



“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: Dora Horváth

Work Package: WP5

Deliverable number: D5.1

Production date: November 29th 2020

Contractual date of delivery: November 30th 2020

Actual date of delivery: November 30th 2020

Dissemination level: Public

Lead beneficiary: University of York

Authors: Sorour Jahanbin, Qurat ul ain Ali

Contributors: Lowcomote partners

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 preliminary approaches for efficient model loading and query optimisation. Firstly, we present an intelligent run time partitioning approach that leverage sophisticated static program analysis of model management programs to identify, load, process and transparently discard relevant model partitions – instead of naively loading the entire models into memory and keeping them loaded for the duration of the execution of the program. 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	4	3.3	Proposed Approach	12
			3.4	Related Work	14
2	Static Analysis	5			
2.1	Architecture	5	4	Query Optimisation	16
2.2	Type Resolution	6	4.1	Research Objectives	16
2.3	Type Checking	7	4.2	Motivating Example	16
2.4	Eugenia - Test case	7	4.3	Proposed Approach	19
2.5	Related Work	8	4.3.1	Query Translation	19
3	Intelligent Run-time partitioning	10	4.4	Related Work	20
3.1	Research Objectives	10			
3.2	Motivating Example	10	5	Conclusion	23

1 Introduction

Low-code platforms use Model-Driven Engineering (MDE) [1] processes such as domain specific languages and code generation to develop applications. In 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 OutSystems¹, Mendix², Google AppMaker³ and ZAppDev⁴, 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 in pay-per-use cloud resources.

In intelligent run-time partitioning of low-code system models, we present our approach [9] that leverages sophisticated static program analysis of model management programs to identify, load, process and transparently discard relevant model partitions – instead of naively loading the entire models into memory and keeping them loaded for the duration of the execution of the program. In this way, 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.

In heterogeneous models query optimisation, we propose an approach [10] 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 aim to 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 high-level model query languages such as Epsilon Object Language (EOL) [11].

This report is submitted as a interim-deliverable for the Lowcomote project. The rest of this report is organised as: Section 2 provides an overview of static analysis architecture along with test case while Section 3 introduces an intelligent run-time partitioning approach based on compile-time static analysis with a motivating example. Section 4 describes the research challenges related to model querying and then briefly presents the proposed solution with an example. Section 5 concludes the report and presents the future research directions.

¹<https://www.outsystems.com>

²<https://www.mendix.com>

³<https://developers.google.com/appmaker>

⁴<http://www.zappdev.com>

2 Static Analysis

The role of the static analyser is to analyse the program at compile-time. Static analysis will provide in-advance knowledge about model management program that can be used for intelligent model partitioning and query optimisation as shown in Figure 1.

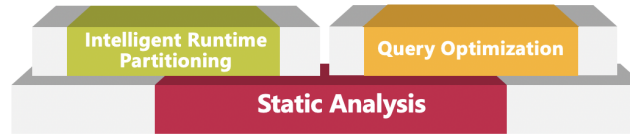


Figure 1: Static analysis architecture

In intelligent model partitioning, by using the information which is obtained by static analyser, the execution engines will be able to identify, load and process model partitions which contain model elements of interest (See Section 3.3).

For query optimisation, the type-resolved abstract syntax graph from static analysis block will be used. It will enable identifying accessed model elements and their patterns in the model management program for the purpose of query optimisation. For handling heterogeneity, static analysis will need to access meta-models at compile-time. It is done using *ModelDeclarationStatement* and discussed in detail in Section 4.3.1

2.1 Architecture

Low-code platforms make use of models to specify the structure and functionality of applications which are then validated/transformed to executable code. Several languages and tools exist for defining such model management programs. Epsilon [12] is a family of consistent languages that consists of task-specific languages that are used for model management. Model management operations are some common activities like model transformation, code generation, merging, validation, and refactoring. The architecture of Epsilon is summarized in Figure 2 which shows that Epsilon supports most of model management tasks including model merging (EML) [13], code generation (EGL) [14], model migration (Flock) [15], model comparison (ECL) [16], model-to-model transformation (ETL) [17], model refactoring (EWL) [18], pattern matching (EPL) [19] and model validation (EVL) [20].

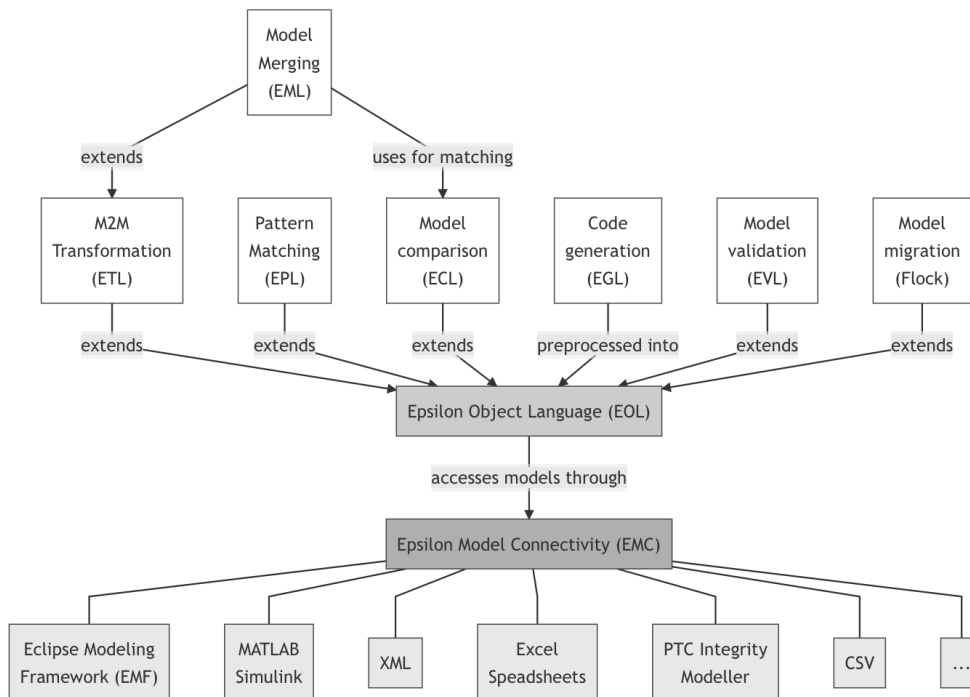


Figure 2: Epsilon

The core language of Epsilon is EOL, which is a general programming language, provides common facilities that are useful for developing domain-specific languages.

Static analysis of Epsilon started in [21] where the Abstract Syntax Tree (AST) of an EOL program is computed, then resolution algorithms including variable resolution (e.g., resolving identifiers to their definitions) and type resolution (e.g., primitive types and collection types) applied to derive an Abstract

Syntax Graph. Thus, using the Abstract Syntax Graph, the static analyser can extract relevant information (i.e., types and properties accessed by the program).

We contribute a static analysis facility in EOL, where various compile-time errors are produced as a by-product. In first step, a type resolver will set the resolved type of expressions then in order to check type compatibility, type of context, parameter and return types for both user-defined and built-in operation is checked by a type checker. This compile-time static analysis of Epsilon is implemented in EOL, as it is the core language of Epsilon.

The reason why Epsilon has been chosen as the basis of this work is that the developed query optimisation and run-time partitioning facilities can benefit a wide range of model management activities (e.g. model validation, code generation, etc.). In addition, it supports a number of model persistence formats and even it can be extended to work with unsupported technologies using Epsilon Model Connectivity (EMC) layer [12]. Beyond Epsilon, several model management tools have static analysis facilities e.g. AnATLyzer [22], Henshin [23].

2.2 Type Resolution

To enable static analysis of Epsilon programs, it was necessary to introduce a number of pseudo-types to the language. Pseudo-types are added inspired from Object Constraint Language (OCL) [24] for the purpose of static analysis. They are called pseudo-types as they cannot be instantiated. They are just used to determine type of “self” in operation signatures. Exact type of these will be determined at the end of compile-time static analysis. Pseudo types (*EolSelf*, *EolSelfCollectionType*, *EolSelfExpressionType*, *EolContentType*) were added in EOL to help in static analysis. Usage of these types are shown in Table 1.

