cater to various tasks, including code generation, model-to-model transformation, and model validation.

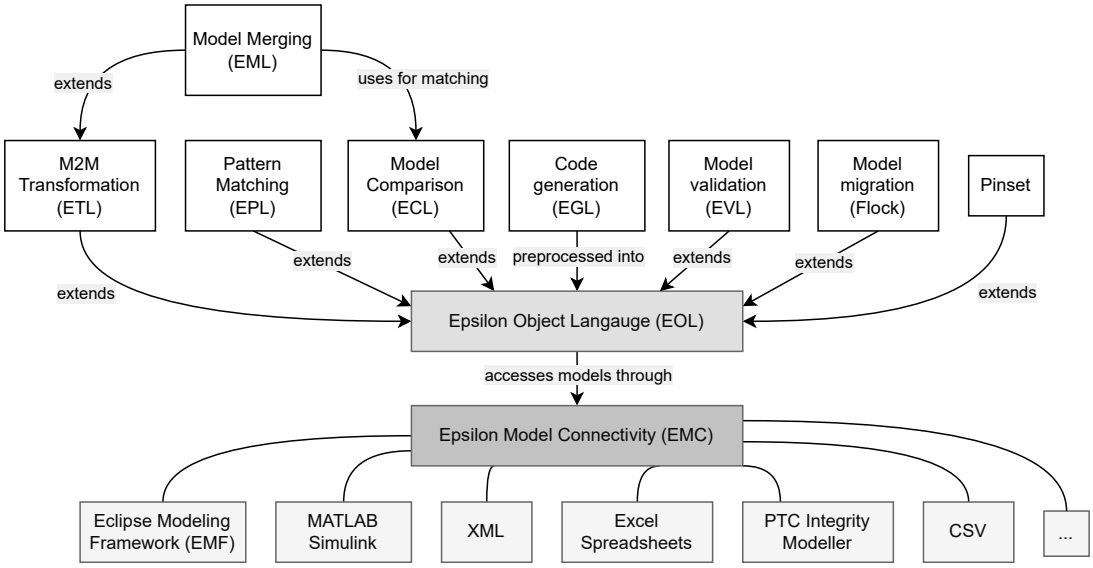


Figure 2.3: Epsilon Architecture [17]

Other general-purpose languages like Java, as well as task-specific languages such

as OCL or ATL, can be employed for performing tasks and manipulating models. However, what sets Epsilon apart in its architecture is the fact that all its languages extend a core language, providing the advantage of an extensible framework. Any feature implemented for the EOL language can be effortlessly extended to other languages with minimal effort, making the entire system highly adaptable and easily expandable.

Also, Epsilon offers support for different modelling technologies (see Figure 2.3), allowing interactions with models in a uniform manner. The Epsilon languages rely on a model connectivity layer that provides a protective shield, isolating them from the complexities of specific modelling technologies. This layer, regardless of the underlying modelling technology used (e.g., EMF, Simulink [44], XML, UML [48], etc.), provides a standardized approach to query, manipulate, and work with these models. This uniformity simplifies the development process and ensures that the same set of operations can be applied to models, regardless of their specific technology, enhancing the flexibility and ease of use of the Epsilon framework.

The implementation of EMC for the specific technology is called *driver*. For instance, EMF models are used by Epsilon languages using the EMF driver.

The execution engine within the Epsilon platform is responsible for running programs written in Epsilon languages. The execution engine first parses the input program written in an Epsilon language (e.g., EOL program). During this step, the engine analyses the code's syntax to understand its structure and validates it for correctness.

If the program involves working with models, the execution engine uses model *driver*s to load the required models from various sources (e.g., files, databases) into memory. Model drivers handle the loading and manipulation of models. Different loading strategies are discussed in section 2.5.

Once the program is parsed and the input models are loaded, the execution engine starts executing the program instructions step-by-step. It follows the control flow specified in the code, executing statements and performing operations (e.g., querying, modifying, transforming, or validating) as required. The execution engine interacts with the loaded models and carries out these tasks based on the program's logic.

As the program executes, it may generate output, such as transformed models, validation results, or code generation artefacts. Once the program execution is complete, the engine releases any resources and memory associated with the execution.

2.4.1 Epsilon Object Language

As mentioned above, EOL forms the core expression language within Epsilon, serving as the building block for task-specific languages designed for model validation, model-to-text transformation, model-to-model transformation, and model migration tasks.

Additionally, EOL can be employed as a standalone, multipurpose model management language capable of automating various tasks that may not fit the predefined patterns of the task-specific languages in Epsilon.

Listing 2.1 shows an example of querying the model in Figure 2.1. In line 1, all instances of *Book* are retrieved from the model. In lines 2 - 3, the *title* of each *Book* is printed.

Listing 2.1: EOL program to print title of book

```
1 var books = Book.allOfKind();
2 for (b in books) {
3     ("Title:" + b.title).println();
4 }
```

2.4.2 Epsilon Validation Language

Using a validation language like Epsilon Validation Language (EVL) [33], model developers can describe the additional constraints and execute the validation program on the model. This process helps to ensure that the model adheres to the specified constraints and is validated accordingly.

An EVL program consists of a series of *constraint*s, each associated with a *context*. This context plays a crucial role in specifying the type of instances on which the constraints will be evaluated. Each *context* can include an optional *guard*, allowing developers to narrow down the scope of *constraint*s to a specific subset of instances within its specified type. If the *guard* condition fails for a particular instance, the *constraint*s within that *context* would not be evaluated for that instance, optimizing the validation process.

EVL constraints include a body (*check*) to enforce rules and requirements. Additionally, developers have the freedom to define a message for each constraint using an *ExecutableBlock*, providing informative explanations when a constraint fails on a particular element.

Listing 2.2 shows an example of an EVL program.

- In line 1, the keyword *context* is used to define the *context* of the *constraint*, which is *Book* in this case.
- Line 2 introduces the name of the *constraint*, which is *ValidTitle*, defined using the *constraint* keyword.
- Line 3 contains the *check* keyword, where the *constraint* logic is specified. It checks that the *title* attribute of all instances of *Book* starts with an upper-case

letter. The *self* keyword refers to the *context* of the constraint, which is *Book* in this case, and *self.title* retrieves the *title* attribute of the *Book* instances.

- Line 4 provides a ”message” to inform the user about the reason for the constraint failure, along with printing the *title* attribute to help the user identify and fix the problem.

Listing 2.2: EVL program to validate title of book

```
1 context Book {
2     constraint ValidTitle {
3         check: self.title = self.title.firstToUpperCase()
4         message: self.title + " should start with an upper-case letter"
5     }
6 }
```

2.5 Model Persistence

Models need to be shared and maintained in a convenient way which makes it easy to collaborate between developers. Model persistence refers to the saving process of a model or how a model is stored. In the context of MDE, model persistence involves preserving models in a storage medium, such as databases, files, or other data repositories, so that they can be accessed, retrieved, and reused by different tools and applications. There are two kinds of model persistence: file-based models and repository-based models.

2.5.1 File-based Model

File-based models are defined as the models that are stored in a file, such as XMI models. The XMI models are stored with *.xmi* extension and use XML tags to present the data. Figure 2.4 shows the XMI representation of the model in Figure 2.1.

In line 1, the XMI file starts with an XML declaration that specifies the XML version and encoding being used. The root element of the XMI file is usually named *XMI* and serves as the container for the entire model representation. The XMI header contains metadata and information about the XMI version, the modelling tool used, and other optional information about the model. This model contains an instance of *Book* class and its *title* attribute. In Figure 2.4, there is a tag in line 2 which represents an instance of *Book* and the *title* attribute is represented as a property inside the *Book* tag. In XMI, each model element is assigned a unique identifier, often referred to as a *xmi:id*. The

xmi:id distinguishes one element from another within the XMI file. This allows different parts of the model to reference each other without ambiguity.

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:book="http://book">
  <Book xmi:id="_cNz_gp-uEeymwOmHBpNTLg" title="MDE in practice"/>
</xmi:XMI>
```

Figure 2.4: XMI representation of model in Figure 2.1

Java SAX Parser

As it is mentioned, XMI is a default persistence in EMF. In EMF, when an XMI file needs to be read and loaded into memory, it is processed by a default parser called Java SAX Parser. The Java SAX (Simple API for XML) parser[4] is used for parsing XMI files, which is an event-based XML parser. This parser navigates through the XML file or stream and triggers callback methods on a listener/handler object. These callback methods are invoked when specific structural elements of the XML are encountered.

For instance, when the parser comes across the start of an XML document, it calls the handler's *startDocument()* method. Similarly, when it encounters the start of an XML element, it invokes the *startElement()* method, and so on.

The responsibility of the handler is to extract the relevant information from these XML elements, achieved by extending the appropriate callback methods. For example, in the context of EMF's SAX parser, the handler would create EObjects (representing model elements) based on the extracted information.

The SAXXMIHandler is a crucial component that manages XML events and performs the creation of model elements (EObjects). It populates the attributes and references (EStructuralFeatures) of these EObjects and subsequently stores them in the *XMI Resource*, effectively completing the process of parsing and loading the XMI-based model.

Let's consider the model in Figure 2.5 that conforms to the *University* metamodel (Figure 2.6). We use Figure 2.7, which illustrates the XMI presentation of the University model, to describe how EMF's XMI parser works [51].

The EMF's XMI parser (and in particular its SAXXMIHandler component) maintains a stack of model elements (EObjects in EMF's terminology) to keep track of its current position in the XMI document. This is needed in order to determine what *EObjects* to create next, as illustrated in the right part of Figure 2.7. When line 1 of the XMI

[4]https://docs.oracle.com/javase/tutorial/jaxp/sax/

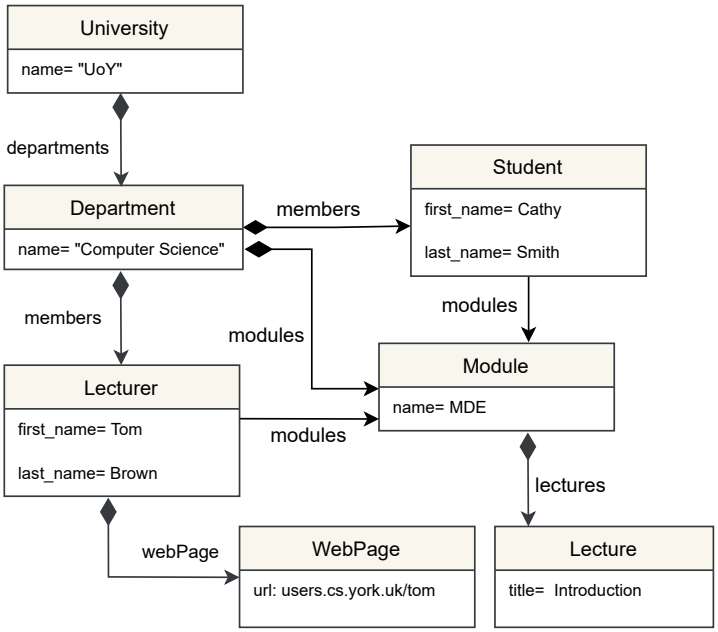


Figure 2.5: A model instance of University metamodel from [51]

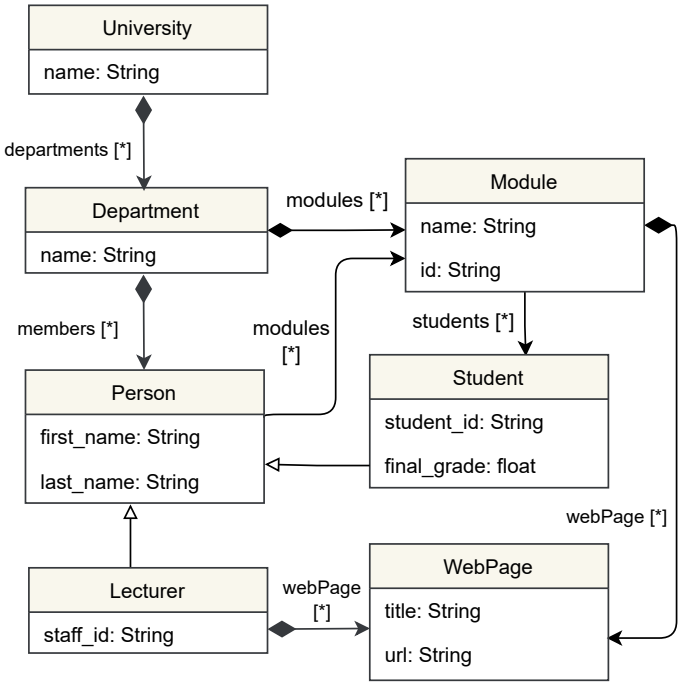


Figure 2.6: University metamodel from [51]

file is read, the callback method *startElement()* is triggered, the <university> element is handled and a new instance of *University* (with its *name* attribute set to UoY) is created. The new *EObject* is pushed into the object stack. When line 4 is read, the parser processes the top of the stack (peekObject in EMF's terminology) together with the <departments> element and decides that an instance of *Department* should be

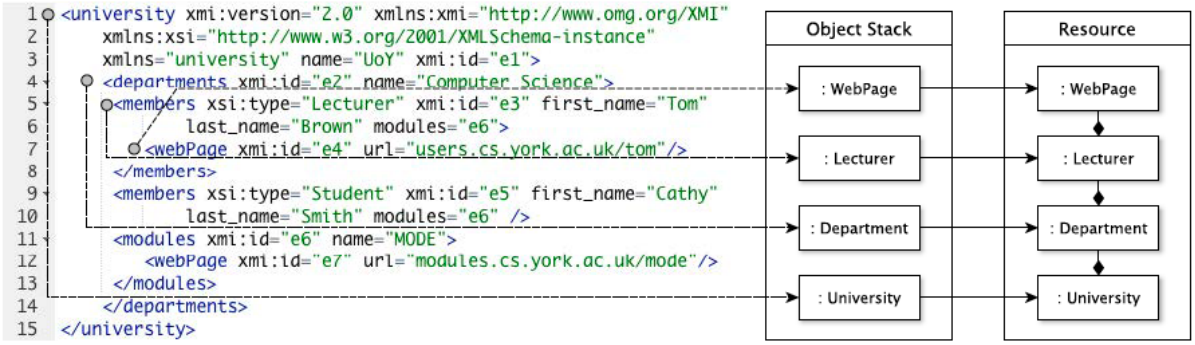


Figure 2.7: EMF's SAX parser loading [51]

created and added to the *departments* reference of the *University* model element.

The created instance of *Department* is also pushed into the object stack. The same principle is applied when line 5 is read, the element <members> is handled, and an instance of *Staff* is created, added to the *members* reference of the *Department* and pushed into the stack. When an element tag ends (e.g. in line 8), the top element of the object stack is popped. Once all XML elements have been processed, a tree structure has been constructed in memory.

XMI Partial Parser

EMF's SAX parser is responsible for loading the model into memory, making it available for model management programs. it reads all the information upfront before executing the program, which can lead to inefficiencies in terms of memory usage. To optimize memory utilization, a more efficient method called partial loading is employed. Partial parser of XMI models was introduced by Wei et al. [51]. Using the partial parser, instead of loading the entire model into memory, only the essential parts of the model required for executing the program are loaded.

As explained earlier, EMF's built-in XMI parser maintains a stack of non-null EObjects to identify the appropriate type of EObject to create when encountering a new XML element and to determine its position in the containment hierarchy.

Wei et al. [51], identified that creating all EObjects solely to maintain a null-free stack is not efficient in terms of time and memory consumption. To overcome this issue, they introduced a placeholder cache that contains empty/placeholder EObjects for types that are unnecessary for program execution. They introduced the SmartSAX parser which supports partial loading of XMI models. When the SmartSAX parser encounters an XML element that can be skipped, it retrieves the corresponding placeholder EObject from the cache and places it in the object stack. Importantly, this process does not involve

processing the actual XML element, resulting in reduced loading time and improved memory consumption.

By adopting partial loading and utilizing the placeholder cache, EMF's SAX parser can optimize memory usage and efficiently handle XMI models without loading unnecessary data into memory.

Listing 2.3: EOL program to lecturer information

```
1 var lecturers = Lecturer.allOfKind();
2 for (l in lecturers) {
3     ("First name:" + l.first_name).println();
4     ("Last name:" + l.last_name).println();
5     "Web page:" + l.webPage.url).println();
6     ("Number of modules taught:" + l.modules.size()).println();
7 }
```

For instance, consider Listing 2.3 that goes through all instances of *Lecturer*s and prints their first name, last name, url of their webpage and number of modules that each lecturer has taught. This program needs all *Lecturer* instances, and the features that are needed from *Lecturer* class are *first_name*, *last_name*, the *webPage* reference with the *url* attribute and the number of modules that each lecturer has taught. Hence, there is no requirement for loading other parts of the model such as *University* and *Student* instances.

Figure 2.8 illustrates a snapshot of the state of the partial parser at the point where it has parsed the University XMI model up to line 7, with reference to the required information for running the program.

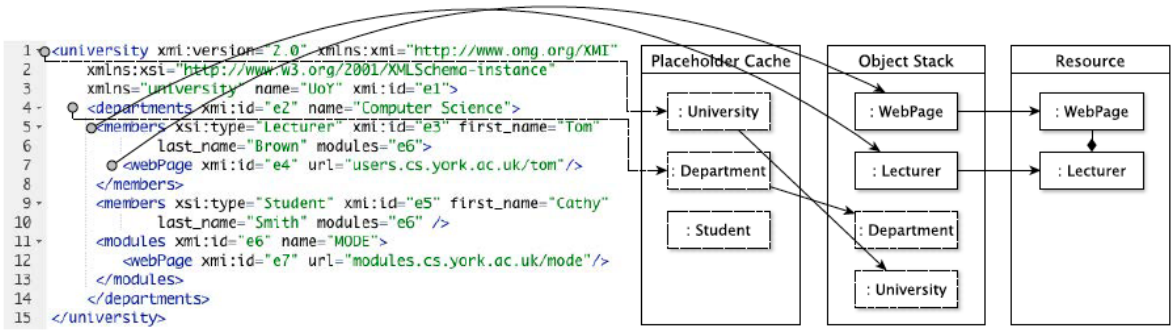


Figure 2.8: SmartSax parser loading

When parsing starts, the SmartSAX parser populates the Placeholder Cache with one placeholder EObject for each type of the full metamodel that is not included in the program's requirement as illustrated on the bottom-left corner of Figure 2.8. Having populated the Placeholder Cache, the parser then handles the XML elements it

encounters as follows:

- When the <university> element in line 1 is encountered, the parser checks the required model elements, determines that instances of *University* do not need to be loaded, and therefore fetches the placeholder instance of *University* from the placeholder cache and pushes it to the object stack.
- When the <departments> element in line 4 is encountered, the parser determines that the type of the object to be instantiated is *Department*. However, according to the program, instances of Department do not need to be loaded either so the parser fetches the placeholder instance of *Department* from the placeholder cache and pushes it to the object stack.
- When the parser encounters the <members> element in line 5 it determines that it needs to create a new instance of *Lecturer* (as Lecturer is part of the necessary information for running the program). After creating the new instance, it populates the values of its *first_name* and *last_name* attributes. Then it looks at the element on the top of the stack (currently the placeholder instance of *Department*), detects that it is a placeholder, and as such adds the populated instance of *Lecturer* to the resource as a top-level element.
- When the <webPage> element in line 7 is encountered, the parser determines that it needs to create an instance of *WebPage* and place it in the webpage containment reference of the top element of the stack. It also populates the *url* attribute of the new instance with the value of the respective attribute of the XML element.
- When </webPage>is encountered in line 7, the top object of the stack is popped (the current top element is now Tom)
- When </members> is encountered in line 8, the top object of the stack is popped (the current top element is now Computer Science)
- When the <members> element is encountered in line 9, the parser determines that it needs to create an instance of *Student*. Since the Student type is not part of the required model elements, it fetches the *Student* placeholder object and puts it at the top of the stack.
- When </members> is encountered in line 10, the top object of the stack is popped (the current top element is again Computer Science)

- When the <modules> element is encountered in line 11, the parser determines that it needs to create an instance of Module and since it is declared that all instances of Module need to be loaded, it creates a fresh object (but does not populate any of its attributes/references as none of these need to be loaded according to the program's requirement). Since the top element in the stack is a placeholder, it adds the new Module instance to the resource as a top-level element and also pushes it to the stack.
- When the <webPage> element is encountered in line 12, the parser determines that it maps to an instance of *WebPage* that should be placed under the webpage containment reference of the top element of the stack (which is currently the MODE module). Since the *WebPage* type of *Lecturer* is required, and its containment reference (Student.webPage) is not of interest, the parser fetches the placeholder WebPage object and pushes it to the stack.
- Each of the last three lines (13-15) cause the parser to pop the top element of its stack, thus ending up with an empty stack. Since all required objects have been loaded, the last step of the algorithm involves resolving non-containment references (in this case, link Tom to the MODE module, through its modules reference).

In [51], the authors conducted extensive benchmarking to demonstrate that the partial loading algorithm exhibits linear scalability concerning the size of the model part of interest. This scalability is evident in both loading time and memory usage. As a result, using the partial parser for loading XMI models is more efficient than using the default EMF parser when only a subset of the model is required for executing the program.

2.5.2 Repository-based Model

Using XMI-based serialization in EMF is highly inefficient due to two main factors. First, XMI files prioritize human readability over compactness, leading to reduced efficiency in input/output (I/O) accesses. Second, fully parsing XMI files is necessary to obtain a navigational model of their contents, resulting in increased memory usage when loading and querying models (as discussed above).

Additionally, XMI-based implementations lack essential advanced features, such as concurrent modifications and model versioning. These limitations can further impact the overall performance and collaboration capabilities of EMF projects.

To tackle the challenges posed by XMI-based serialization, a promising approach is to adopt databases for storing, accessing, and manipulating models. Models that are stored in a database such as NoSQL or Neo4j [40], are considered repository-based models. A repository is considered a persistent solution that is remotely accessible by tools and users. There are two main loading strategies for retrieving data (models in our work scope) from repositories:

Lazy Loading. Lazy loading is the practice of delaying the load or initialization of resources or objects until they're actually needed to improve performance and save system resources. For example, in Listing 2.3, using the lazy loading, in line 1, only skeletons of *Lecturer* elements would be initially fetched from the database. Then in line 3, for each *Lecturer*, the program would need to go back to the database and fetch the value of its *first_name* attribute. It is the same for any other feature that is accessed by the program. So, multiple round-trips to the repository to fetch the value of the attribute or references of each model element are required.

Greedy Loading. Greedy loading is another strategy used in the context of data retrieval and processing. In greedy loading, the system loads and retrieves all the data and resources at once, upfront, instead of loading them on an as-needed basis.

When an operation or query requires specific data, greedy loading aims to fetch not only the requested data but also all related or dependent data that might be needed in the future. This approach ensures that the system has all the required data readily available, potentially reducing the need for further queries or loading operations in the immediate future, but occupying memory with unnecessary information.

For instance, considering Listing 2.3, When all instances of *Lecturer* are fetched in line 1, all their attributes and references would be fetched too (including *Lecturer.webPage* or *Lecturer.staff_id*). As *Lecturer.staff_id* is not accessed by the EOL program, fetching its value from the repository and maintaining it in memory is wasteful.

Model Repositories

There are some mature repositories which are used in MDE for storing or managing models. The Eclipse Connected Data Objects (CDO) [10] project serves as a "distributed shared model framework tailored for EMF models and meta models". Acting as a model repository, CDO offers a comprehensive persistence and access solution capable of handling multiple models. Architecturally, CDO presents an API designed for EMF-based applications, a repository server, and the flexibility to support various persistence back-ends, although relational databases are commonly preferred in practice. As shown in Figure 2.9, in CDO, the models can be stored in all kinds of major databases back-ends like Object Database(ODB), NoSQL and Relational Database(RDB), and it can

transform models in all of the supported back-end types.

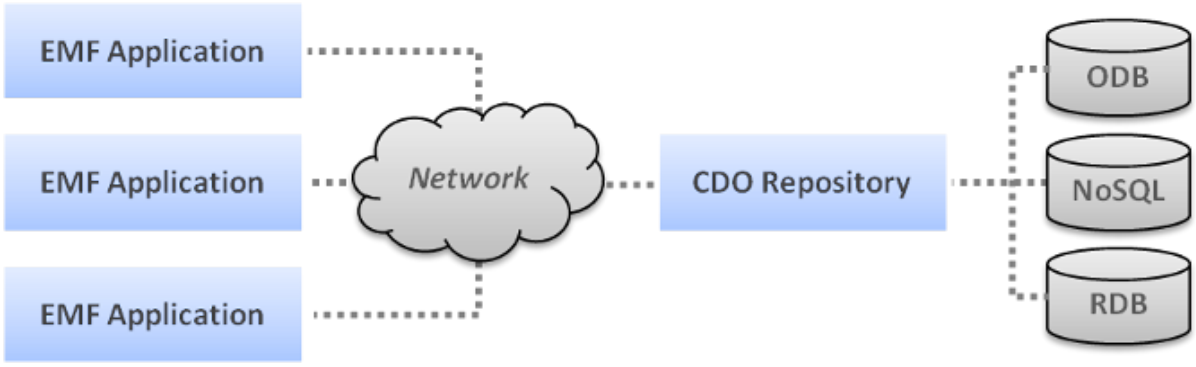


Figure 2.9: CDO Architecture

Another feature of CDO is scalability, which is achieved by loading object on-demand strategies and caching them in the application. Hence, it does not keep the objects which are no longer referenced by the application, and they are collected from the memory automatically. CDO supports lazy loading and greedy loading strategies for retrieving data from database back-ends. By leveraging CDO, users gain access to more advanced functionalities for interacting with data within models, including support for transactions and distributed concurrent interactions with multiple models.

In more recent years, NoSQL database technologies have become increasingly popular forms of persistence backends for large models. These include document databases like MongoDB [37] and, more prominently, graph databases (given that most models are graph-like in structure, and the close relation between the modelling and graph transformation research community) such as Neo4J [40]. Moreover, Barmpis and Kolovos [4] suggest that NoSQL databases would provide better scalability and performance than relational databases due to the interconnected nature of models. Morsa [19] and NeoEMF [14] are repositories that are supported by NoSQL database.

Morsa [19] is a persistent solution for storing and accessing large models based on on-demand strategies, which is supported by the NoSQL database. Figure 2.10 illustrates an overview of Morsa's architecture.

In Figure 2.10, the Morsa driver allows client applications to access models through the modelling framework persistence interface. Further, Morsa loads models using a load-on-demand mechanism, which has been designed to achieve scalability. This mechanism reduces database queries, and it is aimed at managing memory usage based on an object cache that holds loaded model objects. Cache policy is configured to manage the object cache that decides which object must be unloaded when the cache is overloaded.

Four cache policies are supported by Morsa such as First In-First Out (FIFO), Last

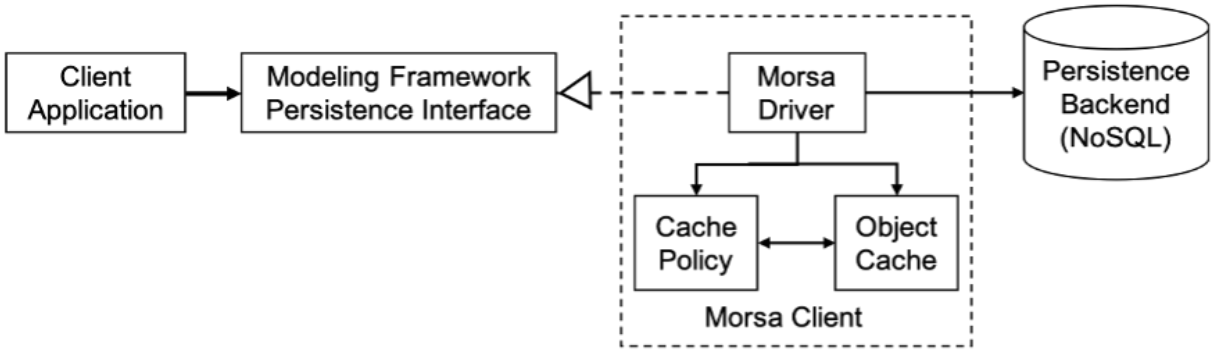


Figure 2.10: Morsa Architecture

In-First Out (LIFO), Less Recently Used (LRU), and Largest Partition First (LPF). The choice of cache policy is currently made by the end-user.

CDO and Morsa adopt a client-server architecture for model persistence, necessitating the integration of an additional API in the client code to ensure effective model access. This architectural approach involves the establishment of a server, handling new connections, and facilitating changes, introducing inherent complexities and overhead [14].

Furthermore, in these approaches, the selection of the datastore is decoupled from the intended model usage, disregarding specific requirements such as complex querying, interactive editing, or intricate model-to-model transformations. While the persistence layer may yield generic scalability improvements, it lacks tailored optimization to address distinct modelling scenarios adequately [14].

NeoEMF is a modelling framework centred on a multi-database architecture, where in each database is designed to deliver optimized performances tailored to the requirements of different modelling scenarios. This paradigm seeks to overcome the limitations of generic persistence solutions and unlock the potential for bespoke and efficient data storage strategies in diverse modelling contexts.

In 2014 [6], NeoEMF was initially developed on the foundation of Neo4j due to its capability to handle vast amounts of data in highly distributed environments. Subsequently, in 2017 [14], it was extended into a multi-database model persistence framework, offering support for models in key-value stores, wide-column databases, and graph databases.

To implement key-value stores, NeoEMF utilizes the Maps provided by MapDB, employing them as an efficient key-value storage solution. For wide column databases, NeoEMF builds upon Apache HBase [2], a distributed, non-relational wide column database that leverages Hadoop Distributed File System (HDFS) for data storage.

Regarding graph databases, NeoEMF relies on Blueprints [3], an interface designed

to create a common API for various graph databases. Many databases, including Neo4j, OrientDB, and Titan, have implemented Blueprints, making it an adaptable choice for NeoEMF's support of graph databases. NeoEMF relies on a lazy-loading capability allowing very large model navigation in a reduced amount of memory, by loading elements when they are accessed.

Neo4j. As mentioned earlier in this chapter, graph databases are extensively utilized for storing models in MDE due to the graph-like structure of most models. Consequently, models can be represented using graph structures, where elements in the model correspond to nodes and relationships between them are represented as edges. This simplifies the storage process within graph databases. By embracing graph representations, MDE can leverage the efficiency of navigating complex relationships, making it particularly advantageous for querying large models or dealing with complex structures. The effective handling of graph-based queries by graph databases further enhances the efficiency of common MDE tasks, such as querying specific elements, their properties, and their relationships, thanks to the utilization of powerful graph traversal algorithms. Overall, the manifold advantages offered by graph databases make them a compelling choice for storing models in the context of MDE.

Neo4j is an open-source NoSQL database. It is a popular and widely used graph database as it delivers features like Scalability and flexibility in data modelling and data structure. It also uses *Cypher* [12], a declarative graph query language, to interact with the database. Neo4j users use Cypher to construct expressive and efficient queries to do any kind of create, read, update, or delete (CRUD) on their graph, and Cypher is the primary interface for Neo4j.

If the Neo4j is used to store models, then accessing the model and retrieving model elements from the Neo4J database is possible by using Cypher. Some of Cypher's features that are helpful and are used in our implementation are listed below:

- Graph pattern matching: Cypher allows users to specify patterns in the graph, describing the nodes, relationships, and properties that they want to retrieve or manipulate. These patterns are intuitive and resemble a visual representation of the graph.

  Listing 2.4 shows an example of Cypher code which retrieves all nodes in a database. In Cypher, nodes are enclosed within parentheses, and they can be assigned a variable if there is a need to reference them later. In Listing 2.4, the variable *b* is used to access the retrieved node. However, it is also possible to omit the use of a variable and simply represent the node as *()* when there is no requirement to reference the node later on.

Listing 2.4: Matching pattern in Cypher

```
1 MATCH(b)
```

- Node and relationship filtering: Cypher supports filtering nodes and relationships based on their labels, properties, and relationship types, allowing users to narrow down their search criteria. In Listing 2.5, only nodes with the *Book* label are filtered, and they can be returned or accessed later by the *b* variable. As shown in line 2, the *title* property of *b* nodes is returned.

Listing 2.5: Matching pattern in Cypher

```
1 MATCH(b: Book)
2 RETURN b.title
```

Table 2.1 shows the result of the query in Listing 2.5.

b.title
“Book 1”
“Book 2”

Table 2.1: Query results of Listing 2.5

- Create, update, and delete operations: Cypher not only allows for data retrieval but also enables creating new nodes and relationships, updating existing data, and deleting elements from the graph. Listing 2.6 shows an example of creating two nodes and creating a relationship between them. In line 1, a node is created for representing a *Book*, titled “MDEInPractice” and in line 2, a node is created to represent an *Author* named “Marco Brambilla”. In line 3, variable *b* refers to the “MDEInPractice” node, variable *a* refers to the “Marco Brambilla” node, and *WRITTEN_BY* relationship is created between two nodes.

Listing 2.6: Create nodes and relationship in Cypher

```
1 CREATE (b:Book {title: 'MDEInPractice'})
2 CREATE (a:Author {name: 'Marco Brambilla'})
3 CREATE (b)-[:WRITTEN_BY]->(a)
```

- Path traversal: One of the powerful features of Cypher is its ability to traverse paths in the graph. It can find paths between nodes based on specific patterns and criteria. In Listing 2.7, Cypher is able to follow the path from each *Book* labelled node to *Author* node by *:WRITTEN_BY* relationship and return the title of all books in the graph with their author’s name.

Listing 2.7: Travesing path in Cypher

```
MATCH (b:Book)-[:WRITTEN_BY]->(a:Author)
RETURN b.title, a.name
```

- Optional matching: Cypher provides the ability to perform optional matches, allowing users to find patterns where certain elements may or may not exist. Using *MATCH* is a strict condition, and if there are no matches in the database, the query will fail to run and it does not return any result (it will be empty). With *OPTIONAL MATCH* on the other hand, if there is no match, the absence of the pattern does not affect the rest of the query, and it returns the results for the other parts of the pattern.

  For instance, in Listing 2.8, two nodes with *Book* label are created in the graph. *Book 1* is written by *Author 1* and *Book 2* is not associated with any author in the database.

Listing 2.8: Creating nodes in graph

