"This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 813884"

Project Number: 813884

Project Acronym: Lowcomote

Project title: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms

Scalable Low-Code Artefact Persistence and Query

Project GA: 813884

Project Acronym: Lowcomote

Project website: https://www.lowcomote.eu/

Project officer: Thomas Vyzikas

Work Package: WP5

Deliverable number: D5.3

Production date: September 30th 2022

Contractual date of delivery: September 30th 2022

Actual date of delivery: October 22nd 2022

Dissemination level: Public

Lead beneficiary: University of York

Authors: Sorour Jahanbin, Qurat ul ain Ali, Jolan Philippe, Benedek Horváth

Contributors: Lowcomote partners

Reviewers: Luca Berardinelli, Jean-Marie Mottu

Proposal Abstract

Low-code development platforms (LCDP) are software development platforms on the Cloud, provided through a Platform-as a-Service model, which allow users to build completely operational applications by interacting through dynamic graphical user interfaces, visual diagrams and declarative languages. They address the need of non-programmers to develop personalised software, and focus on their domain expertise instead of implementation requirements.

Lowcomote will train a generation of experts that will upgrade the current trend of LCDPs to a new paradigm, Low-code Engineering Platforms (LCEPs). LCEPs will be open, allowing to integrate heterogeneous engineering tools, interoperable, allowing for cross-platform engineering, scalable, supporting very large engineering models and social networks of developers, smart, simplifying the development for citizen developers by machine learning and recommendation techniques. This will be achieved by injecting in LCDPs the theoretical and technical framework defined by recent research in Model Driven Engineering (MDE), augmented with Cloud Computing and Machine Learning techniques. This is possible today thanks to recent breakthroughs in scalability of MDE performed in the EC FP7 research project MONDO, led by Lowcomote partners.

The 48-month Lowcomote project will train the first European generation of skilled professionals in LCEPs. The 15 future scientists will benefit from an original training and research programme merging competencies and knowledge from 5 highly recognised academic institutions and 9 large and small industries of several domains. Co-supervision from both sectors is a promising process to facilitate agility of our future professionals between the academic and industrial world.

Deliverable Abstract

Over the last few years, Low-Code Development Platforms (LCDPs) are evolving at high speed. For the use of LCDPs at a larger scale, they should be able to scale well in terms of the size of model, execution time, and memory consumption. As LCDPs are used for larger software domains, the underlying models grow large as well, and this pushes the current generation of low-code and model management tools and technologies to their limits.

When model management runs on pay-as-you-go cloud-based resources, inefficiency and reduced scalability incur an additional cost. Hence, there is vested interest from vendors of cloud-based low-code platforms for efficient and scalable model management tools.

In this deliverable, we present two approaches for efficient model loading and query optimisation. Firstly, we present an approach for partial loading of large models that leverages sophisticated static program analysis of model management programs to identify, load, process relevant model parts – instead of naively loading the entire model into memory. Secondly, an approach based on compile-time static analysis and specific query optimizers/translators is presented to improve the performance of complex queries over large-scale heterogeneous models.

In this way, using intelligent run-time partitioning approach, model management programs will be able to process system models faster with a reduced memory footprint and resources will be freed that will allow them to accommodate even larger models. The query optimization approach aims to bring efficiency in terms of execution time in a way that developers can express model management programs in a technology-agnostic form but still benefit from technology-specific optimisations when compared to the naive query execution for low-code platforms.

Contents

1 Introduction

Low-code platforms use Model-Driven Engineering (MDE) [1] processes such as domain specific languages and code generation to develop applications. In the traditional software development process, models are used for documentation and design, while in MDE, which is an established approach of software engineering [2], models play a crucial role as they are considered as first-class artifacts which drive the software development. In MDE, models are treated as first-class citizens of the development process to enhance productivity [3, 4], maintainability, consistency, and traceability [5].

Many industrial projects attempt to represent the system with models that minimise accidental complexity and use concepts which are close to the domain [5, 6]. Though there are several low-code platforms available like OutSystems[1], Mendix[2] and ZAppDev[3], there are still open challenges limiting the broader adoption of low-code platforms in the industry. One of the main challenges in model-driven environments, including low-code platforms, is scalability [7, 8]. As software systems become more complex, underlying system models grow proportionally in both size and complexity. To keep up, model management languages and their execution engines need to provide increasingly more sophisticated mechanisms for making the most efficient use of the available system resources. Efficiency is particularly important when model-driven technologies are used in the context of low-code platforms where all model processing happens on pay-per-use cloud resources.

In partial loading of large low-code system models, we present our approach [9] that leverages sophisticated static program analysis of model management programs to identify, load, process relevant model parts – instead of naively loading the entire models into memory. In this way, model management programs will be able to process system models faster with a reduced memory footprint, and resources will be used efficiently allowing them to accommodate even larger models.

In heterogeneous models query optimisation, we propose an approach which aims to bring efficiency, when compared to the naive query execution for low-code platforms. Models can be stored and represented in different back-end technologies in low-code systems. To query such heterogeneous models, tailored high-level query languages are used. These languages help interact with heterogeneous technologies but typically have a significant impact on performance. We proposed novel techniques and algorithms for optimisation of queries operating on low-code system models captured using different modelling languages and model representation formats. We proposed approaches for optimisation (1) of queries over MySQL models by translating EOL queries to MySQL queries [10]; (2) of queries over EMF models by (a) custom indices [11], (b) translation of EOL to Viatra [12]; (3) of model-to-model transformation by selective traceability.

This report is submitted as a final-deliverable for the Lowcomote project. The rest of this report is organised as: Section 2 proposes a NeoEMF-based approach for loading serializable models. Section 3 introduces a partial loading approach for large file-based and repository-based models using algorithms and presents the result of the evaluation of our approach compared to the state of the art. Section 4 describes the research challenges related to model querying and then briefly presents the proposed solution with the corresponding evaluation results. Section 5 concludes the report and presents the future research directions.

[1] https://www.outsystems.com
[2] https://www.mendix.com
[3] http://www.zappdev.com

2 Efficient Loading of Serializable Models

Handling very large models rapidly becomes challenging due to the capacity of tools to deal properly with a big amount of data. While several solutions to persist EMF models exist, most of them do not allow partial model unloading and cannot handle models that exceed the available memory. Furthermore, these solutions do not take advantage of the graph nature of the models: most of them rely on relational databases, which are not fully adapted to store and query graphs. Neo4EMF [13] is a persistence layer for EMF that relies on a graph database and implements an unloading mechanism. The main purpose of NeoEMF is to face scalability issues on large-scale models [14]. On the side, additional modules allow developer to use additional persistence solution. For instance, NeoEMF I/O deals with file solution, as the standard XML Metadata Interchange (XMI) files. Moreover, distributed solutions whose purpose is to access and interact efficiently with large-scale models, take advantage of clustering several machines largely increasing the quantity of resources allocated to a computation. The multi-language engine for executing data engineering Apache Spark, distribute objects by streaming. In this section, we propose an extension of NeoEMF for dynamically loading objects, independently of EMF artifacts. Figure 2 gives an overview of how we defined a side ecosystem to load models using the NeoEMF I/O module. The related work presented in Section 3 gives an overview of other existing approach about efficient model loading.

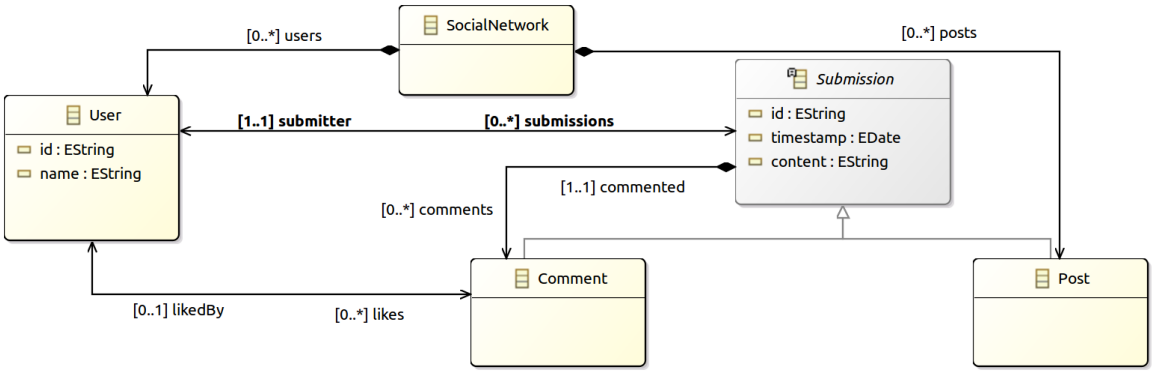

Figure 1: The metamodel of a social network (TTC 2018).

2.1 Motivating example

Social networks grow fast. For instance, the social network Facebook from the Meta company rapidly turned into a trillion edges graph [15]. In Figure 1 we show the simple metamodel for the social graph that we will use in this section. The metamodel has been originally proposed at the Transformation Tool Contest (TTC) 2018 [16], and used to express benchmarks for model query and transformation tools. In this metamodel, two main entities belong to a `SocialNetwork`. First, the `Post`s and the `Comment`s that represent the `Submission`s, and second, the `User`s. Each `Comment` is written by a `User`, and is necessarily attached to a `Submission` (either a `Post` or another `Comment`). Besides commenting, the `User`s can also like `Submission`s. The first task to interrogate a such model is to load its content. The native EMF loader shows very poor performance, and is not able to handle fast loading. For models of only 30k elements and 60k links, the loading is approximately 20 seconds. This amount of time exponentially grows with the number of elements and links. A second drawback of using the EMF loader is its performance strongly depends the model complexity. Since the attributes and references may refer to other objects of the model, a resolution is needed. The complexity of the resolving part depends on the model, and may need a deep investigation in the full model to be solved. It is then not reasonable to use EMF for loading large-scale models, as we can find in social network applications.

2.2 Contribution to NeoEMF

Figure 2 describes the integration of NeoEMF in modeling solutions ecosystem. Standard modeling users using model-based applications which provide high-level modeling features such as a graphical interface, interactive console, or query editor. These features internally rely on EMF's Model Access API to navigate models, perform CRUD operations, check constraints, etc. Modelers might also want to distribute their computation to improve performances and increase computing capacity. Depending on the targeted solution, the input differs on how it is interpreted by the engine. Indeed, most of Java frameworks for distributed computing deals with objects who must implement the `java.io.Serializable` interface. For instance, SparkTE [17], a model transformation engine based on Spark[4], designed in the context of the Lowcomote project, deals with Java serialized elements for communicating between computational nodes. In

[4]https://spark.apache.org

the other case, that is using model-based tools using EMF modeling solution, EMF delegates the operations to a persistence manager using its Persistence API, which is in charge of the serialization/deserialization of the model.

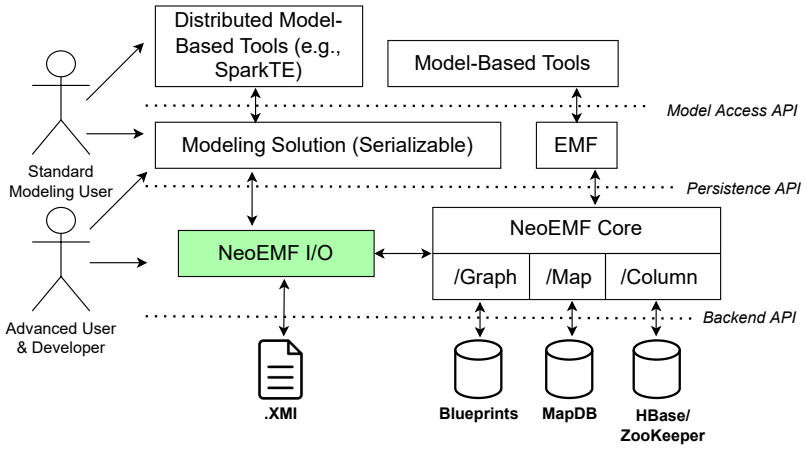

Figure 2: NeoEMF extension integration in a model-based development environment.

Once the core component has received the modeling operation to perform, it forwards the operation to the appropriate database driver (Map, Graph , or Column), which is in charge of handling the low-level representation of the model. These connectors translate modeling operations into Backend API calls, store the results, and reify database records into EMF `EObjects` when needed. Our contribution aims at providing an abstract class `EventListener4Serializable` in the NeoEMF I/O module, that is completely independent of EMF core components. A NeoEMF `EventListener` defines the behaviour for parsing elements as they are read during file reading. To use the features of our new listener, a user defines two behaviours: (i) one for element, overriding `createElement` and (ii) for links overriding `createLink`. An overview of this listener and a concrete implementation for the case presented in Section 2.1 are respectively presented in Listing 1 and Listing 2.

Listing 1: EventListener4Serializable interface.

```
abstract class EventListener4Serializable[E <: Serializable, L <: Serializable]
extends EventListener {
  ...
  def createElement(classname: String, id: Long): E
  def createLink(referenceName: String, parentId: Long, targetIds: List[Long]): L
}
```

Labels for class names and references names are defined as constants, corresponding to the names that are used in the metamodel. Also to allow both the distribution of elements and links, to have a Spark distributed graph, links must be defined as a triplet of source ID, a label and a target ID. The link's structure is independent of elements, but must used valid IDs, existing in the rest of the model.

Our modeling solution presents elements as simple dynamic objects, composed by three fields: (i) a class name, (ii) item ID as a long value, (iii) a map of attributes. In the other side, links are triplets of a source ID, a link label, and a list of target IDs. Compared to EMF, there is a lost of information which would make the serialization impossible. For instance, containment of objects or reversed reference are not naturally serializable and are then removed in our modeling solution.