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

Table 1: Pseudo Types

EolSelf is a pseudo type used in signature (as shown below) of operation to propagate the context type as return type whenever `println` is called. For example, if we call “*abc*”.`println()` return type would then be *EolPrimitiveType.String*.

```
1 operation Any println() : EolSelf {
2   return self;
3 }
```

EolSelfCollectionType is a pseudo type specially for *EolCollectionTypes*. It is used in operations signature as follows:

```
1 operation Collection<Any> test() : EolSelfCollectionType {
2   return self;
3 }
```

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

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

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

Whenever this operation is called with any collection type (*Collection*, *Sequence*, *Bag*, *OrderedSet*, *Set*) as context type the return type would be the type of content type of the collection. For example, if this operation is called on *Sequence<String>*, return type would also be *EolPrimitiveType.String*. For *Bag<Integer>* as context type, return type would be an integer *EolSelfExpressionType* is also a pseudotype just for *EolCollectionTypes*. It is used in operation signature as follows:

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

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

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

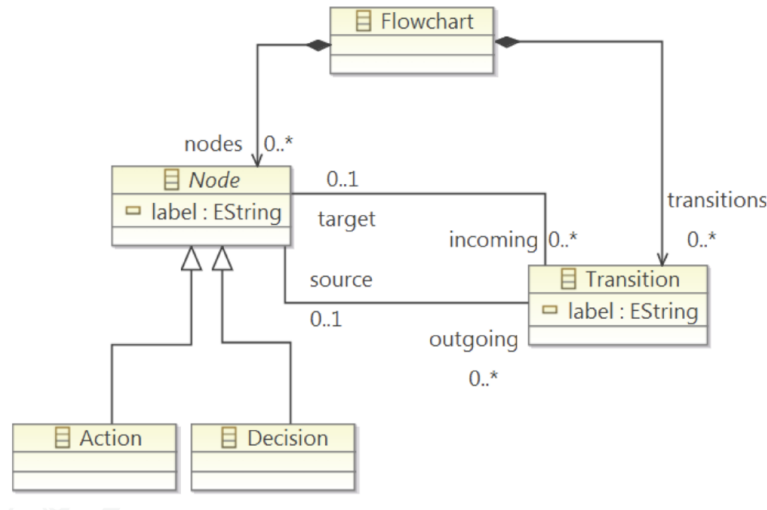


Figure 3: Flowchart metamodel

2.3 Type Checking

Some other static analysis features are as follows: Consider the metamodel shown in Figure 3. In Figure 4 operation Action *printName()* specifies return type as String but it doesn't have any return statement. So, a compile-time error is produced saying missing *ReturnStatement*. For second operation Node *printName()* specifies String as return type but in the return statement, it returns an integer value.

In Figure 4, an operation *greetUser()* is called on *Integer* as context. An error is produced because the definition of operation the required context type to be *String*.

```

TT.eol
1 model flowchart driver EMF {
2   nsuri = "flowchart"
3 };
4
5 var user = "Alex";
6
7 123.greetUser(); //greetUser() cannot be invoked on Integer
8
9 user.greetUser(); //greetUser() may not be invoked on Any, as it requires String
10
11 operation Action printName(): String { //missing return statement
12 }
13
14 operation Node printName(): String { // Return type should be String instead of Integer
15 return 234;
16 }
17
18 operation String greetUser(): String {
19   return "Hello " + self;
20 }

```

Figure 4: Type compatibility errors - Example

A simple example is presented in Figure 4, *greetUser()* operation requires a *String* context type, but the provided type is *Any* which is the parent type of *String*. So, a warning is produced saying *greetUser()* may not be invoked on *Any*, as it requires *String*.

2.4 Eugenia - Test case

These features of the static analyser are evaluated on Eugenia [25], Eugenia is a well-known tool build using Epsilon platform to generate GMF files from an annotated ecore file. Eugenia consists of a large EOL transformations, having 1212 lines of code, which transforms annotated metamodels to 4 different models required by the GMF framework in order to generate a graphical editor.

Eugenia automatically generates *.gmfgraph*, *.gmfmap* and *.gmftool* required by GMF from a single annotated Ecore⁵ metamodel. In Eugenia, there are some warnings where parent type of required type is provided. A screenshot of static analysis on Ecore2GMFEol (a transformation from Eugenia) is shown in Figure 5 and Figure 6.

⁵<https://wiki.eclipse.org/Ecore>

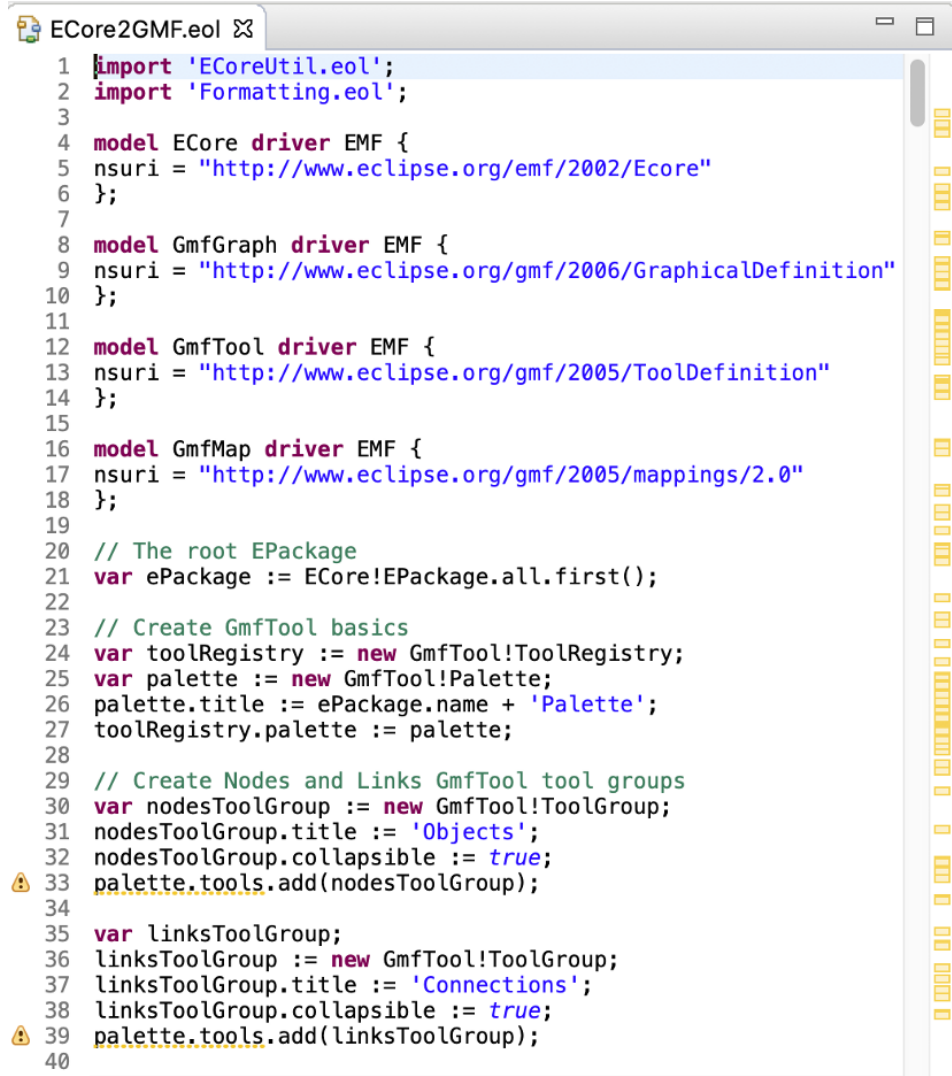


Figure 5: Static analysis on Ecore2GMF.eol

2.5 Related Work

AnATLyzer [22] is a tool for static analysis of ATL model transformations. Basically, it is an IDE that provides type checking, quick fixes and problem explanations. AnATLyzer focuses on three main points: 1) It checks that the source meta model, is correctly typed with respect to the transformation. 2) It ensures that the model generated through transformation conforms to the target meta model. 3) It identifies any conflicting or missing rules. This static analyser is limited to ATL model transformations only.