```
CREATE (:Book{title:'Book 1'})-[:WRITTEN_BY]->(:Author{name:'Author 1'})
CREATE (:Book {title: 'Book 2'})
```

In Listing 2.9, if the query tries to match *:WRITTEN_BY* pattern using the *MATCH* keyword, the result of the query is as shown in Table 2.2. Since there is no match for *book 2*, it does not appear in the result. But if the query is written by *OPTIONAL MATCH* (as it is in Listing 2.10), the result includes *book 2* and the author is returned as *NULL* (see Table 2.3).

Listing 2.9: Match query in Cypher

```
MATCH (book:Book)-[:WRITTEN_BY]->(author:Author)
RETURN book.title, author.name
```

b.title	a.name
"Book 1"	"Author 1"

Table 2.2: Query results of Listing 2.9

Listing 2.10: Optional match query in Cypher

```
OPTIONAL MATCH (book:Book)-[:WRITTEN_BY]->(author:Author)
RETURN book.title, author.name
```

Table 2.4 presents a comparison of the tools reviewed in this study, focusing on their loading strategies and memory management. In this table, a tick in the respective cell indicates support for the feature, while a cross indicates a lack of support.

b.title	a.name
“Book 1”	“Author 1”
“Book 2”	NULL

Table 2.3: Query results of Listing 2.10

In the first row, the default parser for XMI files used in EMF is shown. This parser is capable of handling XMI files, as indicated by the tick in the *File-based* column. However, it lacks the capability to analyze which parts of the model are essential for the program, resulting in a cross in the *program analysis* column. Additionally, this parser uses a strategy of loading the entire model into memory upon execution and keeping it until completion, thereby lacking support for partial loading and memory disposal.

The second row is a Partial parser of XMI models (tick in the file-based column). This parser supports loading the model based on the demand that should be defined by the user, therefore it supports partial loading but does not perform any analysis on the program. Also, it keeps all information in the memory and does not supports the memory disposal feature.

CDO supports models stored in the database and have no support for file-based models. It supports partial loading by using a lazy loading strategy, and it does not perform any program analysis in advance. CDO supports memory disposal using a Java garbage collector but has no analysis or partitioning approach to ensure that data is disposed of from memory.

Morsa is the same as CDO in supporting repository-based models and partial loading, but the memory disposal is based on the cache policy. So it provided some option for the user to specify the cache policy and dispose the memory based on that.

NeoEMF operates similarly to Morsa and CDO. It facilitates repository-based models and partial loading through a lazy loading strategy, while lacking program analysis capabilities. Memory disposal relies on a garbage collector. Although Daniel and colleagues have proposed a method for manual memory management [13], there is currently no automatic memory disposal algorithm or strategy based on program analysis.

Therefore, our research aims to develop a solution that offers both partial loading and automatic memory disposal based on program analysis, a feature lacking in existing approaches.

Table 2.4: Related work comparison

	File-based	Repo-based	program analysis	Partial loading	Memory disposal
Java Sax parser	✓	×	×	×	×
Partial parser	✓	×	×	✓	×
CDO	×	✓	×	✓	✓
Morsa	×	✓	×	✓	✓
NeoEMF	×	✓	×	✓	✓

2.6 Scalability Challenges in Model Driven Engineering

Recent research has demonstrated that MDE has the ability to boost productivity [4, 5] and greatly improve crucial elements of the software development process, such as maintainability, consistency, and traceability [28, 39, 26]. Consequently, numerous industrial projects are now adopting MDE techniques and employing models that reduce accidental complexity while closely aligning with the domain's concepts [24, 54, 28].

With the growing adoption of MDE in larger and more intricate systems, the present generation of modelling and model management technologies faces considerable challenges in accommodating collaborative development and efficient management of models exceeding a few hundred of megabytes in size [32].

To address this issue, it is important to continue research efforts focused on achieving scalability within the MDE technical landscape [32]. This pursuit is crucial to ensure that MDE remains pertinent and capable of providing its well-established advantages in terms of productivity, quality, and maintainability, which are widely acknowledged in the field.

Scalability in the field of MDE relates to the ability of MDE approaches, techniques, and tools to effectively handle the growing complexity and size of models and model-driven systems as they evolve and expand.

In order to maintain efficiency and effectively manage the development process, ensuring scalability becomes vital as projects and systems built using MDE increase in size and intricacy. To achieve scalability in MDE, the following aspects are involved [32]:

- Constructing Large Models and Domain-Specific Languages: As software systems become larger and more intricate, the corresponding models in MDE also grow in size and complexity. To manage and manipulate these substantial models efficiently, sophisticated infrastructure and optimized algorithms are necessary to maintain acceptable performance. Scalable MDE tools should be capable of handling complex models without compromising performance.

- Enabling Collaborative Modeling in Large Teams: In collaborative software development environments, multiple developers may simultaneously work on different parts of the model or even on the same elements. This demands efficient mechanisms for version control, conflict resolution, and concurrent editing, which become more challenging as the scale of the project and the number of contributors increases. Scalable MDE tools should support efficient model differencing, versioning, and merging to manage these changes effectively.

- Advancing Model Querying and Transformations: Dealing with large models (of the order of millions of model elements) requires model querying and transformation tools to be at the cutting edge of technology. Transforming and validating complex models may require significant computational resources, especially when dealing with large-scale systems. Ensuring the correctness and consistency of models becomes more computationally intensive as the project size grows. MDE tools need to be optimized for processing large models swiftly and effectively. Scalability becomes crucial to avoid bottlenecks and ensure timely execution.

- Efficient Storage, Indexing, and Retrieval of Large Models: Storing and persisting extensive models that span multiple megabytes or even gigabytes in size requires scalable storage solutions. Traditional storage systems might not be well-suited for handling such large artefacts efficiently. Also, scalable MDE systems should be designed to use system resources efficiently, minimizing memory consumption and optimizing processing to avoid performance bottlenecks.

As the focus of this work is on the aspect of memory management, we have explored various related work in optimization techniques to improve the scalability of MDE systems. These optimization efforts primarily target the efficient utilization of system resources, such as memory and processing power, to avoid performance bottlenecks and enhance overall scalability.

- **Memory Efficiency:** One aspect of resource optimization involves minimizing memory consumption in MDE tools. Large models can require substantial memory to process and store, which can become a limiting factor in scalability. Researchers have explored techniques and algorithms that reduce the memory footprint of models, allowing MDE systems to handle larger models without exhausting memory resources. As discussed earlier in this chapter, repositories use greedy loading and lazy loading techniques to make the loading process more efficient. Also, there is some research regarding cache mechanisms to improve memory management. Caching involves storing retrieved data that are likely to be used in the near future

or keeping previously computed results to avoid redundant calculations. In [13], Daniel et al. proposed PrefetchML, a domain-specific language that describes prefetching and caching rules over models. PrefetchML is a suitable solution to improve query execution time on top of scalable model persistence frameworks. The rules to describe the event conditions to activate prefetching, the objects to prefetch, and the customisation of the cache policy are defined by designers in PrefetchML.

- **Parallel Processing:** To achieve scalability, MDE tools have explored the use of parallel processing techniques. These techniques involve leveraging multi-core processors and distributed computing environments to efficiently handle large-scale modelling tasks. By dividing the workload among multiple processing units, parallel processing can enhance performance and accelerate model processing.

  In [49], an approach is proposed for parallelization of Eclipse OCL (Object Constraint Language) [41] constraints utilizing Communicating Sequential Processes (CSP). In this work, expressions are executed in parallel, and their results are combined through binary operations. The authors demonstrated the equivalence of behaviour between the parallel and sequential representations of OCL CSP to prove the correctness of their approach. Their implementation utilized CSP as an intermediate representation, which was then transformed into C# code. However, users were required to manually specify the expressions to be parallelized.

  In [35], Madani et al. demonstrated how executing an EVL program concurrently and in a distributed setting can result in a proportional decrease in execution time with more machines and larger models. Their methodology involves breaking down each EVL program into an ordered sequence of rule-element pairs. The EVL distributed execution engine replicates the program execution environment across multiple computers and assigns a subset of this sequence to each computer for independent execution. The main limitation of this approach is that it requires all workers to have a full copy of both the models and the program.

- **Incremental Modeling and Validation:** Scalable MDE approaches often adopt incremental techniques to address the challenge of handling large models efficiently. Instead of reprocessing the entire model when changes are made, incremental techniques only process the parts of the model that are affected by the modifications. This targeted approach significantly improves performance during model editing and validation.

  Cabot and Teniente [9] developed an algorithm for incremental model validation that guarantees the generation of the most efficient expression to validate a

specific constraint when the model undergoes changes (such as Create/Read-/Update/Delete events). They approach the problem of model validation from a conceptual standpoint and demonstrate that their solution automatically generates the most optimal expression for incremental validation in response to a CRUD event. This ensures that the least amount of work is required to execute the constraint.

By exploring these scopes and incorporating the findings into MDE tools and techniques, researchers aim to enhance the scalability of MDE approaches. These efforts are vital to ensure that MDE can effectively handle the increasing complexity and size of models and model-driven systems encountered in modern software development projects.

2.7 Chapter Summary

This chapter covered the essential information needed to understand the work presented in this thesis. It began by introducing fundamental concepts in MDE, including modelling and metamodeling. The Eclipse Modeling Framework, a widely used framework in MDE, was also discussed. Model management tasks such as model transformation and model validation were presented, along with examples of languages used in these tasks. Additionally, the Epsilon framework and some of its languages (EOL and EVL), which formed the basis of the thesis implementation, were highlighted.

Since the thesis focused on model loading, different formats of model persistence were explored, and demonstrations were provided on how programs interacted with these models. The chapter concluded by discussing scalability challenges in MDE and presenting related works in this field of research.

3 Analysis and Hypothesis

This chapter provides an overview of the limitations of state-of-the-art tools in MDE, which serves as the motivation behind the proposal of efficient management of large models and represents the primary contributions of this thesis. After an in-depth analysis of the literature and the identification of current challenges in Section 3.1, we proceed to present the research hypothesis, objectives, and the scope of our research in Section 3.2.

3.1 Analysis

In recent decades, several dedicated languages have been specifically designed to address fundamental model management activities, including tasks like model validation, transformation, and code generation. Unlike their general-purpose counterparts, such as Java, these specialized languages, like OCL [41], ATL [27], and ETL [31], have emerged as pivotal tools in the field.

These dedicated languages bring with them a wealth of advantages. They offer a more streamlined and purpose-tailored syntax, making them inherently better suited for the demands of model management. Furthermore, they provide unique opportunities for in-depth analysis and optimization, which are increasingly essential in the dynamic environment of software systems. As software systems evolve, they get more and more complex. This increased complexity extends to the models that represent these systems. Essentially, these models become larger and more complex, making them harder to handle.

This rising complexity in models presents a significant challenge for the current generation of tools used to manage them. These tools face considerable difficulties when dealing with exceptionally large and complex models [32]. This mismatch between the demands of complex software systems and the capabilities of existing tools emphasizes the need for innovative solutions to manage and manipulate these complex model structures effectively.

This issue with handling large models primarily arises from the way most contemporary MDE tools interact with these models. These tools have certain limitations when it comes to efficiently managing the growing size and complexity of the associated models. It is imperative to understand the specific challenges posed by these tools in the context of MDE and to explore innovative approaches that can address the scalability and efficiency issues faced in this domain.

Consider a scenario where a model management program is responsible for loading and processing a model. This encompasses various tasks, such as transformations and

validations. The process's intricacies depend on the type of model involved.

File-based models, like XMI files, using the SAX parser (the default EMF parser, which is discussed in Section 2.5), require the program to read all the information from the file before it can commence its operations. This upfront loading is a mandatory step. The drawback here is that the execution engine does not discriminate about which parts of the model the program will actually use. It loads the entire model into memory, including information that might never be utilized during the program's execution.

If the model is repository-based and stored in a database, the approach shifts; instead of reading everything at once, the program can send multiple queries to the database progressively, fetching only the specific model elements required during the program's execution. This *on-demand* retrieval approach can be more efficient than loading the entire model upfront. However, it is important to note that multiple round trips to the database can consume additional time.

Furthermore, it is crucial to take into account the memory footprint. Even if the program no longer needs certain model elements, keeping them loaded in memory can lead to inefficiencies and resource wastage. This can have adverse effects on system performance, underlining the significance of optimizing memory usage in model management.

As software and system models continue to expand, the limitations of existing model management languages and tools have become increasingly evident. To effectively address this challenge, it is crucial for the execution engines of model management programs to optimize their use of system resources. In essence, as models grow in size and complexity, it is crucial to enhance the performance of the model management tools to meet the evolving demands of software and system development.

Our proposal centres on the integration of two main features, *partial loading* and *partitioning*, into execution engines to boost the efficiency of model management program execution.

Partial loading revolves around equipping the execution engine with in-advance knowledge about the program's requirements. This knowledge empowers the engine to load and process only the parts of the model which are necessary for the program's execution, thereby eliminating unnecessary data loading. This not only impacts loading time, as less information needs to be loaded, but it also has a positive effect on memory footprint by loading less data in the memory.

Partitioning focuses on keeping model elements in memory only as long as they are actively required during the program's execution. When these elements are no longer in use or referenced by the program, they are promptly released, freeing up memory resources. This approach maintains an efficient memory footprint, ensuring

that only necessary model elements are maintained, ultimately resulting in a notable enhancement in overall execution efficiency.

This thesis introduces an approach that aims to enable the execution engine of model management programs to load only parts of models that are required for their execution and minimise the overhead of loading unnecessary parts in memory. In this work, by using in-advance knowledge about the model management programs provided by static analysis, execution engines are able to identify, load and process model elements of interest only. In addition, instead of keeping elements in memory until the end of execution, the elements will be disposed from memory when they are no longer needed by the model management program. In summary, our goal is to enhance the efficiency of model management program execution by introducing partial loading and partitioning capabilities to their execution engines.

3.2 Research Overview

In this section, we begin by introducing the research hypothesis in Section 3.2.2 and outlining its objectives in Section 3.2.4. To further define the boundaries of our research, Section 3.2.5 provides clarity on its scope.

3.2.1 Research Challenge

The main challenge of this research is to develop methods for optimising the execution of model management programs in terms of execution time and memory footprint. This optimisation should be achieved without the need to modify the existing program code or impose the burden of rewriting it on users. Instead, the goal is to automatically detect opportunities for enhancement within the program's execution engine, including resource management, and refine the behind-the-scenes loading and execution strategies.

3.2.2 Hypothesis

The hypothesis of this work is that static program analysis can substantially enhance the performance and reduce the memory footprint of model management programs in many scenarios.

3.2.3 Research Methodology

The research strategy adopted follows a typical software engineering process [46], incorporating iterative cycles of analysis, design, implementation, and evaluation. Each

phase plays a crucial role in advancing the research objectives and addressing the identified challenges.

- **Analysis:** During the analysis phase, the focus is on understanding the problem domain and identifying key challenges. The analysis involves conducting a thorough literature review to examine existing research and gain insights into the main challenges associated with managing large models in Model-Driven Engineering, particularly in terms of loading large models. It entails examining how existing MDE tools, such as the Eclipse Modeling Framework (EMF), handle complex software projects and large-scale models. Furthermore, the analysis explores the limitations of current model loading and processing approaches, especially concerning memory utilization and execution efficiency when dealing with large models stored in databases.

- **Design and implementation:** In the design phase, the focus is on creating solutions, outlining system structures and translating them into executable code. The goal is to turn the identified needs and obstacles into a clear design and implement it effectively. This phase involves identifying the potential benefits of selectively loading essential information into memory and implementing partial loading functionality within model management programs to address scalability challenges. Furthermore, novel algorithms or strategies are developed to manage model disposal, aiming to improve memory consumption and execution time. Finally, appropriate programming languages, frameworks, and technologies are selected for implementing the proposed solution.

- **Evaluation:** During the evaluation phase, the focus shifts towards testing and validating the implemented solution. This phase aims to assess the performance of the solution in addressing the identified challenges. Evaluation criteria include comparing the proposed approach with existing methods in terms of execution time, memory consumption, and overall performance using the available sets of large models.

3.2.4 Objectives

The general goal of this thesis is to evaluate the feasibility of reducing the execution time and memory footprint of model management programs via in-advance static analysis, particularly when dealing with large input models.

Given the broad scope of this aim, we break it down into the following specific objectives:

- **Objective 1: Implementing a Static Analyser** The first objective of this thesis is to develop and implement a static analyser for Epsilon programs with enhanced functionality compared to the existing static analyser. This analyser plays a crucial role in providing in-advance knowledge about the program. Conducting static analysis before the execution, equips the engine with the essential information needed to load only the necessary parts of the model, thereby enhancing efficiency.

- **Objective 2: Providing the Partial Loading Facility for Repository-based Models** Partial loading of XMI models was already achieved by the researchers in [51], and we need to investigate repository-based models. Therefore, this objective is broken into some sub-objectives:

    1. Investigate and identify a suitable repository capable of supporting the partial loading facility for repository-based models
    2. Develop and implement an algorithm to map models into the structure within the database
    3. Create and implement an algorithm that can automatically generate queries based on the output of the static analyser, facilitating data retrieval for program execution
    4. Evaluate the performance of the partial loading approach for repository-based models, comparing loading time and memory footprint with state-of-the-art method using large models
    5. Evaluating the correctness and reliability of our approach by conducting a comparative analysis of program output between state-of-the-art method and our approach

- **Objective 3: Providing A Partitioning Facility for Validating File-based Models**

    1. Investigate and enhance the implementation of the partial parser for XMI models
    2. Develop and implement an algorithm utilizing the static analyser output to partition the program.
    3. Modify the execution engine to load models based on the partitions created by the algorithm
    4. Assess the approach by comparing loading time to the non-partitioning approach

5. Verify the correctness of the approach by comparing the output of the validation program using the default XMI parser, partial XMI parser, and the partitioning approach

3.2.5 Scope

In this work, our focus is directed towards the Epsilon family of languages. Epsilon is a well-established open-source Eclipse project that has attracted the attention of industrial users and is frequently utilized in academic research projects [18]. Also, Epsilon has a connection to the Automated Software Engineering research group at the University of York, which leads the project's development. This association grants us a deep understanding of the platform, technical proficiency, and the ability to exert control over the codebase.

Epsilon stands out as an extensible platform, and it is not restricted by specific implementations or technologies, and it can seamlessly support a wide array of modelling technologies. The modular framework of Epsilon empowers us to explore an extensive range of modelling technologies, spanning from XML-based formats to spreadsheets, CSV, JDBC databases, and even specialized tools like Simulink.

The research objectives are aligned with the defined research scope:
Chapter 4 is dedicated to achieving the first objective, while Chapter 5 aims to address objective 2 and its five sub-objectives, building upon the infrastructure established in Chapter 4. In Chapter 5, we demonstrate how the partial loading of models, especially when they are stored in a database, can significantly enhance the efficiency of model management program execution compared to alternative approaches.

Our primary focus in this approach is the EOL language of the Epsilon platform. EOL serves as the core language of Epsilon, and extending these capabilities to other languages, such as model transformation or validation, is a more straightforward and less resource-intensive task.

Additionally, we specifically target the Neo4j database as the backend due to its suitability for model storage, given its structural similarity of the graph to the model structure. However, it is worth noting that the principles developed can be extended to other backend systems. At the end of the chapter, we conduct experiments to benchmark the performance of each of the developed implementations, fulfilling the fifth sub-objective of the first objective.

Chapter 6 delves into addressing objective 3 and its five sub-objectives, where the partitioning approach plays an important role in reducing loading time. Building upon the foundations laid out in Chapter 4, we partition the program based on its specific requirements.

Our primary emphasis in implementing the partitioning facility is on the Epsilon Validation Language (EVL), given the widespread application of model validation in the domain of MDE. Since Epsilon is inherently an extensible framework, extending our work to encompass other model tasks, such as model transformation, would be a straightforward process. This is especially true for the ETL language, as the ETL static analyser has been implemented[5], and effective metamodel extraction code is also available. Therefore, integrating these components to adapt the approach for ETL would require minimal effort. Expanding our approach to cover more languages may need extra effort to develop the required analysers. While using EOL's static analyser can help simplify this, we might need to adjust some language-specific statements to match their particular concepts and needs.

Towards the end of the chapter, we conduct a series of experiments to evaluate the performance of each of the developed implementations, thereby fulfilling the fifth sub-objective of the second objective.

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

4 Static Model Management Program Analysis

The concept of a static analyser involves a software tool or system designed to evaluate source code, program specifications, or similar software elements without actually executing the code. Unlike dynamic analysis, which occurs during program execution, static analysis takes place during compilation before the code is run.

Most of the time, a static analyser's primary goal is to find possible issues, mistakes, weaknesses, or places where the code could be improved. It does this by carefully examining how the code is composed, its syntax, and its grammar. It searches for mistakes in how the code is written, differences in the types of data used, and ways of writing code that could make it hard to maintain or lead to security problems. Besides finding issues, the information that a static analyser gathers can be useful for other activities. For example, it can help determine what data the program needs to run, make searches work better, or improve the whole program's design. It also looks for parts of the code that might not be required or used.

In this thesis, the role of the static analyser is also to analyse the program at compile-time. Static analysis will provide in-advance knowledge about model management programs. Indeed, static analysis yields valuable information while studying the program, which can have more comprehensive applications such as partial loading of a model, partitioning a program, or optimising queries.

In the context of partial model loading, providing information by the static analyser, such as resolved expression types, holds the potential to enable the execution engines to recognise, load, and process the parts of models that contain elements of interest. Given that the focus of this study revolves around the Epsilon framework, this chapter explores the static analysis of the Epsilon framework.

It is important to note that creating the static analyser for Epsilon is not considered a novel contribution of this thesis. However, it serves as foundational work upon which our research is built. Therefore, we contribute to enhancing the static analysis of epsilon languages and improve the functionality by adding new features, as discussed in Section 4.3. Other languages like ATL, Henshin, Viatra, and OCL also benefit from having a static analyser, as discussed in Section 4.5.

In this chapter, we start by reviewing the abstract syntax tree (AST) concept and discuss how the AST of the EOL language is structured in Section 4.1. Subsequently, in Section 4.2, we explore the development process of the static analyser for Epsilon, discussing the techniques employed to equip the static analyser, such as variable and type resolution. Furthermore, within Section 4.3, we show the novel additions incorporated into the static analysis as part of this thesis, including type inference

and type checking. The assessment of static analysis is presented in Section 4.4, followed by a discussion on examples of static analysers developed for other languages in Section 4.5. The practical application of the effective metamodel concept, which originates from the static analyser, is outlined in Section 4.6. Lastly, a chapter summary can be found in Section 4.7.

4.1 EOL Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a hierarchical data structure representing the syntactic structure of source code [1]. It provides a structured, abstract representation of the code's syntax, making it easier to be analysed, manipulated, and processed by programs or tools. ASTs are commonly used in compilers to convert source code into an intermediate representation that can be further optimised and transformed. The ASTs are a fundamental concept in compiler design, interpreter development, and other language processing tasks.

An AST breaks down the source code into a tree-like structure composed of nodes. Each node corresponds to a language construct, such as statements, expressions, operators, keywords, variables, and literals. The tree structure of the AST reflects the hierarchical relationships between different language constructs. Figure 4.1 shows an example of AST for the simple equation of *a = b + c*. In this AST, the root node represents the assignment operation (=). The left child of the root represents the variable *a*, and the right child of the root represents the addition operation (+). The left child of the addition node represents the variable *b*. The right child of the addition node represents the variable *c*. This AST visually captures the hierarchical structure of the mathematical expression. It shows how the numbers are combined with the operations to create the overall expression.

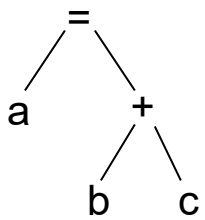


Figure 4.1: Simple example of AST

While some details are abstracted from the AST, it retains all the essential information needed to represent the code's logic and structure accurately. This makes it suitable for analysis, optimization, and transformation tasks. Different tasks may require different

representations of the AST. For example, a type-checking phase might involve an annotated AST, where nodes contain type information.

In this thesis, we need to be able to analyse Epsilon programs to get information from the programs before their execution. A key element of our analysis involves the exploration of the Abstract Syntax Tree (AST) of Epsilon programs. As EOL is the core language of Epsilon and other languages are built on top of this language, our initial focus is directed toward investigating the AST of the EOL language.

Epsilon offers a parser for EOL based on ANTLR [29]. This ANTLR parser generates ASTs, where each node carries a textual representation and an identifier, which are utilized by the EOL execution engine. Wei et al. [50] introduced an EMF-based metamodel for EOL to enable static analysis on a more comprehensive scale. This metamodel is employed to convert these ASTs into models that conform to the EOL metamodel, facilitating the process of transforming ASTs into a more structured representation for enhanced analysis.

Figure 4.2 presents a selection of fundamental components, outlining the essential building blocks that collectively shape an EOL program. Given the complexity of EOL, this presentation is restricted to introducing the foundational and innovative aspects of the EOL metamodel and the EOL standard library.

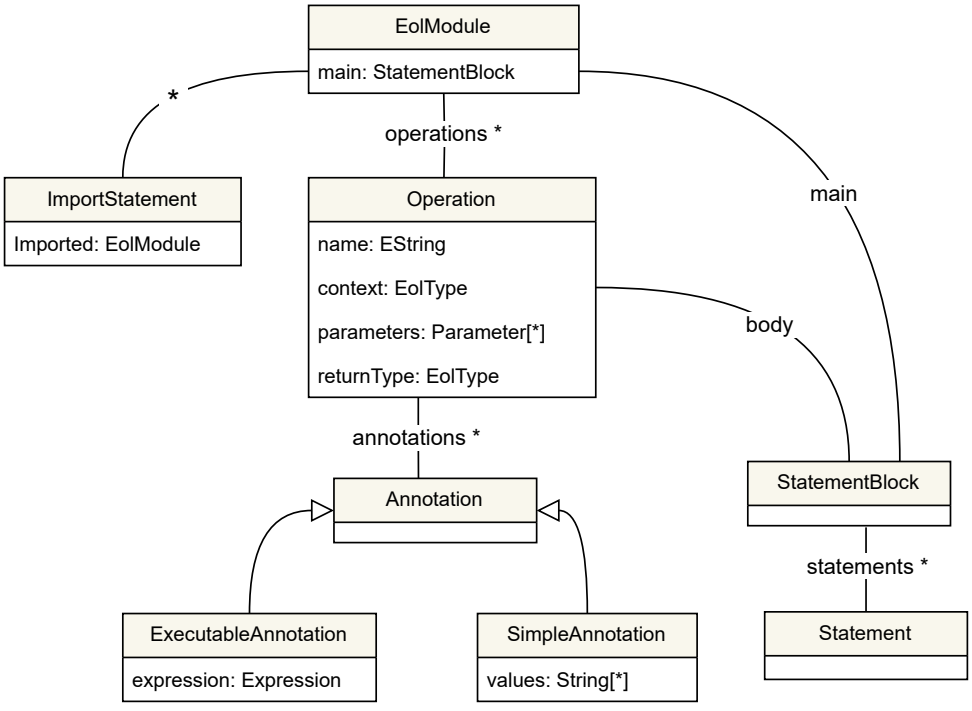


Figure 4.2: The structure the EOL metamodel from [17]

EOL programs are organized in *EolModule*s. Each *EolModule* defines a *body* and a number of *Operation*s. The *body* is a block of statements that are evaluated when the

module is executed. Each *Operation* defines the kind of objects on which it is applicable (*context*), a *name*, a set of *parameter*s and optionally a *return type*. *EolModule*s can also import other modules using *import statement*s and access their operations. The type resolution can be addressed through a resolution process after parsing the program into the EOL model. For instance, consider the simple EOL program in Listing 4.1.

Listing 4.1: EOL program to define and asssign a variable

```
1 var book = Book.all.first();
```

This program includes an assignment that returns the first instance of the *Book* from all instances present in the model and assigns it to the variable named *book*. The metamodel depicting the structure is presented in Figure 4.3, while the corresponding EOL model, computed from the AST, is visualized in Figure 4.4.

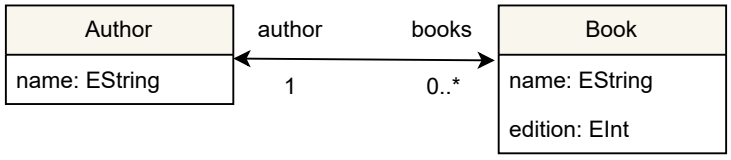


Figure 4.3: Book metamodel

All resolved types in Figure 4.4 are of *Any* type (discussed below) as type resolution has not been applied to the model yet.

4.2 Static Analysis of EOL langauge

Once the EOL model is created from the AST of an EOL program, resolution algorithms are applied. These algorithms include variable resolution (e.g., resolving identifiers to their definitions) and type resolution (e.g., primitive types and collection types). Together, they lead to a type-resolved AST.

With these resolution algorithms in place, the stage is set for Epsilon's static analysis, as introduced in [50][6]. By leveraging the type-resolved AST, the static analyser extracts crucial details from the program. This information encompasses the types and properties accessed by the program, forming a foundational basis for subsequent optimisation activities.

4.2.1 Variable Resolution

The initial phase of static analysis for an EOL program involves the resolution of identifiers to their definitions. In the context of EOL, these identifiers can belong

[6]https://github.com/epsilonlabs/haetae

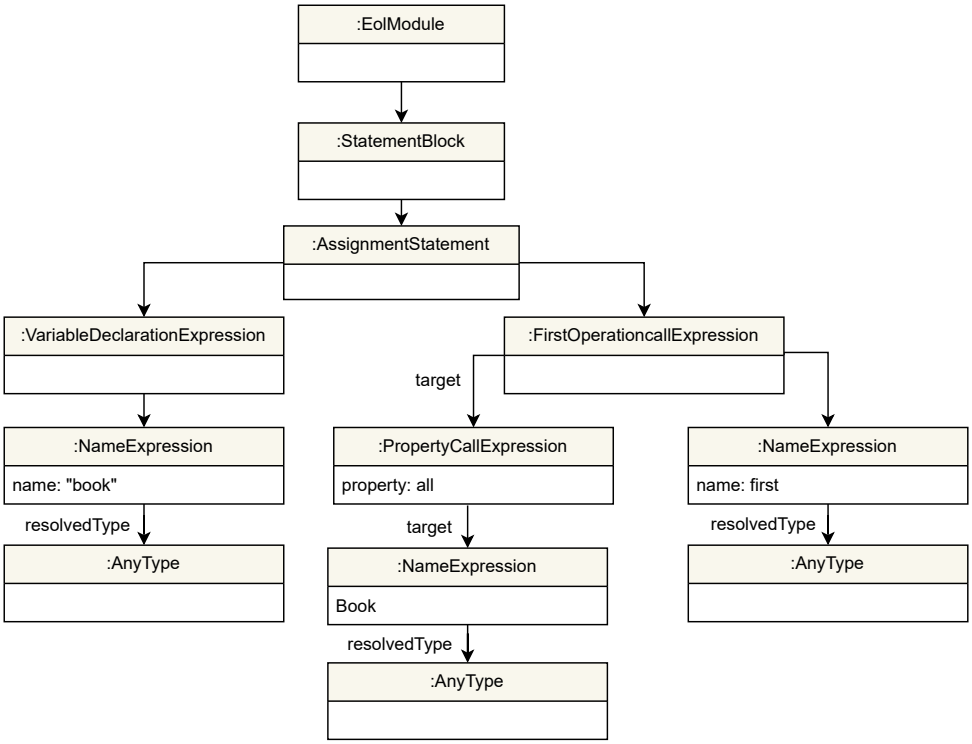


Figure 4.4: EOL model of Listing 4.1

to two main categories:

- **Declared Variables/Operation Parameters**: This includes variables and parameters that have been explicitly declared within the program. The variable resolver is responsible for establishing connections between the declaration and subsequent references of variables. For instance, in Listing 4.2, the variable resolver would link the reference to variable *name* in line 2 to its declaration in line 1.

Listing 4.2: EOL program to define and assign a variable

```
1 var name: String;
2 name = "Tom";
```

- **Undeclared Runtime Variables**: EOL permits referencing variables provided by the execution engine during runtime, even if they haven't been declared in the code by the developer. An example is the keyword *self* in an operation definition, which refers to the model element or object on which the operation is invoked. In the following example, the variable resolver would establish a link between the use of *self* and the corresponding object on which *printName()* is invoked.

Listing 4.3: EOL program to print title of book

```
1 operation Any printName() {
2 self.println();
```

It is important to emphasize that the types of model elements are not resolved during the variable resolution phase. The variable resolver's role is to establish relationships between identifiers and their declarations or runtime contexts, paving the way for subsequent stages of analysis.

4.2.2 Type Resolution

In EOL, several types are all subclasses of *Type* in the EOL metamodel. As shown in Figure 4.5, in EOL, there are several built-in primitive types (Boolean, Integer, Real, String) and collection types (Set, Bag, Sequence and OrderedSet). There is also a *Map* type, *Native* type and the *Any* type. The *Any* type is the basis of all types in EOL, including *Collection* types (inspired by the *OclAny* type of OCL).

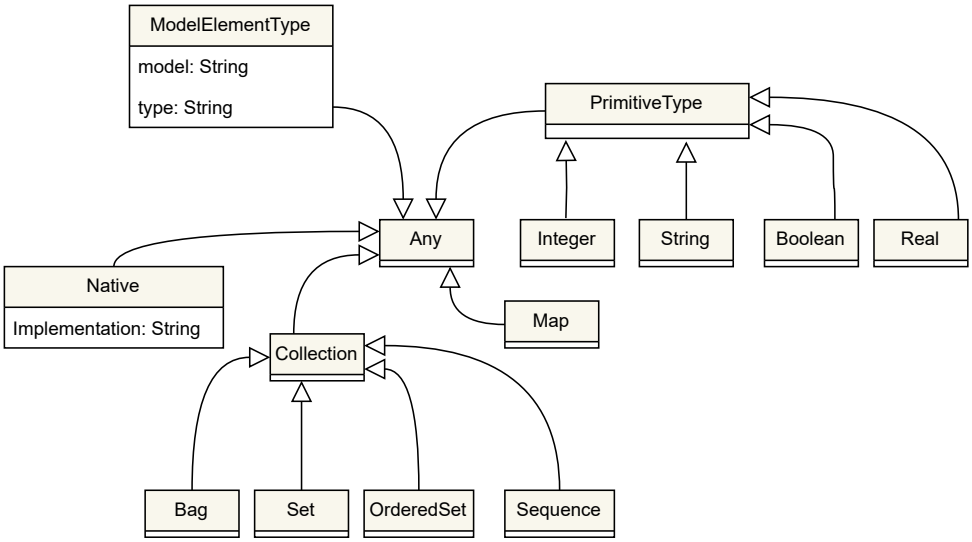


Figure 4.5: EOL type system [17]

Another subclass of Type in the EOL metamodel is *ModelElementType*. This subclass includes specific typing details associated with models defined using different technologies. The process of determining this typing information involves accessing the respective models.

The EOL program includes a *ModelDeclarationStatement* designed for referencing models within the source code. This statement plays a crucial role in facilitating access to models during the design phase. The syntax of a model declaration statement is according to Listing 4.4.

Listing 4.4: EOL program to print title of book