Listing 2: An example usage of EventListener4Serializable for TTC18 case.

```
class SocialNetworkListener
extends EventListener4Serializable[SocialNetworkElement,SocialNetworkLink] {

  override def createElement(classname: String, id: Long)
  : SocialNetworkElement = {
    classname match {
      case USER => new SocialNetworkUser(id)
      case POST => new SocialNetworkPost(id)
      case COMMENT => new SocialNetworkComment(id)
      case _ =>
    }
  }

  override def createLink(referenceName: String, parentId: Long, targetIds: List[Long])
  : SocialNetworkLink = {
    referenceName match {
      case COMMENT_LIKEDBY => new CommentLikedBy(parentId, targetIds)
      case COMMENT_POST => new CommentPost(parentId, targetIds.head)
      case COMMENT_SUBMISSION => new CommentSubmission(parentId, targetIds.head)
      case SUBMISSION_COMMENTS => new SubmissionComments(parentId, targetIds)
      case SUBMISSION_SUBMITTER => new SubmissionSubmitter(parentId, targetIds.head)
      case USER_FRIENDS => new UserFriends(parentId, targetIds)
      case USER_LIKES => new UserLikes(parentId, targetIds)
      case USER_SUBMISSIONS => new UserSubmissions(parentId, targetIds)
      case _ =>
    }
  }
}
```

2.3 Experimental results

This section presents the time it takes for loading models with different sizes and complexity. These times are compared to times to load the same models using the native EMF resource loader. The experiments have been conducted on a shared memory machine with a Intel Core i7-8650U having 8 cores at 1.90GHz and a memory of 32GB. The machine was running Ubuntu 22.04 LTS. We use Java 13, Scala 2.12 with Spark 3.1.0. The used model and their respective sizes are described in Table 1. The smallest model is composed of 1274 elements and 2163 links, while the biggest is made of 859114 elements and 1589908 links. Each recorded timing is the average of 30 executions.

Dataset							**Loading time**	
id	#elements	#links	#users	#posts	#comments	#likes	ext. NeoEMF	native EMF
1	1274	2163	80	554	640	6	61ms	69ms
2	2071	3548	889	1064	118	24	93ms	184ms
3	4350	7594	1845	2315	190	66	146ms	450ms
4	7530	14422	2270	5056	204	129	270ms	878ms
5	15132	27886	5518	9220	394	572	413ms	3921ms
6	30396	56261	10929	18872	595	1598	491ms	19473ms
7	58076	111197	18083	39212	781	4770	1858ms	82535ms
8	115121	218823	37228	76735	1158	13374	2699ms	389953ms
9	224816	424901	1678	74668	148470	36815	3700ms	1711354ms
10	443323	812515	2606	167299	273418	102276	8863ms	> 5 minutes
11	859114	1589908	3699	314510	540905	2684	19201ms	> 15 minutes

Table 1: Loading time of models from TTC18.

Clearly, our solution outperforms EMF native loading mechanism. Because we only pass through the file once, loading elements as they are read, the computation time is proportional to the size of the file. In the contrary, EMF concretely resolves references making its loading mechanism time depending on the topology of the input model.

3 Partial Loading of Models

In MDE, models are manipulated using model management programs that carry out different tasks such as model transformation, model merging, model validation, etc. As models grow in size MDE tools face scalability problems. One of them is the ability of execution engines of model management languages to scale up for models of growing size in terms of execution time and memory usage [18].

The primary reason for this scalability issue lies in the way that most of contemporary technologies interact with models. For example, when a model management program needs to load and process (e. g. transform, validate) a model, if the model is a file-based model (e. g. XMI), then all the required information from it needs to be read upfront, before the execution of the program. If the model is stored in a database (e. g. Neo4j), then we can issue multiple queries to the database to fetch model elements of interest (and their properties) progressively, as they are needed for the execution of the program.

To summarise, the interaction of model management program execution engines with large models (file-based or repository-based) can be too "short-sighted" in the absence of static-analysis-based model loading and caching mechanisms. This can result in increased model loading times and unnecessary memory consumption. This section introduces an approach that can help execution engines of model management programs handle larger models more efficiently. In our approach, by using in-advance knowledge about the program provided by static analysis, execution engines are able to identify and load only parts of model that are likely to be accessed by the program.

3.1 Research Objectives

Objectives: Compared to general-purpose programming languages, dedicated languages such as OCL, ATL and ETL provide a more concise/tailored syntax and additional opportunities for analysis and optimisation. As low-code software systems become more complex, underlying system models grow proportionally in both size and complexity. Existing model management program execution engines evidently struggle with very large models [19]. The aim of this project is to design and implement next-generation execution engines for model management languages, which will leverage sophisticated static program analysis to identify, load, process and transparently discard relevant model partitions [20]– instead of naively loading the entire models into memory and keeping them loaded for the duration of the execution of the program.

Expected results: This project will enable model management languages and engines to reduce the overhead of loading unimportant parts of models (i.e. parts that they will never access) and of unnecessarily keeping obsolete parts (i.e. parts that have already been processed and are guaranteed not to be accessed again) in memory. In this way, model management programs will be able to process low-code system models faster and with a reduced memory footprint, and resources will be freed that will allow them to accommodate even larger models. For example, a model compiler that only exercises 20% of a model, will have the capacity to process models that are five times bigger with the same memory footprint.

The goal of this research is to propose methods for reducing the time of loading and memory footprint when model management tasks are applied on large size models. This overall goal will be divided into the following specific purposes:

1. Provide a static analysis facility for getting in-advance knowledge of the parts of any type of model, which are likely to be exercised by the model management program.
2. Propose an approach for loading only necessary parts of the model into memory.
3. Design an algorithm for partitioning models in an efficient way for loading and unloading necessary parts of models on demand.
4. Propose a strategy for collecting model elements from memory which are no longer referenced by the program.

3.2 Motivating Example

As a motivating example, consider a model that conforms to a contrived Project Scheduling Language (PSL), the UML class diagram of which is shown in Figure 3. 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).

Consider a model that conforms to the PSL language (see Figure 3), and on which we wish 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)[5] [21] as shown in Listing 3.

Epsilon[6] is a platform that provides task-specific languages for common model management activities such as model transformation, code generation, merging, validation, and refactoring. The core language of

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

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

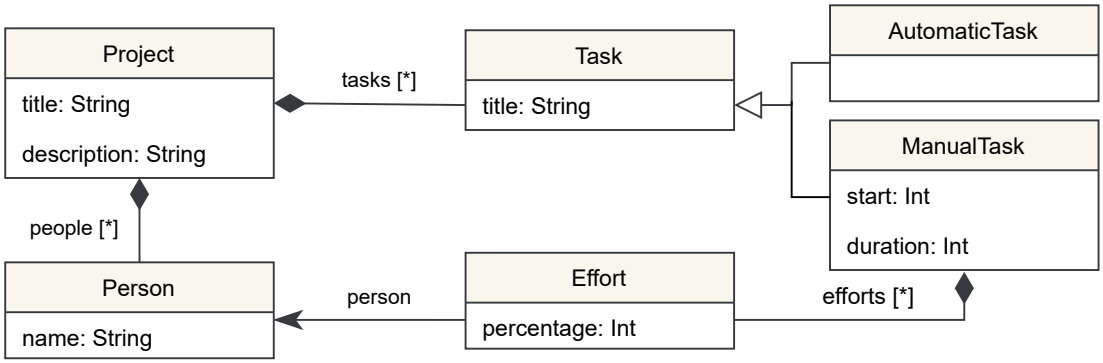

Figure 3: PSL Metamodel.

the platform is the Epsilon Object Language (EOL), which is a model-oriented programming language that provides common facilities for developing task-specific model management languages.

Listing 3: EOL program to print number of people who contribute to each task.

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

In Epsilon's architecture, there is an Epsilon Connectivity Layer (EMC)[7]) which enables Epsilon programs to interact with models in different modelling technologies in a uniform manner by defining the drivers (e. g., EMF, CDO, NeoEMF).

In Listing 3, 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 shown in Listing 3 only accesses the *title* attribute and *efforts* reference of *ManualTask*, the *person* reference of *Effort* and no other properties of the PSL metamodel. If the input model of the program is file-based (e. g. XMI), as the execution engine does not know in advance which parts of the model the program will access, the entire model needs to be loaded into memory. To run this program against a repository-based (e. g., stored in a database-backed repository such as CDO or NeoEMF) model, EOL engine uses the respective driver to fetch all instances of model element types and properties of model elements on demand.

Therefore, 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, there are two alternatives at runtime: either *greedily* fetch all properties and attributes of model elements retrieved from the database or *lazily* fetch 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 3 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: Using this approach, in line 1, only skeletons of *ManualTask* elements would be initially fetched from the database. Then in line 3, for each *ManualTask*, the program would need to go back to the database and fetch the value of its *efforts* reference. So, multiple round-trips to the repository to fetch the value of the attribute or references of each model element are required. These trips are time-consuming.

The former strategy favours execution time over memory consumption, while the second strategy requires less memory, but potentially multiple round-trips to the repository (detrimental to performance).

3.3 Proposed Approach

This section introduces an approach which helps execution engines of model management programs to handle large models more efficiently. The general goal of this approach is to reduce loading time and memory footprint, which is achieved by using static analysis for generating an execution plan. The proposed approach includes three main steps illustrated in Figure 4.

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

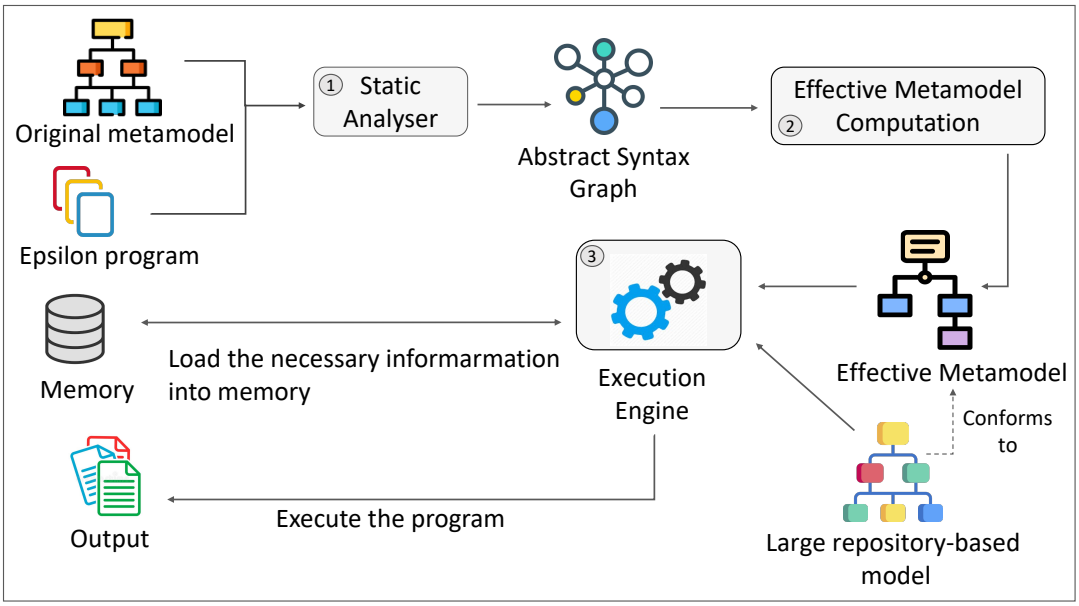

Figure 4: The proposed approach.

3.3.1 Static Analysis

In the first step of our approach (in Figure 21), a model management program and the metamodels of the models it consumes are provided as the 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 an abstract syntax graph [22]. Using the abstract syntax graph, the static analyser can extract relevant information (i.e., types and properties accessed by the program).

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 program [22] (see Section 3.3.2).

While static analysis supports multiple models (and therefore produces multiple effective metamodels), in the remainder of the section we will only consider programs with one model (and therefore one effective metamodel). To illustrate how every step of the approach works, we use the motivating example of Section 3.2.

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 3, 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 2 shows the resolved types of expressions which are extracted from Listing 3 by the static analyser.

Line number in Listing 3	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)

Table 2: Resolved Types Calculated by the Static Analyser.

3.3.2 Effective Metamodel Computation

The second step of the approach is the extraction of the effective metamodel of the model consumed by the program from its abstract syntax graph. The concept of effective metamodel was introduced by Wei et al. [23]. The effective metamodel is constructed with an algorithm (Algorithm 1 which is described later in the section) that uses the resolved types of expressions and contains only types which are 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 apply to other metamodelling technologies too.

As shown in Figure 5, an effective metamodel consists of an *EffectiveMetamodel* class with *name* and *nsuri* attributes. The *EffectiveMetamodel* class is connected to an *EClass* that has *EStructuralFeature*s. The *EffectiveMetamodel* class is connected to an *EClass* by *allOfKind*, *allOfType* and *types* references.

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 of which should be loaded only when they appear in the references of model elements of interest.

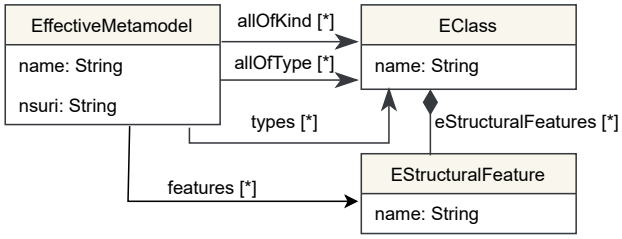

Figure 5: The Structure of Effective Metamodel (adapted from [23]).

For every class used in the program (such as *ManualTask* and *Effort*), 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.

The algorithm which extracts this effective metamodel from an EOL program is described in Algorithms 1 and 2. This algorithm is easily extendable for other Epsilon languages but considering the motivating example, we will discuss the version that works with EOL program in this section.

In Algorithms 1 and 2, the Abstract Syntax Graph which is extracted by the static analyser is visited. Algorithm 1 is interested in calls of *all()* and *allInstances()* operations and *property call*s (such as *ManualTask.efforts*) 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 1 to add the respective *EClass*es to the effective metamodel. If the target of the operation call is a model element type, then it will be added to *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 1 handles property calls to further populate the effective metamodel. In Algorithm 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 the 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).

Figure 6 illustrates the effective metamodel extracted from the EOL program in our motivating example (Listing 3).

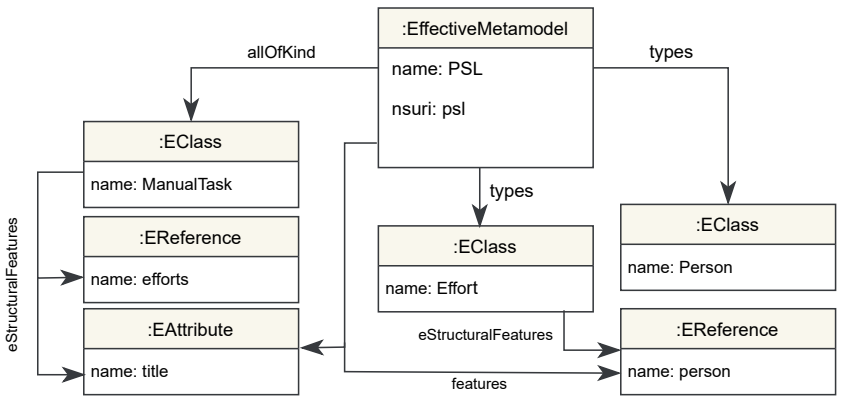

Figure 6: Effective Metamodel of the EOL Program (Listing 3).

