Identification and Optimisation of Type-Level Model Queries

Qurat ul ain Ali
Department of Computer Science
University of York
York, UK
quratulain.ali@york.ac.uk

Dimitris Kolovos
Department of Computer Science
University of York
York, UK
dimitris.kolovos@york.ac.uk

Konstantinos Barmpis
Department of Computer Science
University of York
York, UK
konstantinos.barmpis@york.ac.uk

***Abstract*—The main appeal of task-specific model management languages such as ATL, OCL, Epsilon etc. is that they offer tailored syntaxes for the tasks they target, and provide concise first-class support for recurring activities in these tasks. On the flip side, task-specific model management languages are typically interpreted and are therefore significantly slower than general-purpose programming languages (which can be also used to query and modify models) such as Java. While this is not an issue for smaller models, as models grow in size, naive execution of interpreted model management programs against them can become a scalability bottleneck. In this paper, we demonstrate an architecture for optimisation of model management programs written in languages of the Epsilon platform using static analysis and program rewriting techniques. The proposed architecture facilitates optimisation of queries that target models of heterogeneous technologies in an orthogonal way. We demonstrate how the proposed architecture is used to identify and optimise type-level queries against EMF-based models in the context of EOL programs and EVL validation constraints. We also demonstrate the performance benefits that can be delivered by this form of optimisation through a series of experiments on EMF-based models. Our experiments have shown performance improvements of up to 99.56%.**

***Index Terms*—Model-Driven Engineering, Scalability, Model Querying, Static Analysis**

I. Introduction

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 flavours embedded in model-to-model and model-to-text transformation languages such as Acceleo and ATL – the Epsilon Object Language (EOL) and the task-specific languages that build on top of it, and the VIATRA Query Language (VQL). The main strength of dedicated model management languages is that they offer built-in abstractions for common tasks (e.g. rule-based decomposition and element resolution in model-to-model transformation, protected regions for mixing generated and hand-written content in model-to-text transformation, constraint dependency management in model validation) which facilitate more concise, maintainable and technology-independent model management programs.

The 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 [1], [2].

In this paper, 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. 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, and we have observed performance improvements of up to 99.56%. A speculative overview of the proposed approach was first presented in a workshop paper [3] but without a supporting implementation, or the ability to carry out evaluation experiments, at that stage.

The remainder of this paper is organized as follows: Section II, presents a motivational example and identifies the performance challenges that this work seeks to address. Section III, presents the architecture of the proposed program optimisation approach over EMF models. Evaluation experiments and the obtained results are presented and discussed in Section IV. Section V discusses relevant work in the field of model query optimisation and static analysis. Finally, Section VI concludes the paper and presents direction for further work.

II. Background and Motivation

This section provides some background on tools and technologies used for the implementation of the proposed approach. Then, it presents a motivating example scenario that illustrates the importance of query optimisation.

A. Epsilon

Epsilon [1] is a family of task-specific languages for automating common model-based software engineering activities. These activities include model merging (EML), code generation (EGL), model migration (Flock), model comparison (ECL), model to model transformation (ETL), model refactoring (EWL), pattern matching (EPL) and model validation (EVL). Epsilon supports models from heterogeneous modelling technologies such as EMF, UML, Simulink, XML and others. EOL is the core language of Epsilon and is extended by all other languages. Epsilon languages access models through Epsilon Model Connectivity (EMC) layer, offering a uniform interface for interacting with different modelling technologies. While we have used Epsilon to implement and evaluate the proposed query optimisation approach, the approach is trivially portable to other model OCL-based model management languages such as ATL [4] and Acceleo [5] too.

B. Motivating Example

Consider a scenario where we query a model for the purpose of validating a model conforming to the UML2 [6] EMF-based metamodel. In particular, we wish to check that:

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

The relevant subset of the UML2 metamodel and implementations of the two constraints (using the Epsilon Validation Language) are illustrated in Figure 1 and in Listing 1 respectively. These constraints can be written in any model management language. Here we consider EVL, as it allows cross validation between models of various backend technologies. In Listing 1, the *UniqueName* constraint checks that for every *Class* in the model, its name attribute is unique (lines 9-14). Similarly, *IsCalledInSequenceDiagram* constraint checks that every *Operation* is called in a sequence diagram at least once (lines 16-21).

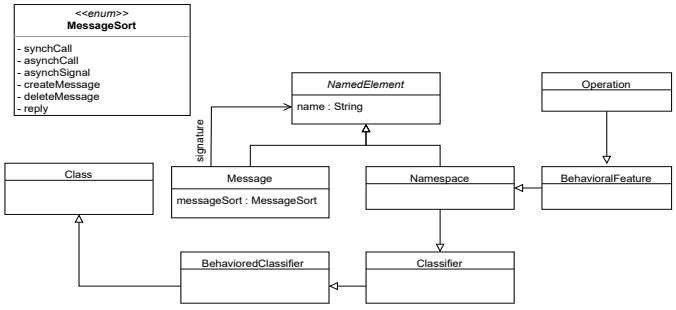

Fig. 1. An excerpt of the UML2 metamodel

```
1 model UML driver EMF {
2   nsuri =  "http://www.eclipse.org
3   /uml2/5.0.0/UML"
4 };
5 pre { }
6 context Class {
7   constraint UniqueName {
8     check: not Class.all.exists
9     (c|c.name = self.name and self != c)   }
10  }
11 context Operation {
12   constraint IsCalledInSequenceDiagram {
13     check: Message.all.exists
14     (m | m.signature = self) }
15 }
```