```
1 model book driver EMF {
2 nsuri = "http://book/1.0"
3 };
```

A *ModelDeclarationStatement* defines the model's name, the modelling technology (driver) that the model conforms to, and a set of driver-specific parameters.

Within this statement, *book* is the local name for a model, while the model's type is also established through the model declaration statement (*EMF* in this case). The *Nsuri*, which represents the namespace URI, serves as the unique identifier for the metamodel in accordance with EMF terminology. This definition is essential for obtaining the respective metamodel. For instance, consider the statement in Listing 4.5.

Listing 4.5: EOL program to print title of book

```
1 var book: Book;
```

At this point, we have knowledge that the variable *book* is of type *ModelElementType* with its specific element name being *Book*. Also, by accessing models/metamodels, the type resolver is able to resolve types with regard to models/metamodels. Hence, the type of *book* variable is set to *ModelElementType*.

The type-resolved AST of Figure 4.4 is demonstrated in Figure 4.6 where the *book* variable and return type of *first* operation is resolved as *ModelElementType*.

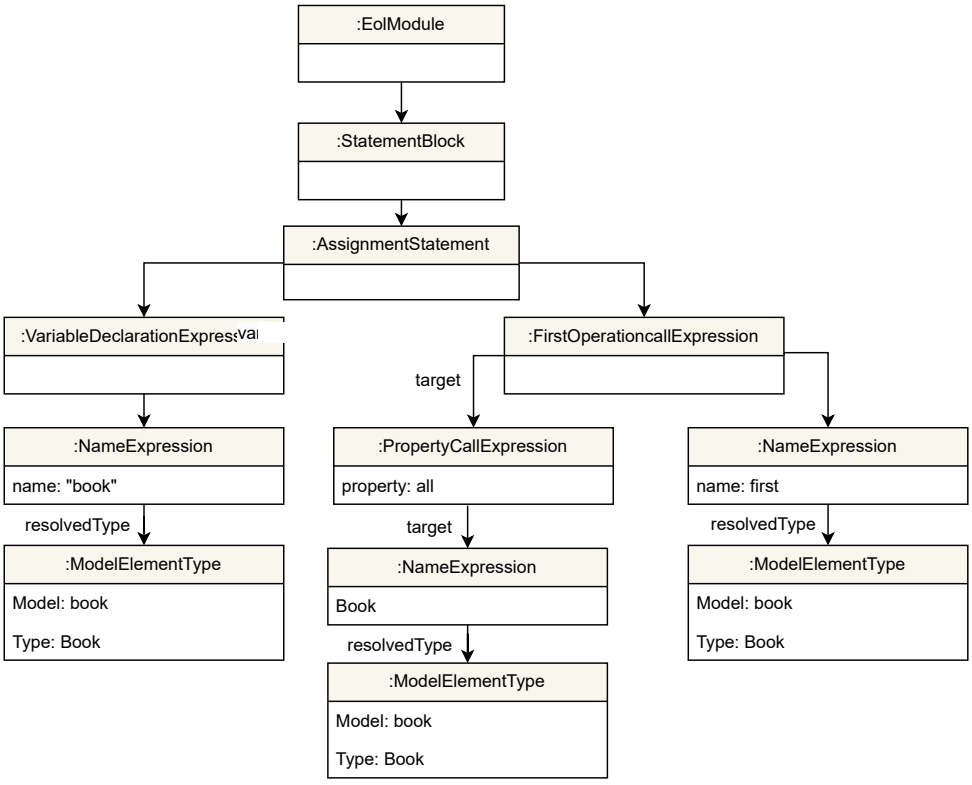


Figure 4.6: Type-resolved AST of Figure 4.4

4.3 Empowering EOL Static Analysis

As mentioned in the previous section, the initial version of the static analyser of Epsilon offers essential features such as type resolution and variable resolution. These components form the foundational infrastructure of the static analysis framework designed for the Epsilon languages.

However, we contributed[7] by encompassing enhancements of the static analyser in two key aspects: type inference and the incorporation of a type checker. These modifications lead to the generation of compile-time errors and warnings. It is important to note that while the identification of compile-time errors is a by-product outcome, the primary focus of our work is to enhance the analysis process. The ultimate objective is to improve the accuracy of design-time information extracted from the program. This project has been released and can be accessed through a public GitHub repository[8].

4.3.1 Type Inference

Type resolution involves explicitly defining and confirming the data type of variables, expressions, and values within a program. On the other hand, with type inference, the compiler or interpreter deduces the data type of variables, expressions, and functions based on their context and usage within the program. This mechanism reduces the need for explicit type annotations, allowing developers to write more concise and expressive code while still benefiting from the advantages of static typing.

To enhance the efficacy of static analysis for Epsilon programs, one step for inferring types in an EOL program is the introduction of pseudo-types into the language. These pseudo-types have been incorporated into the programming paradigm, taking inspiration from the concepts within the Object Constraint Language (OCL) [41], for the purpose of static analysis.

They are called pseudo-types as they cannot be instantiated. They are used to determine the type of *self* in operation signatures. The exact type of these will be determined at the end of compile-time static analysis. These Pseudo types (*EolSelf*, *EolSelfCollectionType*, *EolSelfExpressionType*, *EolContentType*) have been integrated into the EOL language, enriching its capabilities in the domain of static analysis. Usage of these types is demonstrated in Table 4.1.

In EOL language, the operation is defined as below.

Listing 4.6: The syntax of operation signature in EOL

[7] This work was implemented in collaboration with another PhD student (Qurat ul ain Ali) in the context of the Lowcomote Training Network, that funded this research.

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

Type Name	Description
EolSelf	Type of context
EolSelfCollectionType	Collection type of context
EolSelfExpressionType	Resolved type of expression parameter
EolSelfContentType	Content type of collection

Table 4.1: Pseudo Types

```
operation target-type println() : return-type {
}
```

For instance, if we have a signature like operation **String** println(): **String**, it signifies that this operation can be invoked on a *String* variable and will return a *String* as its result.

The pseudotype EolSelf plays a crucial role within the operation signature, enabling the propagation of the contextual type as the return type when the *println()* function is invoked. For instance, in Listing 4.7, *self* is resolved according to the context type and the return type is also set as context type. For instance, if we were to make the call **"abc".println()**, the resulting return type would be *EolPrimitiveType.String*.

Listing 4.7: Println operation

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

Hence, the utilization of *EolSelf* proves advantageous in the sense that if the static analyser successfully resolves the type of the context in the *println()* call, it subsequently gains the capability to infer the type of the operation's return value.

EolSelfCollectionType is a pseudo type specially for *EolCollectionTypes*. It is used in the operation's signature as follows:

Listing 4.8: get() operation

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

Whenever an operation with *EolSelfCollectionType* return type, such as *get()* in Listing 4.8, 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*>, the return type would also be *Sequence*<*String*>. Hence, the static analyser resolved the return type according to

the context type.

EolSelfContentType is also a pseudotype just for EolCollectionTypes. It is used in the operation signature as follows:

Listing 4.9: getContent() operation

```
1 operation Collection<Any> getContentType():EolSelfContentType {
2 }
```

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

EolSelfExpressionType is also a pseudotype just for EolCollectionTypes. It is used in operation signatures as follows:

Listing 4.10: collect operation

```
1 operation Collection<Any> collect(a: Any) : Collection<
    EolSelfExpressionType>{
2 }
```

Whenever an 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 an operation is called as Listing 4.11, the return type would also be *EolPrimitiveType.Boolean* because the expression which is the input of *collect* operation (f<3) is *Boolean*.

Listing 4.11: Example of Collection<EolSelfExpressionType> as return type

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

4.3.2 Type Checking

Once the types of variables, expressions, and statements are resolved or inferred, the static analyser can check the type compatibility. It verifies whether the operations being performed are valid for the determined types, helping to detect type-related errors before running the program. Developers can catch errors early in development by performing static analysis with type resolution and inference. This saves time and effort compared to discovering such issues during runtime testing or in production.

This thesis uses the static analyser to extract crucial information about various model elements (such as variables, functions, classes, etc.) and their properties from the

program. The insights provided by the static analyser help determine which specific parts of the model are required for executing the program. Hence, while the primary purpose of using a static analyser might be to gather information about model elements, warnings and errors that the analyser produces can be seen as valuable by-products. These warnings and errors indicate potential issues or inconsistencies in the code.

Figure 4.7 depicts a snapshot of an EOL editor interface. Each line within the editor corresponds to a statement purposely crafted to highlight errors and warnings generated by the static analyser.

- **Accessing model in compile time:** In lines 1 to 3, there is a *ModelDeclaration* statement which defines the name of a model and the namespace URI (nsuri) for accessing the associated metamodel. The *book* model that conforms to *book* metamodel is being queried in this code.

- **Variable resolution:** There are variable declarations in lines 5 - 8. As *Book* and *Author* are the classes presented in the metamodel, the variable declarations are valid, and the variables are initiated. However, an error arises in line 8, pointing out the *Library* class's absence in the metamodel. This results in the error message as there is no type such as *Library* in the metamodel or no EOL type matched with *library* type. Hence, the static analyser is successfully accessed to the metamodel in compile-time and can recognise which types can be declared according to the metamodel and which are *unknown*.

- **Type compatibility assurance in assignment statements:** In line 10, an error is reported on the right side of an assignment statement. In the assignment statement, the right side of the assignment should be the same as or a sub-class of the type of the left side. In line 10, a property call, which is resolved as *Sequence*$<$*Book*$>$, is assigned to the *b* variable, which is resolved as a *EOLModelElement*. Therefore, a compatibility error here is detected by the static analyser.

- **Inferring resolved types for property call expressions:** Line 14 reports another compatibility error. Static analysis is able to infer resolved types even when it is an operation that is called on a property call. Both sides of the assignment are *ColeIction* type, but an error is shown to the developer as the type of *content* of collections is different.

- **Navigating the type hierarchy:** In line 16, there is another assignment in which its sides hold different types, but the static analyser produces a warning instead of an error. The reason is variable *n* is of *Any* type, and all types are subclasses of *Any* in the EOL type system (see Figure 4.5). Therefore, all types can be assigned

to *Any* variable logically, but a warning is helpful as some information might be missed in the assigned variable.

- **Detecting type inconsistencies in the context of operations:** Line 18 has an operation call to *printName* operation. The produced error is complaining about a mismatch in the context of the operation. The *printName* operation signature in line 24 shows this operation only applies on *String* typed variables, but in line 18, the call is invoked on the return type of *Author.all.first*, which is *Author* (EOLModelElement).
- **Uncovering parameter type mismatches:** Line 20 demonstrates an operation call that does not apply to a specific context but needs a parameter. The signature of *getNumberOfBooks* on line 34 indicates that the expected parameter type should be identical to an Author. Nevertheless, on line 20, the parameter *b* is of type *Book*. This inconsistency generates an error, highlighting the mismatch in parameter types.
- **Mismatch in number of parameters:** The *getNumberOfBooks* operation is designed to accept a single parameter, yet it is called with two parameters in line 21.
- **Detecting Missing Return Statement:** Line 24 shows an error regarding the missing statement for the *printName* operation.
- **Type resolution for *self* variable:** In lines 27 to 30, the *getAuthor* operation is defined, which is applied on *Book* objects. In line 29, an error appears as *self* refers to the context of operation (*Book*), but it is assigned to the *String* type variable, which is not allowed. It shows the static analyser correctly resolves the type of *self* variable.

4.4 Eugenia - Test case

The developed static analyser was put to the test within the context of Eugenia [20], a well-established tool constructed using the Epsilon platform. Eugenia serves the purpose of producing GMF (Graphical Modeling Framework) editors from an Ecore metamodel that has been annotated with additional information. One of the key components of Eugenia is a substantial EOL transformation containing approximately 1.2K lines of code. This transformation carries out the task of converting annotated metamodels into four distinct models that are crucial for the GMF framework's operation. These models are necessary for generating graphical editors.

```
1  model book driver EMF{
2  nsuri= "http://book/0.1"
3  };
4
5  var a : Author;
6  var b : Book;
7  var n: Any;
8  var library : Library; //Unknown type Library
9
10 b = Book.all;        //Sequence<Book> cannot be assigned to Book
11
12
13 var col: Collection<Integer>;
14 col = Book.all.select(b | b.author = a);        //Collection<Book> cannot be assigned to Collection<Integer>
15
16 n.printName();      //printName may not be invoked on Any, as it requires String
17
18 a = Author.all.first().printName();   //printName can not be invoked on Author
19
20 getNumberOfBooks(b);     // Parameters type mismatch
21 getNumberOfBooks(a,b);   // Number of parameters doesn't match, as getNumberOfBooks requires 1 parameters
22
23
24 operation String printName() : String{   //This operation should return String
25 }
26
27 operation Book getAuthor() : Author {
28     var s: String;
29     s = self;               //Book cannot be assigned to String
30     return self.name;       //Return type should be Author instead of String
31 }
32
33
34 operation getNumberOfBooks(author : Author) : Integer{
35     return author.books.size();
36 }
```

Figure 4.7: Type compatibility errors - Example

Eugenia simplifies the process by automatically generating three types of files (with extensions .gmfgraph, .gmfmap, and .gmftool) that GMF requires. All of this is accomplished from a single annotated Ecore metamodel. However, while Eugenia proves highly useful, there are instances where warnings emerge. These warnings specifically revolve around cases where a parent type is being provided for a required type.

For a concrete example, consider the transformation named Ecore2GMF.eol from Eugenia. This transformation is subjected to static analysis, and a snapshot of this analysis is visible in Figure 4.8 on the left. Additionally, insight into one of the imported modules is presented in the figure on the right side. This analysis aids in understanding how the static analyser evaluates the correctness of Eugenia's capabilities in transforming metamodels into the necessary models for GMF's graphical editor generation. As shown in Figure 4.8, it becomes evident that the static analyser does not raise any errors in Figure 3.8 concerning the validity of the accurate transformation. This observation signifies that the analyser operates without producing any incorrect indications, which we refer to as "false positives." In other words, the absence of error notifications in the correct transformation highlights the precision of the static analyser's evaluations.

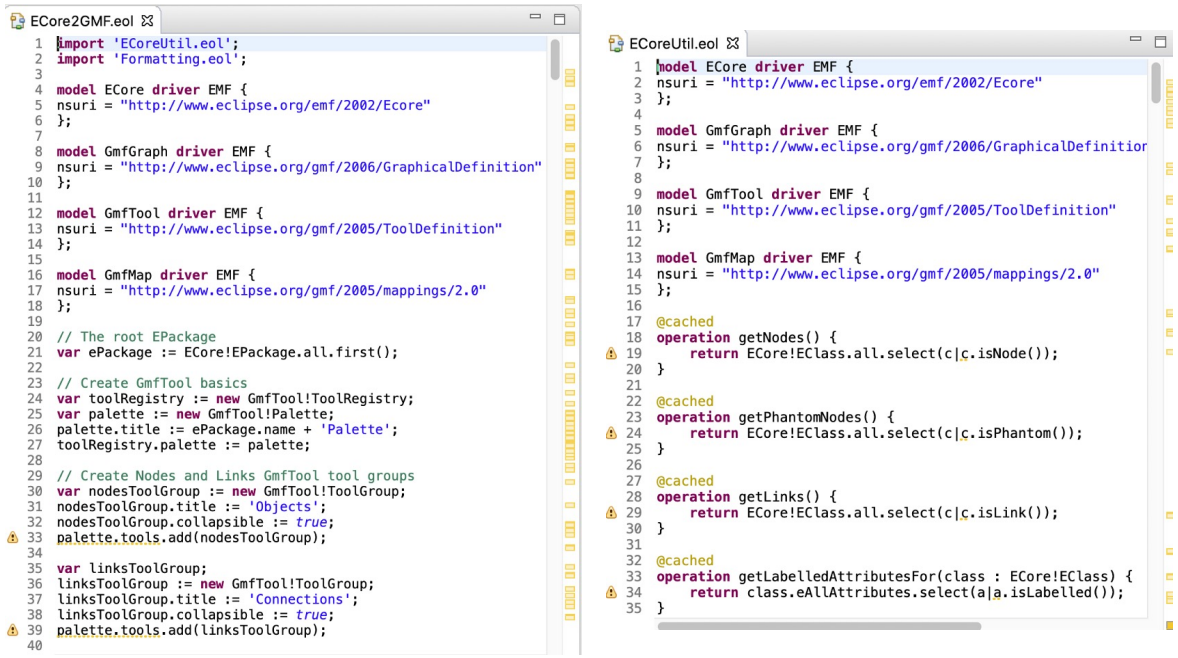


Figure 4.8: Static analysis on Eugenia code

4.5 Related Work

Developing a static analyser to analyse the program and extract information about the code during compile-time is not a novel concept. Some programming languages, general-purpose ones like Java and domain-specific ones like ATL, get advantages from utilizing static analysers. This section provides a brief overview of some of these tools.

AnATLyzer [11] is a tool for static analysis of ATL model transformations. It is an IDE that provides type checking, quick fixes and problem explanations. AnATLyzer focuses on three main points: it checks that the source metamodel is correctly typed concerning the transformation, it ensures that the model generated through transformation conforms to the target metamodel, and it identifies any conflicting or missing rules. This static analyser is limited to ATL model transformations only.

In [7], 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 as critical pairs. Each essential 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 dependence.

Another tool in [47] provides a static analysis facility for graph transformations. This work is based on Constraint Satisfaction Programming (CSP). It also presents a type checker for the 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 [52], a pseudo-type OCLSelf, is introduced to infer the type of context for a few operations such as:

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

Willink [53] 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.

4.6 Effective Metamodel Computation: An Application of Static Analysis

As mentioned earlier, we need to provide in-advance knowledge about the specific parts of the model containing relevant model elements to the execution engine for the purpose of partial model loading in this work, which includes understanding the essential properties and attributes required for the program execution. This vital information can be gained through the static analyser. In this section, we discuss how this information is communicated to the execution engine and provide an explanation of the steps required to extract this information from the program.

The model elements, along with their associated properties (attributes and references), which are essential for the program's execution, are presented in the form of a specialized format called an *effective metamodel*.

The concept of effective metamodel was introduced by Wei et.al [51]. The effective metamodel is a subset of the model's original metamodel, consisting only of types and properties likely[9] to be accessed by the program.

The effective metamodel is constructed using Algorithm 4.1 (discussed in Section 4.6.2), which uses the resolved types of expressions and contains only the types necessary for executing the program from which it is extracted. In our prototype implementation, effective metamodels are only computed for EMF-based models, but in principle, this approach can also be applied to other metamodelling technologies.

4.6.1 The Structure of Effective Metamodel

As shown in Figure 4.9, an effective metamodel consists of an *EffectiveMetamodel* class with *name* and *nsuri* attributes. The *EffectiveMetamodel* class is connected to an

[9] In some cases, the execution engine needs to load unnecessary model properties as well because of a lack of information before the execution (discussed later in detail in section 4.6.2)

EClass that has *EStructuralFeature*s. The *EffectiveMetamodel* class is connected to an *EClass* by *allOfKind*, *allOfType* and *types* references.

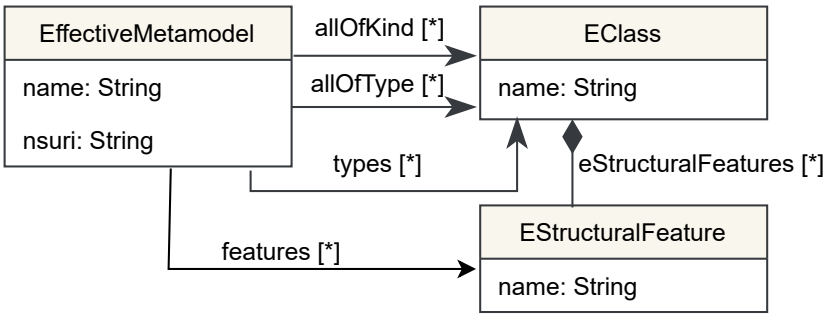


Figure 4.9: The structure of effective metamodel (adapted from [51])

The *allOfKind* and *allOfType* references specify the instances of types that the execution engine should load. The difference between these two references is that *allOfKind* is used when all instances of a class (including subclasses) should be loaded. In contrast, *allOfType* reference means the execution engine should consider only the elements that are direct instances of the class (without considering any of its subclasses). The *types* reference is used for specifying class instances, which should be loaded only when they appear in the references of model elements necessary for executing the program.

For instance, let's consider the EOL program shown in Listing 4.12. This program prints the number of books written by each author, using the metamodel illustrated in Figure 4.3.

Listing 4.12: EOL program to print the number of books written by each author

```
1 model book driver EMF {
2   nsuri = "HTTP://books/1.0"
3 }
4 for(author:Author in Author.allInstances()){
5   author.books.size().println();
6 }
```

For every class used in the program (such as *Author*), an *EClass* is added in the respective effective metamodel. The *EClass* contains collections of *structural features* that reflect the attributes and references of the type accessed by the program (such as *books* reference).

The process which extracts this effective metamodel from an EOL program is described in Algorithms 4.1 and 4.2. This algorithm is easily extendable for other Epsilon languages, but considering the motivating example, we will discuss the version that works with the EOL program in this section.

4.6.2 Effective Metamodel Computation

In Algorithms 4.1 and 4.2, the type resolved AST extracted by the static analyser is visited. The resolved types for the Listing 4.12 is shown in Table 4.2.

Table 4.2: Resolved Types Calculated by the Static Analyser

Line number in Listing 4.12	Expression	Resolved Type
1	model book driver EMF	Model declaration
2	nsuri = ”http://books/1.0”	Assignment statement (String)
4	author	Model Element (Author)
4	author.allInstances()	Operation call expression (Sequence<Author>)
5	author.books.size()	Operation call expression (Integer)
5	author.books	Property call expression (Sequence<Book>)
3	author	Model element (Author)

With the resolved types calculated by the static analyser (as detailed in Table 4.2), the algorithm assesses the statements and expressions within the program. In this process, the algorithm identifies the elements that must be loaded to ensure the successful execution of the program. Subsequently, it includes these elements within the effective metamodel.

Algorithm 4.1 is interested in calls of *all()* and *allInstances()* operations and *property call*s (such as *Author.books*) as they are the only way to navigate to model elements in Epsilon programs. In lines 7-11, the *all()* and *allInstances()* operation calls are handled by Algorithm 4.1 to add the respective *EClass*es to the effective metamodel. Suppose the target of the operation call is a model element type. In that case, it will be added to an *allOfKind* reference of the effective metamodel (line 11) and, if it already exists in the effective metamodel under the *allOfType* or *types* references, the *EClass* is moved to the *allOfKind* reference.

In lines 15-28, Algorithm 4.1 handles property calls to populate the effective metamodel further. In Algorithm 4.2, if the accessed property is *all* (it is an alias for *allInstances()*), then the target element type will be treated identically to the *allOfKind* operation call (lines 2-3). When the target of property call is a model element if an attribute of the model element is accessed, it is added to the effective metamodel as an *EAttribute* (lines 8-9), or if it is a reference, then it is added as an *EReference* (lines 10-11). When a reference is accessed by the program, the instances of the target EClass of the reference need to be loaded. If the reference is a *containment* reference, then it will be added to *types* reference of the effective metamodel; otherwise, it will be added to *allOfKind* reference.

Algorithm 4.1 EOL Effective Metamodel Extraction Algorithm (1 of 2)

```
1:  procedure COMPUTEEFFECTIVEMETAMODEL(effectivemetamodel)
2:      let EM = effective metamodel;
3:      for all operation call expression do
4:          if IsModelElement(operation.target) then
5:              let EC = target.type;
6:              if operation.name.equals(all or allOfKind or allInstances) then
7:                  if allOfType.contains(EC) or types.contains(EC) then
8:                      move EC under EM's allOfKind reference;
9:                  else
10:                     allOfKind.add(EC);
11:             else if operation.name.equals(allOfType) then
12:                 if not allOfKind.contains(EC) and not allOfType.contains(EC) then
13:                     allOfType.add(EC);
14:     for all property call expression do
15:         if IsModelElement(propertyCall.target) then
16:             let EC = target.type
17:             handlePropertyCallExpression(propertyCallExpression,EC)
18:         else if IsCollection(propertyCall.target) then
19:             if IsModelElement(collection.content) then
20:                 let EC = collection.content.type
21:                 handlePropertyCallExpression(PropertyCallExpression,EC)
22:             else if IsAny(collection.content) then
23:                 for all EClasses in EM do
24:                     handlePropertyCallExpression(PropertyCallExpression,EClass)
25:         else if IsAny(propertyCall.target) then
26:             for all EClasses in EM do
27:                 handlePropertyCallExpression(PropertyCallExpression,EClass)
```

Algorithm 4.2 EOL Effective Metamodel Extraction Algorithm (2 of 2)

```
1:  procedure HANDLEPROPERTYCALLEXPRESSION(propertyCallExpression, EClass)
2:      let EC = EClass
3:      if the property.name.equals(all) then
4:          if allOfType.contains(EC) or types.contains(EC) then
5:              move EC under EM's allOfKind reference;
6:          else if not allOfKind.contains(EC) then
7:              allOfKind.add(EC);
8:      else if IsAttribute(property) then
9:          EC.attributes.add(property)
10:     else if IsReference(property) then
11:         EC.references.add(property)
12:         EReference ref = (EReference) property
13:         let EType = ref.type
14:         if IsContainment(ref) then
15:             types.add(EType)
16:         else
17:             allOfKind.add(EType)
```

Figure 4.10 illustrates the effective metamodel extracted from the EOL program in Listing 4.12. In Figure 4.10, the attributes of the *EffectiveMetamodel* class are filled by the original metamodel, which are the *name* and the *nsuri* of the metamodel. All instances of *Author* must be loaded for running the EOL program. The *Author* class is added to *EffectiveMetamodel* under the *allOfKind* reference according to lines 7-11 of Algorithm 4.1.

The *books* reference of *Author* is also required (line 5 of Listing 4.12). Hence, it is added to *Author* as an *EReference* according to lines 10-11 in Algorithm 4.2. The resolved type of *books* reference is equal to *Book* (see Table 4.2), so according to lines 14-17 in Algorithm 4.2, the *Book EClass* is added to effective metamodel using the *allOfKind* reference.

When we compare the metamodel presented in Figure 4.10, displaying the effective metamodel, with the initial metamodel depicted in Figure 4.3, a notable difference emerges. Specifically, information that isn't required for executing the program, like the *name* attribute of the *Book* class, is intentionally excluded from the effective metamodel. Therefore, adopting a loading model strategy to load the model based on an effective metamodel instead of relying on the original metamodel ensures that the execution engine does not unnecessarily consume memory resources to load information that is not exercised by the program at runtime.

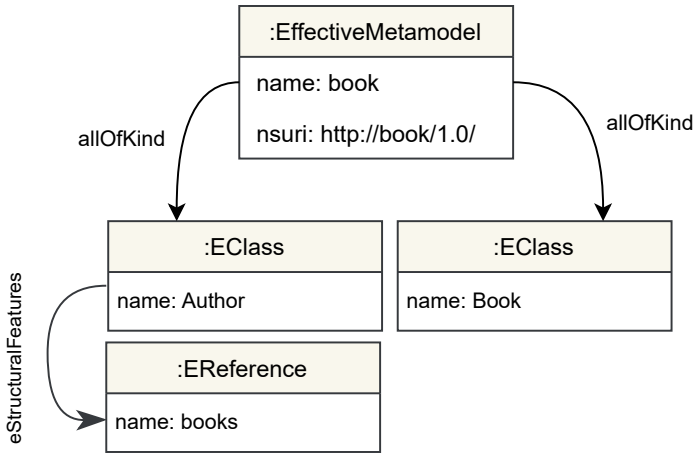


Figure 4.10: Effective metamodel of the EOL program shown in Listing 4.12

Accommodating Untyped Variables and Expressions

Algorithm 4.1 visits the abstract syntax graph to consider all statements and expressions in the code. Thus, constructing an effective metamodel relies on the ability of the static analyser to precisely resolve their types. If in a property call expression, the resolved type of the left-hand side is unknown (*"Any"* in terms of the Epsilon type-system), then the name of the property is added to an *unresolvedProperties* set. After the effective metamodel has been extracted, the *unresolvedProperties* set is used to augment the effective metamodel with additional *FeatureAccess* elements for all the types of the effective metamodel that have features matching properties in the set.

The algorithm's decision to not assign the resolved type of the *p* variable as the value on the right side of the assignment and setting it as ModelElement(*Author*) prompts an important question. This choice is motivated by the need to ensure reliable type resolution for variables assigned in multiple locations within a program.

For instance, if the algorithm were to resolve the type of *p* as ModelElement(*Author*) in line 1, a conflict would arise in line 4, where the *edition* attribute of *t* (which is Integer) could no longer be feasibly assigned to the *p* variable. This conflict arises due to a disparity with the EOL type system, which permits the assignment of any data type to an undeclared type variable using the *var* keyword. Therefore, we keep the flexibility of the EOL program by resolving the type of *p* variable as *Any*.

For example, in Listing 4.13, the *name* of the first *Author* of the input model is printed. All instances of *Author* with *name* attribute and all instances of *Book* with *edition* attribute are required for running this part of the program.

Listing 4.13: EOL Example Code