In Figure 6, 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 EOL program, all instances of *ManualTask* must be loaded. The *ManualTask* class is added to *EffectiveMetamodel* under the *allOfKind* reference according to lines 7-11 of Algorithm 1. The *title* attribute of *task* is added to the *EffectiveMetamodel* according to lines 8-9 of Algorithm 2. The *efforts* reference of *ManualTask* is also required (line 3 of Listing 3), hence, it is added to *ManualTask* as an *EReference* according to lines 10-11 in Algorithm 2. The resolved type of *efforts* reference is equal to *Effort* (see Table 2), so according to lines 12-13 in Algorithm 2, 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.

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

```
1:  procedure computeEffectiveMetamodel
2:      while No further changes are made to the effective metamodel do
3:          let EM = New effective metamodel;
4:          for all operation call expression do
5:              if the type of the target of the operation call is a model element type then
6:                  let EC = retrieve existing EClass for the target model element type;
7:                  if the name of the operation is all(), allOfKind() or allInstances() then
8:                      if EC is already under EM's allOfType or types reference then
9:                          move EC under EM's allOfKind reference;
10:                     else
11:                         add EC under EM's allOfKind reference (if not already there)
12:             else if the name of the operation is allOfType() then
13:                 if EC is not already under EM's allOfType or allOfKind reference then
14:                     add EC under EM's allOfType reference;
15:         for all property call expression do
16:             if the type of the target of the property call is a model element type then
17:                 let EC = retrieve existing EClass for the model element type of target
18:                 handlePropertyCallExpression(propertyCallExpression,EC)
19:             else if the type of the target of the property call is a collection type then
20:                 if the collection's content type is model element type then
21:                     let EC = retrieve existing EClass for the model element type of the collection's content type
22:                     handlePropertyCallExpression(PropertyCallExpression,EC)
23:                 else if the collection's content type is Any then
24:                     for all EClasses in EM do
25:                         handlePropertyCallExpression(PropertyCallExpression,EClass)
26:             else if the type of the target of the property call is Any then
27:                 for all EClasses in EM do
28:                     handlePropertyCallExpression(PropertyCallExpression,EClass)
```

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

```
1:  procedure handlePropertyCallExpression(propertyCallExpression, modelElementType)
2:      if the name of property is "all" then
3:          Consider it as a call to allOfKind in operation call expresion as it is the short term of allInstances()
4:      else
5:          let EC = retrieve existing EClass for the model element type
6:          if EC is not already under the EM's types, allOfType or allOfKind references then
7:              add EC under EM's types reference;
8:          if the property is an attribute then
9:              add the property to EC's structural features as an EAttribute (if not already there)
10:         else if the property is a reference then
11:             add the property to ET's structural features as an EReference (if not already there)
12:             let EType = the type of the reference
13:             add EType under types reference
```

3.3.2.1 Accommodating Untyped Variables and Expressions

Algorithm 1 visits the abstract syntax graph to consider all of statements and expressions in the code. Thus, constructing the 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.

For example, in Listing 4, the *title* of first *Project* of input model is printing. All instances of *Project* with *title* attribute and all instances of *Task* are required for running this part of the program.

Listing 4: EOL Example Code.

```
1 var p = Project.all().first();
2 var t = Task.all().first();
3 p.title.println("Title: ");
```

In the first line of Listing 4, the first item of all instances of *Project* 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, condition in line 22 of Algorithm 1 is not satisfied and *title* attribute of *Project* is not added to the effective metamodel in the first iteration.

Algorithm 1 handles this situation by considering possible (instead of precise) types for variables and expressions. While this is the case for this minimal example, in the general case, the type of variables assigned in more than one place in a program cannot be resolved reliably.

In lines 1-2 of Listing 4, according to lines 9-16 of Algorithm 1, *Project* and *Task EClass*es are added to the effective metamodel. Then, the *title* attribute is accessed by the program but as the resolved type of *p* is *Any*, it adds the *title* attribute to all *EClass*es that are already in the effective metamodel. Hence, according to lines 8-9 in Algorithm 2, *title* will be added to the effective metamodel for the *Project* and *Task* EClasses and the execution engine will load the *title* of all instances of *Project* and the *title* of all instances of *Task* into memory.

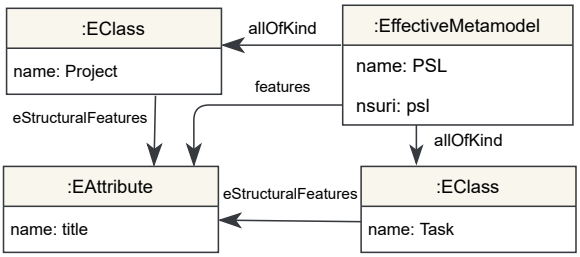

Figure 7: Extracted Effective Metamodel for Listing 4.

As shown in Figure 7, the effective metamodel includes an attribute which is not necessary for running the program (*title* of *Task*) but because of lack of information before the execution, the execution engine loads more information from the repository to be on the safe side. Loading more information and running the program is preferable to loading less information than required at runtime. As during the execution of the Epsilon program, more *EClass*es may be added to the effective metamodel, in line 2 of Algorithm 1, there is a loop which calculates the effective metamodel until the point that no further changes are made to it. As the time consumption of extracting the effective metamodel is negligible, repeating the algorithm will not have a great impact on the efficiency of our approach.

3.3.3 Partial Model Loading

Once the effective metamodels of the different models accessed by a program have been computed, the next step is to use this information in the execution engine to load model partially. The way that execution engine interacts with models depends on their persistence format. In this section, first we discuss partial loading of file-based models and then we focus on partial loading of repository-based models.

3.3.3.1 XMI Model Partial Parser

If the input model is a file-bsed model, then instead of using a default parser of XMI which loads whole model into memory, we need a partial parser.

Once the effective metamodels of the different models accessed by a program have been computed, the next step is to make them available to the partial XMI parser. This is done by calling *load()* function of XMIN instead of default *load()* method of EMF driver.

In EMF, by calling *load()* method, all content of the model will be loaded into memory, however, in XMIN, effective metamodels are computed first, and then the model will be loaded based on the information provided by effective metamodels. As EMF's default XMI parser loads the entire model into memory, for partial loading there is a need for an XMI parser that can load models based on the information provided by an effective metamodel. For this purpose, we have used the parser described in [23] with fixing some bugs that we have done to improve the performance of the parser. Therefore, in step 3 of Figure 4, the parser will load the model based on *Java* effective metamodel, and the execution engine executes the EOL program using parts of models that are loaded into memory.

3.3.3.2 Program Execution

We implement this approach in the form of a new Epsilon driver called XMIN, which is an extension of EMF driver and provides a facility for EOL programs to load EMF models frugally. Epsilon's architecture includes an abstraction layer (Epsilon Connectivity Layer (EMC)[8]), which enables Epsilon programs to interact with models in different modelling technologies in a uniform manner.

Evaluation
In this section, we report on the results of experiments that measure the performance of XMIN against that of the existing Epsilon EMF driver (which uses EMF's built-in XMI parser) in terms of model loading time and memory consumption.

For our experiments, we implemented 34 EVL constraints[9] inspired by the FindBugs[10] tool, which detects a number of "code smells" in models reverse engineered from Java code.

For test models, we used a number of big (reverse-engineered form) Java models from Lintra[11] project, generated using MoDisco [24]. The size and number of elements of the models that were used are summarised in Table 3.

Name	Size (MB)	Number of elements
EclipseModel-1.0	184.1	1,470,286
EclipseModel-2.0	365.5	2,957,627
EclipseModel-3.0	627.7	5,057,953
EclipseModel-all	795.4	6,451,757

Table 3: Models used for evaluation.

We generate three sets of EVL constraints from FindBugs program (Table 4). Each of these sets requires different subsets of models due to a different number of constraints that they validate. *Findbugs-Set1* validate models applying six constraints, and the effective metamodel which is extracted from this program has 52 effective types. Considering Java metamodel that contains 132 types, XMIN parser will load model based on effective metamodel which contains 60% less number of classes compare to Java original metamodel.

Name	Effetive metamodel
Findbugs-Set1	52 classes
Findbugs-Set2	86 classes
Findbugs-all	111 classes

Table 4: EVL constraints sets.

We validate all of models (in Table 3) using *Findbugs-Set1* constraints. For executing *Findbugs-Set1*, different percentage of models elements need be loaded which is as follow:

- for validating eclipsemodel 1.0, 14.09% of model,
- for validating eclipsemodel 2.0, 14.20% of model,

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

[9]https://github.com/Sorour-j/org.eclipse.epsilon/blob/Sorour/org.eclipse.epsilon.partialloading/src/org/eclipse/epsilon/TestUnit/Parser/java findbugs.evl

[10]http://findbugs.sourceforge.net

[11]http://atenea.lcc.uma.es/projects/LinTra.html

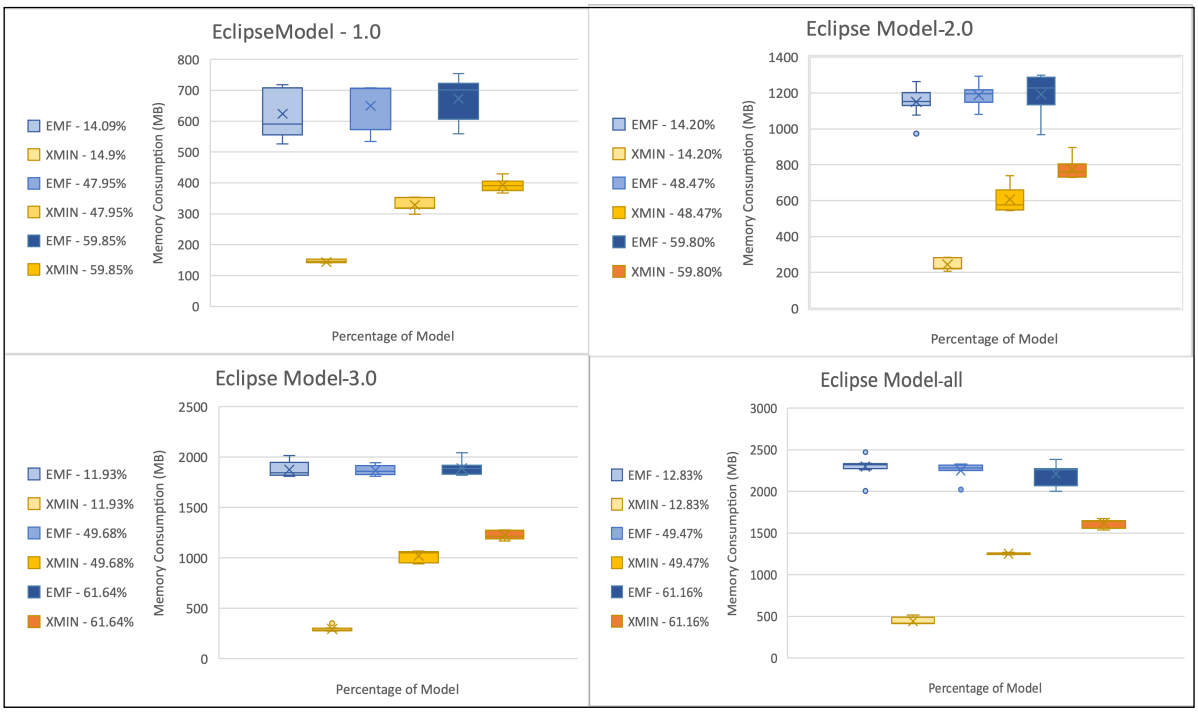

Figure 8: Memory consumption results.

- for validating eclipsemodel 3.0, 11.93% of model,
- for validating eclipsemodel all, 12.83% of model.

The XMIN driver loads models partially based on the types and features used in the EVL constraints. As it is shown in Figure 8, using EMF driver, 100% of EclipseModel-1.0 will be loaded, and 650MB (on average) of memory will be consumed. In contrast, memory consumption is about 150MB (on average) for loading 14.09% of the same model (EclipseModel-1.0) using XMIN driver; which means 77% less memory consumption for loading the same model for executing the same set of constraints using two different drivers (EMF and XMIN).

In the same way, we executed other sets of EVL constraints (in Table 4) against the other three models in Table 3. Then, we recorded loading time and memory consumption and plotted them against the percentage of loaded model elements.

We measured memory consumption and time 12 times for validating each model (overall 108 times) and collected the results illustrated in Figure 8 and Figure 9.

In our results, memory consumption shows a linear behaviour with regard to the percentage of required/loaded model elements. As shown in Figure 8, even when only a subset of the model is required for evaluating the EVL constraints, EMF's XMI parser still loads the entire model. Thus, memory consumption remains practically constant regardless of the set of EVL constraints evaluated on the models.

In contrast, XMIN only loads the elements which are required. Therefore, when only 14.20% of the model (in EclipseModel 2.0 chart) is necessary for executing, model loading through XMIN requires 84% less memory (200 MB compares to 1200MB). As the percentage of required elements increases, memory consumption increases with it.

XMIN has the same behaviour in terms of loading time. When the number of required model elements is smaller, loading time is lower, in contrast to Epsilon's EMF driver, where loading time is constant.

Regarding the XMIN driver's correctness, we validated all models against the full set of constraints with both drivers and verified that the satisfied and unsatisfied constraints produced in all execution pairs were identical. This means that the XMIN driver loads all the information that the constraints need to access from each model.

Logically when 100% of the model should be loaded, XMIN will require more time and memory due to the overhead of extracting effective metamodel. Thus, maybe switching to EMF driver in this situation would be more reasonable.