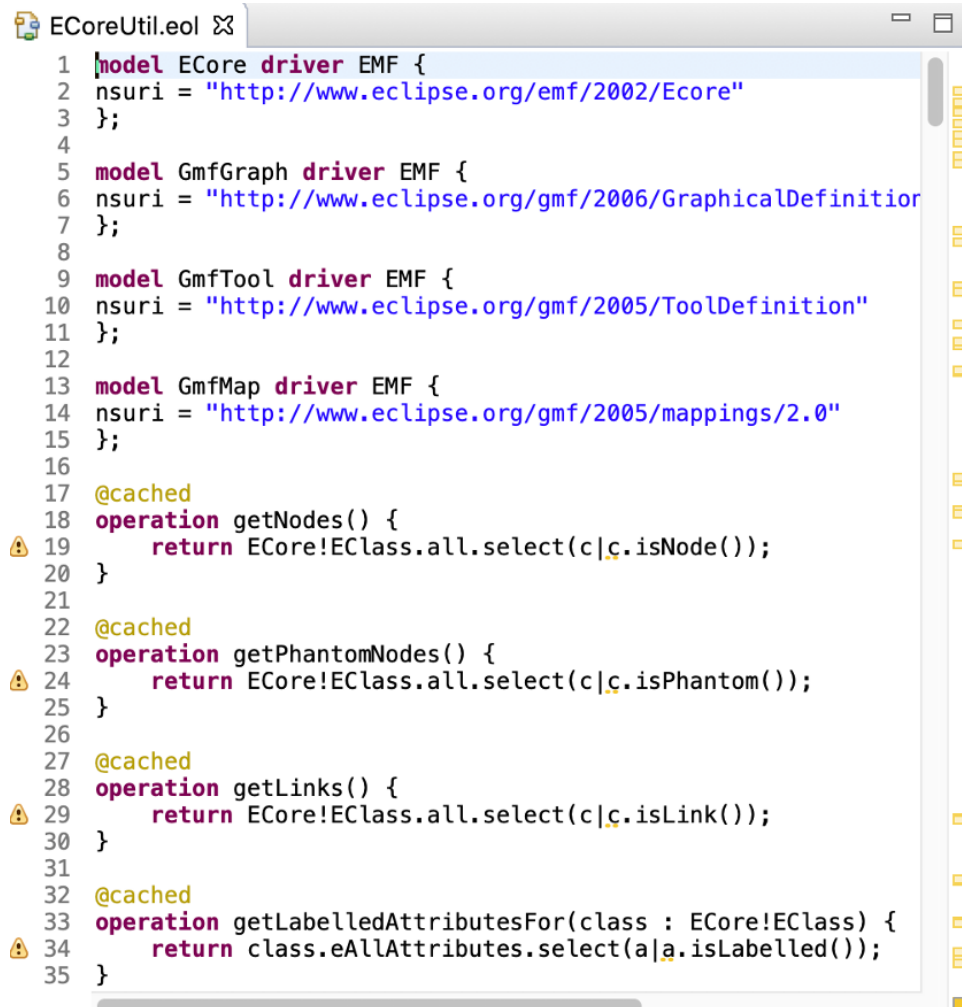
In [23], Born et al. extended Henshin, a rule-based model transformation language adapting graph transformation concepts and being based on the Eclipse Modeling Framework (EMF). This extension computes all potential conflicts and dependencies of a set of rules and reports them in form of critical pairs. Each critical pair consists of the respective pair of rules, the kind of potential conflict or dependency found, and a minimal instance model illustrating the conflict or dependency.

Another tool in [26], provides a static analysis facility for graph transformations. This work is based on Constraint Satisfaction Programming (CSP). It also presents a type checker for Viatra2 framework. As this type checker is based on CSP, it is not possible to find all the errors in a single run using static analysis.

The static analysis of OCL is presented in [27], a pseudo-type OCLSelf, is introduced to infer the type of context for few operations such as:

- OclAny::oclAsSet() – returns Set<Self>
- OclAny::oclType() : Class<OclSelf>

Willink [28] introduced safe navigation operators in OCL. This operator solves the problem of declaring non-null objects and null-free collections. It enables OCL navigation to be fully checked for null safety.



```
1 model ECore driver EMF {
2   nsuri = "http://www.eclipse.org/emf/2002/Ecore"
3 };
4
5 model GmfGraph driver EMF {
6   nsuri = "http://www.eclipse.org/gmf/2006/GraphicalDefinition"
7 };
8
9 model GmfTool driver EMF {
10  nsuri = "http://www.eclipse.org/gmf/2005/ToolDefinition"
11 };
12
13 model GmfMap driver EMF {
14   nsuri = "http://www.eclipse.org/gmf/2005/mappings/2.0"
15 };
16
17 @cached
18 operation getNodes() {
19   return ECore!EClass.all.select(c|c.isNode());
20 }
21
22 @cached
23 operation getPhantomNodes() {
24   return ECore!EClass.all.select(c|c.isPhantom());
25 }
26
27 @cached
28 operation getLinks() {
29   return ECore!EClass.all.select(c|c.isLink());
30 }
31
32 @cached
33 operation getLabelledAttributesFor(class : ECore!EClass) {
34   return class.eAllAttributes.select(a|a.isLabelled());
35 }
```

Figure 6: Static analysis on EcoreUtil.eol

3 Intelligent Run-time partitioning

Over the last two decades, several dedicated languages have been proposed to support model management activities such as model validation, transformation, and code generation. As software systems become more complex, underlying system models grow proportionally in both size and complexity which reveals the limitations of the current generation of tools and technologies in terms of capacity and efficiency [29]. 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.

A reason for these limitations is related to the need of most model management tools to load the entire model into memory. In most contemporary tools, in order to access any element of the model, the entire model must be loaded into memory, which can be wasteful in terms of loading time and memory consumption. Besides, keeping all model elements loaded when they are no longer needed (e.g. in multi-step workflows) affects the memory footprint.

Consequently, partial model loading facilities can be useful for reducing the loading time and memory footprint of large models. In partial loading, models are divided into some partitions where every partition contains model elements that are associated with the set of executed model management operations.

In this section, we present our approach that leverages sophisticated static program analysis of model management programs to identify, load, process and transparently discard relevant model partitions – instead of naively loading the entire models into memory and keeping them loaded for the duration of the execution of the program. In this way, 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.

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 [29]. 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 [30] – 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 5 times as big 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 the 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

Consider the domain-specific modelling language for implementing low-code form-based applications (from Figure 7). According to this modelling language, every *Application* consists of a number of *Entity* and *Form* elements. A *Form* has a reference to an *Entity*. Each *Entity* can be composed of *Properties* where every *Field* is assigned to at most one *Property*. For implementing the low-code form-based application, *Entity* and *Property* elements are used for generating the database schema, back-end CRUD code and web services. Furthermore, from *Form* and *Field* elements we can generate the front-end of the application (e.g. HTML/iOS/Android front-end code).

Let us consider the back-end code generation scenario where a developer wishes to re-generate the back-end of an application (or of all applications hosted in the low-code platform) and leave the front-end

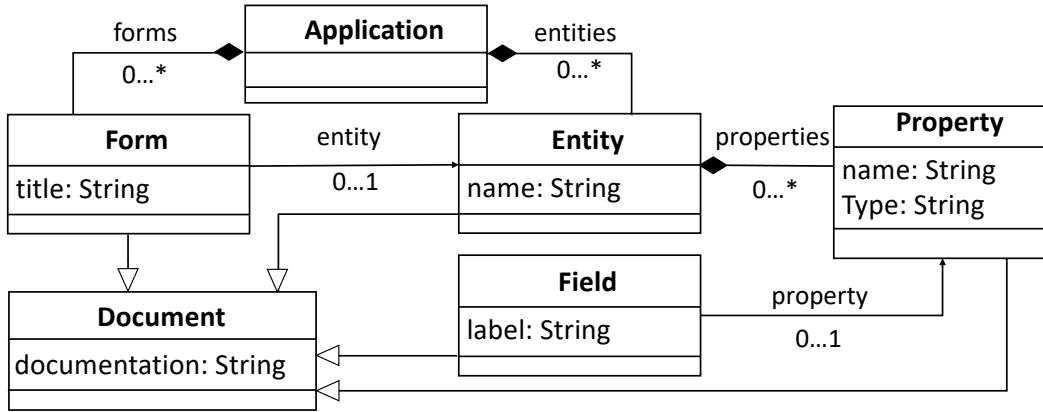


Figure 7: Form-based low-code application metamodel

intact. Assume this re-generation task involves an optimisation in the generated back-end code or the migration to a different database or web-services framework.

The back-end generator could be written in a model-to-text transformation (M2T) language such as the Epsilon Generation Language (EGL) as shown in Listing 1. There is an *Entity2Class* rule which transforms every *Entity* to a Java class using the EGL code. EGL is a member of the Epsilon family language which is used for generating code from a model.

Listing 1: EGL transformation rule for generating part of the low-code application back-end

```

1 rule Entity2Class
2 transform e : Entity {
3 template : 'entity2class.egl'
4 target : "src-gen/" + e.name + ".java"

```

In Listing 1, for every *Entity* (*e* is an instance of *Entity*) in the model, the program invokes the *entity2class.egl* template and stores the generated class in a .java file. The *entity2class.egl* template is shown in Listing 2. In order to generate a class from *Entity*, the name of class is assigned according to the name of *Entity* and from every *Property*, a new field of the class is declared. The *entity2class.egl* only accesses the name of the *Entity* and the names and types of its *properties* but not the *Form* or *Field* elements of the model.

Listing 2: The entity2class.egl template

```

1 public class [%=Entity.name%]{
2     [% for (p in Entity.properties){ %]
3         [%=p.type%] [%=p.name%] = new [%=p.type%]();
4     [%}%]
5 }

```

To load a model for executing this program, there are two possibilities.

- If the models are file-based (e.g. XMI or XML-based), the EGL execution engine needs to decide in advance, which parts of them it will load in memory, as re-parsing the same model file several times can be expensive. In the absence of in-advance static analysis of the generator (M2T), the UI-related parts of the application model would be loaded as well, despite the fact that they will not be used by the back-end generator.
- If the models are not file-based (e.g., stored in a database-backed repository such as CDO or Hawk [31]), the EGL execution engine can retrieve model elements on demand. Still, in the absence of static analysis, there is no way to tell which features of these model elements should be retrieved from the repository for each element. In this situation, there are two alternatives: either *greedily* fetch all features in advance or *lazily* fetch all features 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 (detrimental to performance). Considering Listing 2 that *entity2class.egl* uses the name of the *Entity* and the names and types of its *properties* but not their documentation, the two strategies are sub-optimal:
 - Greedy: The documentation of the entity and its properties is fetched from the repository but is never used by the generator, thus wasting memory.
 - Lazy: Multiple round-trips to the repository are required to fetch the values of the name/type features of each accessed entity and property in the model, thus degrading performance.