```
1 var p = Author.all().first();
```

```
2 var t = Book.all().first();
3 p.name.println("Name: ");
4 p = t.edition;
5 p.println("Book edition: ");
```

In the first line of Listing 4.13, the first item of all instances of *Author* in the model is assigned to *p* variable. As the type of *p* is undefined, the resolved type of *p* is considered as *Any*. Hence, the condition in line 22 of Algorithm 4.1 is not satisfied, and the *name* attribute of *Author* is not added to the effective metamodel in the first iteration.

Algorithm 4.1 handles this situation by considering possible (instead of precise) types for variables and expressions. In lines 1-2 of Listing 4.13, according to lines 9-16 of Algorithm 4.1, *Author* and *Book EClasses* are added to the effective metamodel. Then, the *name* attribute is accessed by the program, but as the resolved type of *p* is *Any*, it adds the *name* attribute to all *EClasses* that are already in the effective metamodel. Hence, according to lines 8-9 in Algorithm 4.2, *name* will be added to the effective metamodel for the *Author* and *Book* EClasses and the execution engine will load the *name* of all instances of *Project* and the *Author* of all instances of *Book* into memory. For *edition* attribute, it is the same, but because according to metamodel, there is no *edition* attribute for the *Author* EClass, it is only added to *Book* EClass.

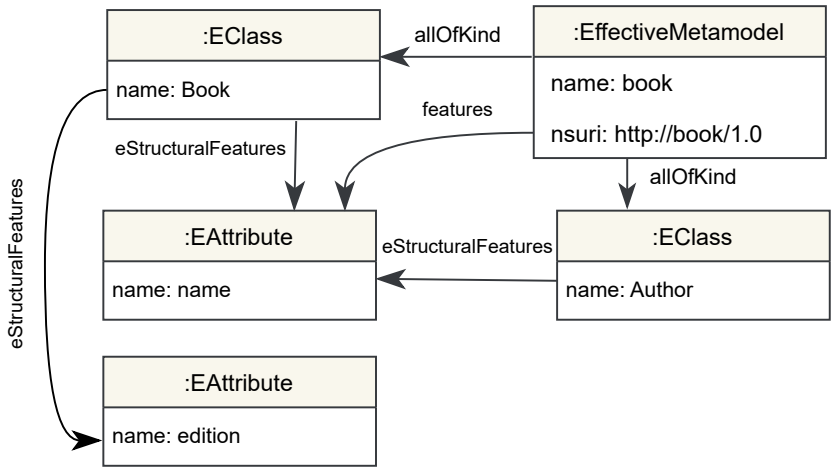


Figure 4.11: Extracted effective metamodel for Listing 4.13

As shown in Figure 4.11, the effective metamodel includes an attribute which is not necessary for running the program (*name* of *Book*), but because of lack of information before the execution and to be on the safe side, the execution engine loads more information from the model. Loading more information and running the program is preferable to loading less information than required at runtime.

An important note to consider is that according to the algorithm, when a property is resolved as *Any* type, it is added to all classes already present in the effective

metamodel. Therefore, if an EClass is added to the effective metamodel later, the property will not be added to the new class. To ensure all classes in the effective metamodel are treated the same, after extracting the effective metamodel, the algorithm is applied again to make sure that *Any typed* properties are added to all effective metamodel's classes. If the effective metamodel remains unchanged after applying the algorithm, it is safe to proceed. If it changes, the algorithm should be reapplied until no further changes occur. As the time consumption of extracting the effective metamodel is negligible, repeating the algorithm will not have a noticeable impact on the efficiency of our approach.

4.7 Chapter Summary

This chapter focused on the role of static analysis and its significance in this research. We began by introducing the concept of Abstract Syntax Tree and proceeded to explore the development of the EOL static analyser. Our contributions to enhancing the Epsilon framework's static analyser were highlighted, including the incorporation of new features. The evaluation of static analysers and a survey of similar tools in other programming languages were also discussed. Furthermore, we introduced the concept of the effective metamodel structure and detailed an algorithm to extract it from the EOL program—a practical application of the static analyser within this thesis.

It is worth noting that the static analyser has been extended to support other Epsilon languages, such as EVL and ETL, which are publicly available on the GitHub repository [10]. Chapter 5 utilizes the EOL static analyser, and in Chapter 6, we employ the EVL static analyser.

The subsequent chapter will further investigate one of the main contributions of our research, providing a clear understanding of how we leverage the effective metamodel to load information from a database-backed model repository for executing EOL programs.

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

5 Partial Loading of Repository-Based Models

This chapter describes the design and implementation of an approach for partially loading repository-based models. As discussed in Chapter 2, the scalability issue in the current generation of model repositories is prominent (see Section 2.6). This thesis hypothesizes that adding a partial loading feature to the execution engine of model management programs can substantially enhance both loading time and memory consumption efficiency.

To test this hypothesis, we present an approach for partial loading of large models that reside in graph database-based model repositories, as these have demonstrated superior performance compared to repositories supported by relational or document databases [5, 6, 14]. This approach leverages sophisticated static analysis of model management programs and auto-generation of graph (Cypher) queries to load only relevant model elements.

This work offers two main contributions. Firstly, a novel algorithm is presented, focused on the translation of a program's effective metamodel into a set of graph queries. These queries are optimized to retrieve only the specific elements, relationships, and properties that the program is expected to interact with during its runtime. This optimization can greatly enhance the efficiency of query execution.

Secondly, this research effort includes the development of a prototype implementation of the mentioned algorithms. The prototype leverages the Eclipse Epsilon family of model management languages and integrates with the Neo4j graph database. This implementation serves as a practical demonstration of the proposed methods and algorithms, showcasing their applicability.

In this chapter, we begin by using a motivating example in Section 5.1 to explain the challenges in more detail. We then discuss our proposed approach in Section 5.2. In Section 5.3, we present the implementation of our approach while also discussing its limitations. Section 5.4 reports the results of evaluating our approach compared to the current state of the art, discusses these results, addresses potential validity concerns, and finally, Section 5.5 concludes the chapter.

5.1 Motivating Example

As a motivating example, consider a model that conforms to a contrived Project Scheduling Language (PSL), the metamodel of which is shown in Figure 5.1. According to the PSL metamodel, each *Project* has a title and a description, and it consists of *Task*s and *Person*s. *Task*s can be completed through automated means (*AutomaticTask*s) or manually (*ManualTask*s). All *Task*s have a title, but only *ManualTask*s have a duration

and a start time. Also, each *ManualTask* specifies the *Effort* that different *Person*s in the project will contribute to it (as a percentage of their time).

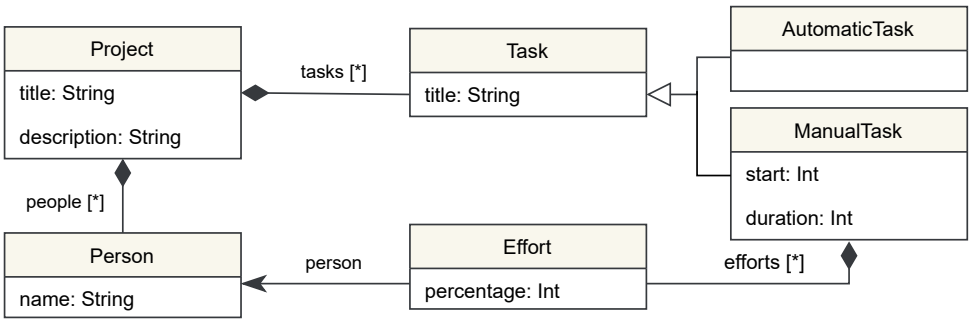


Figure 5.1: PSL Metamodel

Consider a model that conforms to the PSL language (see Figure 5.1), and on which we would like to print the number of people who contribute to each *ManualTask*. This program could be written in a language such as the Epsilon Object Language (EOL) (introduced in Chapter 2) as shown in Listing 5.1.

Listing 5.1: EOL program to print number of people who contribute to each task

```
1 for(task:ManualTask in ManualTask.all()){
2   task.title.print();
3   task.efforts.person.asSet().size().println();
4 }
```

In Listing 5.1, line 1 defines the *task* variable, and it goes through all instances of *ManualTask*. In lines 2-3, the *title* of each *task* and the number of people who contribute to each *task* are printed.

The program only accesses the *title* attribute and *efforts* reference of *ManualTask*, the *person* reference of *Effort* and no other properties of the PSL metamodel. To run this program against a repository-based model (e.g., stored in a database-backed repository such as CDO or NeoEMF), the EOL engine uses the respective driver to fetch all instances of model element types and properties of model elements on demand.

Therefore, as mentioned before, without using in-advance static analysis of the EOL program, there is no way to tell before executing the program which features of the required model elements should be retrieved from the repository. In this situation, the two alternatives at runtime are to *greedily* fetch all properties and attributes of model elements retrieved from the database or to *lazily*, which fetches attributes and references on demand. The former strategy favours execution time over memory consumption, while the second strategy requires less memory but potentially multiple round-trips to the repository, which can be detrimental to performance.

Considering Listing 5.1 that uses the *title* attribute and *efforts* reference of *ManualTask*, the *person* reference of *Effort* and no other attributes or references, the two strategies are sub-optimal:

- Greedy: When all instances of *ManualTask* are fetched in line 1, all their attributes and references would be fetched too (including *ManualTask.duration*). As *ManualTask.duration* is not accessed by the EOL program, fetching its value from the repository and maintaining it in memory is wasteful.
- Lazy: Figure 5.2 shows how the EOL execution engine would interact with the graph database using the lazy strategy. Using this approach, in line 1, only skeletons of *ManualTask* elements would be initially fetched from the database by *id*s. Then, in line 2, for each *ManualTask*, the program would need to go back to the database and fetch the value of its *title* attribute. It is the same for retrieving the *efforts* reference of each *ManualTask*. So, multiple round-trips to the repository to fetch the values of the attribute or references of each model element are required, which can be time-consuming.

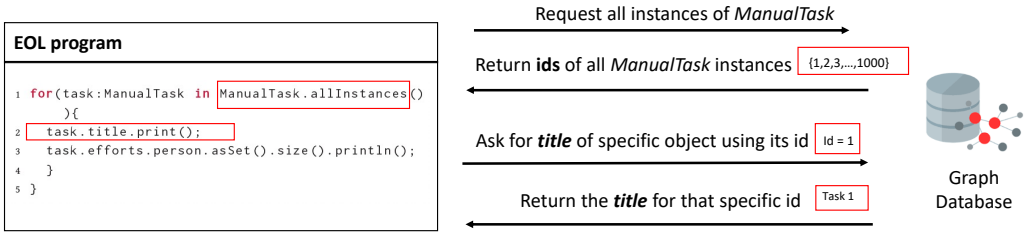


Figure 5.2: Lazy loading interaction with database

5.2 Architecture

The overall goal of this approach is to reduce the loading time and memory footprint of repository-based models consumed by model management programs through static analysis of said programs. In our approach, a static analyser can determine which features or all instances of which specific types (e.g., *ManualTask*) are likely[11] to be accessed by the program in advance.

This information can be used to fetch a subset of the model from the database in one go (e.g., populate the *efforts* reference of each *ManualTask* in one go, but leave out

[11] In some cases, the execution engine needs to load unnecessary model properties as well because of a lack of information before the execution (see Section 4.6.2).

the *start* attribute which is not required). Our expectation is that this approach will be more efficient in terms of memory and time compared to the greedy and lazy strategies described in the Background chapter.

In terms of concrete technologies, we use Neo4J as a graph-based model repository[12] and model management programs written in languages of the Epsilon platform—but the approach is also applicable to other similar technologies (e.g., OrientDB[13] and OCL [41], ATL [27] or Acceleo[14]).

Neo4j has been selected as the backend for this approach due to its widespread popularity and its integration into state-of-the-art solutions like NeoEMF and CDO. Additionally, Neo4j's features, including scalability, performance, and the graph-based structure that closely mirrors the structure found in models, make it a suitable choice for storing and retrieving models efficiently.

Also, there are two primary reasons for choosing Epsilon. First, Epsilon is an open-source project, making it accessible and conducive to collaboration. We have the capability to construct a static analyser that integrates effectively with our approach, ensuring its efficacy. Second, Epsilon's inherent extendibility is advantageous. By initially implementing our approach for the Epsilon Object Language (EOL), we establish a foundation that readily facilitates future extensions to languages like EVL, ETL and other task-specific languages.

A high-level overview of our approach is presented in Figure 5.3. The main components of this approach are represented in grey colour, and they are labelled with numbers 1 to 3.

5.2.1 Static Analysis

In the first step of our approach (see Figure 5.3), a model management program and the metamodels of the models it consumes are provided as input to a static analyser. The static analyser computes the abstract syntax tree of the program. Then, resolution algorithms, including variable resolution and type resolution, are applied to derive a type-resolved syntax tree [50]. Using the type-resolved syntax tree, the static analyser can extract relevant information (i.e., types and properties accessed by the program) as explained in Chapter 4.

The output of the static analyser is an *effective metamodel* for every model accessed by the program. The effective metamodel is a subset of the model's original metamodel, which consists only of types and properties that are likely to be accessed by the

[12] https://neo4j.com
[13] https://orientdb.org/
[14] https://www.eclipse.org/acceleo/

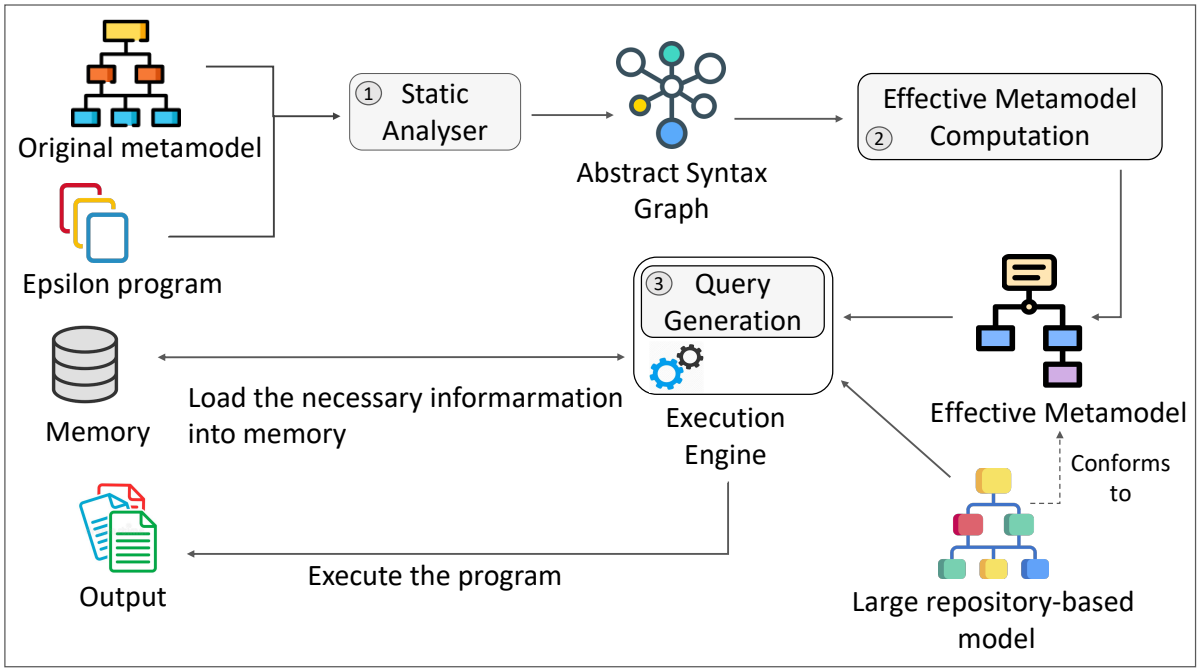


Figure 5.3: The proposed approach

program [50] (see Section 5.2.2).

While static analysis supports multiple models (and therefore produces multiple effective metamodels), in the remainder of the chapter, we will only consider programs operating on one model (and, therefore, one effective metamodel). To illustrate how every step of the approach works, we use the motivating example from Section 5.1.

In the first step of our approach, the static analyser sets the resolved types of expressions to types from the respective metamodels or to primitive types (e.g., String, Integer). For example, in line 1 of Listing 5.1, the resolved type of *task* variable is equal to *ManualTask*. In line 2, the property call is supposed to print the *title* of each *task*. The *title* is an attribute of *task*, and the resolved type is *String*. Then, line 3 accesses *task.efforts.person* and *task.efforts* is a property call where the target of this call is a model element (*ManualTask*) and the feature which is called is *efforts*. In this case, *efforts* is a reference of *ManualTask* of *Effort* type. Table 5.1 shows the resolved types of expressions which are extracted from Listing 5.1 by the static analyser.

5.2.2 Effective Metamodel Computation

The second step of the approach is the extraction of the effective metamodel of the model consumed by the program, which is based on the program's type-resolved abstract syntax tree. In Chapter 4, we introduced the concept of the effective metamodel and presented the algorithm used to extract it. The effective metamodel contains only

Table 5.1: Resolved Types Calculated by the Static Analyser

Line number in Listing 5.1	Expression	Resolved Type
1	task	Model Element (ManualTask)
1	ManualTask.allInstances()	Operation call expression (Sequence<ManualTask>)
2	task.title	Property call expression (String)
2	task	Model Element (ManualTask)
3	task.efforts.person.size()	Operation call expression (Integer)
3	task.efforts.person	Property call expression (Sequence<Person>)
3	task.efforts	Property call expression (Sequence<Effort>)
3	task	Model Element (ManualTask)

attributes and references that are necessary for executing the program from which it is extracted.

Figure 5.4 illustrates the effective metamodel extracted from the EOL program in our motivating example (Listing 5.1).

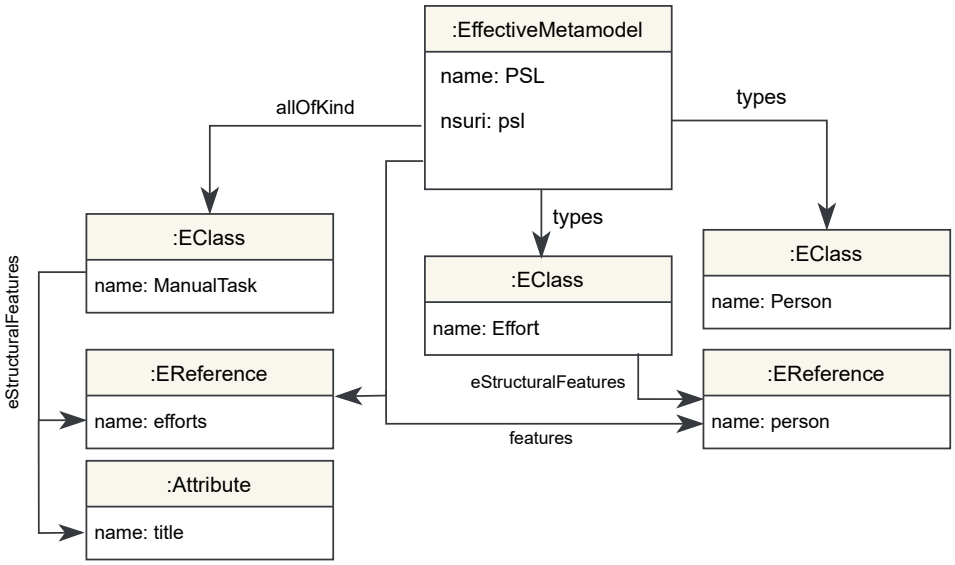


Figure 5.4: Effective metamodel of the EOL program shown in Listing 5.1

In Figure 5.4, the attributes of the *EffectiveMetamodel* class are filled by the original metamodel, which is the *name* and the *nsuri* of the metamodel. For running the EOL program, all instances of *ManualTask* must be loaded. The *ManualTask* class is added to the *EffectiveMetamodel* under the *allOfKind* reference. The *title* attribute of *task* is also added to the *EffectiveMetamodel*.

The *efforts* reference of *ManualTask* is also required (line 3 of Listing 5.1), hence, it is added to *ManualTask* as an *EReference*. The resolved type of the *efforts* reference is

equal to *Effort* (see Table 5.1), so the *Effort EClass* is added to effective metamodel using the *types* reference and the *Person EClass* is added to the effective metamodel using the *types* reference.

5.2.3 Query Generation

Mapping EMF Models to Graph Databases

In this work, we are concerned with the efficient management of models that reside in graph-based model repositories as these have been shown to outperform model repositories backed by relational/document databases [6, 14]. A state-of-the-art graph-based model repository is NeoEMF, which enables the persistence of EMF-based models in Neo4J graph databases (among others). Our first attempt was to implement our efficient model loading approach on top of NeoEMF; however, the framework cannot support partial model element loading without a significant amount of refactoring. Therefore, we chose to implement a custom mapping of EMF models to Neo4J databases, which we discuss in this section.

Figure 5.5 shows a model which conforms to the PSL metamodel in Figure 5.1. The *project* named as "ACME" consists of three *task*s: "Design" is a *ManualTask* which has to be completed by Bob (40% *effort*) and Alice (60% *effort*); "Implementation" is also a *ManualTask* equally split among Alice and Bob (50% each) and "Meeting organisation", which is an *Automated Task*.

In order to map EMF-based models such as this one to a Neo4J graph, we use the mapping strategy illustrated in Figure 5.6. Using this strategy, every model element is mapped to a *Node* in the graph, and the properties of the node are set to the values of the attributes of the element. For example, in Figure 5.6, *ACME* is an instance of *Project* in the model, which has a *title* attribute. Hence, a corresponding node is created in the graph, and the *title* property of the node is set to *ACME*. For *Design*, which is an instance of *ManualTask*, its *duration* and *start* attributes are copied to the respective node.

In Neo4j graphs, nodes can have labels. In our approach, the label of each node is set to the type of the respective model element and its super types. Thus, in Figure 5.6, the label of *ACME* is set to *Project* and the labels of *Design* are set to *Task* and *ManualTask*. As the labels of nodes are unordered, to understand which label corresponds to the exact type of the node, we can load all labels of each node and, by using the structure of the metamodel, find the exact type of node. However, it is more efficient to connect each node to another node that has the name of its type, using an *instanceOf* edge to capture the type of the node. In Figure 5.6, there is an *Edge* in

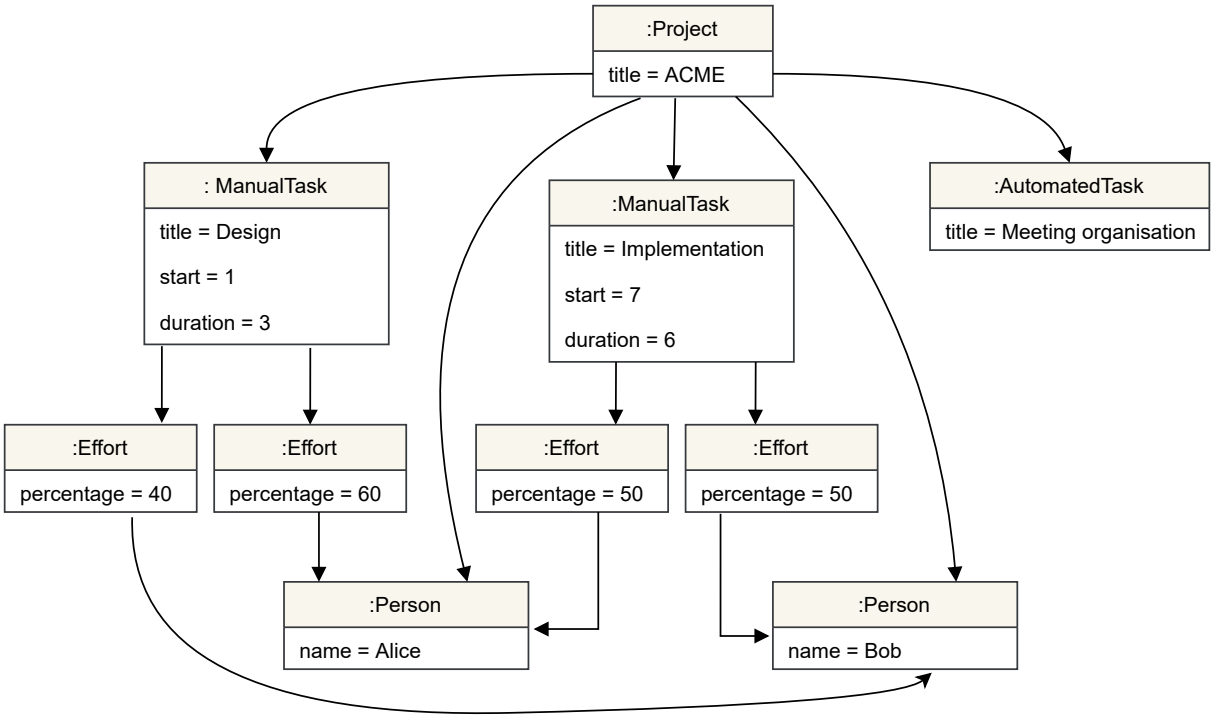


Figure 5.5: PSL model of the ACME project

the graph which connects the created node to the *Project* node to capture the type of the node and an edge which connects the *Design* node to *ManualTask*. Algorithm 5.1 describes the mapping process in detail.

Model Loading

In the third step of our approach, we wish to generate queries in Neo4J's Cypher query language that will fetch (1) only the part of the model that conforms to the effective metamodel extracted in step 1 and (2) do this as efficiently as possible (after statically analysing the program and identifying the part of the model it is likely to access at runtime in the form of an effective metamodel).

Cypher[15] is a language for querying Neo4j databases. Hence, by generating Cypher queries based on the effective metamodel, EOL's execution engine will be able to load the required parts of the model for running the program.

In Section 2.5.2, we discussed the subject of Cypher expressions. To facilitate a clear understanding of these concepts, the expressions used for loading information in our approach have been listed in Table 5.2 (with *var* representing variables). This table serves as a reminder and reference for the discussed concepts.

The first column of Table 5.2 lists the Cypher expressions, the second column is

[15]https://neo4j.com/developer/cypher/

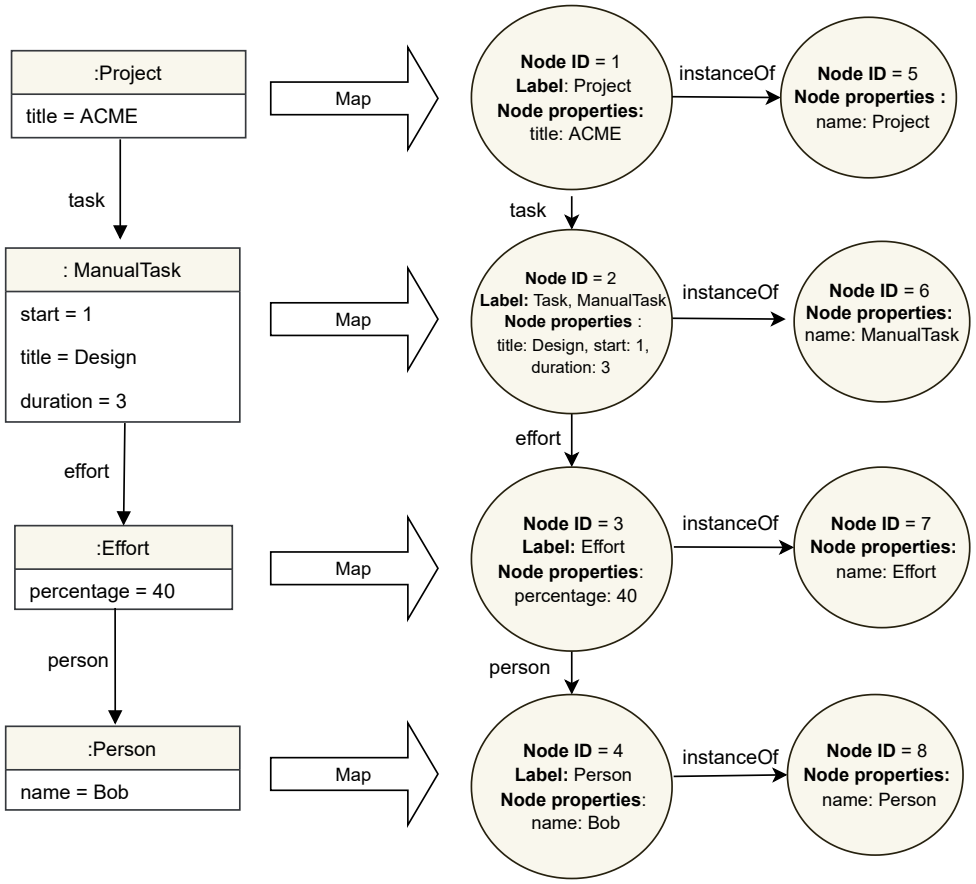


Figure 5.6: Mapping the model of Figure 5.5 into a Neo4J graph

Table 5.2: Cypher Expressions

Cypher expression	Pattern	Functionality
MATCH	(var: label of node)	loading all nodes with the same label
OPTIONAL MATCH	(var:source node)- [var:edge]->(var:target node)	return the target node of matched edge
RETURN	node.property	return the specific property(ies) of the matched nodes

the pattern that each expression follows, and the third column is the functionality of the expression. Using these three expressions, the queries of Listing 5.2 are generated based on the effective metamodel in Figure 5.4.

Considering the effective metamodel in Figure 5.4, all instances of *ManualTask* are required for running the program (EOL code in Listing 5.1). Hence, all nodes with the *ManualTask* label should be loaded. The *MATCH* keyword matches all nodes with the specified label in the graph. The generated query in line 1 of Listing 5.2 loads all *ManualTask* nodes.

Listing 5.2: Generated Cypher Queries According to Effective Metamodel in Figure 5.4

Algorithm 5.1 EMF Model to Graph Conversion Algorithm

```
1: let visitedElements = keep the EMF model elements with corresponding nodes
2: let source = Node, target = Node, newNode = Node
3: for all model elements of EMF model do
4:     let ME = model element
5:     if  visitedElements.contains(a node corresponding to ME) then
6:         source ←node
7:     else newNode = CreateNode(ME)
8:         setNodeProperties(newNode,ME.attributes)
9:         newNode.labels.add(ME.type.name)
10: for all supertypes of ME.type do
11:     newNode.labels.add(supertype.name)
12:     visitedElements.add(newNode)
13:     source ←newNode
14: for all ME.references do
15:     if visitedEl.contains(reference.value) then
16:         target ←visitedElements.get(reference.value)
17:     else newNode = CreateNode(reference.value)
18:         setNodeProperties(newNode,reference.value.attributes)
19:     visitedElements.add(newNode.labels)
20:     target ←newNode
21:     Edge e = CreateEdge(source, target)
22:     e.name = reference.name
```

```
1 MATCH (task:ManualTask)
2 RETURN ID(task), task.title
3 OPTIONAL MATCH (task:ManualTask)-[taskins:instanceOf]->(taskType)
4 RETURN taskType.name
5 OPTIONAL MATCH (task:ManualTask)-[effortRefTask:effort]->(effort:Effort),(effort)-[effortins:
      instanceOf]->(effortType)
6 RETURN ID(effort), effortType.name
7 OPTIONAL MATCH (effort:Effort)-[personRefEffort:person]->(person:Person),(person)-[personins:
      instanceOf]->(personType)
8 RETURN ID(person), personType.name
```

After matching all *ManualTask* nodes, the *id* of each *task* has to be loaded in order to distinguish between different *ManualTask* instances. Line 2 shows the *Return* query to fetch the *id* and the *title* attribute of *ManualTask* nodes from the database.

Beyond the *ID*, the exact type of *ManualTask* instances is also needed, which is specified by the *"instanceOf"* edge. In the case of references, the *MATCH* and *OPTIONAL MATCH* expressions are used to match the edges of the source node and return the respective target nodes. Using *MATCH* is a strict condition, and if there are no matches in the database, the query will not return any results. With *OPTIONAL MATCH* on the other hand, if there is no match, the query will be run and "null" will be

returned as a value of the edge that is not matched. Hence, *OPTIONAL MATCH* is a better fit for matching the references of each node. The query for requesting the type of *ManualTask* nodes is shown in line 3, and the name property of *taskType* is returned using the *RETURN* expression in line 4.

One reference of *ManualTask* instances is required according to the effective metamodel. The *efforts* reference is an edge between *ManualTask* and *Effort* nodes, and the query in line 5 matches this reference. In lines 6-7, the *instanceOf* reference is matched to find the type of the *Effort* node. According to the effective metamodel, attributes of *Effort* are not required, so only the *id*s and *type*s of *Effort* nodes are returned in line 6.

The *person* reference and the *instanceOf* reference of *Effort* are matched in line 7, and the query that returns the *id* and *type* of *Person* is shown in line 8.

Query Optimisation

The goal of our approach is to reduce the number of database hits and load as much information as possible in each access to the database to minimise the overall execution time. The Neo4j documentation[16] offers some recommendations to reduce the execution time of Cypher queries. We have applied some of them to optimise the automated query generation in our approach.

- **Using Labels**: Nodes are labelled by the type and super types of their corresponding model element. These labels are then used by generated queries to efficiently match nodes of different types.

- **Avoid Cartesian Products**: If two different node labels without any relationships between them are matched in a Cypher query, it is considered as a *disconnected pattern*. Generating a query for disconnected patterns will build Cartesian products between two node types, which would not be efficient as it will return a number of records that are not necessary for executing the program. For example, considering the PSL metamodel (Figure 5.1), there is no relationship between nodes with *AutomatedTask* and *Person* labels. So, in Listing 5.3, Neo4j matches each *AutomatedTask* node with all *Person* nodes. Suppose that the number of nodes with *AutomatedTask* label is equal to *m* and the number of nodes with *Person* label is equal to *n*. The number of returned records from the database will be equal to *n*m*. However, if Listing 5.3 is separated into two *MATCH* clauses, then the number of records will be *(n+m)*, which is more efficient.

[16] https://neo4j.com/blog/tuning-cypher-queries/

Listing 5.3: Query Generating Cartesian Products

```
1 MATCH (autoTask:AutomatedTask), (p: Person)
2 RETURN autoTask.title, p.name
```

Thus, we consider a trade-off between generating fewer queries and avoiding Cartesian products. In our approach, *MATCH* clauses are generated for *allOfKind* types in the effective metamodel. This is efficient as in each MATCH clause, the label of each node will be matched and then all required properties and references of a node in the effective metamodel will be returned. Considering the generated queries in Listing 5.2, there are three *MATCH* clauses that are related so they can be combined in one query for efficiency. The combined query is shown in Listing 5.4.

Listing 5.4: Optimised queries of Listing 5.4

```
1 MATCH (task:ManualTask)
2 OPTIONAL MATCH (task)-[taskins:instanceOf]->(taskType),(task)-[effortRefTask:effort]->(
      effort:Effort),(effort)-[effortins:instanceOf]->(effortType),(effort)-[
      personRefEffort:person]->(person:Person), (person)-[personins:instanceOf]->(
      personType)
3 RETURN ID(task), task.title, taskType.name, ID(effort), effortType.name,ID(person),
      personType.name
```

- **Reduce Cardinality**: Since nodes are labelled by the type of their respective model element and its super types, the results of some queries can overlap. For example, in Listing 5.5, all nodes with *Task* and *ManualTask* labels are matched, and the *title*s of matched nodes are returned. Considering the part of the graph in Figure 5.7, the nodes returned in line 2 are “Design”, “Implementation” and “Meeting organisation” nodes since they are labelled as *Task*. The titles of the nodes that are returned in line 4 are “Design” and “Implementation” nodes which are *ManualTask*-labelled nodes. The “Design” and “Implementation” nodes are returned twice (in lines 2 and 4). This redundancy is because of querying two classes (*Task* and *ManualTask*) that have an inheritance relationship. Therefore, it is more efficient to execute only the first query in lines 1-2, which covers all nodes that are loaded by both MATCH clauses.

Listing 5.5: Queries with Overlap

```
1 MATCH (task:Task)
2 RETURN task.title
3 MATCH (manualTask:ManualTask)
4 RETURN manualTask.title
```

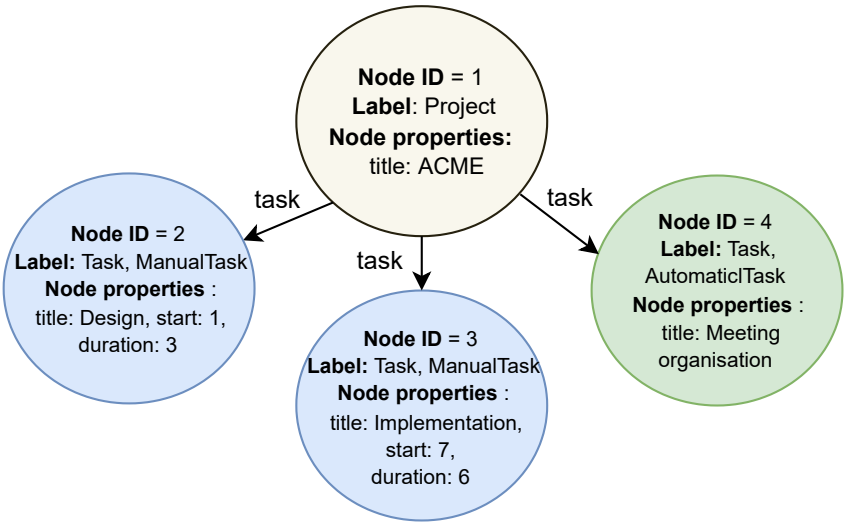


Figure 5.7: A part of graph model

Algorithm 5.2 generates Cypher queries automatically based on the effective metamodel.

Algorithm 5.2 Cypher Queries Generation (1 of 2)