3.3.3.3 Neo4j Query Generation

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. **Mapping EMF Models to Graph Databases**

In this work we are concerned with the efficient management of models that reside in graph-based model

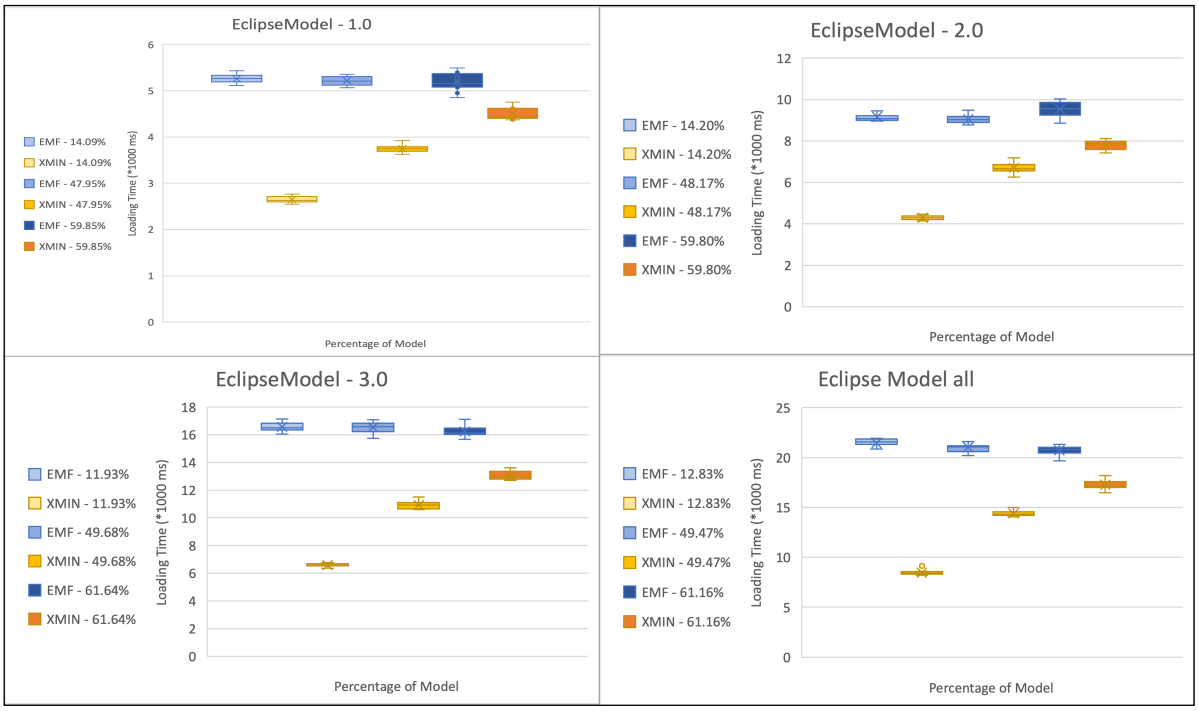

Figure 9: Loading time results.

repositories as these have been shown to outperform model repositories backed by relational/document databases [13, 25]. A state-of-the-art graph-based model repository is NeoEMF, which enables the persistence of EMF-based model 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 10 shows a model which conforms to the PSL metamodel in Figure 3. 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*.

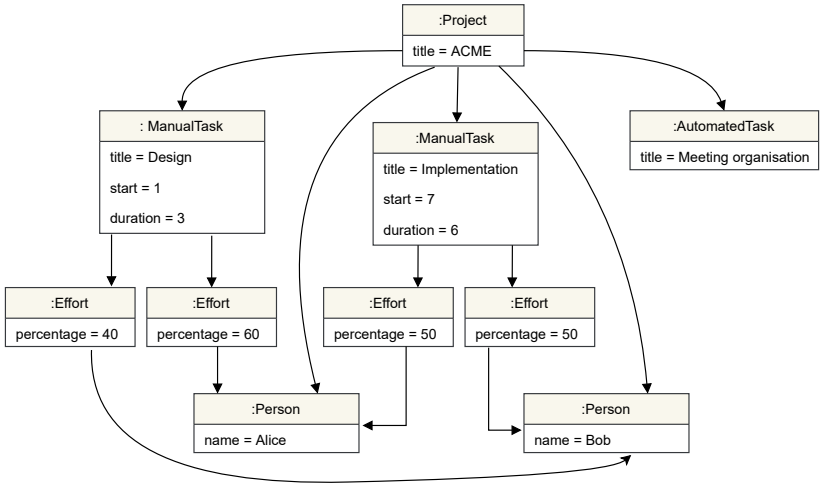

Figure 10: PSL Model of the ACME Project.

In order to map EMF-based models such as this one to a Neo4J graph, we use the mapping strategy illustrated in Figure 11. 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 11, *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 11, 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 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 11, there is an *Edge* in 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*.

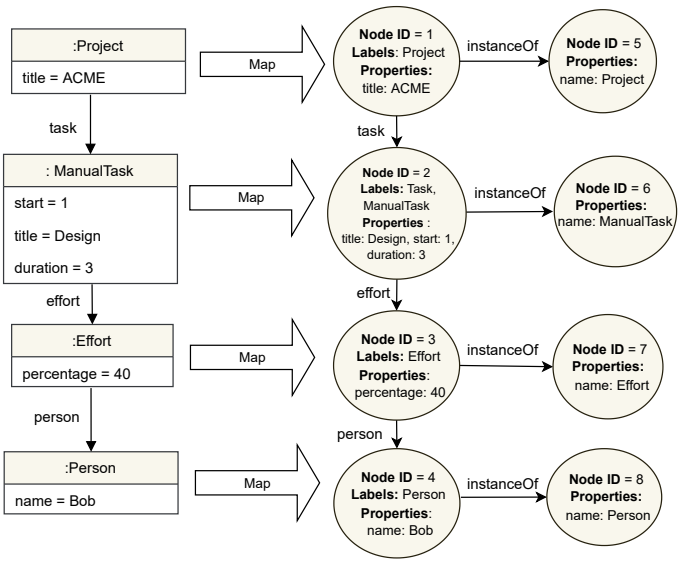

Figure 11: Mapping the Model of Figure 10 into a Neo4J Graph.

Model Loading

Cypher 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 model for running the program.

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 the matched edge
RETURN	node.property	return the certain property(es) of the matched nodes

Table 5: Cypher Expressions

The Cypher expressions that are used for loading information in our approach are listed in Table 5 (*var* stands for variable).

The first column of Table 5 lists the Cypher expressions, the second column is 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 are generated based on the effective metamodel in Figure 6.

Listing 5: Generated Cypher Queries According to Effective Metamodel in Figure 6.

```
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]
6 ->(effortType)
7 RETURN ID(effort), effortType.name
8 OPTIONAL MATCH (effort:Effort)-[personRefEffort:person]->(person:Person),(person)-[personins:instanceOf]
9 ->(personType)
10 RETURN ID(person), personType.name
```

Considering the effective metamodel in Figure 6, all instances of *ManualTask* are required for running the program (EOL code in Listing 3). 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 loads all *ManualTask* nodes.

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 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 fail to run and it does not return any result. With

OPTIONAL MATCH on the other hand if there is no match, query will be run and "null" will be returned as a value of the edge which 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, *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[12] 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 labeled 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 which are not necessary for executing the program. For example, considering the PSL metamodel (Figure 3), there is no relationship between nodes with *AutomatedTask* and *Person* labels. So, in Listing 6, 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 database will be equal to *n*m*. However, if Listing 6 is separated into two *MATCH* clauses, then the number of records will be *(n+m)*, which is more efficient.

Listing 6: A Cypher Query that Generates 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 node in the effective metamodel, will be returned. Considering the generated queries in Listing 5, 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 7.

Listing 7: Optimised queries of Listing 5.

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

- **Reduce Cardinality**: Since nodes are labeled by the type of their respective model element and its super types, the results of some queries can overlap. For example, in Listing 8 all nodes with *Task* and *ManualTask* label are matched and the *title*s of matched nodes are returned. Considering the part of the graph in Figure 12, the nodes returned in line 2 are "Design", "Implementation" and "Meeting organisation" nodes since they are labeled as *Task*. The titles of the nodes that are returned in line 4, are "Design" and "Implementation" nodes which are *ManualTask*-labeled 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. Algorithm 3 generates Cypher queries automatically based on the effective metamodel.

Listing 8: Queries with Overlap.

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

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

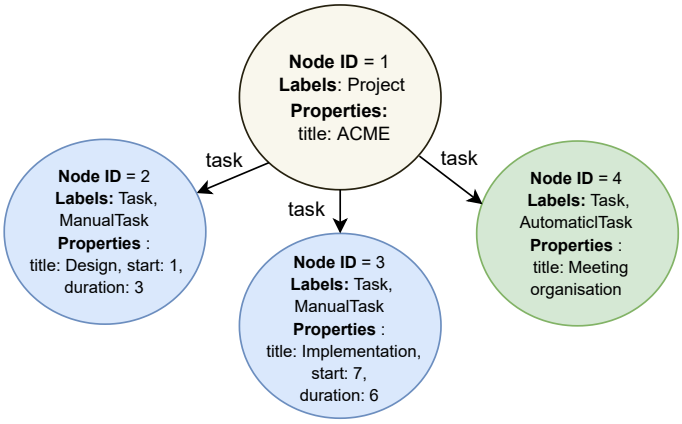

Figure 12: A part of graph model.

Algorithm 3 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:      Add ID of EClass and name of instanceOf reference of EClass to return set
9:      call HANDLEFEATURES (EClass)
10:     let query ="";
11:     query ← "MATCH"+ query
12:     for all item in match array do
13:         query ←item + ",";
14:     query ← "OPTIONALMATCH"+ query
15:     for all item in optionalMatch array do
16:         query ←item + ",";
17:     query ← "RETURN"+ query
18:     for all item in return array do
19:         query ←query + item + ",";
```

3.3.3.4 Program Execution

After retrieving information from the database file, this information is used by the execution engine to run the EOL program. In this approach, we use EMF driver of Epsilon to run the EOL program to avoid implementing a new driver with almost same functionality. Therefore, in the last step of our approach, in the execution engine, the results of queries are wrapped as an EMF input and the program is run by Epsilon EMF driver.

Evaluation

In this section, we report on the results of experiments that measure the performance of our approach against that of NeoEMF in terms of execution time and memory consumption.

We evaluated our approach by 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 [26]. 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 the Neo4j (version 4.4.3) embedded databases.

We have also implemented a query in the Epsilon Object Language (EOL) inspired by one of the GraBaTs[13] test cases. This query finds all classes that declare public static methods whose return type is the

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

Algorithm 4 Cypher queries generation (2 of 2).