A static-analysis-based execution planner could determine which features are (not) accessed by the generator in advance and query the repository accordingly (e.g., populate the *name* and *type* of each *Field* in one go, but leave out the *documentation* which is not required).

As the *entity2class.egl* (Listing 2) only accesses features and properties of the *Entity* itself (i.e., it doesn't reach out to other *Entity* elements), after the execution of rule *Entity2Class* for a given entity, that entity is no longer needed and could be offloaded from memory to reduce the overall footprint of the generator. In the absence of in-advance static analysis, this cannot be determined by the generator. Therefore, all *Entity* elements will be kept in memory until the entire generation has concluded.

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 reducing 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 8.

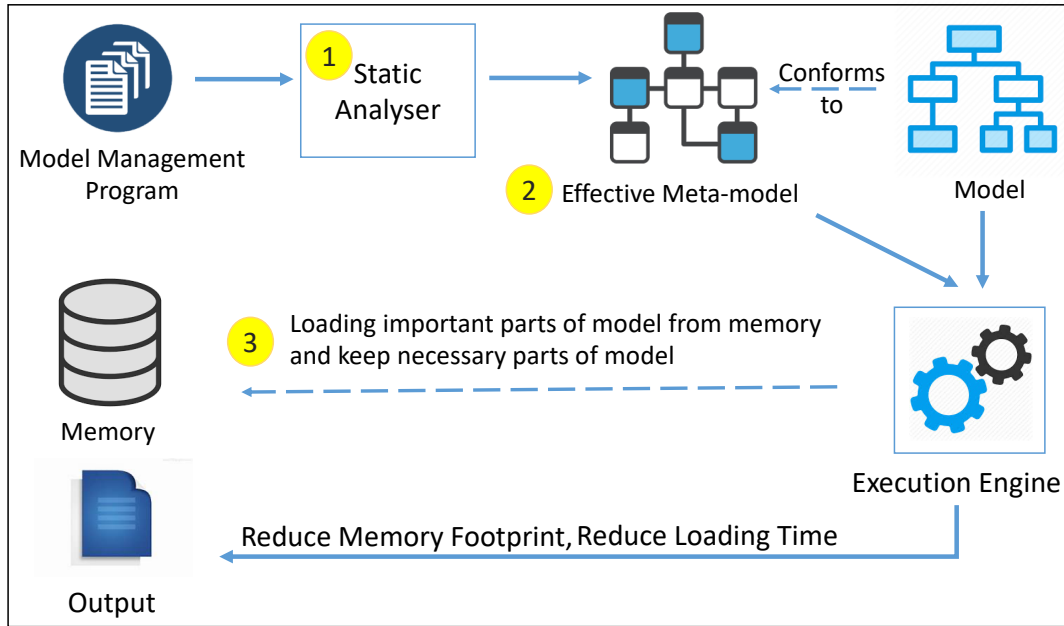


Figure 8: The proposed approach

(1) *Static Analyser*

In the first step of our approach, a model management program is provided as input to a static analyser.

The role of the static analyser is to analyse model management program using the Abstract Syntax Tree at compile-time. While analysing, the type resolution applied to derive an Abstract Syntax Graph. Thus, using the Abstract Syntax Graph, the static analyser can extract relevant information (i.e., types and properties accessed by the program). Type inference is a prerequisite activity which helps the static analyser to extract useful information for execution planning. The execution plan contains the information which helps the execution engine to load and process models efficiently (e.g., what part of the model should be loaded each time, which parts should be disposed of from memory, when each of these activities should occur).

A part of the execution plan is in the form of an effective metamodel which is considered as an output of the static analyser. The effective metamodel is a subset of the model's metamodel which consists of only types and properties of interest [21] (see below).

To illustrate how every step of the approach works, we use the motivating example of Section 3.2. Considering Listing 2, the model management program *entity2class.egl* only uses the name of the *Entity* and the names and types of its *properties*; the remaining information in the model is not required for executing this program (such as their documentation). In the motivating example, the static analyser detects the elements of the model that are necessary for executing the EGL program. Since EOL is the core language of the Epsilon platform, our approach applies to all model management languages of the platform. Also, the underlying principles, subject to suitable technical modifications, can be applied to the other modelling languages, e.g., ATL.

Hence, instead of loading all model elements, we need to load only instances of *Entity* and *Property* and only the values of their *name/type* fields. This information is obtained by static analysis facilities of the EGL program.

It is worth noting that using static analysis is not only about extracting information to load model elements on demand. Using static analysis is useful to define the disposing strategy as well. By using the execution plan, the execution engine can detect which parts of the model are needed and how long this information should be kept in memory. In this way, the execution engine has a plan for executing the program to keep the parts of the model until the program needs them for execution. After that, if the

program does not need elements anymore, the execution engine could unload them from memory (see memory management part).

(2) Effective Metamodel

The output of static analyser is in the form of an effective metamodel for each model involved in the program. The concept of effective metamodel was introduced in [32] in order to support partial loading of XMI files. As for partial loading, we need only the parts of the model which are accessed by the program. In our approach, we load each model according to its effective metamodel instead of the original one.

The structure of effective metamodel is presented in Figure 9. It consists of two classes: *EffectiveMetamodel* class with *name* and *nsuri* attributes and *EffectiveType* with *name*, *attributes* and *references*. The *EffectiveMetamodel* class connects to an *EffectiveType* by *allOfKind*, *allOfType* and *types* references. Figure 10 illustrates the effective metamodel of the EGL program from our motivating example. In comparison with the original metamodel, the effective metamodel does not include any additional information such as Field and Form classes and their properties. Thus, loading the model according to the effective metamodel is expected to be more efficient. The attributes of *EffectiveMetamodel* class are filled by the original metamodel, which are the name and namespace URI (i.e., unique ID in terms of EMF terminology) of the metamodel. Two effective types refer to classes which are necessary for executing the EGL program; Entity and Property in this case. The attributes of classes are according to the attributes which need to be accessed in the code (e.g., name). Thus, in this step, the static analyser helps the execution engine in extracting elements of the model which are necessary for executing the program at compile time in the form of an effective metamodel.

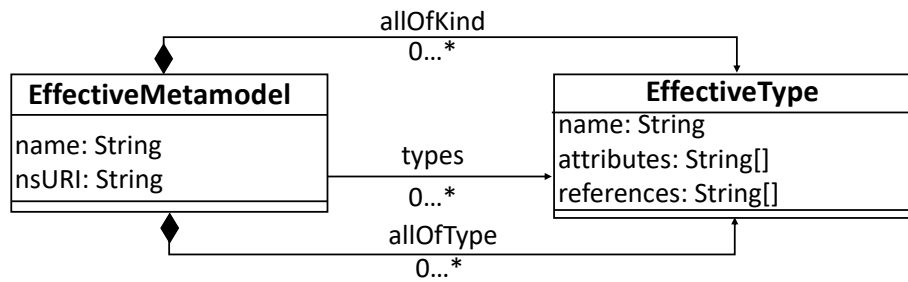


Figure 9: The structure of effective metamodel [32]

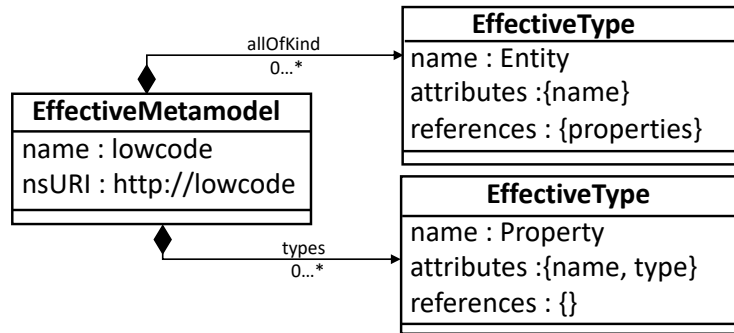


Figure 10: Effective metamodel of EGL code

(3)Memory management

In the next step, the effective metamodel and the model that conforms to it are sent to the execution engine as inputs. Loading only relevant parts of a model into memory is an efficient way to reduce the time of loading but the way that the engine plans to load these parts of model is important.

Considering the motivating example (Listing 1), for applying the *Entity2Class* rule on every *Entity*, one way to partition the model is by *Entity*. Based on this partitioning plan, the execution engine could load every *Entity*, process it and dispose it every time the program finishes using that *Entity*. The engine could load each *Entity* in every network connection (when model is stored in repository) which is efficient in memory but an unsuitable way in terms of performance. On the other hand, the execution engine can load all *Entity* elements in one connection to the network. Loading all *Entity* elements of the model in memory is not efficient as all elements will be kept in memory during the program execution, incurring additional memory consumption. Thus, the execution engine should consider a trade-off between performance and memory consumption for loading model partitions. Consequently, there is a need for intelligent partitioning of models underpinned by sophisticated strategies that use information extracted from code (model management programs) at compile time.