```
1:  let EM = calculated effective metamodel
2:  for all EClass in EM.allOfKind do
3:      let matchString = "";
4:      let visitedClasses = Set of EClasses
5:      let optionalMatch = Set<String>
6:      let return = Set<String>
7:      matchString ←EClass.name
8:      return.add(EClass.id)
9:      return.add(EClass.getReference(instanceOf).name)
10:     Call HANDLEFEATURES (EClass)
11:     let query =""
12:     query ← "MATCH"+ query
13:     for all item in match array do
14:         query ←item + ","
15:     query ← "OPTIONALMATCH"+ query
16:     for all item in optionalMatch array do
17:         query ←item + ","
18:     query ← "RETURN"+ query
19:     for all item in return array do
20:         query ←query + item + ","
```

There are three collections to record MATCH clauses, OPTIONAL MATCH clauses and RETURN clauses in Algorithm 5.2. In lines 1-9, these three collections are filled, and in lines 10-19, the query is generated by combining these clauses together using

the appropriate Cypher expressions. One query for each EClass is generated where all instances of that class are required by the program.

In line 2, the classes that are under the *allOfKind* reference in the effective metamodel are considered by Algorithm 5.2. In line 7, the variable definition for the EClass is added to *matchString*, and the ID of the matched node is added to the *return* collection (lines 7-8). In line 9, the *HANDLEFEATURES* procedure is called to handle the features of the EClass. In Algorithm 5.3, which shows the *HANDLEFEATURES* procedure, each

Algorithm 5.3 Cypher Queries Generation (2 of 2)

```
1:  procedure HANDLEFEATURES(EClass)
2:      visitedClasses.add(EClass)
3:      for all features in EClass.eStructuralFeatures do
4:          if IsAttribute(feature) then
5:              return.add(feature)
6:          else if IsReference(feature) then
7:              let type = reference.type
8:              return.add(reference.target.id)
9:              optionalMatchString ← (EClass.name) − [reference.name] → (type)
10:             if not visitedClasses.contains(type) and not allOfKind.contains(type) then
11:                 handleFeatures(type)
```

attribute of the EClass is added to the *return* collection (in lines 4-5). To follow the references of nodes in the graph, an OPTIONAL MATCH pattern is needed. So, in lines 6-9, an OPTIONAL MATCH pattern is created and added to the *optionalMatch* collection. The OPTIONAL MATCH pattern matches edges that have the same name as the reference of the EClass and it connects the matched node to the target nodes (it is kept in the *type* variable).

As discussed in Section 5.2.3, it is more efficient to generate one query and load all relevant information in one go. So, in lines 10-11, the *HANDLEFEATURES* method is called on *type* to follow the features of the target node. Thus, Algorithm 5.3 uses this recursive method to load a sub-graph that consists of the nodes and their properties and edges.

This recursive call returns when a cycle is found or when there are no further references to follow. In line 2 of Algorithm 5.3, the *visitedClasses* is a collection to keep track of visited classes. So if a *type* is visited once, it will not be visited again according to the *if* condition in line 10 to avoid infinite loops.

After handling all features in Algorithm 5.3, the query is generated for the EClass, and then the process is repeated for the next EClass under the effective metamodel's *allOfKind* reference. The result returned from the database after executing the generated query is shown in Table 5.3.

Table 5.3: Result for the Execution of Generated Query in Listing 5.4

ID(task)	task.title	taskType.name	ID(effort)	effortType.name	ID(person)	personType.name
2	”Design”	”ManualTask”	10	”Effort”	9	”Person”
2	”Design”	”ManualTask”	12	”Effort”	7	”Person”
6	”Implementation”	”ManualTask”	13	”Effort”	7	”Person”
6	”Implementation”	”ManualTask”	14	”Effort”	9	”Person”

After retrieving information from the database, this information is used by the execution engine to run the EOL program.

5.3 Implementation

In our initial efforts, we aimed to develop an efficient model-loading approach using NeoEMF as the underlying framework. However, we faced a notable challenge: the framework itself does not support the selective loading of specific element properties from a model. This limitation would have required substantial refactoring efforts to overcome.

Therefore, we implemented our approach by connecting directly to the Neo4J database and using cypher queries to retrieve data from the database. Our hypothesis was that loading the **required** attributes and references in one go (partial loading) works better than loading **all** attributes and references in one go (greedy) because of loading less information. This hypothesis also performs better than the lazy approach, which only loads the necessary attributes and references with multiple round trips to the database (because of fewer trips to the database). So, to evaluate this hypothesis, we ran experiments using the Twitter dataset[17], which is available as a large standard dataset on the Neo4j website. In the experiment, we tried to load *name* and *screen_name* of all Twitter users using three loading approaches.

The first query, as shown in Listing 5.6, loads all Twitter Users from the database. We required only *name* and *screen_name*, but the greedy approach works regardless of which attribute or references are required. Therefore, all users with all information will be returned.

Listing 5.6: Retrieving data using greedy approach

```
1 "MATCH (u: User) RETURN u"
```

[17] https://github.com/neo4j-graph-examples/twitter-v2

In the second query (Listing 5.7), the *name* and *screen_name* attributes are loaded using lazy loading. First, all users are returned by their *id*. Then, for each object, a query is run to get the specific attribute of the object.

Listing 5.7: Retrieving data using lazy approach

```
1 "MATCH (u: User) RETURN u.identity As id";
2 For every object {
3 "MATCH (u: User) where u.identity =" + object.id + "RETURN a.name AS name");
4 "MATCH (u: User) where u.identity =" + object.id + RETURN screen_name AS scname");
5 }
```

The third query is using the partial approach. This query matches all users and returns the two necessary attributes that are required in only one query.

Listing 5.8: Retrieving data using partial approach

```
1 "MATCH (u: User) RETURN a.name AS name, screen_name AS scname");
2 get("name");
3 get("scname");
```

The results depicted in Figure 5.8 validate the time-saving benefits of partial loading. This advantage stems from a reduction in database queries and the amount of information being loaded. Consequently, an effective approach involves **minimizing query numbers** while **maximizing the data loaded per transaction**. The positive outcomes of this experiment provide strong support for our initial hypothesis, reinforcing our decision to adopt this strategy when working with Cypher and Neo4j.

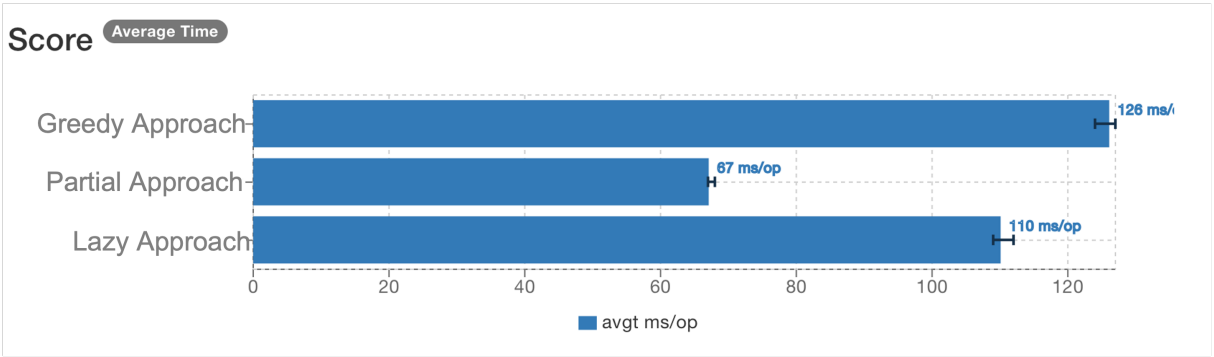


Figure 5.8: Testing different approaches on Neo4J

5.3.1 Epsilon Implementation

In this approach, in terms of technical work, we consider a new type of model in Epsilon as *Neo4j* model which *extends* the EMF model class in Epsilon and *overrides* the *load()* function. Figure 5.9 illustrates the sequence of activities in *load()* function.

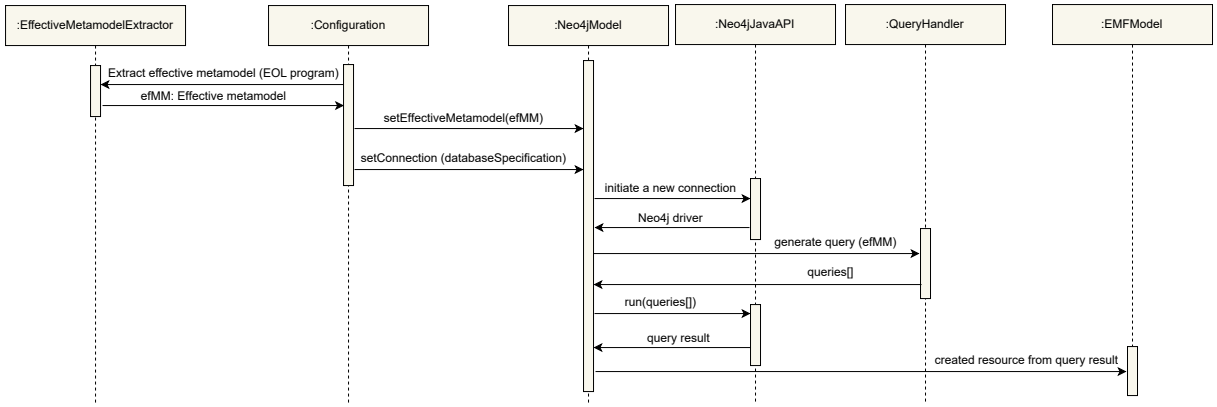


Figure 5.9: Sequence diagram of the implementation of load() function in Neo4j model

In the first step, the *configuration* class sends the EOL module to *effectiveMetamodelExtractor* to extract the effective metamodel and send it back to the configuration. After that, the effective metamodel and database specification (includes information such as the name of a database or database path), which is set by the user, is set for *Neo4jModel*. We use the Neo4j Java API in order to connect to the Neo4j database and retrieve the required data from the graph. Hence, the *Neo4jModel* class sends a request to the API for creating a new connection. The Neo4j API returned a driver as a result, which confirms that the connection is initiated. Then the effective metamodel, which is set in the Neo4J model from the configuration, is sent to *QueryHandler* to generate queries based on Algorithm 5.2 and Algorithm 5.3. The generated queries are returned to *Neo4jModel*. *Neo4jModel* sends the queries to API, which is capable of running them and sending the result to *Neo4jModel* in the forms of *record*s. In the last step, *Neo4jModel* create *EObject*s from *record*s to make it compatible with EMF. *Neo4jModel* creates a resource using the created *EObjects*, and the program will be run using the EMF resource.

5.3.2 Limitations

There are two noteworthy limitations in our proposed approach. First, our approach is limited to read-only input models of model management programs. Changing models (updating, deleting or adding model elements) is not supported in our approach. Also, our prototype implementation does not attempt dead code elimination, which means that the extracted effective metamodel can contain types and features that may never be accessed at runtime.

5.4 Evaluation

In this section, we report on the results of experiments that measure the performance of our approach against that of NeoEMF. We have chosen NeoEMF because it outperforms other repositories [14]. NeoEMF follows the lazy loading strategy.

We evaluated our approach on a system using Java VM 14.0.1 with Intel(R) Core(TM) i7, 16 GB memory and CPU @ 2.80 GHz running Mac OS X Catalina.

For our experiments, we have used the models proposed in the GraBaTs 2009 contest [45]. The models conform to an Ecore-based metamodel of the Java programming language and have been reverse-engineered from open-source Java projects. There are five XMI models, from Set0 to Set4, each one larger than its predecessor (from a 8.8 MB XMI file with 70 447 model elements representing 14 Java classes to a 646 MB file with 4 961 779 model elements representing 5984 Java classes). We produced Neo4j graphs from the Grabats XMI files and saved them in Neo4j (version 4.4.3) embedded databases.

For our experiments, we ran the EOL programs against Set0-Set4 graphs in the database using our approach and NeoEMF. During these experiments, we measured both the execution time and memory consumption. It is important to note that the execution time considered in our measurements includes not only the time it takes for the program to execute but also the loading time. Therefore, the total execution time comprises the loading time and the time taken for the program to execute.

Two test cases were considered to evaluate our approach compared to NeoEMF using the GraBaTs dataset. The first test case consists of five programs that utilize 20%, 40%, 60%, 80% and 100% of each model, as used in the work by Wei et al. [51]. The second test case involves a query known as the GraBaTs query. The results were computed after five warm-up iterations and represent the average over 10 executions of the program.

5.4.1 Experiment 1: Model percentage

We used five EOL programs for each model from [51], each utilizing specific percentages (20%, 40%, 60%, 80%, and 100%) of the GraBaTs models, and evaluated their performance. The results, depicted in Figure 5.10, consistently demonstrate the superiority of our approach over NeoEMF across all models, irrespective of the percentage of the model utilized by the program.

Notably, as illustrated by the slope of the line, our approach exhibits increasingly pronounced advantages as the size of the model grows, showcasing its exceptional efficiency in handling large models.

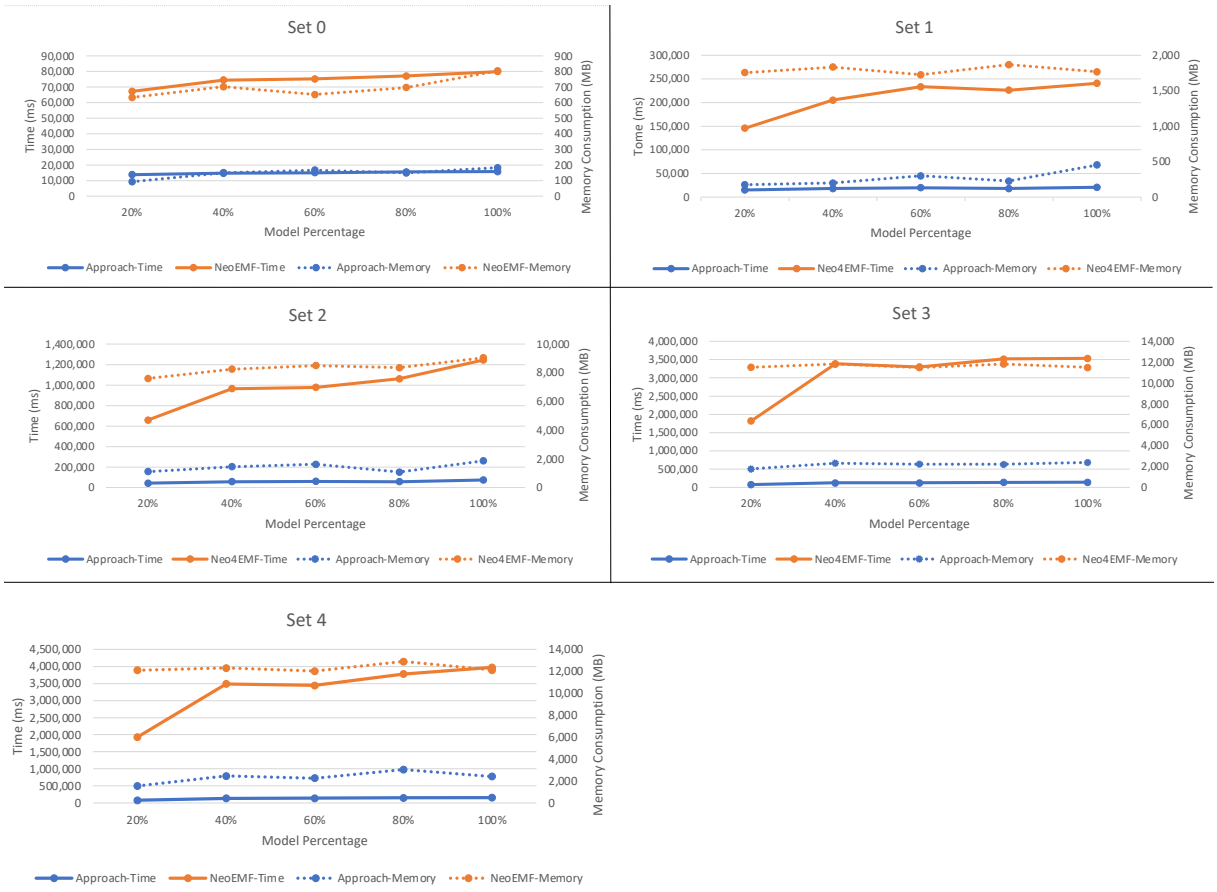


Figure 5.10: Performance comparison of our approach and NeoEMF

In Set0, we observed a substantial improvement in execution time, ranging between 74-75%, and a notable enhancement in memory consumption, with improvements ranging from 65-83%.

In Set 2, we observed a substantial improvement in execution time, ranging between approximately 93-94%, indicating the remarkable efficiency of our approach compared to the NeoEMF framework. Additionally, there was a notable improvement in memory consumption, with enhancements ranging from approximately 69-85%.

In Set 3, we observed a significant improvement in execution time, ranging between approximately 95-96%, underscoring the substantial efficiency gains achieved by our approach compared to NeoEMF. Furthermore, there was a notable enhancement in memory consumption, with improvements ranging from approximately 74-76%.

In Set 4, we observed a considerable improvement in execution time, ranging between approximately 95%, demonstrating the significant performance improvements realized by our approach over NeoEMF. Additionally, there was a notable enhancement in memory consumption, with improvements ranging from approximately 73-85%.

The most significant improvement was recorded for running the program using the

60% of Set 3, boasting a remarkable 96% improvement.

5.4.2 Experiment 2: Grabats query

We have also implemented a query in the Epsilon Object Language (EOL) inspired by one of the Grabats test cases[18]. This query, shown in Listing 5.9, finds all classes that declare public static methods whose return type is the containing class itself.

Listing 5.9: EOL implementation of Grabats query

```
model Core driver Neo4j {nsuri = "org.amma.dsl.jdt.core"};
model DOM driver Neo4j {nsuri = "org.amma.dsl.jdt.dom"};
model PrimitiveTypes driver Neo4j {nsuri = "org.amma.dsl.jdt.
    primitiveTypes"};
var matches:Set;
matches.addAll(
    TypeDeclaration.all.collect(td: TypeDeclaration|td.bodyDeclarations.
        select(
              md: MethodDeclaration|
              md.modifiers.exists(mod: Modifier|mod.public==true)
              and
              md.modifiers.exists(mod: Modifier|mod.static==true)
              and
              md.returnType.checkName(td)
          ))
    .flatten()
    .collect(names:MethodDeclaration|names.returnType.name.
        fullyQualifiedName)
);
matches.size().println();

operation Type checkName(td : TypeDeclaration) : Boolean{
if (self.isTypeOf(SimpleType)){
    var st : SimpleType;
    st = self;
    return (st.name.fullyQualifiedName == td.name.fullyQualifiedName);
}
else
   return false;
}
```

For our experiments, we ran the EOL program against Set0-Set4 graphs in the database using our approach and NeoEMF. During these experiments, we measured

[18] http://https://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=GraBaTs_2009_Case_Study

both the execution time and memory consumption. The results are shown in Table 5.4.

Table 5.4: Experiment Results

	Our Approach		NeoEMF	
Model	**Execution Time (s)**	**Execution Memory (MB)**	**Execution Time (s)**	**Execution Memory (MB)**
Set0 (70447 model elements)	8.1	108	5.7	319
Set1 (198466 model elements)	8.3	118	9.5	656.2
Set2 (2082841 model elements)	12.2	210	61.5	2537.5
Set3 (4852855 model elements)	37.3	232.8	114.6	3718
Set4 (4961779 model elements)	41.5	318.1	126.4	2987.2

For instance, for Set1, it takes $8.3\,\mathrm{s}$ and $118\,\mathrm{MB}$ of memory to run the Grabats query using our approach while for NeoEMF, the average time is equal to $9.5\,\mathrm{s}$ and the memory consumption is $656.2\,\mathrm{MB}$ which is about 5.5 times higher than our approach. The most significant difference in memory usage is in Set3, where the memory footprint of our approach is 93% lower than NeoEMF.

On average, using our approach, memory consumption is lower by 84%, and the execution time is lower by 37% compared to NeoEMF.

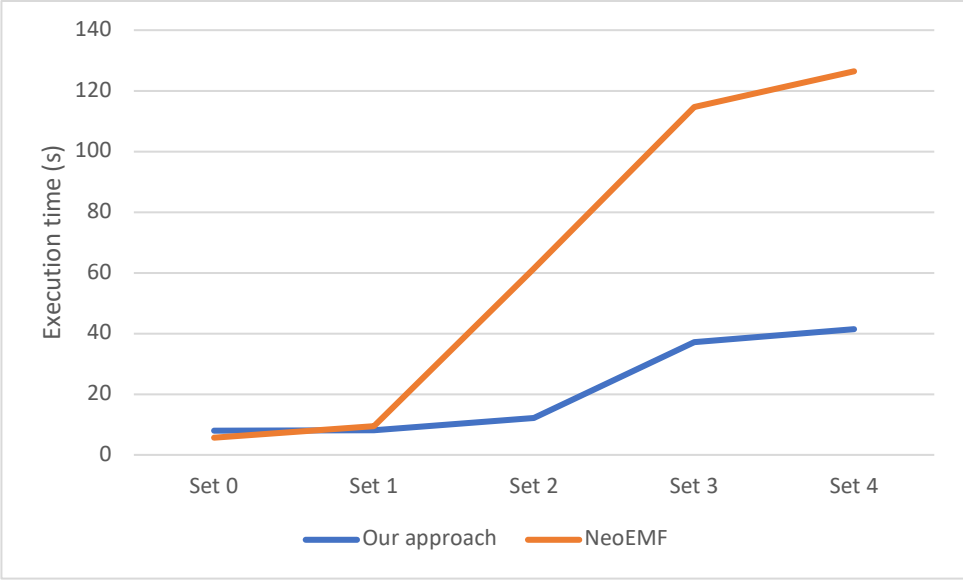


Figure 5.11: Execution time comparison of our approach and NeoEMF

The charts illustrated in Figures 5.11 and 5.12 show the linear behaviour of our approach and NeoEMF. In Figure 5.11, in the Set0 model, as the model is not very large, the overhead of effective metamodel extraction and loading data from the database in our approach is not compensated at runtime and the execution time is better for NeoEMF.

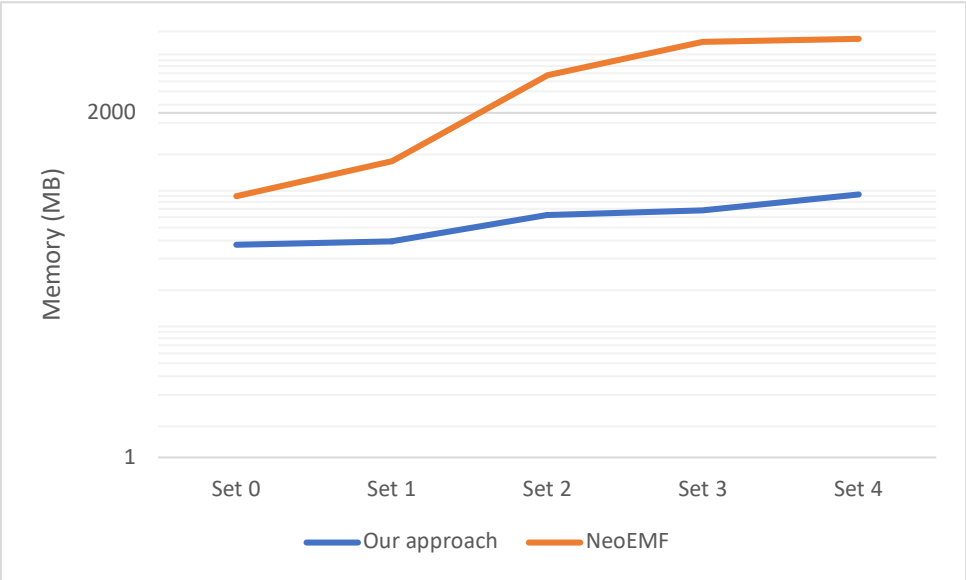


Figure 5.12: Memory comparison of our approach and NeoEMF (logarithmic scale)

As the size of the model grows, the slope of the diagram is greater in NeoEMF compared to our approach, which means our approach is more efficient in terms of execution time. The highest percentage of time-saving is 74% for Set2.

In Figure 5.12, both approaches have linear behaviour, and our approach consumes less memory compared to NeoEMF. From Set0 to Set4, as the size of the model increased, the memory consumption grew in two approaches.

Regarding correctness, we validated all models by executing each program with two approaches (our approach and NeoEMF) and verified that the output produced by all execution pairs is equivalent. Table 5.5 shows the generated output of Grabats query using our approach and NeoEMF. It is worth noting that the output remains consistent, even as we optimize memory and time usage, signifying that these optimizations do not alter the program's behaviour.

	Generated Output	
Model	**Our approach**	**NeoEMF**
Set0	1	1
Set1	2	2
Set2	38	38
Set3	150	150
Set4	159	159

Table 5.5: Generated output of Grabats query with different loading approaches

5.4.3 Discussion

In our approach, we pre-load all necessary information before executing the program, allowing the execution engine to run the program using the data already available in memory. In contrast, NeoEMF initially loads only object IDs before execution and retrieves the required data while the program is running.

Therefore, in our approach, loading time tends to be the more time-consuming component, but program execution is fast (due to the availability of all required information in the memory). This contrasts with NeoEMF's lazy loading approach, where the execution of the program is the time-consuming part, as the required data are fetched during program execution.

Figures 5.13 and 5.14 display 10 iterations of program execution using each model.

While the previous section discussed the total execution time, these figures provide a more detailed breakdown of loading and execution times.

In Set0, the blue line demonstrates the loading time in our approach, which, as expected, exceeds the execution time (depicted by the orange line). On the other hand, the yellow line that shows the execution time in NeoEMF is longer compared to the loading time (grey line). As the model is not very large, the overhead of effective metamodel extraction and loading data from the database in our approach is not compensated at runtime, resulting in better end-to-end execution time for NeoEMF.

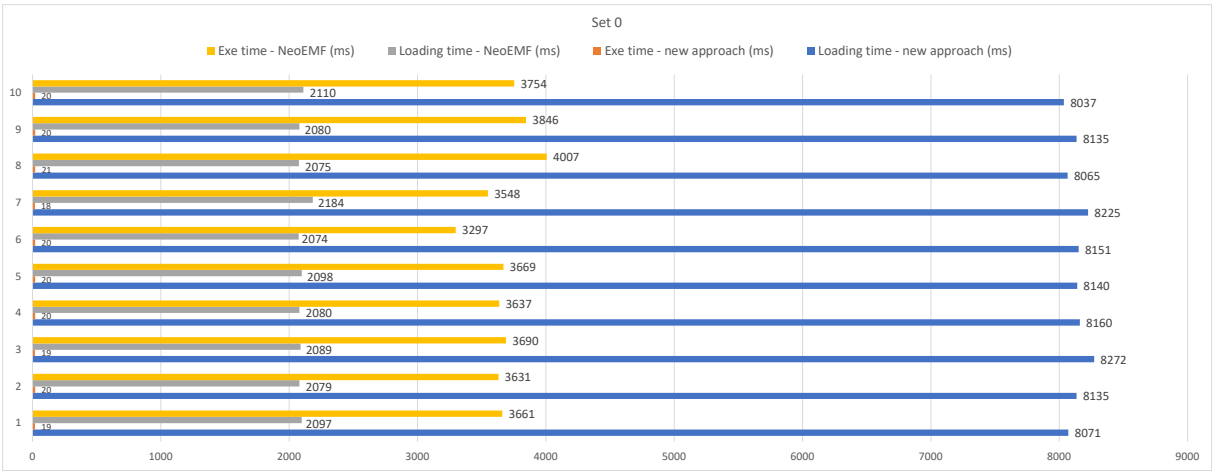
In Set1, as the model size expands, we observe an increase in loading time in our approach while the execution time remains relatively low. In contrast, NeoEMF experiences a situation where execution time surpasses loading time. Consequently, the overall execution time, which encompasses loading time and execution time, is higher for NeoEMF due to the longer total execution time.

For Set2, Set3 and Set4, the situation remains the same. With larger model sizes, the requirement for elements essential for program execution grows. Consequently, NeoEMF requires an increased number of trips to the database, contributing to longer execution times. In contrast, loading times in our approach also increase with the growing model size, but as depicted in the charts, the rate of increase in loading time for our approach is less steep compared to the execution time of NeoEMF.

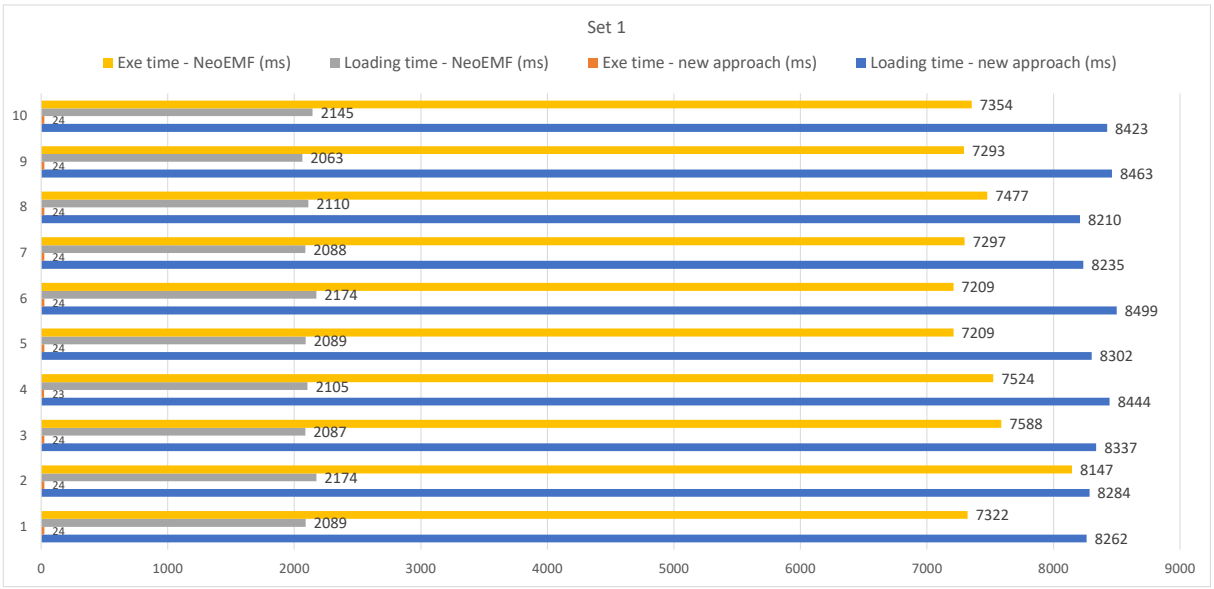
5.4.4 Threats to Validity

We limit construct validity threats by considering big models from the widely used GraBaTs 2009 contest [45] that conform to the Grabats metamodel. The results reported in this chapter consider these test cases.

We limit internal validity threats by reporting results after executing 5 warm-up

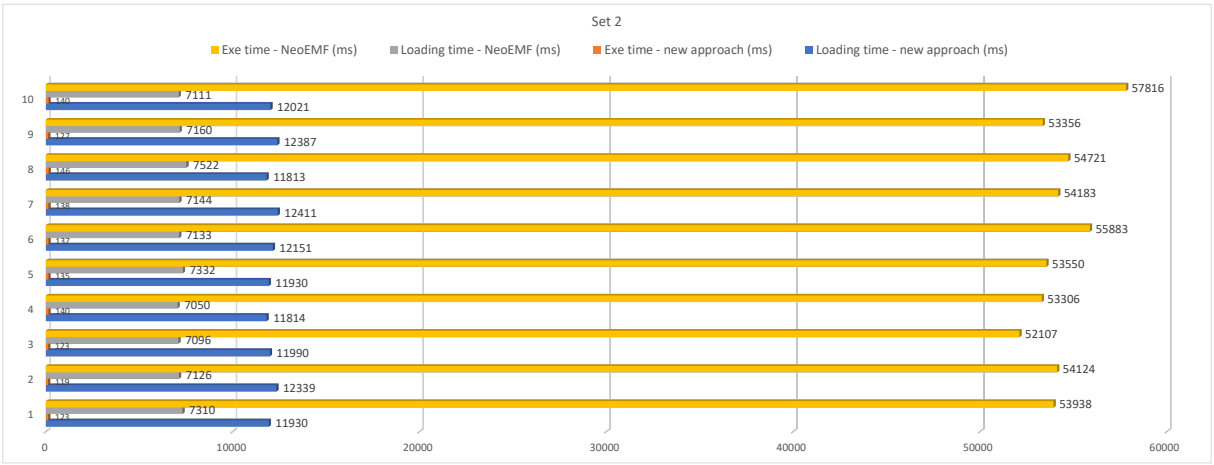


(a) Set 0

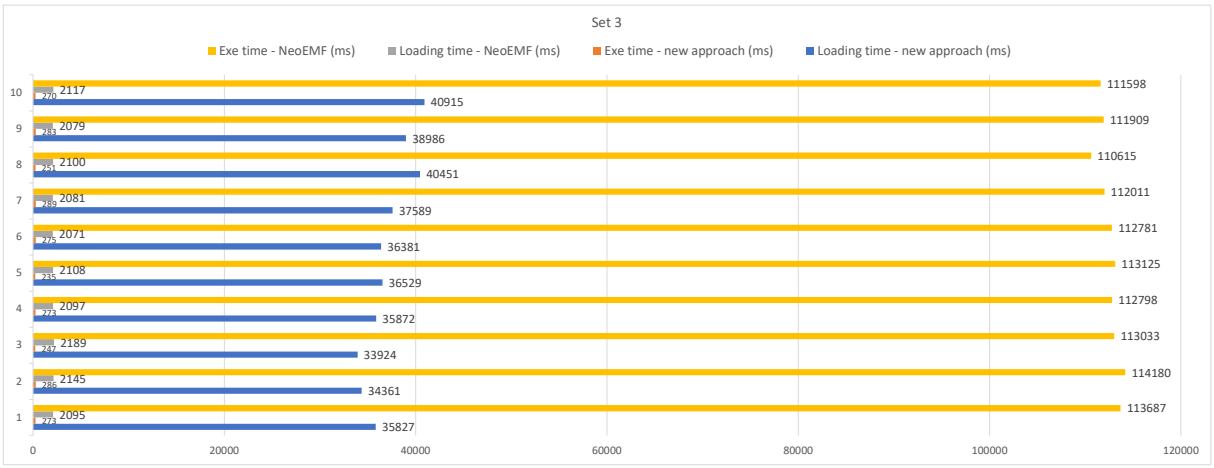


(b) Set 1

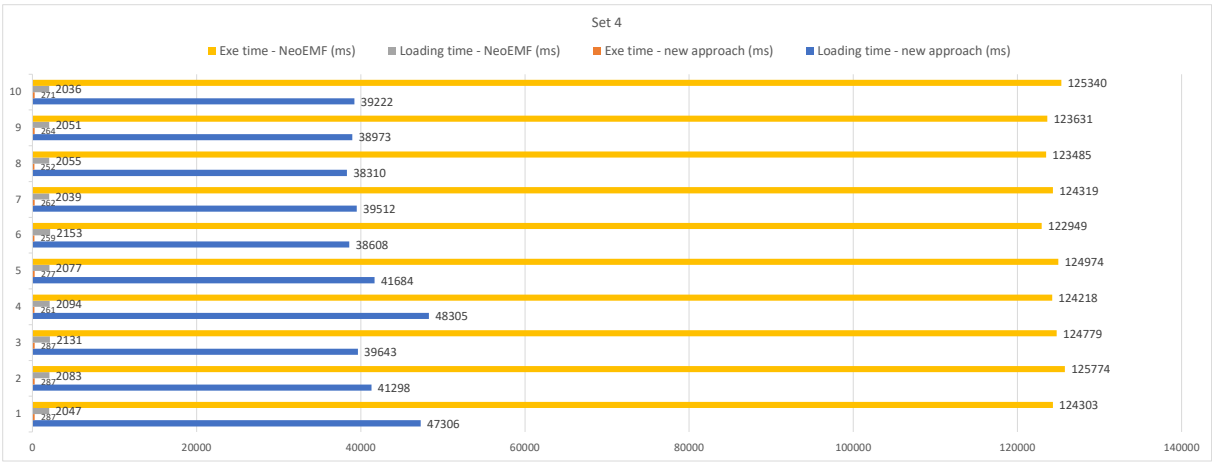
Figure 5.13: Ten iterations execution result



(a) Set 2



(b) Set 3



(c) Set 4

Figure 5.14: Ten iterations execution result

iterations of the program, thus reducing the potential impact on memory and execution time of starting and initialising the JVM.

We reduce external validity threats by building our approach atop mature and robust MDE technologies, including the Epsilon suite of model management programs and the Neo4j graph database. As discussed in Section 5.2, extending our approach to support other technologies is relatively straightforward with modest effort. However, more experiments are required to establish the applicability and scalability of our approach in domains and metamodels/models with characteristics different from those used in our experimental evaluation.

5.5 Chapter Summary

In this chapter, we introduced the first contribution of this thesis: a method for partially loading repository-based models. This method relies on insights gained from analyzing model management programs statically.

We began by presenting an example in Section 5.1 to illustrate the idea, followed by a detailed explanation of the method's architecture using algorithms. We also discussed how the method is implemented and acknowledged its limitations.

To validate the effectiveness of our approach, we conducted evaluation experiments using large models from the Grabats test suite. The results demonstrate that our approach can notably reduce the time and memory required to run model management programs on repository-based models. This efficiency is achieved when the program only interacts with a specific subset of the model's elements without altering the program's overall behaviour or its output. Consequently, our thesis hypothesis stands validated, affirming that partial loading significantly impacts the performance of the execution engine.

6 Partitioning of XMI Models

When dealing with large file-based models, such as those serialized in XMI format, issues related to scalability arise during the validation of these models. As explored and demonstrated in the Background chapter, the default XMI parser in EMF loads the entire model into memory, even parts of the model that remain unused by the program. This inefficient process results in wasted memory and can lead to execution failures when the model's size exceeds the available heap memory, rendering it impossible to run the program on a single machine.

Similar problems arise when attempting to execute constraints in parallel within a distributed setting. In this scenario, all constraints are spread across multiple machines, and each machine loads the entire model, regardless of which constraints they are assigned to execute. This results in inefficient model loading and validation within the execution engine, leading to longer model loading times and unnecessary memory usage on each machine.

We introduced a partial parser for XMI models in Section 2.5, enabling model management program execution engines to load only the necessary parts of the models, thus accommodating larger models. However, the same problem may still arise if the available heap memory is insufficient to allocate even the required model elements for validation. Thus, our hypothesis suggests that in addition to partial loading, the incorporation of partitioning functionality offers further enhancements. By dynamically loading and unloading data during execution, partitioning contributes to the overall improvement of the execution engine's performance.

In this chapter, we introduce an approach to enhance the performance of execution engines when handling the validation of large XMI models. By utilizing static analysis and prior knowledge about the program, our proposed approach divides the constraints into groups. The execution engine incorporates a partial XMI parser, allowing it to selectively load the relevant parts of the model expected to be accessed by each group. Moreover, the execution engine can discard model parts that are no longer referenced by the program, freeing up memory for the ongoing execution. Applying the proposed approach, execution engines of validation programs can handle large models more efficiently. The use of static analysis allows for better resource allocation, thus reducing unnecessary memory consumption for loading and validating models.

Section 6.1 provides an explanation of the challenges of interest through a motivating example. In Section 6.2, the proposed approach is presented and discussed in detail. The implementation of this approach is elaborated upon in Section 6.3, where we also acknowledge the limitations associated with our approach. Section 6.4 reports

on the results of the evaluation of this work, and finally, Section 6.5 summarises the contributions of this chapter.

6.1 Motivating Example

Figure 6.1 shows the metamodel of a simplified component-connector language, which is used as a motivating example in this chapter. The metamodel has a *Component* class which has a *name* and consists of *ports*, which is shown by a containment reference from *Component* to the *Port* class. Each *Port* has a *name* and a *type*. Also, it has a reference to its *Component*. There are two types of *Ports*, *InPort* and *OutPort* that are connected to each other by a *Connector*. The *source* of a *Connector* is always an *OutPort* of a *Component*, and the *target* of *Connector* is an *InPort*.

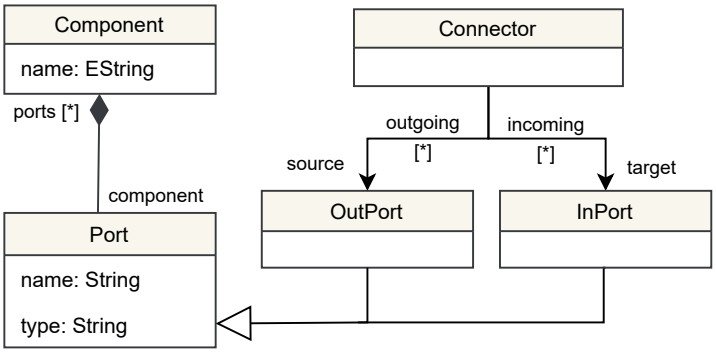


Figure 6.1: Component Language Metamodel

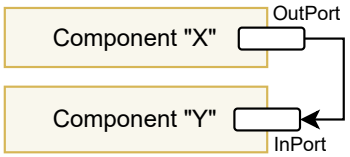


Figure 6.2: Sample model that conforms to the metamodel of Figure 6.1

Consider a model that conforms to the Component language (a Component diagram is shown in Figure 6.2), which we would like to validate with additional constraints. In this work, we have selected Epsilon's EVL language due to specific reasons we have addressed in Chapter 3. Nevertheless, it is worth noting that the proposed approach is applicable to various other types of validation languages as well. Listing 6.1 contains the constraints expressed in EVL.

Listing 6.1: EVL constraints to validate Component Language instances