```
1:  procedure HANDLEFEATURES(EClass)
2:      Add the EClass to visitedClasses
3:      for all features in EClass.eStructuralFeatures do
4:          if feature is an EAttribute then
5:              add the feature to return set
6:          else if feature is an EReference then
7:              let type = Type of reference value
8:              Add ID of reference's target to return set
9:              optionalMatchString ← (EClass.name) − [reference.name] → (type)
10:             if Eclass is not in visitedClasses and EClass is EM but not in allOfKind then
11:                 handleFeatures(type)
```

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”

Table 6: Result for the Execution of Generated Query in Listing 7.

containing class itself (i. e. like the *getInstance* method of the *Singleton* pattern).

For our experiments, we ran the EOL program against Set0-Set4 graphs in the database using our approach and NeoEMF and measured the execution time and memory consumption. The results are shown in Table 7. The results were computed after 5 warm-up iterations and represent the average over 10 executions of the program. For instance, for Set1, it takes 8.3 s and 118 MB of memory to run the GraBaTs query using our approach while for NeoEMF, the average time is equal to 9.5 s and memory consumption is 656.2 MB which is about 5.5 times higher than our approach. The most significant difference in memory usage is in Set3 case, where the memory footprint of our approach is 93 % lower than NeoEMF.

The charts illustrated in Figures 13 and 14 show the linear behaviour of two approaches. In Figure 13, in the Set0 model, as the model is not very large, the overhead of effective metamodel extraction and loading data from database in our approach is not compensated at runtime and the execution time is better for NeoEMF. 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 14, both approaches have a linear behaviour, and our approach consumed less memory compared to NeoEMF. From Set0 to Set4, as the size of model increased, the memory consumption grew in two approaches.

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

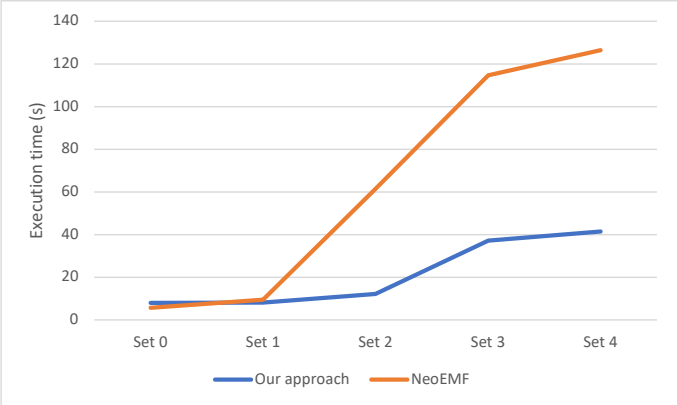

Figure 13: Execution time comparison of our approach and NeoEMF.

Regarding correctness, we validated all models by executing each program with two approaches (our approach and NeoEMF) and verify that the output produced by all execution pairs should be equivalent (e. g. in the number of people for each task in EOL and NeoEMF).

Model	Our Approach		NeoEMF	
	Execution Time (s)	Execution Memory (MB)	Execution Time (s)	Execution Memory (MB)
Set0 (70 447 model elements)	8.1	108	5.7	319
Set1 (198 466 model elements)	8.3	118	9.5	656.2
Set2 (2 082 841 model elements)	12.2	210	61.5	2537.5
Set3 (4 852 855 model elements)	37.3	232.8	114.6	3718
Set4 (4 961 779 model elements)	41.5	318.1	126.4	2987.2

Table 7: Experiment Results.

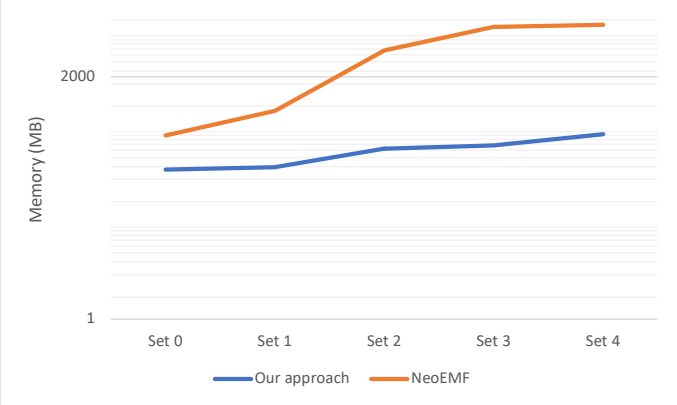

Figure 14: Memory comparison of our approach and NeoEMF (logarithmic scale).

3.4 Related Work

Providing infrastructure for storing and indexing large models is an essential aspect of scalable MDE. Scalable Model Persistence researches are divided into two subcategories [27]: *Efficient Model Storage* and *Model Indexing*.

In the case of efficient model storage, the common format which is currently used for storing models is XML Metadata Interchange. While XMI is a suitable way of storing small models, it has some limitations when it deals with large models.

A limitation we are facing in XMI format is the lack of support for partial loading. It means that in order to access any element of model, the whole of model file should be parsed and loaded to memory first. Many state-of-art modelling tools such as EMF have this issue, which is not efficient in terms of time and memory consumption.

Another issue is about XMI file size. As XMI is a verbose XML-based format, the XMI model files are larger than needed for storing the information they do. To address these limitations, Jouault et al. [28] introduced an open binary format known as Binary Model Syntax (BMS), and they claimed that initial experiments show that BMS files are three times smaller than corresponding XMI files. Also, due to lazy loading support and persistence indexing, BMS format is more efficient to access model elements, and it can be a high-performance alternative to XMI. However, while BMS was introduced in 2009, there has not been any update or release of this format in the public domain until now.

There are works related to model indexing. Some of the related works in this field are related to repositories. Repository is considered as a persistence solution that is remotely accessible by tools and users. Model bus [29] is a basic model repository based on EMF that allows modelling services to be connected. While the back-end provides useful features, it only offers limited scalability (in terms of model size) as it does not support partial access to models.

The Connected Data Object (CDO) [30] is another model repository for EMF models and metamodels. It is also a framework built on top of the EMF, which provides the persistence of large models. Figure 15 provides an overview of CDO architecture.

As shown in Figure 15, CDO supports persistence, which means users can store their models in all kind of major databases back-ends like ODB, NoSQL and RDB and transforms models in all of the supported back-end types quickly.

Another property 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.

However, one concern with CDO is that it implements its own version control management system, and regarding industry adoption, using a separate version control system for models only is not practical. Moreover, although CDO claims to be able to load models up to 4 GB, experimental evaluation with Intel

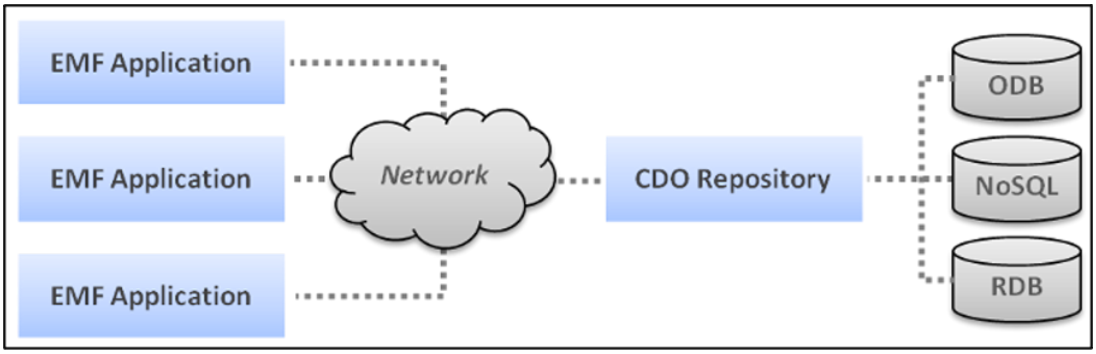

Figure 15: CDO Architecture [30].

CoreI5 760 PC at 2.80 GHz with 8 GB of physical RAM in [31] reported an upper bound of 271 MB.

Morsa [31] is a persistence solution for storing and accessing large models based on on-demand strategies, which is supported by the NoSQL database. Figure 16 illustrates an overview of Morsa architecture.

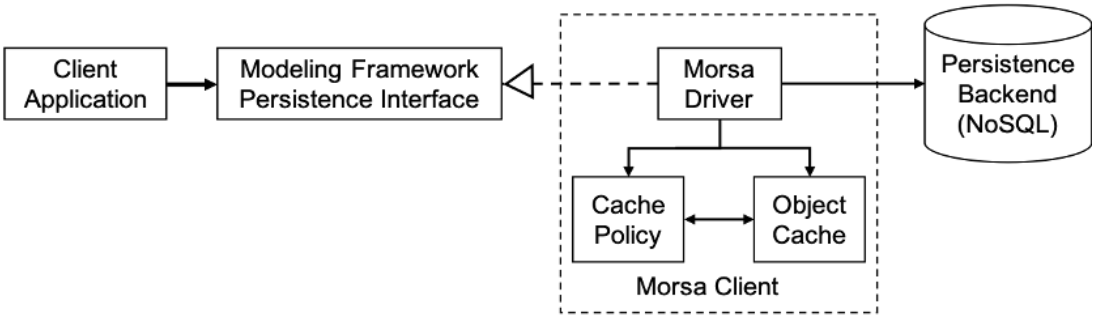

Figure 16: Morsa Architecture [31].

Morsa driver allows client applications to access models through the modeling framework persistence interface. Further, Morsa provides clients with a partial load of large 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 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.

Morsa satisfies scalability requirements, but it is just a prototype, and there is no update that they plan to consider crucial issues like security in order to deploy this prototype in an industrial context. Also, choosing the cache policy is manual, and the user should select the policy using a GUI [31].

Neo4EMF [13] is a persistence layer for EMF models. It is built on top of Neo4j that is a graph-based database, as these databases are able to manage large-scale data on highly distributed environments. Moreover, Barmpis and Kolovos [32] suggest that NoSQL databases would provide better scalability and performance than relational databases due to the interconnected nature of models. Neo4EMF is similar to Morsa in several aspects (notably in on-demand loading), but it aims at exploiting the optimized navigation performance offered by graph-databases. While Neo4EMF is a more performant alternative to XMI due to high-performance access and on-demand loading, its raw performance do not surpass a more mature solution like CDO.

SmartSAX is another prototype which was introduced in [23]. It supports partial loading of XMI model files. That is, instead of loading the entire file when we need even an element of the model, it makes it possible to load the elements of model based on need. The main idea is providing in-advance knowledge of program (which kind of model elements and which of their properties are accessed by the model management program) can be used to partially load only a subset of XMI-based EMF model into memory. While in full XMI loading, the time consumption and memory footprint are practically negligible for loading of small XMI models but for large models in the industrial context, it would be problematic.

SmartSAX also has some limitations. First, it just supports read-only XMI files (does not support changes made to partially loaded model). Also, as partial loading can affect the internal structure of the XMI model, elements should have IDs that do not depend on their position in the containment hierarchy. Finally, SmartSAX currently does not support loading models that are persisted in multiple XMI files.

In [33], Daniel et al. propose 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. While the rules to describe the event conditions to activate the prefetch, the objects to prefetch, and the customisation of the cache policy are defined by designers in PrefetchML, the automatic generation of PrefetchML scripts based on static analysis of available queries

and transformations for the meta-model would be more efficient in term of optimization.

Although recent research has made advancements in this area, existing solutions have clear shortcomings in accessing and processing large models. The first shortcoming is about loading models. Repositories such as Morsa, CDO provide remote access of large models and store them in a graph-based or relational database. Still, as some tools are based on EMF, and the common format for storing models is XMI, there is a need for partial access to XMI models, which loads models using on-demand strategies. In addition, loading and storing models by elements is not an efficient way, so the second challenge is about partitioning models. Intelligent strategies for grouping model elements as a partition are needed. Finally, intelligent unloading strategies are needed. Keeping part of models loaded into memory that will not be used further increases the memory footprint unnecessarily. Hence, unloading them when the program does not refer to them anymore would be a solution for reducing the memory consumption of model management programs.

4 Query Optimisation

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

The main shortcoming of dedicated model management languages compared to general-purpose languages such as Java is performance. While widely-used general-purpose languages are typically compiled and benefit from advanced runtimes offering advanced features such as adaptive optimisation and microarchitecture-specific speed-ups, model management languages are predominately interpreted, and therefore their execution speed is substantially lower. This can become a scalability bottleneck as models grow in size and inhibit their applicability to projects that involve large models [34, 35].

4.1 Research Objectives

Objectives: As software systems become more complex, underlying models in LCEPs grow proportionally in both size and complexity. Such models can be persisted in a variety of proprietary or standard formats (such as XMI), and in different types of back-ends (e.g. file systems, relational databases, document databases). High-level, concise and tailored model query languages such as OCL and EOL can be used to shield query developers from the intricacies of the underlying model formats/back-ends but this typically has a significant impact on performance. Recently, we have shown how sophisticated runtime query optimisation can be used to drastically improve the execution time of high-level OCL-style queries executed over models stored in relational [36] and non-relational [37] databases. The objectives of this project are to: (1) investigate the applicability of runtime query optimisation techniques to a wide range of model persistence formats and back-ends, (2) identify reusable optimisation primitives and patterns across different formats and back-ends, and (3) evaluate the obtained benefits in terms of performance and memory footprint.

Expected results: This project will produce novel techniques and algorithms for optimisation of queries operating on low-code system models captured using different modelling languages and model representation formats. It will also produce an open-source prototype that will implement the identified algorithms and techniques on top of existing model query languages. While the precise performance benefits will depend on the nature of individual queries and the underlying model representation formats, based on our preliminary results in [36], we expect an increase of at least one order of magnitude in query execution time for certain classes of queries (e.g. filtering all instances of a type). The breakdown of the overall research objective is as follows:

RO-1: Identify the performance challenges involved in executing complex queries over large models represented in heterogeneous formats (such as XMI etc.) and stored in different back-ends (Simulink and relational databases etc.).

RO-2: Identify reusable optimisation primitives and patterns across different formats and back-ends using static analysis of high-level language code.

RO-3: Investigate the applicability of compile-time query optimisation techniques to a wide range of model persistence formats and back-ends.

RO-4: Propose algorithms for optimisation of queries operating on low-code system models captured using different modelling languages and model representation formats.

RO-5: Evaluate the results of proposed algorithms in terms of execution time and memory footprint over various back-end technologies.

Considering the research challenges and objectives, we propose an approach for optimising queries operating over heterogeneous low-code system models. The primary purpose of this framework is to be able to automatically rewrite expensive queries in an input model management program written in technology-agnostic language operating over heterogeneous models to a more efficient form. The rewritten program should be efficient in terms of execution time. The rewriting process will consider memory footprint to make a trade-off. Program rewriting is based on information extracted through static analysis. The rewriting

and optimisation will vary based on the specific backend technology or technologies the query is operating over. To our knowledge, we have not found a solution to this problem in the literature.

In a low-code platform, the underlying models can be of different modeling technologies and stored in different backend formats. After a model management program is parsed, an Abstract Syntax Tree (AST) is generated. This AST may not include any type information attached to its nodes. Before execution, the static analyser component will analyse the program and populate type-related information into the AST, also referred to as an Abstract Syntax Graph. Abstract Syntax Graph would then be used by the rewriters involved in a program (depending on the type of models it needs to access), to rewrite this program behind the scenes into an optimised form.

4.2 MySQL Models

Listing 9: Example EOL program interacting with MySQL DB.

```
model DB driver MySQL {
   server = "localhost",
   port = "3306",
   database = "requirements",
   username = "root",
   password = ""
}
var functionReq = MySql!Requirement.all.select(m | m.type = "functional");
```

As a preliminary query optimisation idea, we have implemented query translation for a subset of EOL expressions as in Listing 9 to MySQL queries as in Listing 10. A part of this is presented in [10]. This translation is done before the execution of program. Query translation enables to query large-scale relational databases while still using tailored model query languages. In an MDE platform, sometimes some information such as requirements about another model are stored in relational databases. So for tasks like cross-validating a Simulink model and its requirements stored in a MySQL database, there is a need to interact with relational database using model management language.

Listing 10: Rewritten EOL program interacting with MySQL DB.

```
model DB driver MySQL {
   server = "localhost",
   port = "3306",
   database = "requirements",
   username = "root",
   password = ""
}
var functionReq = MySql.runSql("Select * from Requirement where type = functional");
```

4.3 Custom Indices

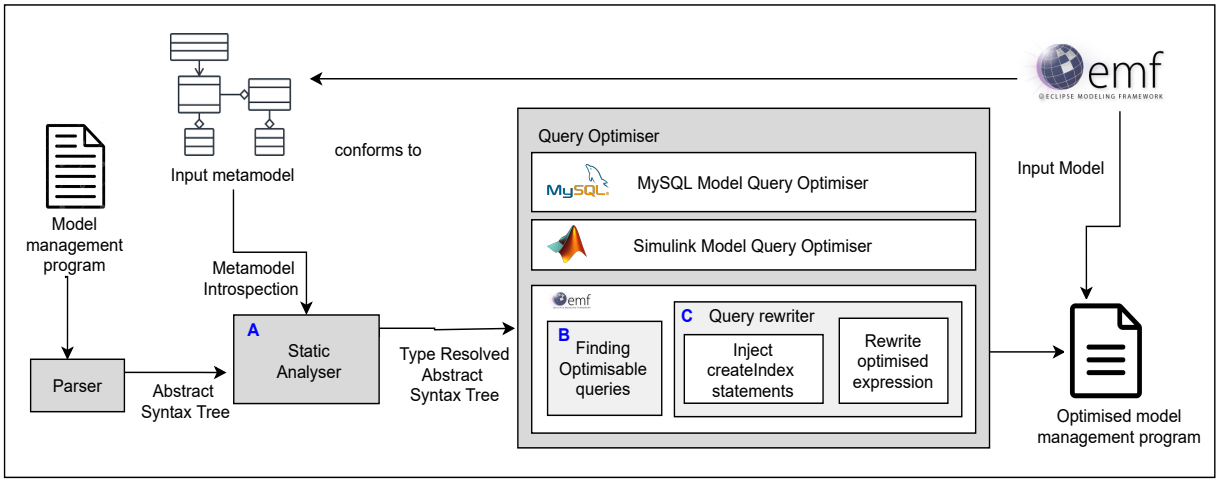

Figure 17: Proposed approach for custom indices.

We introduce an architecture for improving the execution speed of interpreted model management programs written in languages of the Epsilon platform, using static analysis and program rewriting techniques. The architecture of the proposed approach is illustrated in Figure 17. We then demonstrate an application of this architecture for detecting repeated queries on all instances of types in EMF-based models and for speeding-up their execution through the construction of relevant indices. We have evaluated the proposed optimisation technique using large models that have been reverse engineered from Java code and a set of existing constraints.

4.3.1 Motivating Example

In this section, motivation behind this work will be presented through an example scenario. Let us consider an example where we query a model for the purpose of validating it. The considered model is conforming to the UML2[14] EMF-based metamodel. An excerpt of the UML2 metamodel is shown in Figure 18. In this example, suppose we wish to validate the model on the following constraints:

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

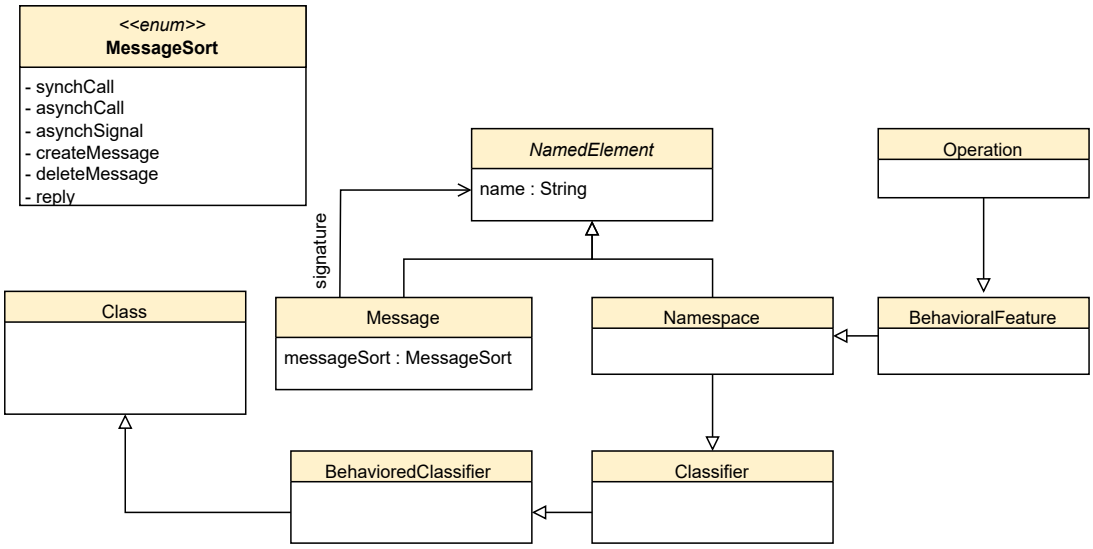

Figure 18: Excerpt of UML metamodel.

The implementation of the two above-mentioned constraints (using the Epsilon Validation Language) is presented in Listing 11 respectively. These validation scripts can conceptually be written in any model management language. Here we consider EVL, as it allows cross-validation between models of various backend technologies. It can take benefit of heterogeneous model query optimisation in a single program.

In Listing 11, the *UniqueName* constraint checks that for every *Class* in the model, its *name* attribute is unique (lines 7- 12). Similarly, *IsCalledInSequenceDiagram* constraint checks that every *Operation* is called in a sequence diagram at least once (lines 13-18). An *Operation* call is represented by *Message* in a sequence diagram.

Epsilon languages use *all* as an alias for *allInstances()*. In Listing 11, *Class.all* in line 9 and *Message.all* in line 15 retrieve all instances of *Class* and *Message* anywhere in the model respectively.

Evaluating these constraints over a large UML model containing a large number of classes and operations would be computationally expensive. Theoretically, the complexity of *UniqueName* constraint is *O(N*N)* if we consider the number of Classes to be *N*. The complexity of evaluating *IsCalledInSequenceDiagram* over *M* operations and *P* messages would be *O(M*P)*.

This sort of reverse navigation is a recurring issue in model management programs [38] when working with EMF models. A common workaround to reduce complexity in such scenarios is to define opposite references (e.g we could define an opposite reference from *NamedElement* to *Message*) however this pollutes the metamodel and in the case of standard meta models (e.g. such as the UML2 metamodel used in this example) adding opposite references is not an option. Moreover, we have to either anticipate the needs of future model management programs when we're constructing the metamodel or to naively add opposites for all references in the metamodel.

Listing 11: Example EVL validation constraints before optimisation.