In addition to loading the necessary partitions of the model, another concern is memory consumption. When loaded model elements are no longer needed by the model management program, a sophisticated strategy is required for disposing them from memory. In Listing 1, if the execution engine executes the program for all entities, after the Java class of an entity is generated, there is no need to keep that entity in memory any more. Hence, the memory consumed by that entity can be freed or become available for

loading more elements.

The last phase of this approach is producing an output after executing the program. As there are different types of model management programs, the output would be a model (executing a model transformation program) or code (executing a code generation program). Using this approach, we expect that the program output will be produced more efficiently with reduced loading time because of using intelligent loading strategies and with less memory consumption due to unloading unnecessary model parts.

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 [33]: *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. [34] 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 [35] 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) [36] 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 11 provides an overview of CDO architecture.

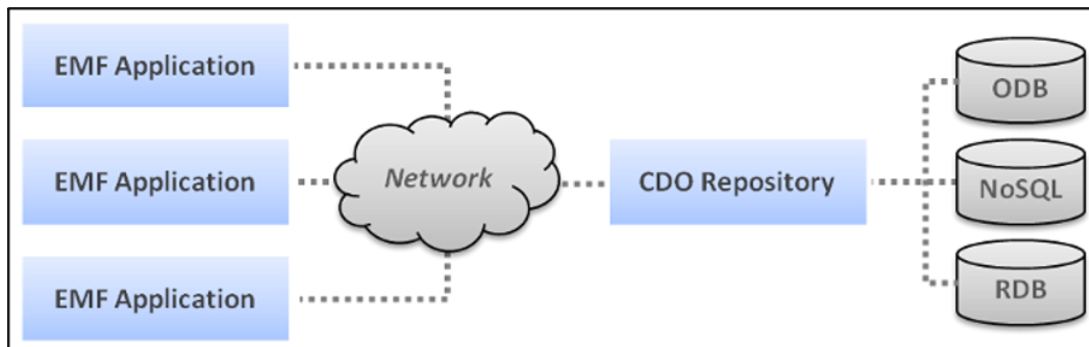


Figure 11: CDO Architecture [36]

As shown in Figure 11, 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 4GB, experimental evaluation with Intel CoreI5 760 PC at 2.80GHz with 8GB of physical RAM in [37] reported an upper bound of 271MB.

Morsa [37] is a persistence solution for storing and accessing large models based on on-demand strategies, which is supported by the NoSQL database. Figure 12 illustrates an overview of Morsa architecture.

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.

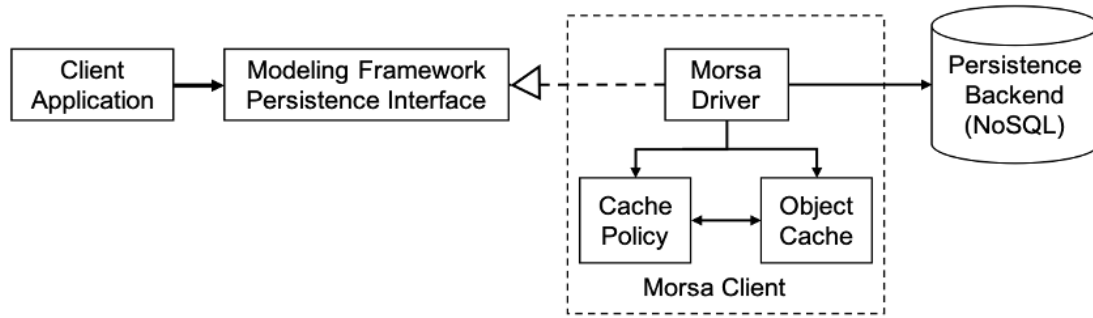


Figure 12: Morsa Architecture [37]

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 [37].

Neo4EMF [38] 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 [39] 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 [32]. 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 [40], 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

In low-code platforms, there is a need to handle very large models (VLMs) [41] for some domains, for example, the models of the Automotive Open System Architecture (AUTOSAR) [42], containing millions of model elements. Other areas with elements in the order of millions include Building Information Modelling and reverse-engineered code from complex systems [41]. While executing complex and computationally expensive queries over such large models, there is a significant performance cost in terms of execution time [43].

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 [44] and non-relational [45] 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 [44] 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.

4.2 Motivating Example

In a low-code platform, often, there can be a need to access heterogeneous models concurrently. Consider a Simulink and UML activity diagram metamodel, as an example as shown in Figures 13 and 14. There are certain requirements and risks for subsystems that are stored in a relational database, an excerpt of the requirements table is shown in Figure 15.

Considering both metamodels and the table structure in the figure, some constraints can be written in the Epsilon Validation Language (EVL) [20] as shown in Listing 3. EVL is the validation language of the Epsilon platform, built on-top of the OCL-based Epsilon Object Language (EOL), which is used to evaluate constraints on the models. In Listing 3, we have a constraint named *SubsystemCounterpart* (Line 1-6) that checks that for every *Activity* in UML model there exists a corresponding *Subsystem* in the Simulink model with the same name and vice-versa (Line 7-10). The constraint *ValidSubsystem* checks that the requirements refer to valid subsystem names (Line 16-21) and the constraint *HasRequirements* checks that there is at least one requirement for every subsystem (Line 11-15).

Now, if we consider evaluating this constraint over a pair of large UML/Simulink models, it would become computationally expensive and slow to execute, as each UML activity will be checked against a

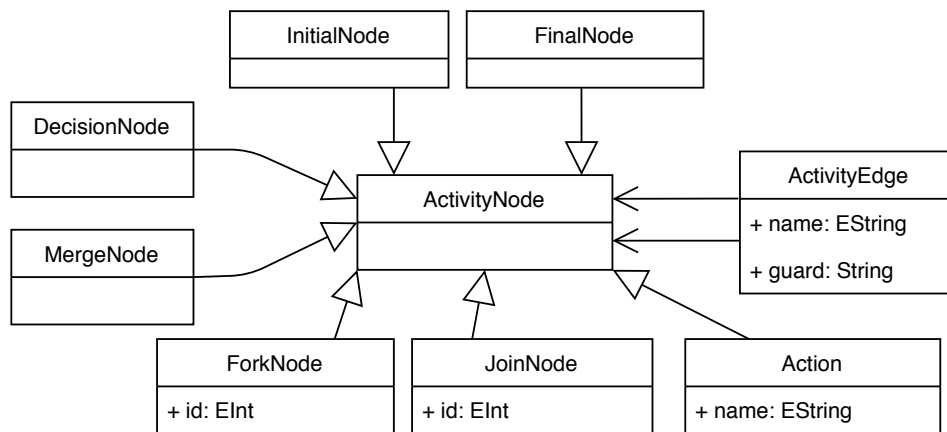


Figure 13: UML Activity Diagram Metamodel

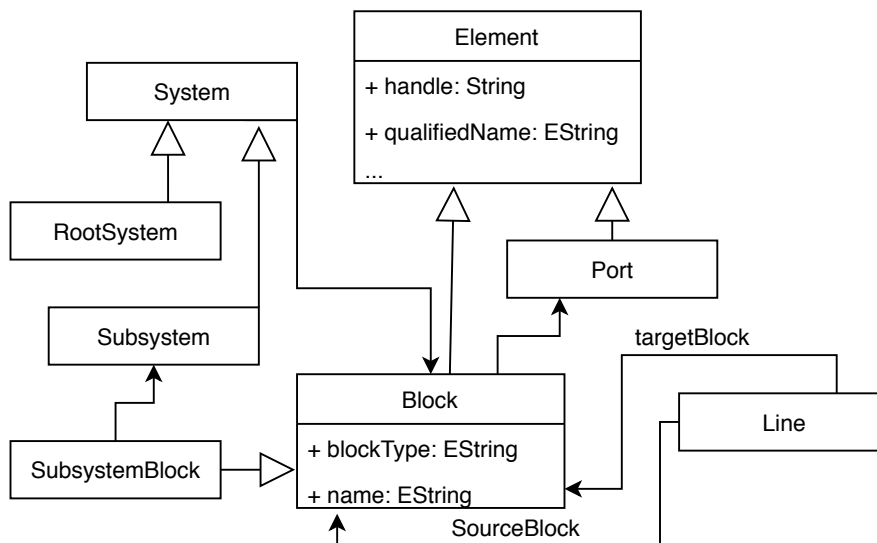


Figure 14: Excerpt of Simulink Metamodel

large number of subsystems within the Simulink model. Writing in a high-level language such as EVL makes it easy to write constraints over heterogeneous modeling technologies using a uniform syntax, but on the other hand, it can also increase computational complexity and memory footprint. The complexity of evaluating these constraints via naive iteration for a Simulink model with N subsystems and a UML model with M activities would be $O(N*M)$ for each constraint.