```
1 context Component {
```

```
2   constraint hasValidName {
3     check: self.name = self.name.ftuc()
4     message: self.name + " should start with an upper-case letter"
5   }
6   constraint hasUniqueName {
7     check: Component.all.select
8       (c:Component|c.name = self.name).size() == 1
9     message: "Duplicate component name" + self.name
10  }
11 }
12 context Connector {
13  constraint portTypesMatch {
14    check:self.source.type=self.target.type
15    message:"The types of the source and target ports don't match"
16  }
17 }
```

The first constraint (*hasValidName*) checks that the *name* of each *Component* should start with an upper case letter (using EVL's *ftuc()*[19] method), while the second one (*hasUniqueName*) makes sure that *Component* names are globally unique. The third constraint (*portTypesMatch*) checks that the *type*s of two *Port*s connected by a *Connector* are the same.

According to Listing 6.1, the information that is required for executing the *hasValidName* constraint is the *name* attribute of all instances of *Component* in the model. The *hasUniqueName* constraint also needs the same information. For running the *PortTypeMatch* constraint, the *source* and *target* references of all instances of *Connector* are of interest and the *type* attribute of *Port*, too. It is worth noting that some parts of the model, such as the *name* attribute or the *component* reference of *Port*, are not accessed by any of these constraints.

To run these constraints, if the execution engine uses EMF's default XMI parser, the whole model will be loaded into memory (see Section 2.5). Hence, the memory will be occupied by model elements or features not required for executing the program (such as the *name* attribute or the *component* reference of *Ports*). Also, all loaded elements will be kept in memory until the end of execution, which is not optimal. For example, after running the second constraint (*hasUniqueName*), instances of the *Component* class are no longer needed by the program; there is no need to keep these model elements and their features in memory while executing the third constraint.

To improve efficiency, it is possible to identify groups of constraints that access the same sub-sets of the model. For instance, the *hasValidName* and *hasUniqueName*

[19] first to upper case

constraints in the example require the same information (the *name*s of all *Component*s in the model). In our approach, by recognising this, the execution engine can execute these constraints together without the need for additional loading and unloading overhead. The necessary information, such as the *name* attribute of *Component*, can be loaded into memory, the constraints can be executed, and the information can be unloaded once it is no longer needed for the *PortTypeMatch* constraint. By selectively loading and unloading the relevant information, the execution engine can reduce the loading time and overall execution time.

The current execution engine of EVL programs is not aware of features of required model elements that have to be fetched from the XMI model and loaded into memory. Also, there is no hint for the execution engine to dispose of the parts of models from memory that are no longer required by the program. Hence, a static analyser becomes essential to analyse the EVL program in order to obtain advanced knowledge about the program. This knowledge allows the execution engine to selectively load relevant information and group the constraints based on this acquired understanding.

6.2 Architecture

The memory management and loading time when working with large XMI models can be improved through the implementation of two key features in the EVL execution engine: partial loading and partitioning. In our approach, we combine these capabilities to enable the execution engine to load only specific subsets of the model that are required for executing the program, resulting in a reduced memory footprint. Additionally, splitting constraints into sub-groups (which is referred to as partitioning the constraints in this chapter) allows for selective loading and unloading subsets of the model based on the constraints being executed, further improving the overall efficiency of the program. Together, these features can facilitate more efficient memory utilization and improved performance in running the program.

In terms of concrete technologies, we use XMI as a file-based model format and programs written in the EVL language of the Epsilon platform, but the approach can be extended to other languages of Epsilon, and it is also applicable to other similar technologies (e.g. OCL). A high-level overview of our approach is presented in Figure 6.3. The main components of this approach are represented in grey and labelled with numbers 1 to 3. The two first steps are the same as the partial loading of the repository-based model approach (previous chapter), while the third step is different.

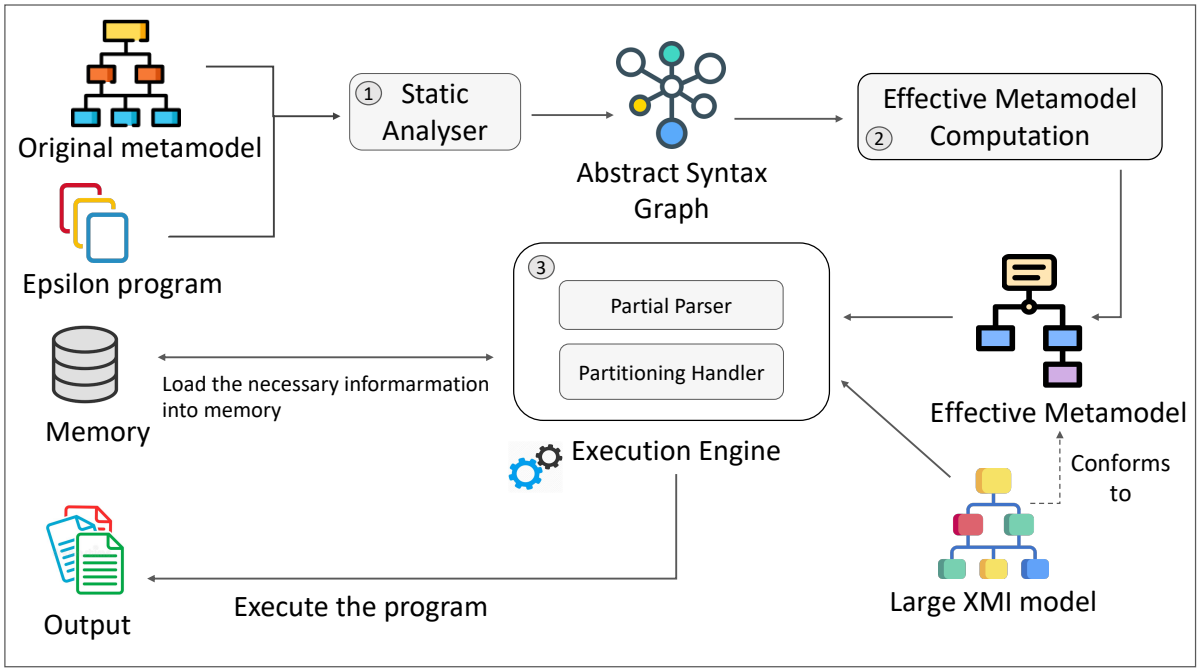


Figure 6.3: The proposed approach

6.2.1 Static Analyser

In this work, we use a static analyser for Epsilon Validation programs[20]. The inputs of the EVL static analyser are an EVL program (set of constraints) and the metamodels of the models it consumes (first step of our approach in Figure 6.3).

In-advance knowledge about the program (i.e., types and properties accessed by the program) is extracted by EVL static analyser using the abstract syntax tree. The extracted information is in the form of an *effective metamodel* for every model the constraints access. Detailed discussions regarding static analysis and the effective metamodel concept can be found in Chapter 4.

Similarly to the previous chapter, it is worth noting that the EVL static analyzer has the capability to handle multiple models, resulting in multiple effective metamodels. However, for the remainder of this chapter, we will focus on programs with a single model, thus having one effective metamodel. To illustrate how every step of the approach works, we use the motivating example of Section 6.1.

In the first step of our approach, the static analyser sets the resolved types of expressions to types from the respective metamodels, to primitive types (e.g., String, Integer) or to collection types.

For example, in line 3 of Listing 1, the resolved type of the *self* variable is equal to *Component* as the context of constraint is *Component* (Line 1). In line 3, the property

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

Table 6.1: Resolved Types Calculated by the Static Analyser

Line number in Listing 6.1	Expression	Resolved Type
1	Component	Model Element (Component)
3, 4, 9	self.name	Property call expression (String)
3, 4, 9	self	Model Element (Component)
3	self.name.ftuc()	Operation call expression (String)
8	Component.all.select(c | c.name = self.name).size()	Operation call expression (Integer)
8	Component.all.select(c | c.name = self.name)	Operation call (Collection<Component>)
8	c.name	Property call expression (String)
8	c	Model Element (Component)

call checks the *name* of each *Component*. The *name* is an attribute of *Component*, and the resolved type is String. The type of the *self.name* in line 3 (as a target of *ftuc()* operation call) and in line 4 (as a property call) is resolved as String (as same as property call in line 3).

In the following constraint (*hasUniqueName*), the target of the *select* operation call is a property call which retrieves the *name* of all *Component*s. In line 8, variable *c* is resolved as model element (*Component*) as the target of the *select* operation is all instances of *Component*. The type of *self.name* is resolved as String, as same as in the *hasValidName* constraint.

In line 14, the types of *self.source.type* and *self.target.type* are resolved as String. The type of the *self* variable is resolved to *Connector*, and the *source* and *target* are the references of *Connector*s which are resolved as model elements (source is resolved to *OutPort* and the target is *InPort* type) (see the metamodel in Figure 6.1). Table 6.1 shows the resolved types of expressions extracted from Listing 6.1 by the static analyser.

6.2.2 Effective Metamodel Computation

The extraction of the effective metamodel of constraints is the second step of the approach. The concept of the effective metamodel and the algorithm constructing the effective metamodel are discussed in Section 4.6. Figure 6.4 illustrates the effective metamodel extracted from the EVL program in our motivating example (Listing 6.1).

In Figure 6.4, the attributes of the *EffectiveMetamodel* class are filled by the original metamodel, which are the *name* and the *nsuri* of the metamodel. For running the EVL program in Listing 6.1, all instances of *Component* and *Connector* must be loaded. The *Component* and *Connector* classes are added to the *EffectiveMetamodel* under the *allOfKind* references. The *name* attribute of *Component* is added to the *EffectiveMetamodel* as well. The *source* and *target* references of *Connector* are also

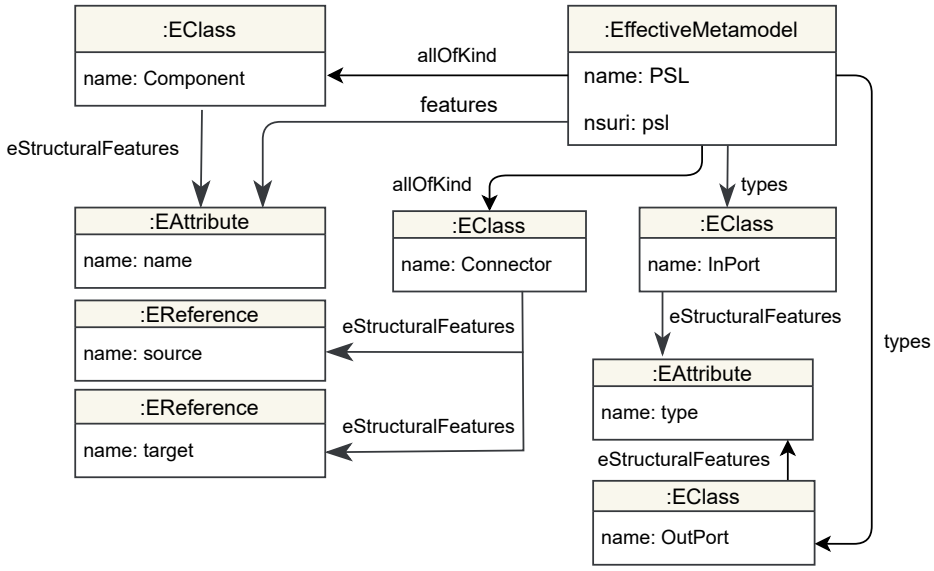


Figure 6.4: The effective metamodel extracted from Listing 6.1

required (line 14 of Listing 6.1), hence, they are added to *Connector* as *EReference*s using the *EStructuralfeatures* references.

The resolved type of *source* reference is equal to *OutPort* (see Table 6.1), so the *OutPort EClass* is added to the effective metamodel using the *types* reference which is the same for the *target* reference that is resolved to the *InPort EClass*. The last attribute is the *type* attribute which is added to the *InPort* and *OutPort EClasses*.

6.2.3 Partial Parser

As EMF's default XMI parser loads the entire model into memory, for partial loading, a custom XMI parser is needed to load models based on the information provided by an effective metamodel. For this purpose, we use the partial parser described in Section 2.5 after fixing some bugs and making further improvements to its performance. Therefore, in step 3 of Figure 6.3, the partial parser will load the model based on the effective metamodel.

This parser scans the whole XMI file and checks the tags of elements with the effective metamodel classes and features. If the element's tag is represented in the effective metamodel, then the element is loaded; otherwise, the parser ignores the element, pushes a placeholder in the parser stack and moves on to the next element in the file. For a more detailed explanation of the parser function, please refer to Section 2.5.

Using a partial parser in the EVL execution engine saves time in the loading process and saves memory [51] because less information is loaded compared to the default XMI

parser. Considering the example in Listing 6.1, in step 3 of our approach, the partial parser loads the XMI models based on the extracted effective metamodel (Figure 6.4). All instances of *Component* with their *name* attribute, all instances of *Connector* with their *source* and *target* references, and *InPort* and *OutPort* instances with their *type* attribute are loaded. Other information, like the names of ports, is skipped by the parser.

However, by only using the partial parser, the necessary information for executing all constraints is kept in memory until the end of execution. The execution engine performance can be improved by keeping this information in memory only while it is required and then disposing it. This feature can be supported by grouping the constraints, loading the parts of the model required for executing the group and disposing of it after group execution is finished.

6.2.4 Partitioning Handler

Loading all necessary information from the model upfront and running the program would be efficient in terms of loading time as it would avoid the cost of re-parsing a model, but it would keep elements and their property values in memory for longer than needed, which is not efficient in terms of memory footprint. Hence, if the machine's memory is insufficient to accommodate all the necessary model elements, the execution engine will be unable to load the model, resulting in a failure to execute the program. Partitioning the constraints and loading information for each constraint separately can be a solution for this issue. Although loading and unloading information for each constraint is not considered optimal (discussed below), it serves as an intermediate step to introduce our approach.

In this solution, instead of extracting an effective metamodel for the whole program and loading information based on that in one go, an effective metamodel is extracted for each constraint separately. After executing the constraint, all model elements will be unloaded, and the memory becomes available to run the next constraint. In Listing 6.1, there are three constraints to validate the model. Figure 6.5 shows three effective metamodels that are extracted for each constraint.

To execute the EVL constraints provided in Listing 6.1 using the intermediate solution, the EVL execution engine follows the procedure below:

- Loading Model Elements for *hasValidName* Constraint: The EVL execution engine begins by loading model elements based on the effective metamodel (EfMetamodel 1) associated with the *hasValidName* constraint, as illustrated in Figure 6.5. This involves loading all instances of *Component* with their *name* attributes. The constraint is then executed using these loaded model elements and properties.

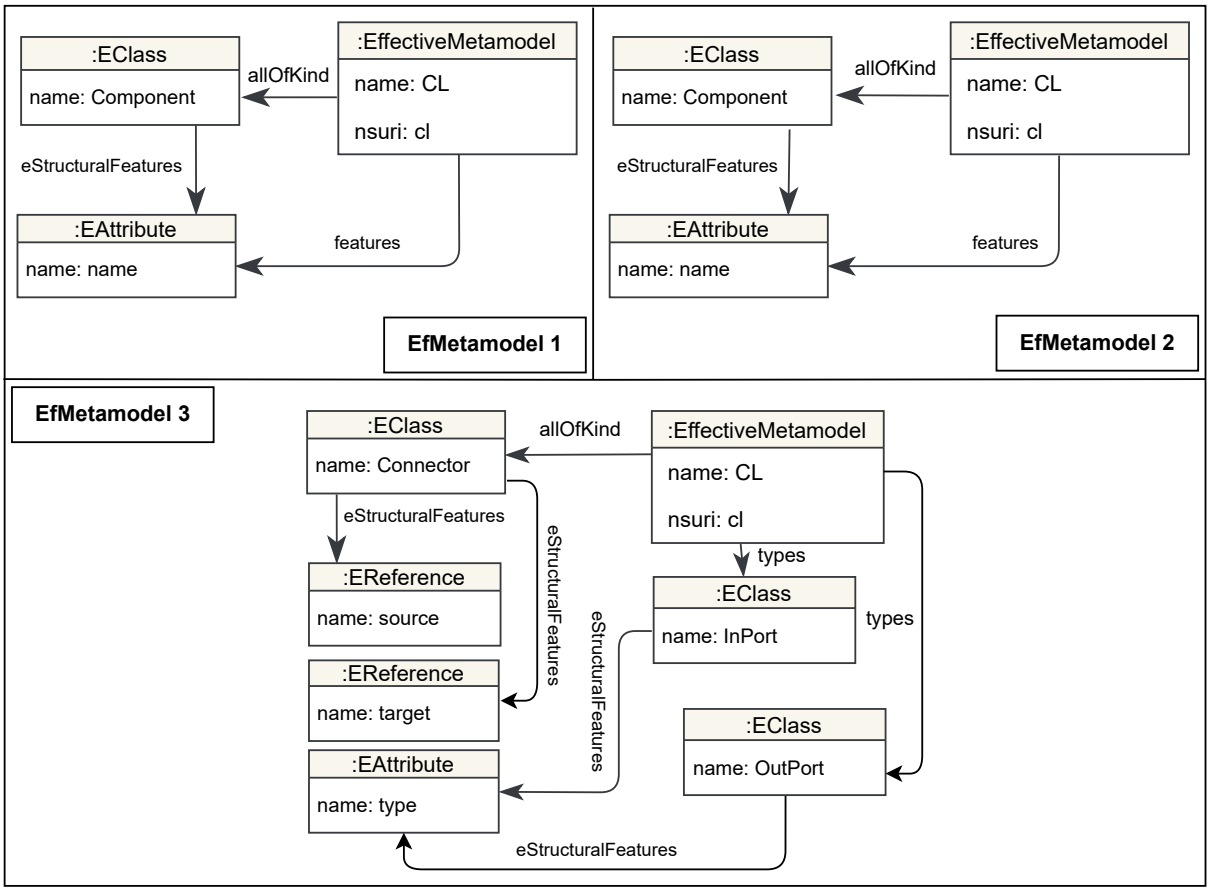


Figure 6.5: Effective metamodels of constraints in Listing 6.1

- Cache Clearing and Unloading: After the *hasValidName* constraint execution, the cache is cleared, and all information related to this constraint is unloaded from the memory.
- Repetition for *hasUniqueName* Constraint: The same process is repeated for the second constraint, *hasUniqueName*. Again, all instances of *Component* with their *name* attributes are loaded, and the constraint is executed. Once the constraint execution is complete, all information related to this constraint is unloaded.
- Repetition for *PortTypesMatch* Constraints: The described process continues for the third constraint and any subsequent constraints in a similar manner. Each constraint is executed with the necessary model elements loaded, and after execution, the information is cleared from memory.

In this solution, an *execution plan* serves as a roadmap for the execution engine, helping it navigate the steps of the constraint validation and loading process in a way that optimizes memory usage. This *execution plan* essentially outlines the specific execution steps for a given task or program within the system. Figure 6.6 illustrates

the execution plan in the intermediate solution. This execution plan consists of a cycle for loading models based on the effective metamodel, executing the constraint, and unloading information for each constraint.

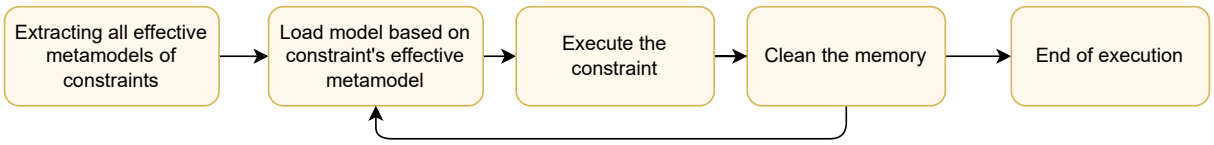


Figure 6.6: Overview of an execution plan in intermediate approach

Constraint Grouping

Partitioning the program based on constraints is efficient regarding memory footprint as model elements are not kept in the memory for the entire execution. However, it is more time-consuming as the model is loaded multiple times.

Figure 6.5 shows that the effective metamodels of *hasValidName* and *hasUniqueName* are identical (EfMetamodel 1 and EfMetamodel 2), and these two constraints require the same data. Hence, disposing of the information from memory after running *hasValidName* constraint and loading the same information again to execute *hasUniqueName* is sub-optimal.

By grouping the *hasValidName* and *hasUniqueName* constraints and associating their effective metamodel with the group, the execution engine can identify that these constraints can be executed using the same data. As a result, the execution engine optimizes the process by avoiding the unnecessary unloading of information after loading model elements for the *hasValidName* constraint. Instead, it loads the required information based on the effective metamodel, executes both constraints and unloads the information.

Therefore, adopting a strategy of grouping constraints based on their effective metamodel, specifically when they require the same information or a subset of information for execution, presents a promising approach to minimize loading time more than the intermediate solution. By identifying these relationships and grouping the constraints accordingly, the execution engine can optimize the loading process by reusing the already loaded information, thereby decreasing the overall loading time. In our approach, the partitioning handler is a component in the execution engine which is responsible for grouping the constraints.

Hence, the suggested execution plan in our approach is modified compared to the intermediate solution, as demonstrated in Figure 6.7, where data unloading occurs following the execution of a group of constraints instead of a single constraint.

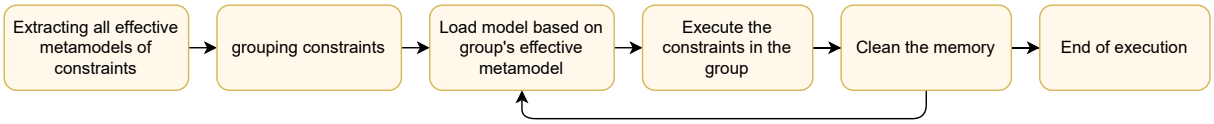


Figure 6.7: Overview of an execution plan in our approach

Algorithm 6.1 describes an algorithm to group the constraints. The strategy of this algorithm is to make a group of constraints with the same effective metamodel, or the constraints that their effective metamodels are a subset of each other.

Algorithm 6.1 Partitioning Algorithm

```
1:  let Map<Constraint, EffectiveMetamodel> constraintSets
2:  let Map<Set<Constraint>, EffectiveMetamodel> constraintgroups
3:  for all constraint1 in constraintSets do
4:      let gpEfModel = getEffectiveMetamodel(constraint1)
5:      let group = New Set<Constraint>
6:      for all constraint2 in constraintSets do
7:          if isSubSet(gpEfModel, getEffectiveMetamodel(constraint2)) or isSubSet(
    getEffectiveMetamodel(constraint2), gpEfModel) then
8:              group.add(constraint2)
9:              geEfModel = MergeMetamodel(gpEfModel, getEffectiveMeta-
    model(constraint2))
10:         constraintgroups.add(group, geEfModel)
```

In line 3 of Algorithm 6.1, the algorithm goes through all constraints and computes their effective metamodels. In line 6, the algorithm searches in constraints to evaluate if there are other constraints, the effective metamodel of which is a subset or a super-set of this constraint. If it is found, the second constraint will be added to the same group as the first one, and the super-set effective metamodel will be mapped to the group (lines 7 to 9).

For example, if a constraint accesses the *target* reference of *Connector* class, then the effective metamodel is a subset of EfMetamodel 3 (as EfMetamodel 3 includes the *target* reference of *Connector* class to execute the constraint), and it would be efficient to put them in the same group.

Another possible case for grouping constraints is when the constraints' effective metamodels are not the same or superset/subset of each other but there is a significant overlap between them (the elements that are required for running the constraints). In this situation, grouping constraints becomes less straightforward.

On the one hand, grouping constraints saves time in the loading/unloading process, but on the other hand, loading more information can increase the memory footprint of the loaded model. For example, in Figure 7.1, there are two constraints that both require

all instances of *InPort* class with the *name* attribute (they are shown by green colour). Constraint 1 needs the *incoming* reference of *InPort* class, while Constraint 2 requires the *type* attribute to be executed. In this scenario, the partitioning handler can take two different strategies:

- Loading model elements for each constraint and executing the two constraints separately. The execution engine loads the *name* attribute and *incoming* reference of *InPort* class, executes the Constraint 1 and unloads the model elements. Then, it loads the *name* and *type* attributes of *InPort* class again and executes Constraint 2.

- Grouping the two constraints, merging their effective metamodels and executing the constraints in sequence (or in parallel). The execution engine loads the *name* and *type* attributes and the *incoming* reference of *InPort* class and executes both constraints.

Therefore, there is a trade-off between loading more information in one go, having less memory available for execution and saving time, and loading elements separately and spending more time on loading but making more space available in memory for running the constraint. For example, in Figure 7.1, for executing Constraint 1, if the number of *InPort* instances is large enough to occupy most of the memory, then it is highly possible that loading an additional attribute (*type* attribute) would not be efficient (execution needs the memory). If the number of elements occupies less memory, then loading more data and running more constraints would be recommended.

The partitioning algorithm needs more information about the model to decide on partitioning and automatically evaluate which strategy is the optimal option. This information is not available in our approach, as the static analyser gains in-advance knowledge about the program. It works on the metamodel abstraction level and has no information about the model and its content. Therefore, the lack of this knowledge is to be compensated by the user.

To address this limitation, we empower users to group the constraints themselves using annotations in the EVL program. By leveraging the expertise and insights of the user, the constraints can be grouped in a way that takes into account the specific characteristics and dependencies of the model. This enables users to make informed decisions about constraint grouping, leveraging their domain knowledge and understanding of the model structure.

As shown in Listing 6.2, we consider a new annotation (*@group*) in EVL language. Using this annotation, users can group constraints manually. In Listing 6.2, the *@group* annotation is followed by an *id* . Constraints with the same group id are in the same

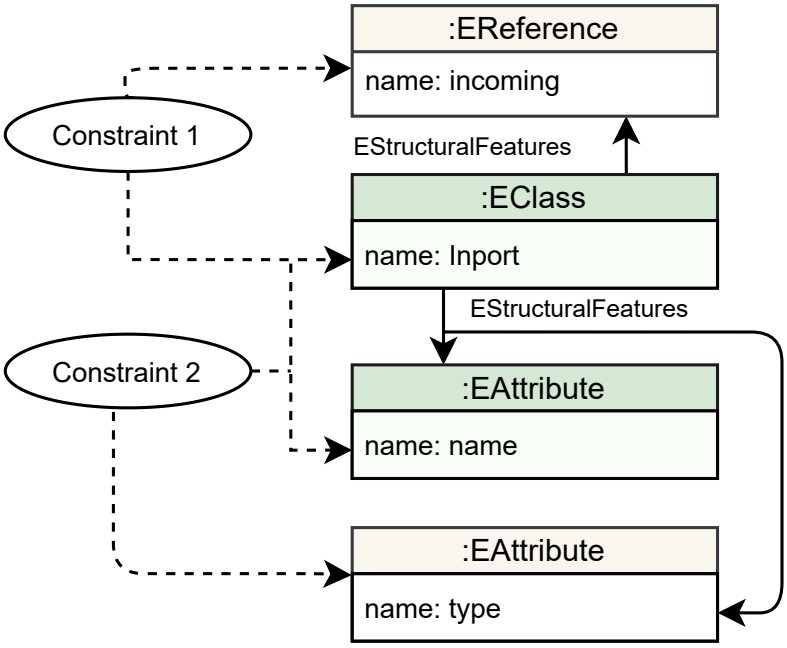


Figure 6.8: Constraints with overlap

group. Therefore, constraint *hasValidName* and *hasUniqueName* are grouped as they have the same id, and *portTypeMatch* is executed separately.

Listing 6.2: Groupig EVL constraints to validate Component Language instances