```
1  model UML driver EMF {
2      nsuri = "http://www.eclipse.org
3          /uml2/5.0.0/UML"
4  };
5  pre {
6  }
7  context Class {
8      constraint UniqueName {
9          check: not Class.all.exists
10             (c|c.name = self.name and self != c)
11     }
12 }
13 context Operation {
```

[14] https://www.eclipse.org/modeling/mdt/?project=uml2

```
14      constraint IsCalledInSequenceDiagram {
15          check: Message.all.exists
16              (m | m.signature = self)
17      }
18  }
```

To speed up this type of model validation, one optimisation strategy is to programmatically create in-memory indices and then use them for look-ups. Existing languages such as Acceleo offer different facilities for this e.g. search for *eInverse*[15]. Another approach for using reverse navigation in OCL is shown in [38]. This can significantly reduce the complexity compared to the naive iteration through all instances of the relevant model element types. Such an optimised validation program is depicted in Listing 12. These constraints are semantically equivalent to the ones in Listing 11 but are much faster to execute. In Line 6 of Listing 12, an index is constructed which maps names to lists of classes with their name attribute, rather than naively iterating through all the instances of *Class*. Similarly, in Line 7, an index is constructed which maps names to lists of messages with their signature attribute, rather than naively iterating through all the instances of *Message*. Then in constraints, these constructed in-memory indices (Lines 11-13, 18-20) are searched instead. As in-memory indices can be stored as hashmaps, finding UML classes by names and similarly finding messages by signature, the computation cost would be that of a hash function. Considering the complexity of hash functions being *O(1)*, the overall complexity of both the constraints would be reduced to *O(N)* and *O(M)*, respectively. Another possible optimisation could be translating to the model management languages that supports incremental approach.

Listing 12: Example EVL validation constraints after optimisation.

```
1  model UML driver EMF {
2      nsuri = "http://www.eclipse.org
3          /uml2/5.0.0/UML"
4  };
5  pre {
6      UML.createIndex("Class", "name");
7      UML.createIndex("Message", "signature");
8  }
9  context Class {
10     constraint UniqueName {
11         check: not UML.findByIndex
12             ("Class", "name", self.name)
13             .select(c|c.self != c).size() > 0
14     }
15 }
16 context Operation {
17     constraint IsCalledInSequenceDiagram {
18         check: UML.findByIndex("Message",
19             "signature", self).size() > 0
20     }
21 }
```

4.3.2 Proposed Approach for Custom Indices

EMF is the most widely used modeling technology in the MDE community. We developed a query optimisation strategy for EMF-based models of building in-memory indices (hashtables) using static analysis and automated program rewriting. This is presented as a conference paper in [11]. This approach supports both EOL and EVL programs. A program is traversed to find optimisable expressions i.e., Iterating the results of *allInstances()*. This gives us the possible in-memory indices to create. Then, through call graph analysis we identify if any index is likely to be used multiple times in a program. We add an index in list of potential indices only if that is detected to be used multiple times. Finally, we rewrite the program which includes injecting *createIndex()* statements to construct indices from list of potential indices (as shown in Algorithm 5) and secondly to rewrite the original expressions to make use of the indices as shown in Listing 12. In-memory indices create an overhead of populating indices which is paid off if an index is used multiple times in a program. This is the reason we use call graph analysis to manage the overhead.

4.3.3 Evaluation

The execution-time performance of the proposed approach to optimise EVL programs over large-scale EMF models has been evaluated. In the rest of the section, the first approach is referred to as EVL – since it executes the EVL programs in a naive parallel mode, while the second one is referred to as EVL-QR – since it makes use of the query rewriting strategy, on the top of the EVL engine in parallel mode. We also compare our results and presented speedups compared to OCL. For OCL evaluation, we wrote the same Java FindBugs in OCL and reported the execution time. The computation time taken for the static analysis

[15]https://www.eclipse.org/acceleo/documentation/

Algorithm 5 Algorithm for Finding Potential Indices.