Listing 3: EVL constraint before optimisation

```

1 context UML!Activity {
2   constraint SubsystemCounterpart {
3     check: Simulink!`simulink/Ports & Subsystems/Subsystem`.allInstances
4       .exists(s|s.name = self.name)
5   }
6 }
7 context `simulink/Ports & Subsystems/Subsystem` {
8   constraint ActivityCounterPart {
9     check: UML!Activity.allInstances.exists(a|a.name = self.name)
10  }
11  constraint HasRequirements {
12    check: Requirements!Requirement.allInstances.exists(r|r.subsystem = self.name)
13  }
14 }
15 context Requirements!Requirement {
16   constraint ValidSubsystem {
17     check : Simulink!`simulink/Ports & Subsystems/Subsystem`.allInstances
18       .exists(s|s.name = self.subsystem)
19   }
20 }

```

One possible optimization here is to translate these into their native query languages, which are often more efficient to execute in. In this case, Simulink has a built-in index-backed *findBlocks* method for looking up elements by type and properties. Here, to speed up this query, a native query that makes use of the

Requirement	
PK	<u>RequirementID</u>
	Subsystem
	FunctionalRequirement

Figure 15: Requirement Table

findBlocks method as shown in EVL Listing 4. This constraint is semantically equivalent to the one shown in Listing 3 but is much faster to execute. Assuming a complexity of $O(1)$ for a hash-based index in Simulink, this would reduce the overall complexity of the constraint to $O(M)$.

To reduce the complexity of the 2nd constraint, we could extend Epsilon’s EMF driver with two new methods. A new *index* method would create a property-based index of type instances in the UML model (i.e. a name-based index of activities in line 2), which could be then used in a *find* method to retrieve instances of that type by property value (i.e. activities by name in line 12), without having to naively iterate through them. With a complexity of $O(M)$ for creating the index in line 2 and a complexity of $O(1)$ for querying it in line 12, the complexity of the 2nd constraint would drop to $O(M) + O(N)$. For the third constraint *HasRequirements*, the query can be translated to the native query language of relational databases (SQL) as shown in the Listing 4, to improve performance. Native queries rewritten in SQL will be much faster to execute for retrieving data stored in relational databases.

Listing 4: EVL constraint after optimisation

```

1 pre {
2   UML.index('Activity', 'name');
3 }
4 context UML!Activity {
5   constraint SubsystemCounterpart {
6     check : Simulink.findBlocks('simulink/Ports & Subsystems/Subsystem', 'name', self.name)
7       .notEmpty()
8   }
9 }
10 context 'simulink/Ports & Subsystems/Subsystem' {
11   constraint ActivityCounterPart {
12     check: UML.find('Activity', 'name', s.name).isDefined()
13   }
14 }
15 constraint HasRequirements {
16   check: Requirements.runSql("select * Requirement where subsystem = '"+ self.name + "'")
17     .size() > 0
18 }
19 }
20 context Requirements!Requirement {
21   constraint ValidSubsystem {
22     check : Simulink.findBlocks('simulink/Ports & Subsystems/Subsystem', 'name', self.subsystem)
23       .notEmpty()
24   }
25 }

```

There are two notable downsides to manually rewriting the constraints to make explicit use of driver technology-specific issues (i.e. Simulink’s *findBlocks* method and the EMF driver’s *find* and *index* methods).

- This kind of optimisation requires expert knowledge of the capabilities of the different modelling tools and drivers.
- Model management programs that make use of these optimisation mechanisms are more verbose and hence difficult to understand and maintain.
- Model management programs become tightly-coupled with the underlying technologies. This would hinder migration to a different modelling technology in the future (e.g. to a non-EMF based UML tool)

The main aim of this work is to investigate how such optimisations can be performed behind the scenes, using static analysis and automated program rewriting so that developers can express model management programs in a technology-agnostic form (as in Listing 3) but still benefit from technology-specific optimisations.

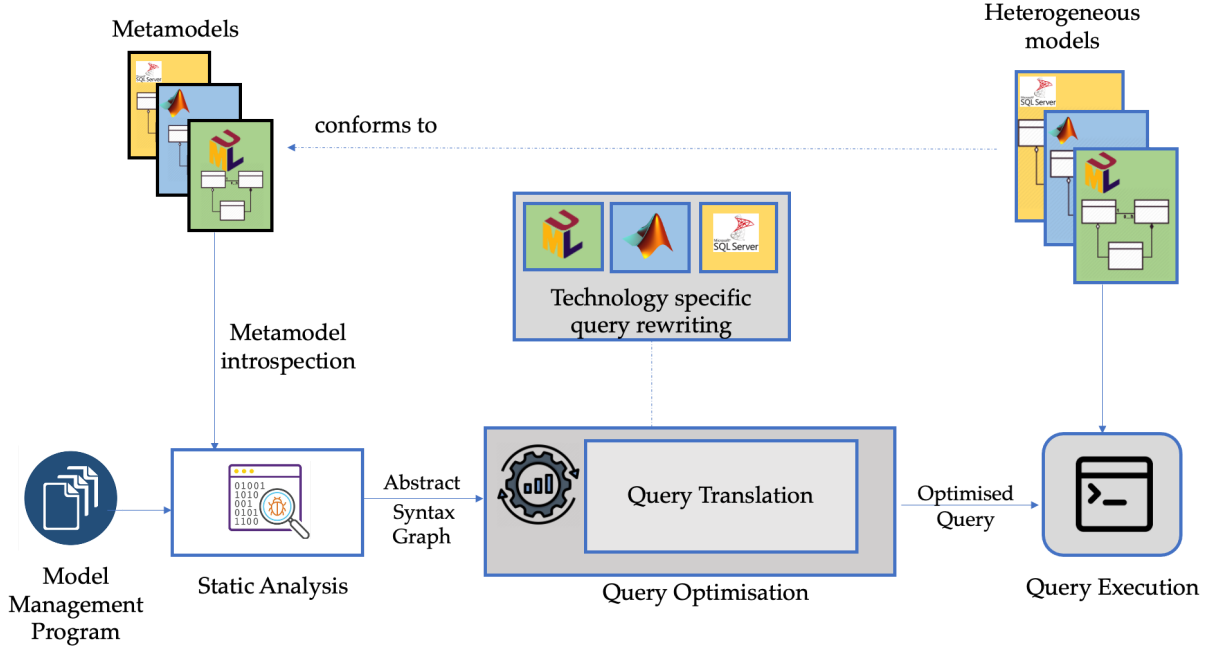


Figure 16: Proposed methodology

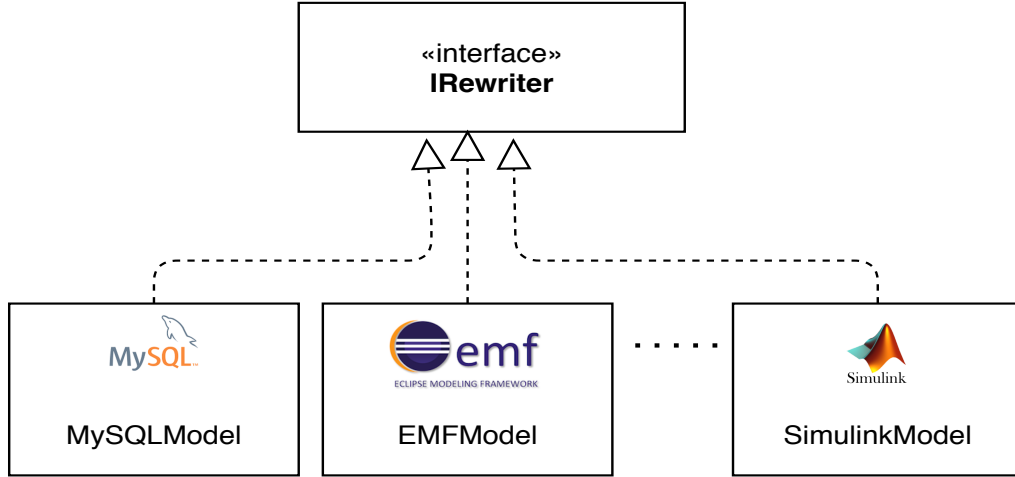


Figure 17: Query Optimisation Architecture

4.3 Proposed Approach

In this section, we present a framework for query optimisation over heterogeneous models in a low-code platform [10]. The aim of this framework is to be able to automatically rewrite expensive queries to make them more efficient in terms of execution time. Query rewriting/translation is based on compile time static analysis. To our knowledge, we have not found the solution to this problem in literature.