```
context Component {
  @group gp1
  constraint hasValidName {
    check: self.name = self.name.ftuc()
    message: self.name + " should start with
             an upper-case letter"
  }
  @group gp1
  constraint hasUniqueName {
    check: Component.all.select
      (c:Component|c.name = self.name).size() == 1
    message: "Duplicate component name" + self.name
  }
}
context Connector {
  @group gp2
  constraint PortTypesMatch {
    check:self.source.type=self.target.type
    message:"The types of the source and target ports don't match"
  }
}
```

By allowing users to manually group constraints, we provide a flexible and customisable approach that empowers users to optimize the performance of the constraint execution process based on their knowledge and understanding of the model.

In summary, our approach possesses the capability to automatically group constraints when efficiency can be assured, especially in cases where effective metamodels are subsets or exhibit similarities. However, in scenarios necessitating a trade-off between constraint grouping and considering more extensive model information, user intervention is required.

6.3 Implementation

As mentioned in Chapter 2, in Epsilon's architecture, there is the Epsilon Connectivity Layer (EMC)[21] which enables Epsilon programs (including EVL constraints) to interact with models in different modelling technologies in a uniform manner by defining drivers (e.g., EMF, CDO, NeoEMF).

Our frugal file-based model loading approach has been implemented in the form of a new Epsilon driver called XMIN, which is an extension of the existing EMF driver and provides facilities for Epsilon programs to load XMI-based models partially. The location of the XMIN driver within the Epsilon architecture is depicted in Figure 6.9. In the XMIN driver, the execution method of the EVL engine is overridden based on the proposed execution plan, which is shown in Figure 6.7.

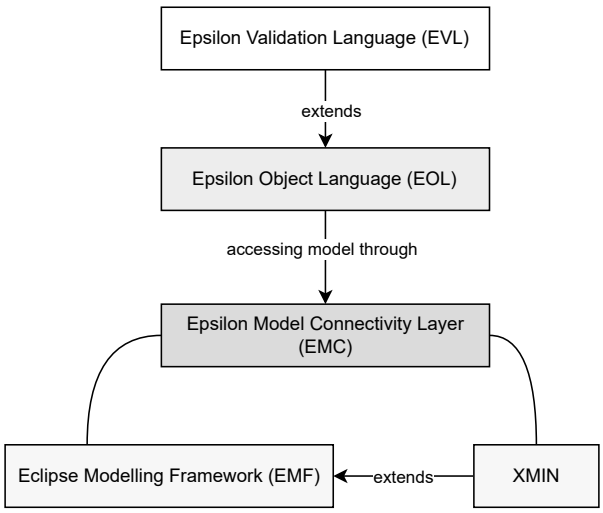


Figure 6.9: XMIN driver in Epsilon architecture

[21] https://www.eclipse.org/epsilon/doc/emc/

6.3.1 Limitations

Our proposed approach has two notable limitations that should be highlighted. Firstly, it is designed specifically for read-only input models in model management programs. This means that our approach does not support any modifications to the models, such as updates, deletions, or additions of model elements.

Secondly, from a technological perspective, this approach has been implemented and evaluated only for EMF-based models. Extending it to support other model types is part of our future work.

6.4 Evaluation

In this section, we report on the results of experiments that measure the performance of different approaches discussed in Section 6.2. By evaluating and comparing partitioning techniques, we aimed to provide insights into the advantages and time-saving potential associated with the constraint grouping approach.

6.4.1 Memory Consumption

The memory efficiency benefits of loading large models partially are reported in [51]. Wei and colleagues highlight the advantages of employing an XMI partial parser for this purpose.

In the context of benchmark experiments in their work, various models were used, all derived from reverse-engineered Java code from the GraBaTs 2009 contest. These models denoted as set0, set1, set2, set3, and set4, varied in size from $9.2\,\mathrm{MB}$ to $676.9\,\mathrm{MB}$, and they were all stored in XMI format.

The experimental procedure began with counting the loading units within each model, spanning from set0 to set4. Subsequently, Wei et al. crafted EOL programs aimed at achieving loading unit coverage levels of 20%, 40%, 60%, 80%, and 100% for models within the set0 to set4 range. Finally, set0 to set4 were loaded using the effective metamodel derived from the EOL programs and the partial parser. The experiment then recorded and compared the performance in terms of loading time and memory consumption with the performance of the built-in EMF XMI parser applied to the same models.

The benchmark results yielded valuable insights:

- Resource consumption (both in terms of time and memory) exhibited a linear relationship with loading unit coverage when using the partial parser.

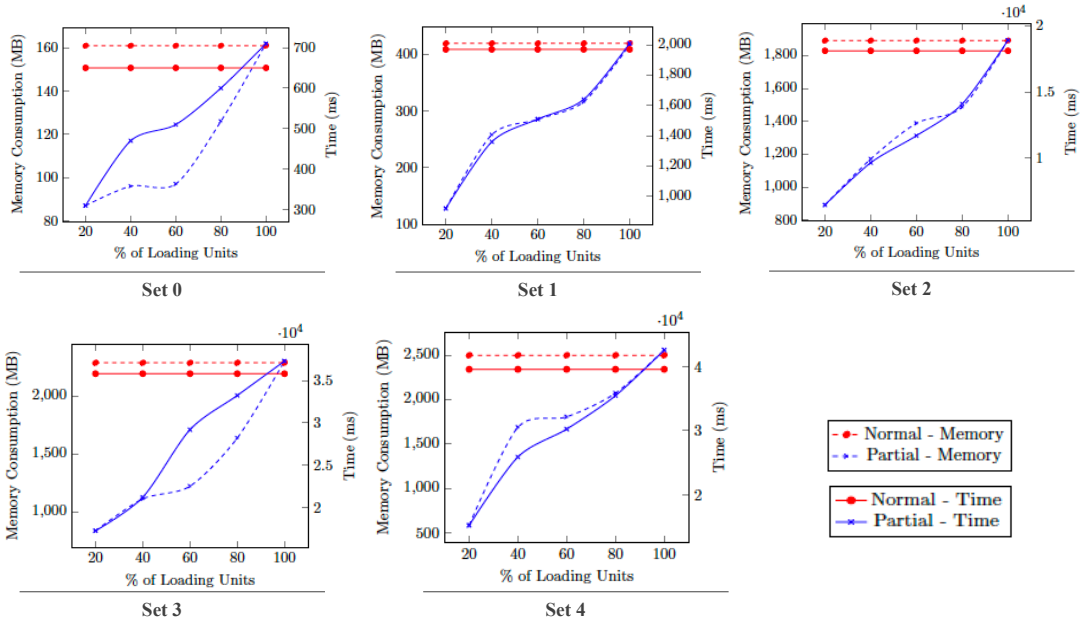


Figure 6.10: Performance report of using a partial XMI parser [51]

- Notably, for set0, the partial parser demonstrated substantial improvements in resource consumption, both in terms of time and memory. However, when aiming for 100% coverage, the partial parser incurred slightly higher time and memory usage due to upfront costs associated with effective metamodel extraction and reconciliation. Additionally, during parsing, redundant comparisons with the effective metamodel also contributed to increased time consumption.
- For set 1 to set 4, significant improvements were observed, with at least a 10% reduction in loading time and a 20% decrease in memory usage up to 80% coverage.
- The results are graphically presented in Figure 6.10 for all five data sets. It is noteworthy that the relationship between time consumption and memory consumption was not strictly proportional. This non-linearity was attributed to certain attributes of EObjects containing substantial amounts of strings, which led to increased memory consumption. This phenomenon was particularly pronounced in the case of set0, set3, and set4.

6.4.2 Time performance

Given that Wei's work has already demonstrated the memory efficiency of employing a partial parser, our focus in this section is to assess the time efficiency of different

approaches, including executing each constraint separately, and using the constraint grouping approaches presented in Section 6.2.4.

We evaluated the approaches on a system using Java VM 11.0.10 with Intel(R) Core(TM) i7, 16 GB memory and CPU @2.80 GB running Mac OS X Catalina.

For our experiments, we generated random large models using EMF (pseudo) random instantiator[22] developed by the AtlanMod Team[23]. The models conform to an Ecore-based metamodel of the Component language (see Figure 6.1). We validated five XMI models, from model1 to model5 (from a 212.9 MB XMI file with 1 million model elements to a 1.29 GB file with 6 million model elements). The models are listed in Table 6.2.

Table 6.2: Evaluated models

Model name	Number of elements	Size
model1	1,000,000	212.9 MB
model2	3,000,000	563.6 MB
model3	3,700,000	806.6 MB
model4	5,000,000	1.05 GB
model5	6,000,000	1.29 GB

We assessed our work using four validation experiments. Figure 6.15 illustrates the Component Language metamodel, and all constraints are used to validate the models conform to this metamodel. The initial experiment utilizes a validation program comprising three constraints with overlapping effective metamodels. The second experiment involves a program with two constraints having non-overlapping effective metamodels. The third experiment includes two programs, each with a mix of constraints where some are grouped and some are not. Lastly, the fourth experiment featured a program with ten constraints, with a mix of grouped and ungrouped constraints determined by both an algorithm and a user.

Experiment 1: all constraints in the same group

We have implemented an EVL program to validate the model1 to model5 models. These constraints are listed in Listing 6.3.

Listing 6.3: EVL constraints to validate Component instances in the models

[22] https://github.com/atlanmod/mondo-atlzoo-benchmark/tree/master/fr.inria.atlanmod.instantiator
[23] https://www.imt-atlantique.fr/fr

```
model ccl driver XMIN {nsuri = "http://componentlanguage"};
context Component {
    /* Component names must start with an upper-case letter */
    constraint ValidName {
        check: self.name = self.name.firstToUpperCase()
        message: self.name + " should start with an upper-case letter"
    }
    /* Component names must be globally unique */
    constraint UniqueName {
        check: getComponentsByName().get(self.name).size() == 1
        message: "Duplicate component name " + self.name
    }
    /* Components must be connected to at least one more component */
    constraint IsConnected {
        check: self.ports.exists(p:InPort|p.incoming.notEmpty()) or
        self.ports.exists(p:OutPort|p.outgoing.notEmpty())
        message: "Component " + self.name + " is disconnected"
    }
}
```

To evaluate the constraints in Listing 6.3, we executed the program in two modes. First, we measured the execution time when each constraint was executed separately. In this mode, data was loaded according to the requirement of a single constraint and unloaded before the execution of the next constraint. Second, the constraints were executed as a group where they were grouped based on Algorithm 6.1.

In Listing 6.3, the *ComponentIsConnected* constraint required all instances of the *Component* class with the *name* attribute and *ports* reference. The *ComponentValidName* constraint required the *name* attribute of all instances of *Component*, which overlaps with the *ComponentIsConnected* constraint. The *ComponentUniqueName* constraint also required the same information as *ComponentValidName*. These constraints were grouped together as they required a subset of the same information to be executed.

The results of the experiments are shown in Figure 6.11. The blue column represents the execution time for model1 to model5 when the constraints are executed separately on the models. The orange column represents the execution time when all constraints are executed as a group. As is evident by the numbers, grouping has a positive impact on execution time and boosts the execution engine performance. on average, the grouped approach improves the performance by approximately 26.37%.

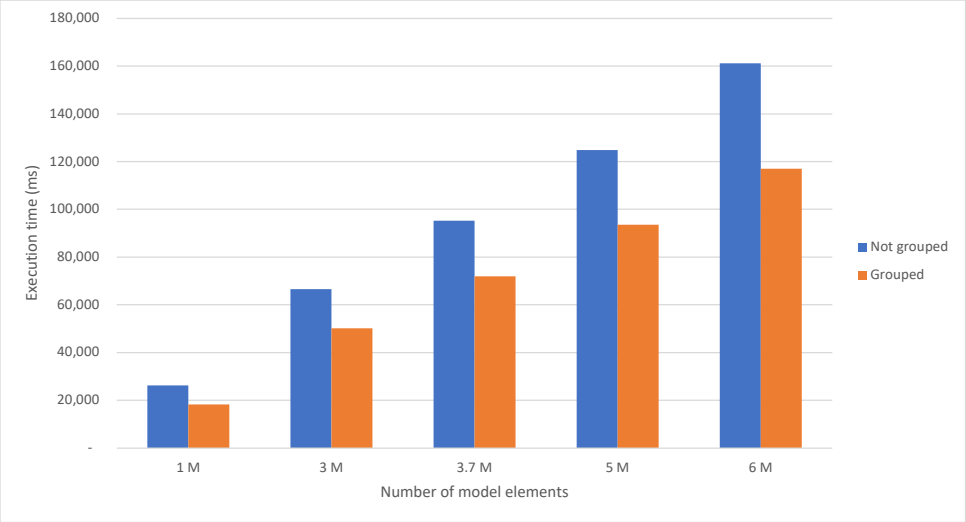


Figure 6.11: The execution time of Listing 6.3

Experiment 2: all constraints in different groups

In this test case, we aim to evaluate the effect of grouping when constraints do not have an overlap, and they are not grouped by the algorithm, but the user chooses to group them based on the availability of resources.

To illustrate the impact of grouping in this scenario, we consider two constraints as shown in Listing 6.4: *ComponentValidName*, which requires the *name* attribute of *Component*s, and *PortTypeMatch*, which needs all instances of *Connector* with their *source* and *target* references, along with all instances of *Port*s with the *type* attribute. Since there is no overlap between their requirements, these constraints would not be grouped together based on Algorithm 6.1. However, the user can choose to group them together.

Listing 6.4: EVL constraints to validate Component and Connector instances of the models

```
model ccl driver XMIN {nsuri = "http://componentlanguage"};
context Component {
    /* Component names must start with an upper-case letter */
    constraint ComponentValidName {
        check: self.name = self.name.firstToUpperCase()
        message: self.name + " should start with an upper-case letter"
    }
}
context Connector {
    /* If a connector connects two ports, their types must match */
    constraint PortTypesMatch {
        check: self.source.type = self.target.type
```

```
13          message: "The types of the source and target ports don't match"
14      }
15  }
```

The results of measuring execution time (in milliseconds) and memory consumption (in GB) are depicted in Figure 6.12. As illustrated, the orange column, representing the execution time when the constraints are not grouped, is higher than the blue one, representing the execution time when they are grouped. Therefore, grouping can save time. This is because all information is loaded in one go without the cost of unloading. Grouping constraints significantly reduces execution time, especially as the dataset size increases. For instance, at 6M, the grouped mode is approximately 10 times faster than the non-grouped mode.

Grouped execution significantly reduced execution time by approximately 58.17% to 90.23% as dataset size increased, demonstrating better scalability. However, it required more memory, with usage increasing by 28.57% to 55.56% compared to the non-grouped mode. As discussed in Section 6.2.4 and illustrated in Figure 6.12, the trade-off between memory usage and execution time is evident. The orange line (non-grouped) consistently shows lower memory usage than the blue line (grouped). While grouping constraints demands more memory, the significant reduction in execution time often justifies this cost, particularly for larger datasets. Therefore, grouped execution is recommended for large datasets due to its substantial performance benefits. For smaller datasets, the decision can depend on the availability of memory and the importance of execution time efficiency.

In general, grouping constraints, when they are not a subset, depend on the available memory. For example, if only 6 GB of memory is available for executing the constraints on a model with 6 million elements, grouping them together would require approximately 7 GB (see Figure 6.12), causing the program to fail and rendering grouping inefficient. Conversely, if sufficient memory is available, grouping can aid the execution engine in validating the model faster.

Experiment 3: mixed constraints

1. **Grouping three out of five constraints:** This experiment evaluates the impact of grouping on the execution time of a program with mixed constraints. In this scenario, we consider a program (in Listing 6.5) with five constraints, three of which are grouped according to Algorithm 6.1, while the remaining two are not.

   As explained in *Experiment 1*, the constraints *ComponentValidName*, *ComponentUniqueName*, and *ComponentIsConnected* are grouped by Algorithm 6.1 due to

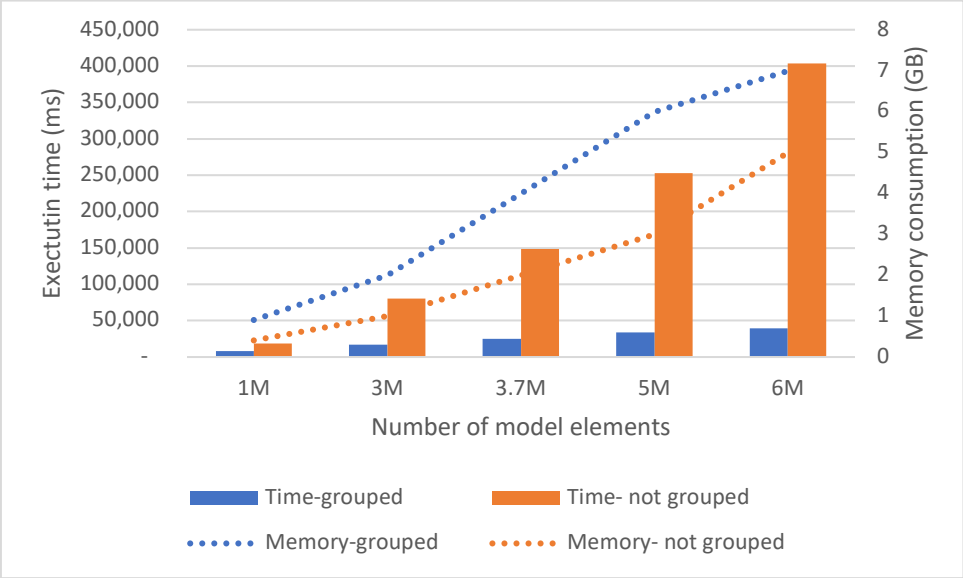


Figure 6.12: The execution time and memory consumption of Listing 6.4

the overlap between their effective metamodels. The *PortTypeMatch* constraint requires all instances of *Connector* with their source and target references, along with all instances of *Ports* with the type attribute to be executed successfully. The *PortValidName* constraint requires all instances of *Port* with their *name* attribute. Since there is no subset relation between the *PortTypeMatch* and *PortValidName* constraints, they are not grouped and are executed individually.

Listing 6.5: EVL constraints to validate Component Language instances

```
model ccl driver XMIN {nsuri = "http://componentlanguage"};
context Component {
    /* Component names must start with an upper-case letter */
    constraint ValidName {
      check: self.name = self.name.firstToUpperCase()
      message: self.name + " should start with an upper-case letter"
    }
    /* Component names must be globally unique */
    constraint UniqueName {
        check: getComponentsByName().get(self.name).size() == 1
        message: "Duplicate component name " + self.name
    }
    /* Components must be connected to at least one more component */
    constraint IsConnected {
        check: self.ports.exists(p:InPort|p.incoming.notEmpty()) or
        self.ports.exists(p:OutPort|p.outgoing.notEmpty())
        message: "Component " + self.name + " is disconnected"
    }
```

```
19 }
20 context Connector {
21     /* If a connector connects two ports, their types must match */
22     constraint PortTypesMatch {
23         check: self.source.type = self.target.type
24         message: "The types of the source and target ports don't
               match"
25     }
26 }
27 context Port {
28     /* Port names must start with an lower-case letter */
29     constraint ValidName {
30         check: self.name.isDefined() implies self.name = self.name.
               firstToLowerCase()
31         message: self.name + " should start with a lower-case letter"
32     }
33 }
34 @cached
35 operation getComponentsByName() {
36     return Component.all.mapBy(c|c.name);
37 }
```

We first run the program with no grouping, executing each constraint separately, and then execute the program by grouping three constraints and not grouping the other two.

As it is shown in Figure 6.13, the execution time is consistently lower when constraints are grouped. The most significant reduction in time is observed in the model with 6 million elements, with a decrease from 347 milliseconds to 303 milliseconds. Grouping constraints saves loading time for the grouped constraints, leading to more efficient execution. The average percentage reduction in loading time when constraints are grouped is approximately 10.07%.

2. **Grouping two out of five constraints:** To further investigate the impact of the number of grouped constraints, we conducted a second experiment with a different program containing five constraints. This time, only two constraints were grouped based on the algorithm, while the other three were not. Listing 6.6 demonstrates the constraints that are used for validation in this experiment.

Listing 6.6: EVL constraints to validate Component Language instances

```
1 model ccl driver XMIN {nsuri = "http://componentlanguage"};
2 context Connector {
3     /* If a connector connects two ports, their types must match */
```

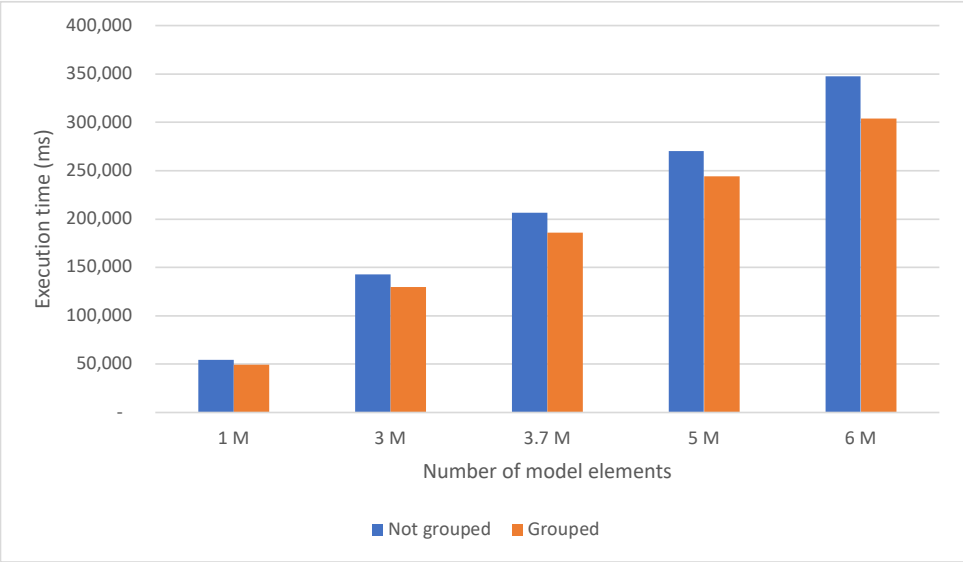


Figure 6.13: The execution time and memory consumption of Listing 6.5

```
4       constraint PortTypesMatch {
5           check: self.source.type = self.target.type
6           message: "The types of the source and target ports don't
                match"
7       }
8       /* The source and target ports of a connector must belong to
            different components */
9       constraint DifferentComponents {
10          check: self.source.component <> self.target.component
11          message: "Cannot connect ports of the same component"
12      }
13  }
14  context Port {
15      /* Port names must start with an lower-case letter */
16      constraint ValidName {
17          check: self.name.isDefined() implies self.name = self.name.
                firstToLowerCase()
18          message: self.name + " should start with a lower-case letter"
19      }
20  }
21  context OutPort {
22      /* The name of a port must be unique within its container
            component */
23      constraint UniqueName {
24          check: self.component.ports.select(p : OutPort|p.name = self.
                name).size() == 1
25          message: "Duplicate port name " + self.name
26      }
```

```
27
28    /*Unnamed out ports are only allowed for components with one out
          port*/
29     constraint NamedIfMultiple {
30         check: self.component.ports.select(p: OutPort|true).size() >
               1 implies
31              self.name.isDefined()
32         message: "Unnamed out ports are only allowed for components
               with one out port"
33     }
34 }
```

DifferentComponents constraint needs all instances of *Connector* with their *source* and *target* references, along with all instances of *Ports* with the components reference, while for *PortTypeMatch*, the *type* attribute of *Port* is needed instead of *components* reference.

Constraint *OutPortUniqueName* and *NamedIfMultiple* are grouped as both of them need all instances of *OutPort* with their *components* reference and *name* attribute and all instances of *Component* with their *ports* reference.

Figure 6.14 illustrates a similar trend as observed in the first program: the grouped condition consistently exhibits lower execution times across all models. The most significant reduction occurs in the model with 6 million elements, where the execution time decreases from 357 milliseconds to 216 milliseconds. On average, grouping constraints results in a 24.25% reduction in loading time.

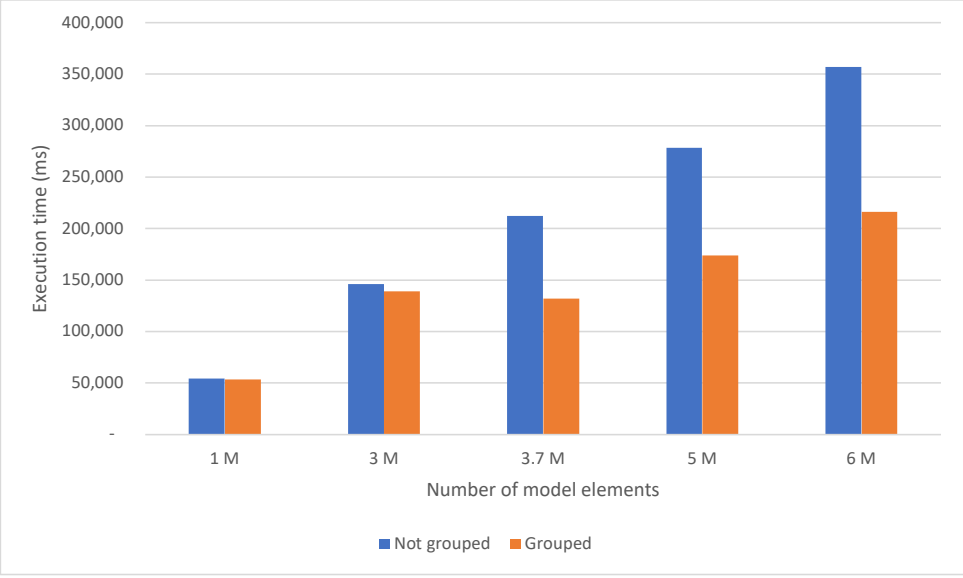


Figure 6.14: The execution time and memory consumption of Listing 6.6

The results indicate a more noticeable impact on time consumption compared to the first program, which is attributed to the specific design of the model and the number of elements of each class in the models.

The time saved through grouping constraints comes from the avoidance of loading redundant information, resulting in reduced loading time. In the first program, loading time was saved by fetching all instances of *Component*s with their *name* attribute just one time instead of three times (for each constraint seperately), thereby reducing loading time is equal time needed for loading all instances of *Component* with a single attribute. Conversely, in the second program, time was saved is equal to time needed for loading all instances of *OutPort* with their *components* reference and *name* attribute, alongside all instances of *Component* with their *ports* reference.

It's essential to consider not only the size of properties but also the quantity of each element, as both factors play a crucial role in this analysis.

Experiment 4: constraints in the three different modes

We have also implemented 10 constraints[24] in the EVL language to validate these models. These constraints are listed in Listing 6.7.

Listing 6.7: EVL constraints to validate Component Language instances

```
model ccl driver XMIN {nsuri = "http://componentlanguage"};

context Component {
    /* Component names must start with an upper-case letter */
    constraint ValidName {
        check: self.name = self.name.firstToUpperCase()
        message: self.name + " should start with an upper-case letter"
    }
    /* Component names must be globally unique */
    constraint UniqueName {
        check: getComponentsByName().get(self.name).size() == 1
        message: "Duplicate component name " + self.name
    }
    /* Components must be connected to at least one more component */
    constraint IsConnected {
        check: self.ports.exists(p:InPort|p.incoming.notEmpty()) or
        self.ports.exists(p:OutPort|p.outgoing.notEmpty())
        message: "Component " + self.name + " is disconnected"
    }
}
```

[24] https://eclipse.dev/epsilon/playground/?858b6314

```
21 context Connector {
22     /* If a connector connects two ports, their types must match */
23     constraint PortTypesMatch {
24         check: self.source.type = self.target.type
25         message: "The types of the source and target ports don't match"
26     }
27     /* The source and target ports of a connector must belong to different
           components */
28     constraint DifferentComponents {
29         check: self.source.component <> self.target.component
30         message: "Cannot connect ports of the same component"
31     }
32 }
33 context Port {
34     /* Port names must start with an lower-case letter */
35     constraint ValidName {
36         check: self.name.isDefined() implies self.name = self.name.
               firstToLowerCase()
37         message: self.name + " should start with a lower-case letter"
38     }
39 }
40 context InPort {
41     /* Either all input ports of a component are connected or none */
42     constraint IsConnected {
43         check: self.component.ports.exists(p : InPort|p.isConnected())
               implies self.isConnected()
44         message: "Either all or no input ports must be connected"
45     }
46     /* The name of a port must be unique within its container component */
47     constraint UniqueName {
48         check: self.component.ports.select(p : InPort|p.name = self.name).
               size() == 1
49         message: "Duplicate port name " + self.name
50     }
51 }
52 context OutPort {
53     /* The name of a port must be unique within its container component */
54     constraint UniqueName {
55         check: self.component.ports.select(p : OutPort|p.name = self.name)
               .size() == 1
56         message: "Duplicate port name " + self.name
57     }
58
59    /*Unnamed out ports are only allowed for components with one out port*/
60     constraint NamedIfMultiple {
```