```
let model = current model rewriter (separate rewriters for every model)
let inLoop = false
let allOperations = all, allInstances
let optimisableOperations = select, exists
let callGraph = call graph of the input program
    optimiseBlock(main statement block)
procedure optimiseBlock(StatementBlock)
    for all statement s in StatementBlock do
        if s is a ForStatement or WhileStatement then
            inLoop = true optimiseBlock(body of s)
        else if
     then
            visit every DOM element recursively
        else
    optimiseStatement(s)
procedure optimiseStatement(Statement)
    if s is an OperationCallExpression then
        repeat
    optimiseStatement(target of s)
        until targetExpression is instance of NameExpression
        for all Parameters of s do
            repeat
    optimiseStatement(parameterExpression)
            until parameterExpression is instance of NameExpression
    if s is an FirstOrderOperationCallExpression then
        if target of s is a PropertyCallExpression or OperationCallExpression then
            if allOperations contains name of target  then
                if optimisableOperations contains operationName of s then
                    if target of propertyCallExpression is owned by model then
                        if inLoop then
                            add to Potential Indices
for all op in getDeclaredOperations do
    if path p from main to op exists then
        if p contains an edge labelled as loop then
            inLoop= true
    optimiseBlock(body of op)
```

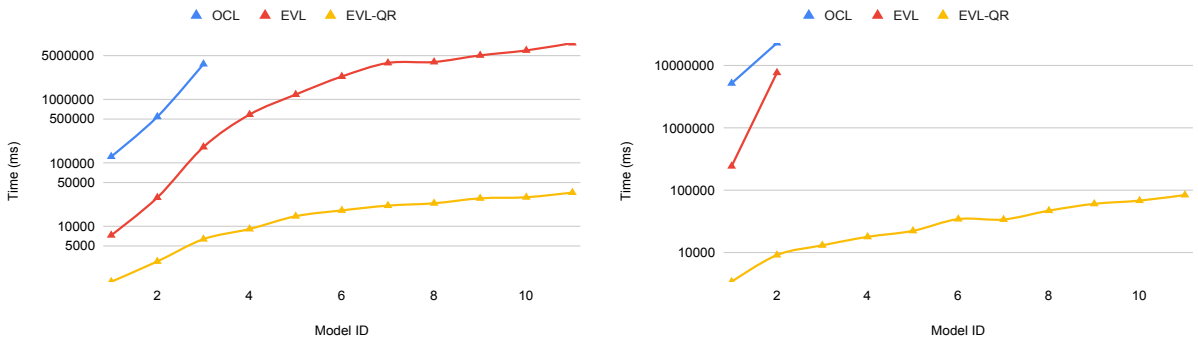

Figure 19: Comparison of OCL, EVL and EVL QR (a) with eOpposites (b) without eOpposites.

and query rewriting processes has been measured to assess the overhead they incur. Then, the execution time of the program itself is recorded, as this approach does not interact with model loading and thus has no effect on model loading times.

Observing the comparison graph shown in Figure 19, we see that EVL with query rewriting is substantially more performant than EVL. In a naive EVL execution, observing the comparison graph shown in Figure 6, we see that EVL with query rewriting is substantially more performant than EVL. In a naive EVL execution, as the model size grows, the execution time increases non-linearly, in this case from about 7 seconds to 130 minutes for models with 100k elements to 4.35M elements, respectively (a three order of magnitude increase, for models of around one order of magnitude in variance). In comparison with EVL, EVL-QR speeds up the validation by 5.6x for the smallest model and 228.18x for the largest model. While in comparison with OCL, EVL-QR speeds up the validation by 97.6x for the smallest model and upto 579.2x and even more for larger models where OCL's performance is timed out. This gives us confidence that the proposed query rewriting approach is scalable and efficient for very large models. Overall, these results illustrate that automated query rewriting has performance benefits both for small and large models.

4.4 Translating EVL to VIATRA

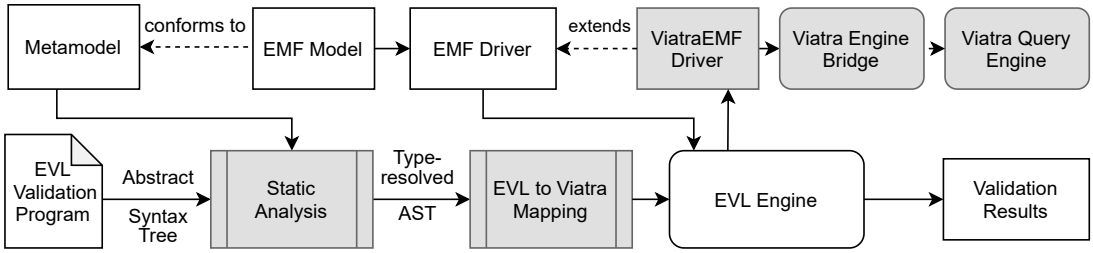

Figure 20: EVL to VIATRA mapping architecture.

VIATRA supports scalable querying through incremental engine. We proposed an approach[16] [12] for mapping certain expensive EOL expressions to VIATRA graph patterns as shown in Figure 20. Through the information extracted through static analysis, we detect first-order operations in an EOL/EVL program and then translate them to equivalent VIATRA patterns at compile time. So when such a translated VIATRA pattern is to be executed, a *runViatra* method provided by the EMC EMF VIATRA's extension is executed. It sends the textual pattern to VIATRA Engine Bridge which generates query specificaiton for VIATRA Engine. VIATRA Engine executes the query specification provided and return the found matches to EVL which are combined with the other results (unoptimised expressions) evaluated by EVL. This work is in collaboration with Benedek Horváth (ESR 13) at IncQuery Labs Budapest, Hungary.

4.4.1 Evaluation

In order to measure the query execution time and memory use of the proposed approach we adopted three validation constraints from the FindBugs validation suite [39] and evaluated them on the Java MoDisco EMF model of the Eclipse source code [39]. We compared the incremental (RETE) and local search (LS) engines of VIATRA with the sequential EVL engine. In the query evaluation phase, the models are already loaded in memory, and the EOL expressions are already translated to VQL. The query rewriting took 9 ms for all queries. The measurements were conducted on a machine with Windows 10, Intel i7-9750H CPU @ 2.60GHz, 32 GB RAM, Java HotSpot^{TM} 64-Bit Server VM 13.0.1+9 (with 16 GB max heap size).

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

Table 8: Execution time of the engines (in min:sec.milliseconds).

As execution-times shows, the RETE engine provides the shortest execution time, due to the incremental caching of pattern matches. The local search engine with base index gives similar results, due to the caching of the base relations and objects in the model. The LS engine without base index and the sequential EVL engine were several magnitudes slower compared to the previous engines.

The largest speed-up is 1481x between the RETE and the sequential EVL engine, in the case of models with 500k elements.

[16]https://github.com/lowcomote/evl-viatra-prototype

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

Table 9: Memory use of the engines (in MB).

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

4.5 Selective Traceability of Model-to-Model Transformations

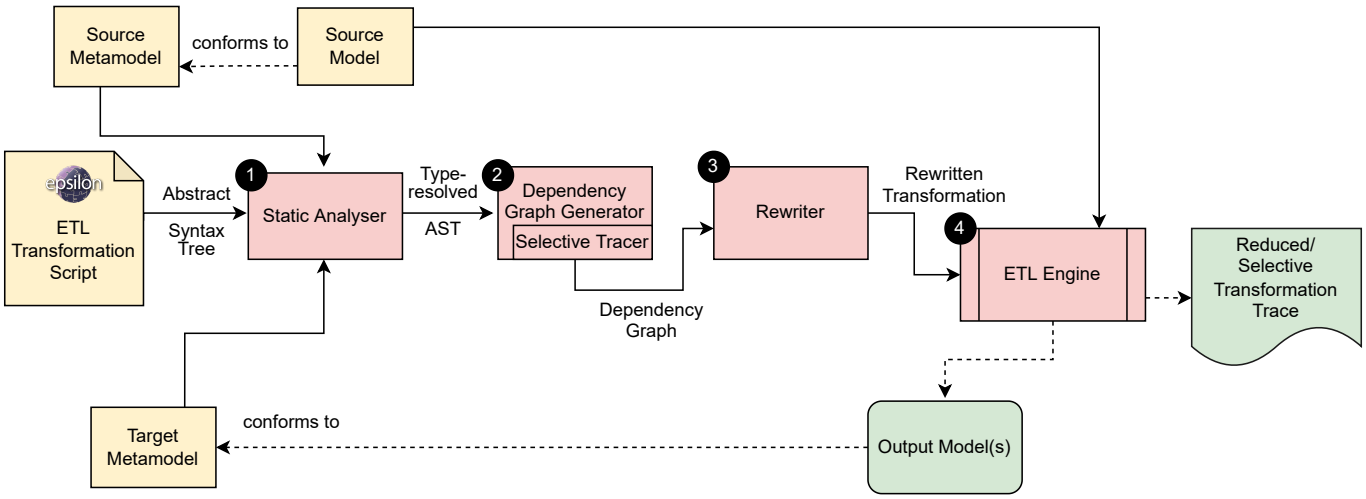

Figure 21: An overview of the proposed approach.

4.5.1 Motivating Example

In this section, we discuss our proposed approach for the efficient execution of rule-based model transformation programs using static analysis and automatic program rewriting. The main goal of this approach is to reduce the execution time and memory footprint of transformations without changing or compromising any of the language semantics. This approach is illustrated in Figure 21.

Let us consider the example of a partial (for conciseness) *OO2DB* transformation. It describes the transformation of a model conforming to an object-oriented schema metamodel, as shown in Figure 22, into a model conforming to a relational database metamodel as shown in Figure 23. This transformation has been adapted from [40] and an excerpt is shown in Listing 13. The transformation contains four transformation rules:

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

The size of the trace of the transformation (as shown in Listing 13), which relates source to target elements in the current implementation of the ETL execution engine will be $O+M+N$, if we evaluate it over a source model containing O classes, M number attributes and N references.

However, at a closer look, in this *OO2DB* transformation, only trace links created by the rule *Class2Table* are needed by the other rules. The remaining trace links, created by the three other rules (*SingleValuedAttribute2Column*, *MultiValuedAttribute2Table* & *Reference2ForeignKey*) are not used anywhere in the transfor-

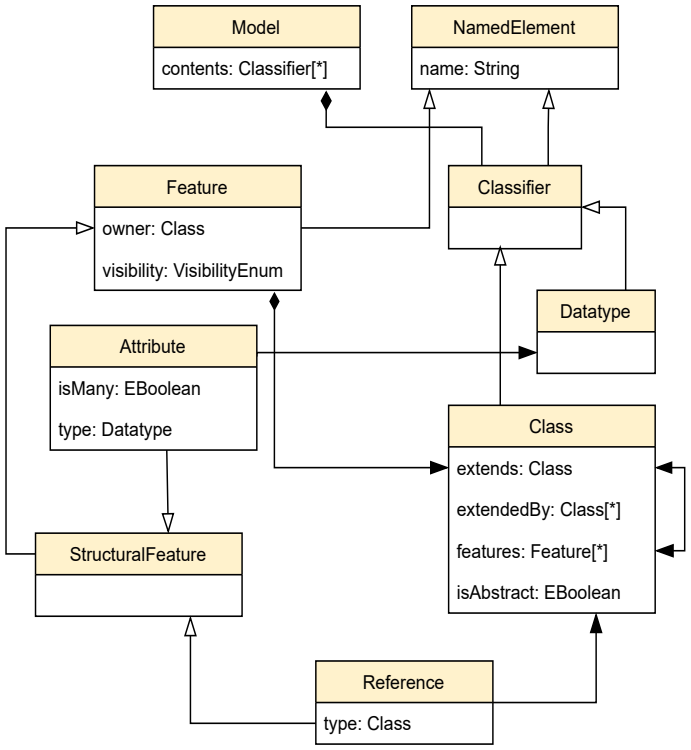

Figure 22: Object-Oriented Metamodel.

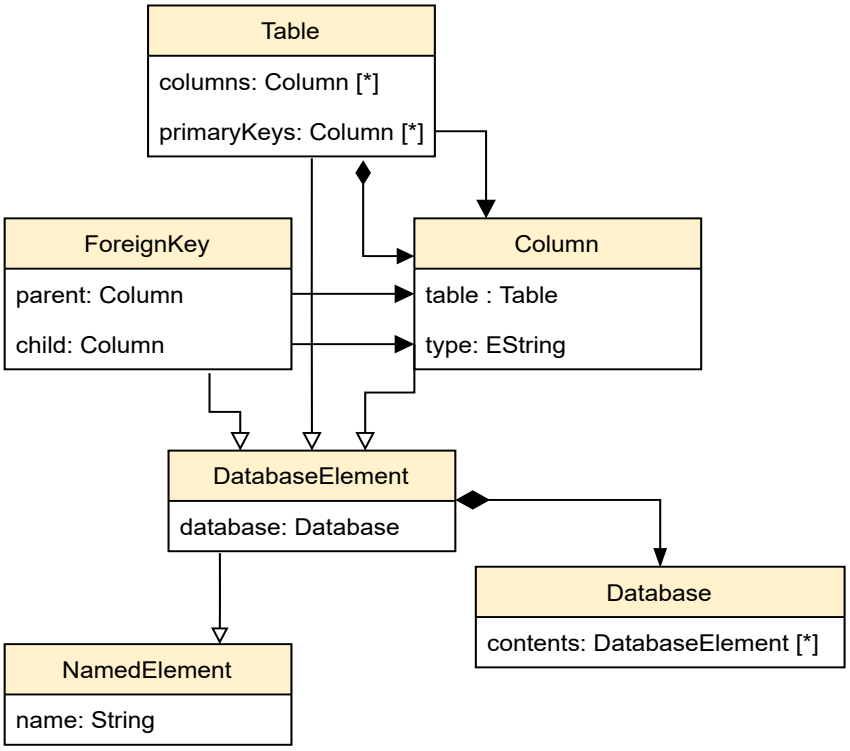

Figure 23: Database Metamodel.

mation and therefore establishing and keeping them in memory is wasteful. This would reduce the size of transformation trace to O.

4.5.2 Selective Traceability Approach

The proposed approach contains four main components, with a source metamodel, a source model, a target metamodel and a transformation being its inputs. The *Static Analyser* ① component is given the model-

to-model transformation and the source and the target metamodel(s), extracting the type information and yielding a type-resolved abstract syntax tree (AST). Then, using the *Dependency Graph Generator* ②, we extract the dependencies between the different transformation rules in the transformation: the dependency graph uses the type-resolved AST to populate these dependencies. Following this, we have a *Selective Tracer* to selectively create hash map caches to store the result of source target key-value pairs generated by corresponding operations and use them as a minimal transformation trace. In the next step, we pass this dependency graph (see Algorithm 6)to the *Rewriter* ③ (see Algorithm 7), where the transformation is rewritten i. e. the transformation rules are converted to the corresponding operations to optimise the resolution of elements by other rules. Finally, we pass the rewritten optimised transformation to the *ETL Engine* ④ for execution. ETL Engine is the default engine already provided by Epsilon. We extended the default engine to provide the resolution of operation calls to their corresponding user defined methods mapping provided by the static analyser.

Listing 13: Object-oriented 2 Database Transformation.

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

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

pre {
   var db : new Target!Database;
}
rule Class2Table
    transform c : Source!Class
    to t : Target!Table{
       t.name = c.name;
       t.database = db;
}

// Transforms a single-valued attribute
// to a column
rule SingleValuedAttribute2Column
    transform a : Source!Attribute
    to c : Target!Column {
       guard : not a.isMany
       c.name = a.name;
       c.table = a.owner.equivalent();
}

// Transforms a multi-valued attribute
// to a table where its values are
// stored and a foreign key
rule MultiValuedAttribute2Table
    transform a : Source!Attribute
    to t : Target!Table,
      fkCol : Target!Column {

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

// Transforms a reference into
// a foreign key
rule Reference2ForeignKey
    transform r : Source!Reference
    to fkCol : Target!Column {

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

}
```

4.5.3 Evaluation

We evaluate the proposed approach against the default ETL execution engine to measure its benefits in terms of execution time & memory footprint. First, we perform a comparison of the proposed approach with the default ETL engine. Second, we compare the proposed optimisation approach with other state-of-the-art languages for M2M transformation.

The experiment referred to as "ETL" evaluates running the transformation using naive ETL without any optimisations. The "Optimised ETL" one uses our optimisation/rewriting strategy. We also compare our results with two other widely-used model-to-model transformation languages, ATL and YAMTL [41],

Algorithm 6 Algorithm for extracting dependency graph.

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

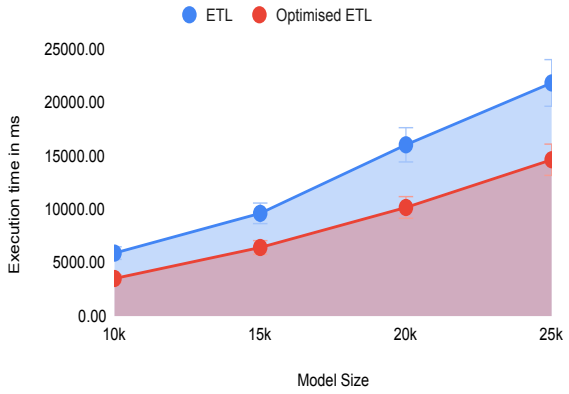

Figure 24: Execution Time Comparison.

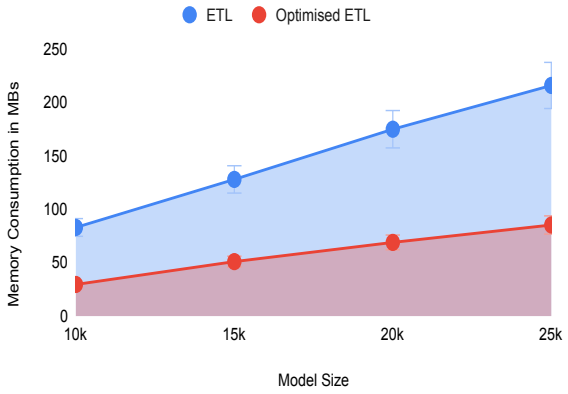

Figure 25: Memory Comparison.

to position our work in the broader context of M2M langauges. For the ATL and YAMTL evaluation, we rewrote the same transformation in ATL and YAMTL and we report on the execution time and memory footprint.

The results of the execution time of the naive ETL vs Optimised ETL are visualised in Figure 24. We can observe that overall ETL's execution time is significantly improved in Optimised ETL version. This is because of the optimisations provided by static analysis and program rewriting to avoid operation call resolutions at runtime and also because of efficient resolution of equivalents before the execution.

The memory use for the naive ETL and optimised ETL can be seen in Figure 25, where we can observe that the Optimised ETL consumes less memory compared to ETL due to the reduced (selective) transformation trace provided by the selective traceability mechanism.

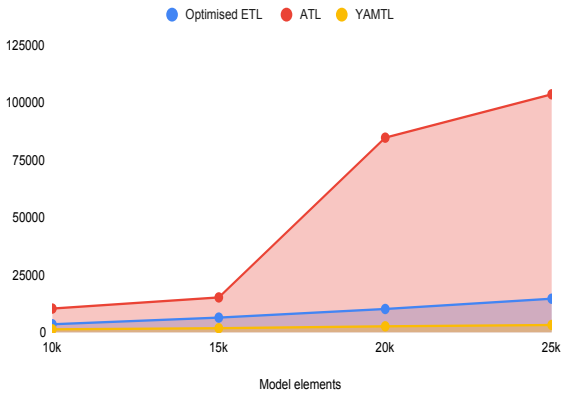

Figure 26: Execution Time Comparison.

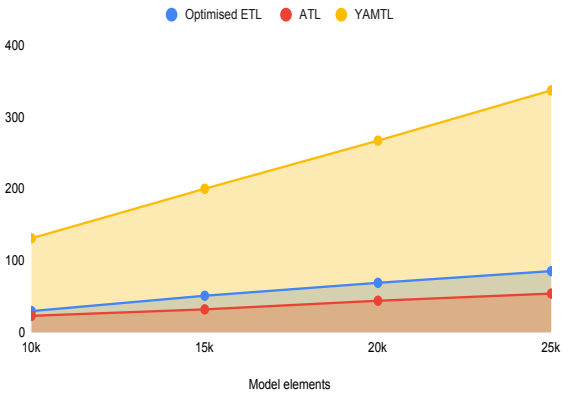

Figure 27: Memory Comparison.

Algorithm 7 Algorithm for rewriting the transformation.

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

We present the results of execution time and memory consumption of our proposed approach in comparison with ATL and YAMTL, respectively (depicted in Figure 26 and 27). We can clearly see YAMTL executes faster than the others, because YAMTL is compiled to Java, while ETL and ATL are interpreted.

4.6 Model-to-Model Transformation Chain Optimisation

Model transformation is a key concept in the field of model-driven engineering. To scale up the model transformation tasks in a cloud-based low-code platform, reusing and composing smaller model transformations enables a complex transformation to be executed sequentially and optimally. To achieve reusing and composition of smaller transformations, we need to identify all the model transformation chains possible in the repository/folder to transform the source model to the target model. Then, with the help of static analysis, the total structural features involved in a model transformation are calculated and the chain with minimum structural features is considered to be the best chain. We propose an approach as shown in the Figure 21 for the optimisation of model transformation chains that leverages static analysis of the model-to-model transformation and performs analysis of transformation chain and rule to execute automated program rewriting behind the scenes. The main idea is to remove those transformation rules and bindings which generate the intermediate model elements that are not required for generating the elements of the final target model. We expect to improve the performance by reducing the execution time required to execute the selected transformation chain. The optimization of the execution of the model transformation chain helps in achieving the scalable solution that manages several work flows originating from heterogeneous lowcode platforms. This work is done in collaboration with Apurvanand Sahay (ESR 15) and is

explained in detail in Deliverable 5.4.