An overview of the proposed approach is shown in Figure 16. In a low-code platform, the underlying metamodels can be of different modelling technologies, as depicted in our running example. Furthermore, model management programs, such as queries or transformations, are compiled. At compile-time, a static analysis component will analyze both the program and the metamodels to which its input and output models conform and will yield a type-resolved abstract syntax graph. Static analysis after type resolution can also produce the necessary compile-time errors based on type compatibility as a by-product. The query optimisation block will use the results of such static analysis.

4.3.1 Query Translation

For the whole process of query optimization, let us consider the running example, as shown in Listing 5. For static analysis, a metamodel is extracted from database schema with the following rules:

- Each Database D is mapped to a respective metamodel MM .
- Each Table T in Database D would be mapped to a meta-class of that metamodel MM .
- Each Column C in a table T is mapped to a structural feature or attribute of meta-class $Class$ of that metamodel MM .

To access models at compile-time for the purpose of static analysis, we use *ModelDeclarationStatement*. The syntax of the model declaration system is shown in Listing 5. Model declaration statement specifies model's local name, model's type (in this case MySQL), as well as a set of model-type- specific key-value parameters (in this case server, port, database, username, password, name) that is used to fetch the model's metamodel. This model declaration statement for static analysis is technology-agnostic i.e. we can specify different modelling technologies.

The query optimisation block will have specific optimizers for each back-end technology such as MySQL or Simulink as shown in Figure 17. The architecture supports several orthogonal optimizers as all optimisers operate on the same *AST*, so it is possible that they may interfere with each other. If there is just one optimiser then it would have to know about all the other models accessed by the program in question.

Listing 5: Syntax of Model Declaration Statement

```

1 model Requirements driver MySQL {
2   server = "localhost",
3   port = "3306",
4   database = "requirements",
5   username = "root",
6   password = "",
7   name= "Requirements"
8 };
9 Requirements!Requirement.allInstances.exists(f|f.subsystem = self.name);

```

Every back-end technology can provide different optimizing strategies that can be utilized for efficient querying. For instance, if a program queries three different models conforming to different modelling technologies concurrently, then three individual optimizers for each back-end technology would be invoked. They will each be responsible for the optimisation of queries on their models.

These optimizers would translate queries written in high-level languages such as the Epsilon Object Language and automatically rewrite them in the native query language of their model persistence technology. Query translation and rewriting would be different for different model formats (such as database-backed models, Simulink). All modeling technologies supported by the Epsilon (drivers) implement an EMC-provided Java interface *IModel*. For instance, in the running example, we want to do query optimisation for two types of models, Simulink, and database-backed models. Now, both these drivers already implement *IModel*. We created a new interface for query optimisation known as *IRewriter*. Both these drivers will now implement this *IRewriter* interface. We introduce a new method *rewrite()* in this interface *IRewriter* which will take in an *IEolModule* as a parameter. In EOL, programs are organized in modules i.e. *EolModule* that implement the *IEolModule* interface. Each *EolModule* defines a main body and a number of operations.

Now all model drivers that support compile-time optimisation, will implement the *IRewriter* interface and its *rewrite()* method. One example of this approach is shown in Figure 17. At compile-time, the *rewrite* method is called for all declared models to perform technology-specific query optimisations. In the *rewrite()* method, the *AST* of each statement is passed, which is then translated/rewritten to its native query language, and is replaced with the original *AST* in *EOLModule*.

Now, we will explain by an example how the type-resolved abstract syntax graph can be used to translate certain types of EOL expressions to SQL. For the query translation process, we will consider the running example in Listing 5. In particular, the EOL expression can be translated to more efficient SQL representations:

- *.allInstances* is a property that retrieves all records from a table of database. In translation process *Requirements!Requirement.allInstances* would be translated to *select * from Requirement*.
- *.exists()* is a *FirstOrderOperationCallExpression* that returns true if there is at least one instance in the collection that satisfies the given condition.
- *Requirements!Requirement.allInstances.exists(f—f.subsystem=self.name* would be translated to *Requirement.runSql(select * from Requirement where subsystem = ' + self.name + ').size() > 0*

These optimised queries would be executed on collections of models. For evaluation, we will consider execution time and memory footprint. The decrease in execution time will depend on the underlying model persistence technology. These optimisations can be tested for chain transformations as well (Collaboration with ESR-15).

4.4 Related Work

We can classify related model querying approaches into two main categories as shown in Figure 18.

- Native Querying
- Back-end Independent Querying.

Native querying is efficient as it is tailored for the specific back-end persistence technology: the native language of the model back-end is used. For example, if a model is stored in a relational database, SQL would be used as a query language. The most prominent advantage is this efficiency, as native query languages can have index-based methods, but it also contains several drawbacks [46]:

- Query conciseness: Native queries can be verbose and challenging to write and maintain
- Query abstraction level: Native queries are technology-specific, often requiring considerable effort to change queries if the back-end technology is changed

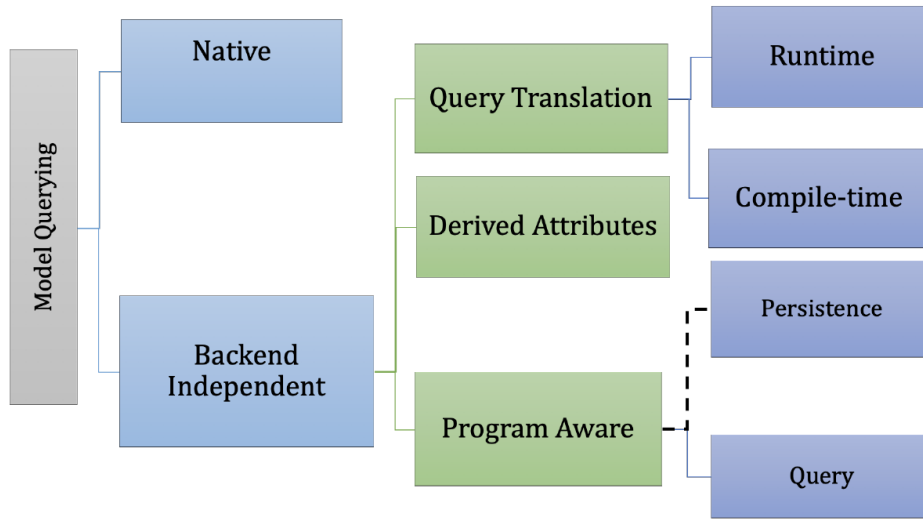


Figure 18: Taxonomy of model querying approaches

Another common way is the use of high-level languages that abstract over model representations and persistence formats. ATL (Atlas Transformation Language) [47], OCL (Object Constraint Language) [24], EOL (Epsilon Object Language) are some examples of such high-level languages. They make use of intermediate layers (such as the OCL pivot metamodel and the Epsilon model connectivity layer) to shield developers from the complexity and particularities of the underlying persistence technologies. The OCL pivot metamodel only supports EMF-based models, while EMC supports several model persistence formats (such as relational database, spreadsheets, Simulink, and EMF-based models). Epsilon offers a driver-based approach, so new technologies can easily be integrated by adding a driver that implements the *IModel* Java interface.

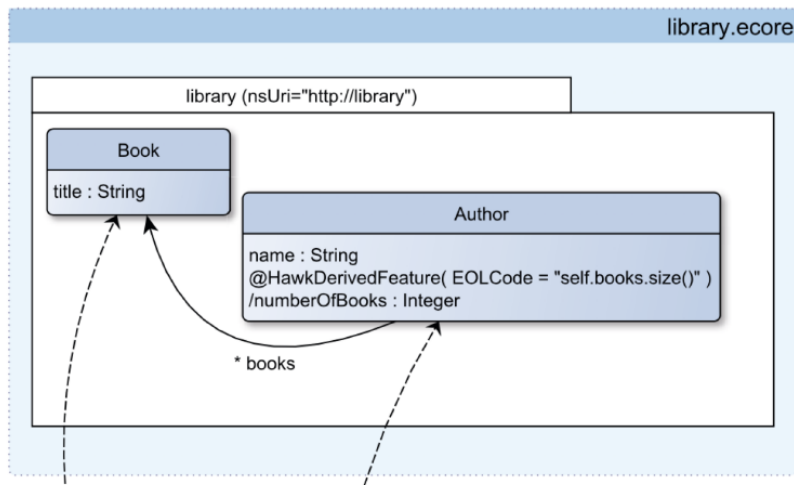


Figure 19: Addition of derived attributes [31]

In [44], the authors discuss the challenges of running OCL based queries on relational database-backed models and propose an approach for translating queries written in higher-level query languages (EOL) to native query languages (such as SQL) at run-time. In [48, 49] authors have proposed ways to generate SQL from OCL expressions. In the Hawk model index [31], an approach has been introduced based on derived features. Authors suggest precomputing such features and cache them in the model index itself as shown in Figure 19. Results have shown a decrease in execution time by using such derived attributes and references, but it has certain shortcomings as well. Firstly, it adds an overhead of computing these derived attributes, which increases the model insertion time containing derived attributes, as well as the overhead of updating the values of these features when the model changes.