```
61          check: self.component.ports.select(p: OutPort|true).size() > 1
                implies
62               self.name.isDefined()
63          message: "Unnamed out ports are only allowed for components with
                one out port"
64      }
65  }
66  operation InPort isConnected() : Boolean {
67      return self.incoming.notEmpty();
68  }
69  @cached
70  operation getComponentsByName() {
71      return Component.all.mapBy(c|c.name);
72  }
```

Every constraint is represented by a coloured dot with a number, and classes and features of the metamodel required for its execution, are marked by the same colour. For example, the *ComponentValidName* constraint requires all instances of *Component* class and their *name* attribute. In Figure 6.15, *ComponentValidName* constraint is represented by a dark blue dot (number 2). Hence, there is a dark blue dot in the *Component* class compartment and its *name* attribute.

Within this structure, when multiple dots always appear together in the metamodel, it means they are using the same information and can be grouped. In Figure 6.15, *ComponentValidName* belongs to the same group as the *ComponentIsConnected* and *ComponentUniqueName* constraints, as they require a subset of the same information to be executed. By following the colour of these constraints (dark green (1), dark blue (2) and red (3)), it is shown that they appear in Figure 6.15 next to each other. Therefore, they are allocated to the same group according to Algorithm 6.1.

The execution time was measured for three partitioning modes. The modes are mentioned below:

1. Partitioning the model based on constraints and running each constraint independently (intermediate solution): In this mode, the model is partitioned based on individual constraints, and each constraint is executed separately. By running constraints independently, any dependencies or overlaps between constraints are ignored.

2. Partitioning the model based on groups of constraints using our proposed algorithm: In this mode, we use our algorithm to group the constraints based on specific criteria (see Section 6.2.4). The algorithm determines which constraints should be executed together as a group. By grouping constraints, we aim to

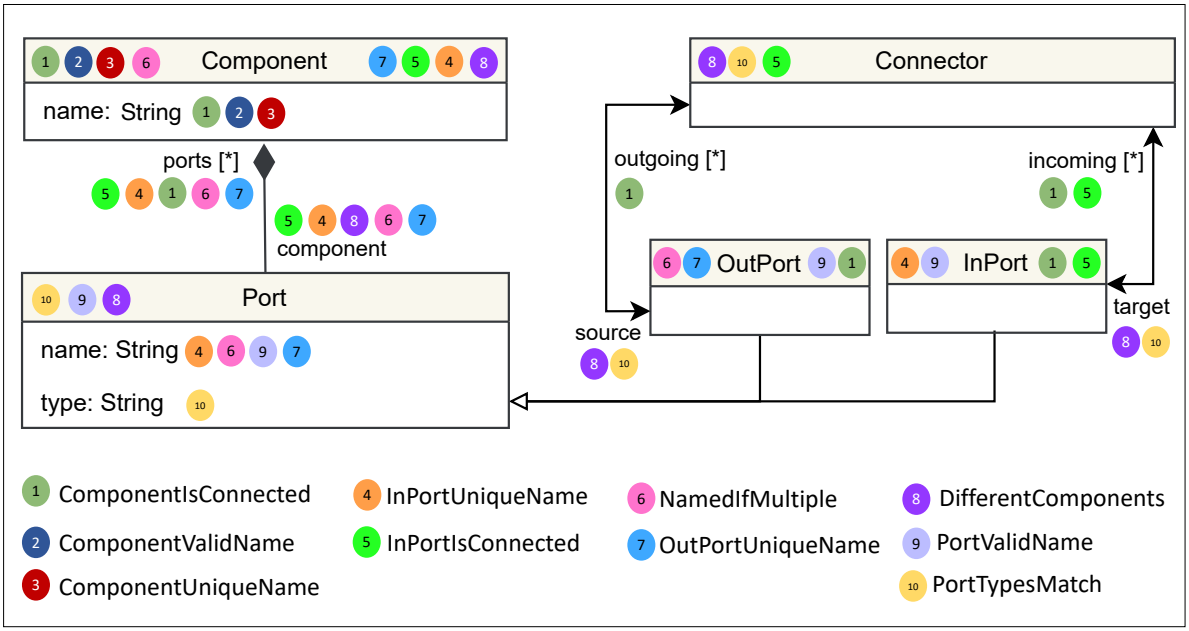


Figure 6.15: The coverage of evaluated constraints

optimise the execution time by reducing the overhead of repeatedly loading the same information. This approach considers the overlaps between constraints.

3. Partitioning the model based on groups of constraints specified by the user: In this mode, the user defines how constraints should be grouped based on their understanding of the model and the relationships between constraints.

To evaluate the execution time in each partitioning mode, we measured the time taken to execute the constraints for each mode separately. By comparing the execution times across the three modes, we can assess the efficiency and effectiveness of each approach in terms of constraint execution time.

The results are shown in Figure 6.16. In this figure, the blue column represents execution time in mode 1 for model1 to model5 where all constraints are executed separately. The orange column represents the execution time in mode 2 for the XMI models, the constraints are grouped based on Algorithm 6.1, and the grey column demonstrates execution time when the user groups the constraints manually.

As demonstrated in Table 6.3, constraints are divided into seven groups in mode 2.

In the third mode of partitioning, two groups remain the same as in mode 2 (first and second rows in Table 6.3), based on Algorithm 6.1. However, an additional group is added by the user (third row of mode 3 in Table 6.3). The information needed for executing the *InPortUniqueName* constraint is similar to that for the *InPortIsConnected* constraint, with just one additional attribute needed for executing the former. Upon examining our test models, it was realised that, for example, out of the 5 million elements,

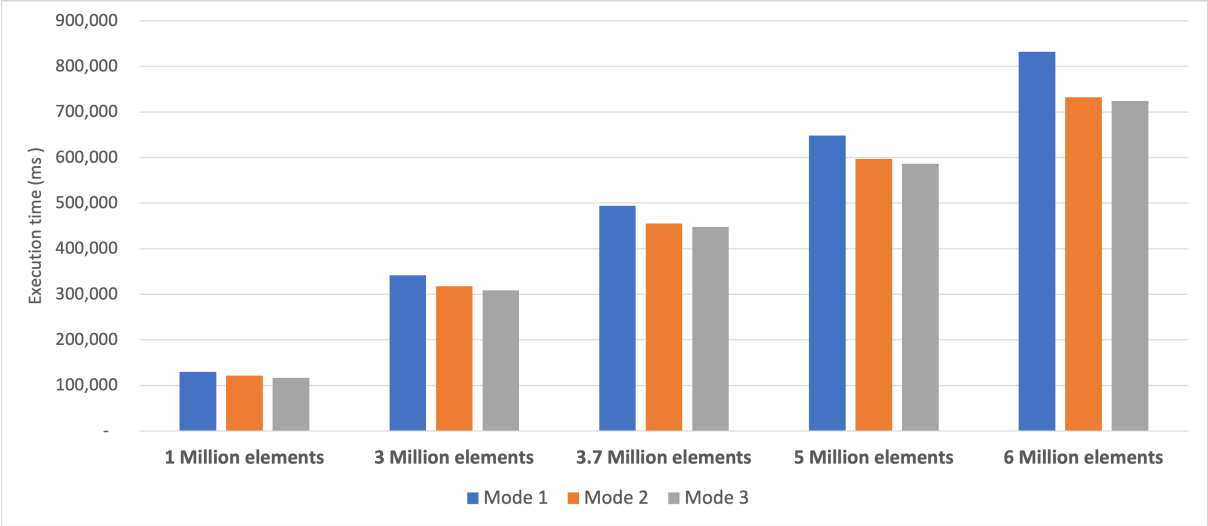


Figure 6.16: Execution time for different models

there are over 2 million instances of InPort. Considering this insight, unloading 2 million elements and loading them again would likely be more time-consuming compared to loading one more attribute.

To address this, in this experiment, we grouped the *InPortUniqueName* and *InPortIsConnected* constraints together to save loading time for the *InPortUniqueName* constraint. By grouping these constraints, a part of the loading time for loading the *InPortUniqueName* constraint's required information is saved, as the necessary information is already loaded while considering the additional attribute. This grouping strategy takes advantage of the machine's available memory and assumes that accommodating one more attribute is feasible.

Indeed, comparing the three columns in Figure 6.16 underscores the significant impact of constraint grouping on execution time. When comparing the orange column(mode 2) and blue columns (mode 1), it's evident that the grouping of constraints influences execution time. This comparison highlights the benefits of algorithmic grouping, as seen in the orange column.

Furthermore, comparing the grey column (mode 3) with the orange column (mode 2) sheds light on the potential for further optimization through user input. The observed improvements emphasize the value of incorporating user insights and preferences into the constraint grouping process, resulting in even more pronounced enhancements in execution time.

As a general observation in our experiment, considering the models and constraints that we evaluated, grouping constraints can result in a time saving of approximately 10% to 12% in the execution time of EVL constraints. Comparing the three columns for each model in Figure 6.16 demonstrates that when constraints are grouped together, there is

Table 6.3: Partitions in two modes

groups in mode 2	groups in mode 3
ComponentUniqueName ComponentValidName ComponentIsConnected	ComponentUniqueName ComponentValidName ComponentIsConnected
outPortUniqueName NamedMultiple	outPortUniqueName NamedMultiple
InportIsConnected	InportIsConnected InportUniqueName
InportUniqueName	
DifferentComponents	DifferentComponents
PortTypeMatch	PortTypeMatch
PortValidName	PortValidName

a significant improvement in the overall execution time. It shows how manual grouping of constraints based on insights about the model and the trade-off between loading elements versus loading additional attributes can lead to significant further time savings in constraint execution. Also, the increasing differences between the columns as the model gets larger demonstrate that our approach is capable of handling larger models more efficiently. It highlights the scalability of our approach and indicates that larger models benefit even more from the grouping of constraints. These findings validate our hypothesis that partitioning has a notable impact on execution time, thereby enhancing the performance of the execution engine.

To provide more detailed insights and justify the results, a subset of the obtained result is presented in Table 6.4, specifically focusing on three out of ten constraints, both before and after grouping.

Table 6.4: Loading time and execution time for model4 (in seconds)

	Time in Mode 1		**Time in Mode 2**	
Constraint	**Loading**	**Execution**	**Loading**	**Execution**
ComponentUniqueName	13,012	10,299	17,238	73,786
ComponentIsConnected	17,338	58,141		
ComponentValidName	14,107	7,466		

In Table 6.4, it is shown that the execution engine, under the first mode of partitioning, requires approximately 13 seconds to load the necessary model elements and properties

for executing the *ComponentUniqueName* constraint. Similarly, it takes roughly 17 seconds to load the required information for the *ComponentIsConnected* constraint and about 14 seconds for the *ComponentValidName* constraint.

In the second mode, the constraints are grouped based on Algorithm 6.1. The *ComponentIsConnected* constraint required all instances of *Component* class with the *name* attribute and *ports* reference. Furthermore, this constraint belongs to the same group as the *ComponentValidName* and *ComponentUniqueName* constraints. These constraints are in the same group as they require a subset of the same information to be executed (See Figure 6.15). As shown in Table 6.4, the time for loading the necessary information for all of these constraints is about 17 seconds (which is equal to the loading time of *ComponentIsConnected* constraint). The loading time for the two other constraints can be saved by grouping these constraints together, as they can now reuse the information loaded for the *ComponentIsConnected* constraint.

Reviewing the execution time in both modes in Table 6.4, it is worth noting that there is no execution time overhead observed when executing the constraints (the total execution time of constraints in Mode 1 matches that of Mode 2). This implies that the constraint execution time remains consistent regardless of the constraint grouping method applied (only the model loading times differ).

Regarding the correctness of our approach, we conducted validation by executing the EVL constraints using both the default EVL execution engine and our approach. The aim was to ensure that the output generated by both execution methods is identical in terms of the number of satisfied and unsatisfied constraints, regardless of whether partial loading and partitioning were applied. This validation process ensures that our approach does not introduce conflict or any inconsistencies in the constraint execution process compared to the default execution engine. By establishing the equivalence of the outputs, we can conclude that our approach for partial loading and partitioning maintains the correctness of the constraint evaluation, producing results consistent with the default execution engine.

6.4.3 Threats to validity

To address construct validity threats, we specifically consider large models generated by a random generator that conform to the Component Language metamodel. The results presented in this chapter are based on these test cases.

To mitigate internal validity threats, we ensure reliable and consistent results by executing three warm-up iterations of the program before reporting the final results. Additionally, we take an average of 10 executions to minimize any potential impact on memory usage and execution time caused by JVM startup and initialization.

To minimize external validity threats, we build our approach on top of mature and robust Model-Driven Engineering (MDE) technologies, such as the Epsilon suite of model management programs and the XMI format. These technologies provide a solid foundation for our approach. We acknowledge that extending our approach to support other technologies is relatively straightforward with reasonable effort, as discussed in Section 6.2. However, further experiments are needed to determine the applicability and scalability of our approach in domains and with metamodels/models and constraints that have different characteristics from those used in our experimental evaluation. By addressing these threats and taking measures to ensure reliability and generalizability, we strive to provide a comprehensive evaluation of our approach in this work.

6.5 Chapter Summary

In this chapter, we introduced an approach for grouping EVL constraints through the static analysis of EVL programs. We began by presenting a motivating example in Section 6.1 and outlined the architecture of our approach in Section 6.2. Subsequently, we provided an overview of the implementation and discussed the limitations of our approach in Section 6.3.

Furthermore, we conducted evaluations utilizing five large randomly generated models in Section 6.4. The results have shown that the act of grouping constraints can lead to a reduction in both loading and overall execution times of EVL programs when compared to a non-grouping approach. Importantly, this improvement in efficiency does not have any adverse effects on program behaviour or output.

7 Conclusions

In the final chapter, we evaluate the overarching aims, objectives, and achievements of the thesis. Additionally, we emphasise the necessity for further research. Section 7.1 provides a brief summary of this thesis. Section 7.2 integrates the contributions of the thesis along with its key insights. Section 7.3 suggests potential directions for future complementary research.

7.1 Summary

The general aim of this thesis is to execute model management programs more efficiently. The main focus of this work was on scalability issues when such programs deal with very large models. The approach we presented provided partial loading and partitioning features to the execution engine of EOL and EVL languages. The main idea is to statically analyse the model management program and gain information about the program in compile-time. Using this information, the execution engine is able to load and process only the parts of the model that are required for executing the program and save space in memory by skipping unnecessary parts. Also, instead of keeping all information in memory for the whole execution, the execution engine can keep model elements till they are needed by the program and dispose of them when they are no longer referenced by the program.

In Chapter 1, we provided a concise introduction to the motivation behind this thesis. Chapter 2 provided a comprehensive background for this research. It began by introducing the fundamental concepts of Model-Driven Engineering, discussing various levels of abstraction within this methodology. The chapter then delved into the Eclipse Modeling Framework, a prominent framework in MDE, introducing its key components such as EObject, EClassifier, EClass, and EDataType. The discussion extended to various model management tasks, including model transformation and validation. The Epsilon framework, serving as the foundation for this thesis, was introduced, along with elaborating its two languages: EOL and EVL. Additionally, two types of model persistence were discussed to familiarise the reader with the different model varieties explored in the subsequent thesis chapters: file-based models and repository-based models. The chapter explored related works in each model persistence category, shedding light on the partial parsing of XMI, Morsa, CDO, and NEOEMF repositories. We highlighted their loading techniques, lazy and greedy while emphasising their limitations. The introduction of Neo4j, a widely-used database, and the Cypher language for querying such databases further enriched the reader's understanding. Towards the end of the chapter, the research addressed the stability challenges within MDE,

referencing various efforts by researchers to enhance its scalability. Nevertheless, a gap was identified, establishing the foundation for the motivation behind this work.

In Chapter 3, we elaborated on how the thesis would demonstrate its intended contributions in practice. We initiated by offering a brief overview of the foundational motivation driving this research and subsequently introduced the thesis hypothesis. The objectives were presented concisely as bullet points and subdivided into sub-objectives for more detail. Later, we outlined the scope. It was worth emphasising how each section of this work aligns precisely with the defined objectives.

Chapter 4 serves as the foundation of our approach. The chapter started by introducing the concepts of Abstract Syntax Trees and clarifying the composition of an EOL program. Following this, the discussion explored the static analyser of the EOL language, along with a review of previous efforts to implement a static analyser for Epsilon using variable resolution and type resolution. We then detailed our contributions to the static analysis of Epsilon, highlighting the improvements we made to its functionality, including features like type inference and the generation of compile-time errors as a byproduct. Additionally, we presented our approach for evaluating this static analyser, and we made note of similar analysers implemented for other languages. In the latter part of the chapter, we introduced the concept of an effective metamodel, a product of the static analyser's output. We also presented an algorithm for extracting the effective metamodel from the resolved-type Abstract Syntax Tree, a key component in our approach.

The first contribution of this thesis was presented in Chapter 5. The primary focus of this chapter revolved around the partial loading of models stored in a graph-based database. At the beginning of this chapter, a motivating example was presented to aid readers in understanding the underlying problem and recognising the necessity of the approach outlined in this chapter to address it. Subsequently, the architecture of our approach was presented, illustrating how partial loading can be accomplished when the execution engine interacts with repository-based models. An algorithm was introduced to map the model's structure to a graph and store it in the database, utilising a graph-based database due to its structural similarity with models and superior performance compared to relational or document-based databases [5]. The proposed architecture considered the static analysis, the effective metamodel of the program, as well as the program serving as inputs to the execution engine. The execution engine employed a query generator to create queries for model loading based on the effective metamodel. Since Neo4j was the backend used in this approach, the queries were automatically generated in the Cypher language. We designed and implemented an algorithm for generating Cypher queries based on the effective metamodel. Additionally, we applied

optimisation techniques to generate queries that are optimised. Consequently, only the parts of the model necessary for program execution were loaded into memory. This approach effectively eliminated the overhead associated with loading extraneous information. The chapter concluded by providing insights into the implementation of this approach. It presented a performance analysis of memory usage and execution time, comparing our approach with the state-of-the-art method. We employed the GraBats test case, and the results were quite significant. On average, our approach exhibited a remarkable 84% reduction in memory consumption and a 37% decrease in execution time when compared to NeoEMF.

Chapter 6 presented the second contribution within this thesis. The main emphasis of this chapter was on partitioning large XMI models, a key strategy aimed at improving the performance of model validation programs. The chapter initiated with a motivating example that effectively clarified the problem and underscored the necessity of integrating partial loading and partitioning when dealing with large models in the execution engine.

Then, an approach was presented that used static analysis to analyse the validation program and extract effective metamodels. Following this, by integrating a partial XMI parser and a partitioning handler within the execution engine, we empowered the engine to manage large XMI models efficiently. The partial parser of XMI was introduced before. Hence, we refined the partial XMI parser by addressing and rectifying bugs. This enhanced partial parser is now seamlessly integrated into the execution engine, allowing us to load models based on the program's effective metamodels. In the next part, we designed and implemented a new algorithm to group the constraints in the validation program. We elaborated on how this innovative approach can enhance performance when compared to the alternatives of either loading all required information upfront or loading information for each constraint individually. We extended the flexibility to users, allowing them to make grouping decisions as they see fit and granting them control over constraint execution. Following this, we provided a brief overview of the implementation details. To conclude this chapter, we evaluated our approach and presented the results obtained from the loading and validating of a large XMI model, ranging from 1 million to 6 million elements.

7.2 Thesis Contributions

The main aim of this thesis was to optimise and enhance the scalability of model management programs. To achieve this, we have strengthened the capabilities of the execution engine of EOL and EVL languages, specifically in handling large models more

efficiently concerning memory usage and processing time. The contributions enabling these improvements are as follows:

7.2.1 Static analysis

The main idea of this work was to provide in-advance information about the program and load models based on the program's requirements. Therefore, we enhance the static analyser of Epsilon languages to analyse the program and provide information about model elements, attributes and references that are necessary for executing the program. It is helpful to gain a deeper understanding of the program's behaviour during the compilation phase. We enhanced the functionality of the static analyser using two features:

1. **Type Inference.** We added the type inference ability to the static analyser to make it able to infer types of expressions, statements and variables. In this way, the analysis will be more accurate, and the execution engine can be more reliable regarding the type resolution.

2. **Type Checking.** We added the type-checking feature to Epsilon static analyser, which generates compile time errors and helps developers to write better quality code.

Also, we used the static analyser output to keep all required information in the form of effective metamodels:

- **Effective Metamodel.** We introduce an algorithm that provides a strategy for applying the static analyser in our research. We demonstrate how to effectively store the valuable information obtained from the static analyser in a format that can be easily interpreted by the execution engine.

7.2.2 Partial Loading of Repository-based Models

As part of the contributions in this thesis, we have introduced a novel approach for selectively loading models from the Neo4j database, drawing upon insights obtained from the static analyser's output.

1. **Efficient Graph Query Generation.** Another critical component of our approach is the creation of an algorithm that translates a program's effective metamodel into a series of efficient graph queries. These queries are optimised to retrieve only the elements, relationships, and properties that the program is likely to utilise during runtime.

2. **Query Optimisation Techniques.** To further enhance the performance, we have applied query optimisation techniques. These optimisations result in queries that can retrieve information from the database in a significantly reduced time.

3. **Evaluation.** We have rigorously evaluated our approach regarding both memory footprint and execution time. This evaluation provides valuable insights into the efficiency and practicality of our methodology.

4. **Prototype Implementation.** As a practical demonstration of these algorithms, we have developed a prototype implementation using the Eclipse Epsilon family of model management tools. This implementation is a tangible illustration of the concepts and methods outlined in this research.

In summary, the contributions of this thesis encompass a comprehensive set of algorithms and techniques aimed at selectively loading models from a repository-based database. This approach represents a balanced compromise between traditional lazy and greedy loading strategies. Based on our experimental results, it becomes evident that this approach can significantly reduce the time and memory resources required to execute model management programs, particularly when working with large models stored in a database and when the program necessitates only a subset of the models for execution.

7.2.3 Partitioning

In our final contribution, we directed our focus towards partitioning:

1. **Algorithm for Model Partitioning.** We introduced an algorithm crafted to partition model validation programs efficiently. This algorithm allows for the selective loading of models based on the specific data required for the execution of distinct parts of the program. It streamlines the validation process by optimising memory usage and enhancing the overall program's efficiency.

2. **Prototype Implementation.** To validate the effectiveness of this approach, we implemented a prototype. This prototype allows users to group validation constraints according to their specific requirements and preferences. This hands-on implementation allows users to experience the benefits of partitioning firsthand and tailor it to their unique needs.

To the best of our knowledge, there is no other existing work that performs static program analysis to determine the essential data required for program execution and

subsequently loads the model automatically based on this analysis, be it in a file-based model or a repository-based model. This innovative approach appears to be unique in its methodology and contributions.

7.3 Future Work

While this thesis has uncovered important aspects of model loading and the challenges related to scalability in MDE, it is essential to recognise that there are still interesting questions to explore, new areas to investigate, and emerging technologies that can push the boundaries of this field. In the next section, we will point out areas that need more research. This will serve as a roadmap for future researchers to build on what we have learned, overcome existing limitations, and contribute to the ever-changing landscape of scalability in MDE.

- **Evaluate the Static Analyser with Additional Test Cases.** Our evaluation focused on the performance of the static analyser using the Eugenia test case. This choice was made because of our confidence in the correctness of Eugenia's transformation programs. While this initial evaluation demonstrates the analyser's ability to assess Eugenia's transformations accurately, it is essential to consider a broader spectrum of test cases. Expanding the evaluation to include a variety of test cases will provide a more comprehensive understanding of the static analyser's capabilities and its adaptability to different transformation scenarios. This extended evaluation will offer a more robust assessment and contribute to a more comprehensive analysis of the analyser's effectiveness in MDE scenarios.

- **Expand Database Compatibility.** We have successfully implemented the partial loading approach using Neo4J, a well-known graph-based database. Although graph-based databases are generally recommended due to their performance advantages in this context, it is worthwhile to explore the possibility of extending support for other types of databases. Various tools are designed to work with different database models, including document-based, relational, or key-value databases, and they, too, can benefit from partial loading. For example, within Epsilon, there is the JDBC [22] driver, which facilitates interactions with relational databases. Investigating the compatibility and potential adaptations of this driver to our partial loading approach could be an interesting effort. By broadening our database compatibility, we open up the opportunity to apply partial loading benefits to a wider range of database systems, thus enhancing the performance of tools and optimising the utilisation of resources.

- **Integration with CDO and NeoEMF Repositories.** In the initial stages of our investigation, we considered implementing our approach on top of CDO and NeoEMF repositories. However, we found that CDO and NeoEMF are structured to provide access to model element properties in a particular manner. CDO offers facilities for greedy access (requesting all properties) and lazy access (requesting one property at a time from the database). Similarly, NeoEMF primarily supports lazy loading. Unfortunately, neither of these repository systems inherently supports our approach's key feature, which involves requesting specific properties from the database based on the program's footprint (i.e., effective metamodel). The success of our evaluation, as outlined in Section 5.4, in demonstrating the efficiency benefits of partial model loading underscores the potential advantages of this approach. We hope that our findings and the demonstrated benefits can motivate the developers of repositories like NeoEMF and CDO to consider incorporating support for this feature in future iterations. Such enhancements could lead to more flexible and efficient usage of these repository systems in a broader range of scenarios.

- **Automated Optimisation of Constraint Grouping Strategies.** As discussed in Section 6.2, the task of grouping constraints becomes complicated when the effective metamodels of these constraints are not identical or do not exhibit a clear superset/subset relationship, yet they share a significant overlap in terms of the elements required for constraint execution.

  One approach involves grouping constraints by considering common parts, which reduces the time spent on loading and unloading uncommon elements. However, this method incurs the cost of loading uncommon parts, making it time-consuming. Alternatively, the other approach suggests not grouping constraints and loading them separately by considering only the non-common parts. This approach prioritises efficiency in loading and unloading, but it comes at the expense of loading and unloading the same information. A trade-off exists between two competing factors: loading more information in a single operation, which leaves less memory available for constraint execution but saves time, and loading elements separately, which requires more loading time but makes more memory available for executing constraints. For instance, consider the scenario depicted in Figure 7.1.

  When executing Constraint 1, if the number of instances of *InPort* is significant enough to occupy a substantial portion of memory, it may not be efficient to load an additional attribute, such as the *type* attribute, as it could negatively impact execution due to memory constraints. On the other hand, loading additional data and executing more constraints could be recommended when the number

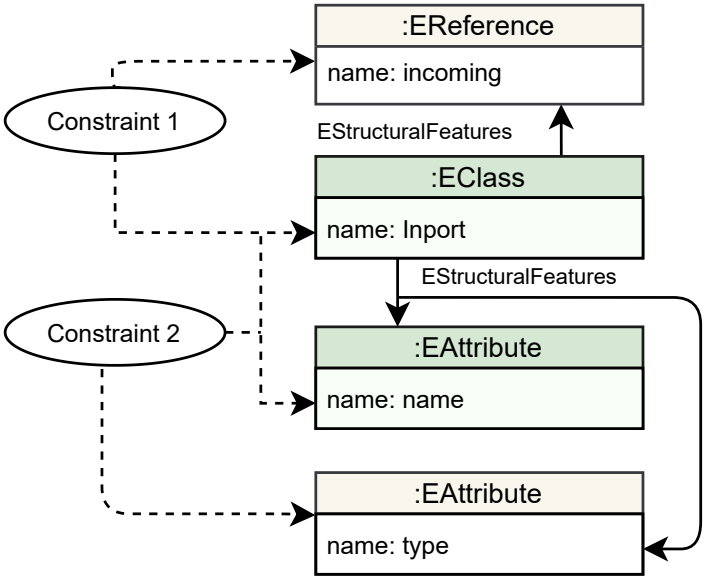


Figure 7.1: Constraints with overlap

of elements occupies a smaller portion of memory. This trade-off highlights the importance of making informed decisions about loading strategies based on the available memory and specific execution requirements.

The partitioning algorithm requires a deeper understanding of the model to make informed decisions regarding partitioning strategies and automatically evaluate the most optimal strategy. However, this level of information is currently unavailable within our approach. The static analyser operates at the metamodel level and lacks specific knowledge about the model and its contents. To address this gap, the integration of a model analyser becomes essential. Such a tool can provide the necessary insights and awareness for our proposed algorithm to make well-informed decisions about grouping constraints. With access to detailed information about the model's structure and content, the model analyser may enable automatic constraint grouping, enhancing the efficiency and effectiveness of our partitioning approach.

- **Optimising Parallel Constraint Execution on Distributed Systems.** Looking ahead, one potential avenue of research involves exploring the use of partial loading and the strategic grouping of constraints within validation programs to enhance parallel constraint execution on distributed systems. At present, challenges persist when executing constraints in parallel. Constraints are distributed across multiple machines, each loading the complete model, regardless of the specific constraints they are assigned to execute. This approach results in inefficient interactions, impacting both loading and model validation processes in the execution engine.

Consequently, we face increased model loading times and unnecessary memory consumption across all machines.

The concept of leveraging partial loading and grouping constraints offers the promise of more efficient execution. Partial loading entails loading only the information relevant to the constraints allocated to each machine, reducing loading times and optimising overall execution. In addition, a strategic grouping of constraints can enhance efficiency further. The current inefficiencies in memory utilisation can lead to wasted resources and execution failures when the model size surpasses available heap memory, hindering the execution engine from loading the model and running the program on a single machine. The future adoption of these approaches could be a valuable direction for research, promising to reduce execution time and maximise system performance in the context of parallel constraint execution on distributed systems.

- **Integration with Runtime Optimisation.** Our current efforts have primarily revolved around enhancing efficiency in the loading process and conserving memory resources for model management programs. However, we see the potential for further improvements by integrating these approaches with runtime optimisation strategies.

  For example, there exists valuable research in query optimisation and incremental execution techniques that can significantly aid the execution engine in optimising the execution process. These techniques aim to enhance the runtime performance of programs and queries, ensuring they execute more efficiently and effectively. By combining our work on loading efficiency and memory management with runtime optimisation strategies, we can explore the synergies between these areas, potentially achieving substantial advancements in the execution of model management programs. This collaborative effort promises to deliver not only efficient loading and memory management but also optimised runtime performance, contributing to a holistic approach for improving the overall execution of model management programs.

8 References

[1] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers Principles, Techniques & Tools*. 2007.

[2] *Apache HBase*. https://hbase.apache.org/. Accessed: 2023-07-26.

[3] *API Blueprint. A powerful high-level API description language for web APIs*. https://apiblueprint.org/. Accessed: 2023-07-26.

[4] Konstantinos Barmpis and Dimitrios S Kolovos. "Comparative analysis of data persistence technologies for large-scale models". In: *Proceedings of the 2012 Extreme Modeling Workshop*. 2012, pp. 33–38.

[5] Konstantinos Barmpis and Dimitrios S. Kolovos. "Comparative Analysis of Data Persistence Technologies for Large-Scale Models". In: *Proceedings of the 2012 Extreme Modeling Workshop*. XM '12. Innsbruck, Austria: Association for Computing Machinery, 2012, pp. 33–38.

[6] Amine Benelallam et al. "Neo4EMF, a scalable persistence layer for EMF models". In: *European Conference on Modelling Foundations and Applications*. Springer. 2014, pp. 230–241.

[7] Kristopher Born et al. "Analyzing Conflicts and Dependencies of Rule-Based Transformations in Henshin". In: *Fundamental Approaches to Software Engineering*. Ed. by Alexander Egyed and Ina Schaefer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 165–168. ISBN: 978-3-662-46675-9.

[8] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2017.

[9] Jordi Cabot and Ernest Teniente. "Incremental evaluation of OCL constraints". In: *Advanced Information Systems Engineering: 18th International Conference, CAiSE 2006, Luxembourg, Luxembourg, June 5-9, 2006. Proceedings 18*. Springer. 2006, pp. 81–95.

[10] *CDO Model Repository Overview*. https://eclipse.dev/cdo/documentation/. Accessed: 2023-07-24.

[11] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. "AnATLyzer: An Advanced IDE for ATL Model Transformations". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 85–88. ISBN: 9781450356633. DOI: 10.1145/3183440.3183479. URL: https://doi.org/10.1145/3183440.3183479.

[12] *Cypher Query Language*. https://neo4j.com/developer/cypher/. Accessed: 2023-07-27.

[13] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. “Prefetchml: a framework for prefetching and caching models”. In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. 2016, pp. 318–328.

[14] Gwendal Daniel et al. “NeoEMF: A multi-database model persistence framework for very large models”. In: *Science of Computer Programming* 149 (2017), pp. 9–14.

[15] John Daniels. “Modeling with a sense of purpose”. In: *IEEE software* 19.1 (2002), pp. 8–10.

[16] *EMF Ecore Javadoc*. https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html#details. Accessed: 2023-07-26.

[17] *Epsilon Documentation*. https://eclipse.dev/epsilon/doc/. Accessed: 2023-07-26.

[18] *Epsilon Users*. https://www.eclipse.org/epsilon/users/. Accessed: 2023-10-27.

[19] Javier Espinazo Pagán, Jesúss Sánchez Cuadrado, and Jesús García Molina. “Morsa: A Scalable Approach for Persisting and Accessing Large Models”. In: *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2011, pp. 77–92.

[20] *Eugenia*. https://eclipse.dev/epsilon/doc/eugenia/. Accessed: 2023-08-22.

[21] Eclipse Foundation. *Eclipse Modelling Framework (EMF)*. https://eclipse.org/modeling/emf/. Accessed: 2023-07-20.

[22] Automated Software Engineering Research Group. *Relational Databases (JDBC)*. https://github.com/epsilonlabs/emc-jdbc/. Accessed: 2023-10-18.

[23] *Henshin*. https://projects.eclipse.org/projects/modeling.emft.henshin. Accessed: 2023-07-31.

[24] John Hutchinson et al. “Empirical assessment of MDE in industry”. In: *Proceedings of the 33rd international conference on software engineering*. 2011, pp. 471–480.

[25] *IBM Engineering Systems Design Rhapsody – Developer*. https://www.ibm.com/products/uml-tools. Accessed: 2023-07-20.

[26] Ari Jaaksi. “Developing mobile browsers in a product line”. In: *IEEE software* 19.4 (2002), pp. 73–80.

[27] Frédéric Jouault et al. “ATL: A model transformation tool”. In: *Science of Computer Programming* 72.1 (2008). Special Issue on Second issue of experimental software and toolkits (EST), pp. 31–39. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2007.08.002. URL: https://www.sciencedirect.com/science/article/pii/S0167642308000439.

[28] Juha Kärnä, Juha-Pekka Tolvanen, and Steven Kelly. “Evaluating the use of domain-specific modeling in practice”. In: *Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling*. 2009.

[29] Dimitriois Kolovos. “An extensible platform for specification of integrated languages for model management”. PhD thesis. Citeseer, 2008.

[30] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. “The epsilon object language (EOL)”. In: *European conference on model driven architecture-foundations and applications*. Springer. 2006, pp. 128–142.

[31] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. “The epsilon transformation language”. In: *Theory and Practice of Model Transformations: First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008 Proceedings 1*. Springer. 2008, pp. 46–60.

[32] Dimitrios S Kolovos 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.

[33] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. “On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages”. In: *Rigorous Methods for Software Construction and Analysis: Essays Dedicated to Egon Börger on the Occasion of His 60th Birthday*. Ed. by Jean-Raymond Abrial and Uwe Glässer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 204–218.

[34] Jochen Ludewig. “Models in software engineering–an introduction”. In: *Software and Systems Modeling* 2 (2003), pp. 5–14.

[35] Sina Madani, Dimitris Kolovos, and Richard F Paigea. “Towards optimisation of model queries: A parallel execution approach”. In: *Journal of Object Technology* (2019).

[36] *MAGICDRAW*. https://www.3ds.com/products-services/catia/products/no-magic/magicdraw/. [Online; 2023-07-20].

[37] *Mango Database*. https://www.mongodb.com/. Accessed: 2023-07-31.

[38] *MOF Query/View/Transformation*. https://www.omg.org/spec/QVT/1.3/About-QVT. Accessed: 2023-07-31.

[39] Parastoo Mohagheghi and Vegard Dehlen. “Where is the proof?-a review of experiences from applying mde in industry”. In: *Model Driven Architecture–Foundations and Applications: 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings 4*. Springer. 2008, pp. 432–443.

[40] *Neo4j Graph Database*. https://neo4j.com/product/neo4j-graph-database/. Accessed: 2023-07-26.

[41] *Object Constraint Language*. http://www.omg.org/spec/OCL/. Accessed: 2023-07-31.

[42] Louis Rose et al. “Genericity for model management operations”. In: *Software & Systems Modeling* 12 (2013), pp. 201–219.

[43] Shane Sendall and Wojtek Kozaczynski. “Model transformation: The heart and soul of model-driven software development”. In: *IEEE software* 20.5 (2003), pp. 42–45.

[44] *Simulink is for Model-Based Design*. https://www.mathworks.com/products/simulink.html. Accessed: 2023-07-20.

[45] Jean-Sébastien Sottet, Frédéric Jouault, et al. “Program comprehension”. In: *5th International Workshop on Graph-Based Tools (GraBaTs 2009)*. Citeseer. Zurich (Switzerland), 2009.

[46] Klaas-Jan Stol and Brian Fitzgerald. “The ABC of Software Engineering Research”. In: *ACM Trans. Softw. Eng. Methodol.* 27.3 (Sept. 2018). ISSN: 1049-331X. DOI: 10.1145/3241743. URL: https://doi.org/10.1145/3241743.

[47] Zoltán Ujhelyi. “STATIC ANALYSIS OF MODEL TRANSFORMATIONS”. MA thesis. May 2009.

[48] *Unified Modeling Language*. https://www.uml.org/. Accessed: 2023-07-20.

[49] Tamás Vajk et al. “Runtime model validation with parallel object constraint language”. In: *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*. 2011, pp. 1–8.

[50] Ran Wei and Dimitris S Kolovos. “Automated Analysis, Validation and Suboptimal Code Detection in Model Management Programs.” In: *Bigmde@ Staf*. 2014, pp. 48–57.

[51] Ran Wei et al. “Partial loading of XMI models”. In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. 2016, pp. 329–339.

[52] Edward D. Willink. “Modeling the OCL Standard Library”. In: *ECEASST* 44 (2011). DOI: 10.14279/tuj.eceasst.44.663.

[53] Edward D. Willink. “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. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pp. 81–88. URL: http://ceur-ws.org/Vol-1512/paper07.pdf.

[54] Marc Zeller, Daniel Ratiu, and Kai Höfig. “Towards the adoption of model-based engineering for the development of safety-critical systems in industrial practice”. In: *Computer Safety, Reliability, and Security: SAFECOMP 2016 Workshops, ASSURE, DECSoS, SASSUR, and TIPS, Trondheim, Norway, September 20, 2016, Proceedings 35*. Springer. 2016, pp. 322–333.