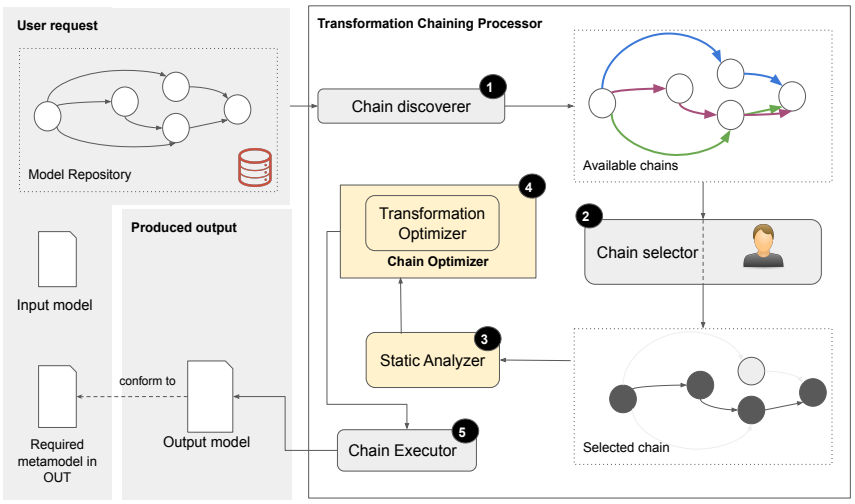

Figure 28: Model-to-model transformation chain optimisation.

4.7 Related Work

This section summarises existing work within the scope of this deliverable in three main categories. First, it lists existing tools in MDE that provide static analysis facilities; second, it discusses model query optimisation strategies; and finally, it discusses model-to-model transformation optimisation.

AnATLyzer [42] is a tool for static analysis of ATLAS Transformation Language (ATL) transformations that provides type checking, problem reporting and quick fixes. AnATLyzer checks that the transformation is correctly typed with respect to the source metamodel. It ensures that the generated target model conforms to the target metamodel. It also identifies any conflicting or missing rules. AnATLyzer is limited to static analysis of ATL model transformations only.

In [43], Born et al. extended Henshin, a rule-based model transformation language adapting graph transformation concepts based on the EMF. This extension computes all potential conflicts and dependencies of a set of rules and report them in the form of critical pairs. Each critical pair consists of the respective pair of rules, the kind of potential conflict or dependency found, and a minimal instance model illustrating the conflict or dependency.

Another tool [44] provides a static analysis facility for graph transformations. This work is based on Constraint Satisfaction Programming (CSP), containing a type checker for the Viatra2 framework. As this type checker is based on CSP, it is not guaranteed to find all the errors in a single run using static analysis. This tool is limited to static analysis of Viatra2 transformations. Static analysis of OCL is discussed in [45], where a pseudo-type *OCLSelf* is introduced to infer the type of built-in operations such as *oclAsSet()* and *oclType()*. Willink [46] introduced safe navigation operators in OCL. This operator solves the problem of declaring non-null objects and null-free collections and enables OCL navigation to be fully checked for null-safety.

In [47], the A2L compiler is introduced for parallel execution of ATL transformations. It uses static analysis through ATLyzer (discussed above), to generate efficient code at the transformation level.

Gremlin-ATL is another approach presented in [48]. It is a model-to-model transformation framework that translates ATL transformations into Gremlin, a query language supported by several NoSQL databases.

Another model-to-model transformation language, YAMTL, was introduced in [49]. YAMTL provides an efficient engine to transform EMF-based models with transformations defined in the internal DSL of Xtend. Support for incremental transformations was also added in [50] using the forward change propagation mechanism.

Several approaches and languages are available for incremental model-to-model transformations, such as the Tefkat tool, by Hearnden et al. in [51]. Here, changes to the source models are directly mapped to their effects on transformation execution, allowing modifications to target models to be computed efficiently.

5 Conclusion

In this report, we have discussed how scalability is a challenging aspect in a low-code platform and proposed approaches for intelligent model partitioning and query optimisation using static analysis. We described the architecture of static analyser for EOL. We added new features to the EOL engine to get more static information such as return type compatibility, type compatibility of context and parameters from model management programs at compile time. As the static analyser has been implemented for EOL, we plan to extend this facility to other specific languages of the Epsilon framework like the Epsilon Validation Language (EVL).

First, in Section 2, we have proposed an approach for the efficient loading of serializable models from the Neo4EMF database. We proposed an extension of NeoEMF for dynamically loading objects independently of EMF artifacts. The approach was demonstrated on the social network case study of TTC 2018. As future work, we plan to integrate this approach to the other research lines that were introduced in the report.

Then, in Section 3, we have introduced an approach for partial loading of file-based and repository-based models based on information extracted through static analysis of model management programs that access them. We have evaluated our approach against large models from the FindBugs and Grabats test suites. The results demonstrate that program-aware partial loading can significantly reduce the time and memory required to run model management programs against file-based and repository-based models when only a subset of the model's elements is accessed by the program, without otherwise affecting the behaviour or the output of the program.

As future work, we plan to investigate how the results of static analysis can also improve memory footprint when the execution engine unloads obsolete parts of the model (i. e. parts that have already been processed and are guaranteed not to be accessed again) in memory instead of keeping them for the duration of the execution of the program. In this way, resources will be freed that will allow management programs to accommodate even larger models.

Then in Section 4, we presented how in a low-code platform, models stored in various back-end formats, often need to be accessed concurrently in a model management program. It is essential to have a query optimization strategy for this scenario so that large-scale models can be queried efficiently. We have shown that static analysis and query rewriting can deliver benefits both in terms of memory footprint and execution time of complex queries by experimental results.

According to the proposed architecture of query optimisation, we use a static analyser to extract the information from the model management program. The type-resolved abstract syntax graph is then passed through different query translators/rewriters. We provided an approach for EOL to SQL query translation, optimisation of queries over EMF models by custom indices, and translation of EOL expression to VIATRA graph patterns. We also proposed an approach for selective traceability to efficiently execute rule-based model-to-model transformation programs. We also presented the results of the experiments conducted to compare the proposed approaches with state of the art. The results demonstrate that static analysis based optimisation can significantly reduce the execution time and memory footprint.

References

[1] Jean Bézivin. Model driven engineering: An emerging technical space. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 36–64. Springer, 2005.

[2] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis lectures on software engineering*, 3(1):1–207, 2017.

[3] 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.

[4] Ari Jaaksi. Developing mobile browsers in a product line. *IEEE software*, 19(4):73–80, 2002.

[5] Parastoo Mohagheghi and Vegard Dehlen. Where is the proof?-a review of experiences from applying mde in industry. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 432–443. Springer, 2008.

[6] 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 *International Conference on Computer Safety, Reliability, and Security*, pages 322–333. Springer, 2016.

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

[8] Antonio Bucchiarone, Jordi Cabot, Richard F Paige, and Alfonso Pierantonio. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, 19(1):5–13, 2020.

[9] Sorour Jahanbin, Dimitris Kolovos, and Simos Gerasimou. Intelligent run-time partitioning of low-code system models. MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Qurat ul ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. Efficiently querying large-scale heterogeneous models. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Qurat Ul Ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. Identification and optimisation of type-level model queries. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 751–760, 2021.

[12] Qurat Ul Ain Ali, Benedek Horváth, Dimitris Kolovos, Konstantinos Barmpis, and Ákos Horváth. Towards scalable validation of low-code system models: Mapping evl to viatra patterns. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 83–87, 2021.

[13] Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. Neo4emf, a scalable persistence layer for emf models. In *European Conference on Modelling Foundations and Applications*, volume 8569 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2014.

[14] Gwendal Daniel, Gerson Sunyé, Amine Benelallam, and Massimo Tisi. Improving memory efficiency for processing large-scale models. volume 1206, 07 2014.

[15] Avery Ching, Sergey Edunov, M. Kabiljo, Dionysios Logothetis, and S. Muthukrishnan. *One trillion edges: Graph processing at facebook-scale*, pages 1804–1815. 01 2015.

[16] Antonio García-Domínguez, Georg Hinkel, and Filip Krikava, editors. *Proceedings of the 11th Transformation Tool Contest, co-located with the 2018 Software Technologies: Applications and Foundations, TTC@STAF 2018, Toulouse, France, June 29, 2018*, volume 2310 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.

[17] Jolan Philippe, Massimo Tisi, Hélène Coullon, and Gerson Sunyé. Executing certified model transformations on apache spark. SLE 2021, page 36–48, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Dimitrios S Kolovos, Louis M Rose, Nicholas Matragkas, Richard F Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan De Lara, István Ráth, Dániel Varró, Massimo Tisi, et al. A research roadmap towards achieving scalability in model driven engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, pages 1–10, 2013.

[19] Dimitrios Kolovos, Richard Paige, and Fiona Polack. The grand challenge of scalability for model driven engineering. In *International Conference on Model Driven Engineering Languages and Systems (MODELS2008)*, volume 5421 of *Lecture Notes in Computer Science*, pages 48–53. Springer, Berlin, Heidelberg, 04 2008.

[20] Antonio Garmendia, Esther Guerra, Dimitrios S Kolovos, and Juan De Lara. Emf splitter: A structured approach to emf modularity. *XM@ MoDELS*, 1239:22–31, 2014.

[21] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The epsilon object language (eol). In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture – Foundations and Applications*, pages 128–142, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[22] Ran Wei and D.S. Kolovos. Automated analysis, validation and suboptimal code detection in model management programs. In *CEUR Workshop Proceedings*, volume 1206, pages 48–57, 01 2014.

[23] Ran Wei, Dimitrios S. Kolovos, Antonio Garcia-Dominguez, Konstantinos Barmpis, and Richard F. Paige. Partial loading of xmi models. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, page 329–339. Association for Computing Machinery, 2016.

[24] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. Modisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 173–174, 2010.

[25] Gwendal Daniel, Gerson Sunyé, Amine Benelallam, Massimo Tisi, Yoann Vernageau, Abel Gómez, and Jordi Cabot. Neoemf: A multi-database model persistence framework for very large models. *Science of Computer Programming*, 149:9–14, 2017.

[26] Grabats2009: 5th int. workshop on graph-based tools (2012).

[27] Dimitrios Kolovos, Louis Rose, Richard Paige, Esther Guerra, Jesús Cuadrado, Juan De Lara, István Ráth, Dániel Varró, Gerson Sunyé, and Massimo Tisi. Mondo: scalable modelling and model management on the cloud. 2015.

[28] Frédéric Jouault, Jean Bézivin, and Mikaël Barbero. Towards an advanced model-driven engineering toolbox. *Innovations in Systems and Software Engineering*, 5(1):5–12, 2009.

[29] Xavier Blanc, Marie-Pierre Gervais, and Prawee Sriplakich. Model bus: Towards the interoperability of modelling tools. In *Model driven architecture*, pages 17–32. Springer, 2004.

[30] Eike Stepper. Cdo, November 2016.

[31] Javier Espinazo Pagán, Jesúss Sánchez Cuadrado, and Jesús García Molina. Morsa: A scalable approach for persisting and accessing large models. In *International Conference on Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 77–92. Springer, 2011.

[32] 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, page 33–38. Association for Computing Machinery, 2012.

[33] 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*, pages 318–328, 2016.

[34] Sina Madani, Dimitris Kolovos, and Richard F Paige. Towards optimisation of model queries: a parallel execution approach. *Journal of Object Technology*, 18(2), 2019.

[35] Massimo Tisi, Salvador Martínez, and Hassene Choura. Parallel execution of atl transformation rules. In *International Conference on Model Driven Engineering Languages and Systems*, pages 656–672. Springer, 2013.

[36] Dimitrios S Kolovos, Ran Wei, and Konstantinos Barmpis. An approach for efficient querying of large relational datasets with ocl-based languages. In *XM 2013-Extreme Modeling Workshop*, volume 48, 2013.

[37] Konstantinos Barmpis and Dimitrios S. Kolovos. Towards scalable querying of large-scale models. In Jordi Cabot and Julia Rubin, editors, *Modelling Foundations and Applications*, pages 35–50, Cham, 2014. Springer International Publishing.

[38] Martin Hanysz, Tobias Hoppe, Axel Uhl, Andreas Seibel, Holger Giese, Philipp Berger, and Stephan Hildebrandt. Navigating across non-navigable ecore references via ocl. *Electronic Communications of the EASST*, 36, 2010.

[39] Sina Madani, Dimitrios S. Kolovos, and Richard F. Paige. Parallel model validation with Epsilon. In *Proceedings of the 14th European Conference on Modelling Foundations and Applications, ECMFA@STAF 2018*, volume 10890 of *LNCS*, pages 115–131. Springer, 2018.

[40] Michael Lawley, Keith Duddy, Anna Gerber, and Kerry Raymond. Language features for re-use and maintainability of mda transformations. In *Workshop on Best Practices for Model-Driven Software Development*, 2004.

[41] Yet Another Model Transformation Language. https://yamtl.github.io, 2022. [Online; accessed 29-April-2022].

[42] 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, page 85–88, New York, NY, USA, 2018. Association for Computing Machinery.

[43] Kristopher Born, Thorsten Arendt, Florian Heß, and Gabriele Taentzer. Analyzing conflicts and dependencies of rule-based transformations in henshin. In Alexander Egyed and Ina Schaefer, editors, *Fundamental Approaches to Software Engineering*, pages 165–168, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

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

[45] Edward D. Willink. Modeling the OCL standard library. *ECEASST*, 44, 2011.

[46] 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*, volume 1512 of *CEUR Workshop Proceedings*, pages 81–88. CEUR-WS.org, 2015.

[47] Jesús Sánchez Cuadrado, Loli Burgueño, Manuel Wimmer, and Antonio Vallecillo. Efficient execution of atl model transformations using static analysis and parallelism. *IEEE Transactions on Software Engineering*, PP:1–1, 07 2020.

[48] Gwendal Daniel, Frédéric Jouault, Gerson Sunyé, and Jordi Cabot. Gremlin-atl: a scalable model transformation framework. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 462–472. IEEE, 2017.

[49] Artur Boronat. Expressive and efficient model transformation with an internal dsl of xtend. MODELS '18, page 78–88, New York, NY, USA, 2018. Association for Computing Machinery.

[50] Artur Boronat. Incremental execution of rule-based model transformation. *International Journal on Software Tools for Technology Transfer*, 23(3):289–311, 2021.

[51] David Hearnden, Michael Lawley, and Kerry Raymond. Incremental model transformation for the evolution of model-driven systems. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*, MoDELS'06, page 321–335, Berlin, Heidelberg, 2006. Springer-Verlag.