Another approach for query optimisation as proposed in [50] is to efficiently compute calls to *allInstances()* queries. The *allInstances()* operation retrieves a collection containing all the members of the element (type) the operation is invoked on. This approach is based on greedy computation instead of on-demand computation. It uses metamodel introspection and compile-time static analysis of queries to:

- Check if the program makes multiple calls to *allInstances()*.
- If yes, then precompute all *allInstances()* collections. Cache all the precomputed collections in one pass.

In [43], the authors present how combining three optimization techniques (parallelization, lazy evaluation, and short-circuiting) can significantly increase the performance of queries over large models. In [51], a tool called Mogwai is proposed for efficient and scalable querying. Mogwai translates OCL and ATL expressions to Gremlin scripts- a query language for NoSQL databases. This shifts the computation of queries at the database (persistence) side, and it makes use of the benefits of optimisation strategies of the specific back-end technology for large models. To address scalability challenges in MDE, one solution is through the use of distributed systems. Pagan et al. [52] propose an efficient query language: MorsaQL (Morsa Query Language) for the MORSA repository [37] – a repository for storing large models in NoSQL databases. The design of MorsaQL is based on the SQL SELECT – FROM – WHERE schema. SELECT describes the type of resulting element, FROM specifies search scope, and WHERE specifies the constraints or condition. Experimentation has shown better performance as compared to OCL, EMF Query, and Plain EMF in terms of efficiency and usability for queries over models stored in Morsa repositories. Some open challenges that still exists in state-of-the-art model querying approaches can be summarized as:

- There is a limited static analysis functionality in high-level query languages that supports heterogeneous modelling technologies. Also, there is limited use of static analysis in model management program analysis and optimisation (e.g query translation, rewriting etc.)
- For the translation of queries to the native language of the persistence technology, mapping from high-level language to different native languages is a challenge for heterogeneous model formats (XMI) and back-ends (Simulink, spreadsheets, databases etc.)
- Query optimisation through derived attributes increases upfront startup cost for pre-computing if those attributes are never used, then it is wasteful to compute derived attributes and also its increasing loading time for models.
- Pre-computing and caching (greedy caching) derived attributes or in case of computing *allInstances* collection in one pass as in [50] not just upfront cost but also increases memory footprint.

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 3, we presented a novel approach that will enable model management languages and engines to eliminate 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 re-accessed) in memory.

According to the designed approach in Section 3.3, we use a static analyser for extracting the effective metamodel, which has information about referenced model elements in code. We are currently working on extracting the effective metamodel in order to detect relevant model elements starting from the incomplete algorithm introduced in [32]. In the future, we plan to devise an algorithm to partition the model in order to load every part of model in an efficient way. This algorithm would be useful in a way that enables the engine to load models in an efficient way.

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 argued that compile-time static analysis and query translation can deliver benefits both in terms of memory footprint and execution time of complex queries. Query translation is used to take benefit from each back-end technology's specific optimizations.

According to the proposed architecture of query optimisation in Section 4.3, we use static analyser to extract the information from model management program. The type-resolved abstract syntax graph is then passed through different query translators/rewriters. Currently, we are working on EOL to SQL query translation. In future, we plan to extend query rewriting for EMF models to create indices based on statically analysed information and make use of those indices at run-time to query models.

To evaluate the intelligent run-time partitioning approach, we will compare it to other state of the art approaches like Neo4EMF and CDO. As the goal of this work is to propose methods for reducing the loading time and memory footprint, they are the factors that we will compare by running model management programs with and without the intelligent model partitioning and disposal facilities. In order to evaluate query optimisation approach, we will preferably use programs that access and query different models (MySQL, Simulink etc.) concurrently to see if the approach performs efficiently in case of heterogeneous models. These model management programs (EOL, EVL) can be both self-written and mined from public repositories on GitHub.

We will consider the models proposed in the GraBaTs 2009 contest as test cases [53] for both scalable persistence and query. The models conform to the JavaMetamodel metamodel. There are five models, from Set0 to Set4, each one larger than its predecessor (from a 8.8MB XMI file with around 70k model elements representing 14 Java classes to a 646MB file with 5M model elements representing 5984 Java classes).

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 Kärrä, 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] 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.
- [12] Dimitris Kolovos, Louis Rose, Antonio Garcia-Dominguez, and Richard Paige. *The Epsilon Book. Eclipse*. 2010.
- [13] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. Merging models with the epsilon merging language (eml). In *International Conference on Model Driven Engineering Languages and Systems*, pages 215–229. Springer, 2006.
- [14] Louis Rose, Richard Paige, Dimitrios Kolovos, and Fiona Polack. The epsilon generation language. In *European Conference on Model Driven Architecture - Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science book*, pages 1–16. Springer, 2008.
- [15] Louis M Rose, Dimitrios S Kolovos, Richard F Paige, Fiona AC Polack, and Simon Poulding. Epsilon flock: a model migration language. *Software & Systems Modeling*, 13(2):735–755, 2014.
- [16] Dimitrios S Kolovos. Establishing correspondences between models with the epsilon comparison language. In *European Conference on Model Driven Architecture—Foundations and Applications*, pages 146–157. Springer, 2009.
- [17] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*, pages 46–60. Springer, 2008.
- [18] Dimitrios S Kolovos, Richard F Paige, Fiona AC Polack, and Louis M Rose. Update transformations in the small with the epsilon wizard language. *Journal of Object Technology (JOT)*, 6(9), 2003.
- [19] Dimitris S Kolovos and Richard F Paige. The epsilon pattern language. In *2017 IEEE/ACM 9th International Workshop on Modelling in Software Engineering (MiSE)*, pages 54–60. IEEE, 2017.
- [20] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. On the evolution of ocl for capturing structural constraints in modelling languages. In *Rigorous Methods for Software Construction and Analysis*, pages 204–218. Springer, 2009.
- [21] 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.
- [22] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Analyzer: An advanced ide for atl model transformations. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE '18*, page 85–88, New York, NY, USA, 2018. Association for Computing Machinery.
- [23] 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.
- [24] Edward Daniel Willink. Aligning ocl with uml. *Electronic Communications of the EASST*, 44, 2011.
- [25] Louis M Rose, Dimitrios S Kolovos, and Richard F Paige. Eugenia live: a flexible graphical modelling tool. In *Proceedings of the 2012 Extreme Modeling Workshop*, pages 15–20, 2012.
- [26] Zoltán Ujhelyi. Static analysis of model transformations. Master’s thesis, Budapest University of Technology and Economics, May 2009.
- [27] Edward D. Willink. Modeling the OCL standard library. *ECEASST*, 44, 2011.
- [28] 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.
- [29] 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.
- [30] 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.
- [31] Konstantinos Barmpis and Dimitrios S Kolovos. Towards scalable querying of large-scale models. In *European Conference on Modelling Foundations and Applications*, pages 35–50. Springer, 2014.
- [32] 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.
- [33] 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.
- [34] Frédéric Jouault, Jean Bézivin, and Mikael Barbero. Towards an advanced model-driven engineering toolbox. *Innovations in Systems and Software Engineering*, 5(1):5–12, 2009.
- [35] 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.
- [36] Eike Stepper. Cdo, November 2016.
- [37] Javier Espinazo Pagán, Jesús 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.
- [38] 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.
- [39] 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.
- [40] 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.
- [41] Massimo Tisi, Salvador Martinez, and Hassene Choura. Parallel execution of atl transformation rules. In *International Conference on Model Driven Engineering Languages and Systems*, pages 656–672. Springer, 2013.
- [42] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkampfer, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. Autosar—a worldwide standard is on the road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, volume 62, page 5, 2009.
- [43] Sina Madani, Dimitris Kolovos, and Richard F Paige. Towards optimisation of model queries: a parallel execution approach. *Journal of Object Technology*, 18(2), 2019.
- [44] 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.
- [45] 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.
- [46] Konstantinos Barmpis and Dimitrios S Kolovos. Evaluation of contemporary graph databases for efficient persistence of large-scale models. *Journal of Object Technology*, 13(3):3–1, 2014.
- [47] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl: a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720, 2006.
- [48] Marina Egea and Carolina Dania. Sql-pl4ocl: an automatic code generator from ocl to sql procedural language. *Software & Systems Modeling*, 18(1):769–791, 2019.
- [49] Marina Egea, Carolina Dania, and Manuel Clavel. Mysql4ocl: A stored procedure-based mysql code generator for ocl. *Electronic Communications of the EASST*, 36, 2010.
- [50] Ran Wei and Dimitrios S Kolovos. An efficient computation strategy for allinstances (). In *BigMDE@ STAF*, pages 32–41, 2015.
- [51] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. Scalable queries

- and model transformations with the mogwai tool. In *International Conference on Theory and Practice of Model Transformations*, pages 175–183. Springer, 2018.
- [52] Javier Espinazo Pagán and Jesús García Molina. Querying large models efficiently. *Information and Software Technology*, 56(6):586–622, 2014.
- [53] Grabats2009: 5th int. workshop on graph-based tools (2012).