Listing 1. Example EVL validation constraints before optimisation

Epsilon languages use *all* as an alias for *allInstances()*. In Listing 1, *Class.all* in line 10 and *Message.all* in line 17 retrieve all instances of *Class* and *Message* anywhere in the model respectively. Evaluating these constraints over a UML model containing a large number of classes and operations would be computationally expensive. More specifically, the complexity of *UniqueName* constraint is *O(N*N)* if we consider the number of Classes to be *N*, and the complexity of evaluating *IsCalledInSequenceDiagram* over *M* operations and *P* messages would be *O(M*P)*. Reverse navigation is a recurring issue in model management programs [7] when working with EMF models. For example in UML model, navigating from *Operation* to *Message*. A common workaround to reduce complexity in such occasions is to define opposite references (e.g 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 are constructing the metamodel or to naively add opposites for all references in the metamodel.

```
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 context Operation {
16   constraint IsCalledInSequenceDiagram {
17     check: UML.findByIndex("Message",
18     "signature", self).size() > 0 }
19 }
```

Listing 2. Example EVL validation constraints after optimisation

To speed up this type of model validation, one optimisation strategy is to programmatically create in-memory indices and then use them for look-ups. Existing languages such as Acceleo offer different facilities for this e.g., search for *eInverse* [5]. Another approach for OCL is shown in [7]. 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 2. These constraints are semantically equivalent to the

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

ones in Listing 1 but are much faster to execute. In Line 7 of Listing 2, an index is constructed which maps names to lists of classes with their name attribute, rather than naively iterating through all the instances of *Class*. Similarly, in Line 8, an index is constructed which maps names to lists of messages with their signature attribute, rather than naively iterating through all the instances of *Message*. Then in constraints, these constructed in-memory indices (Lines 13-15, 21-22) are searched instead. As in-memory indices can be stored as hashmaps, finding UML classes by names and similarly finding messages by signature, the computation cost would be that of a hash function. Considering the complexity of hash functions being *O(1)*, the overall complexity of both the constraints would be reduced to *O(N)* and *O(M)*, respectively.

This paper provides an approach for detecting optimisation opportunities such as the ones shown in the above example and then automatically rewriting relevant model management programs accordingly. This research focuses on investigating how such optimisations can be performed behind the scenes, using static analysis and automated program rewriting so that developers can express model management programs in a naive form (as in Listing 1) and benefit from index-based optimisation (as in Listing 2) as seemlessly as possible.

III. Query Optimisation

This section discusses the proposed query optimisation architecture in detail, an overview of which is illustrated in Figure 2. This approach takes a model management program as input and passes it through a static analyser component to compute an abstract syntax graph. The abstract syntax graph (type-resolved abstract syntax tree) is input to the query optimiser block, (this part can involve multiple query optimisers, up to one for each modelling technology) which outputs the rewritten optimised program to be executed.

This query optimisation architecture works with heterogeneous modelling technologies as each such technology can offer its own technology-specific optimisations. Also, this architecture is extensible: any modelling technology that has to offer query optimisation can register its own rewriter. A rewriter has its own *rewrite()* method whereby before program execution, after static analysis, all the rewriters of the modeling technologies involved in the program are invoked to perform technology specific query optimisations. The responsibility of rewriting into semantically equivalent programs is delegated to the individual optimisers involved and hence each optimiser should be tested for its correctness, further elaborated in Section IV. The optimised program that is semantically equivalent to the input program, is then executed instead.

A. Static Analysis

Static analysis is the first step of the proposed query optimisation approach. An overview of the process of static analysis is depicted in Figure 3. Beyond the typical activity of checking the program for type-related errors and warnings, static analysis is useful for extracting information useful for program optimisation. This information mainly include type information of elements and control flow of the program. Static analysis of Epsilon programs was initially discussed in [8]. We extended the open-source static analyser presented in [8] by adding features such as type inference and type resolution using metamodel introspection. Static analysis capabilities were implemented for EOL, as it is the core language of Epsilon with all other languages extending it. We then extended this EOL static analyser to create an EVL static analyser, with both EOL and EVL static analysers providing error and warning reporting as well.

Let us consider Listing 1 to see how type resolution works. *Class* in line 8 would be resolved to the respective model element type of the UML model. In line 11, *c* and *self* are variables and are inferred to be of type *Class*, as *Class.all* is a collection of Classes. Hence, *c.name* and *self.name* would be resolved to be of String type. Overall the resolved type of the expression in the *check* part of the constraint *UniqueName* in Line 9-12 is boolean. Similarly for the second constraint *IsCalledInSequenceDiagram*, *UML!Operation* in line 15 would be resolved to the *Operation* model element type. *Message.all* in line 17 returns a collection of *Messages* and therefore the type of *m* is inferred as *Message*. Similarly, In line 18, the types of *self* and *m.signature* are resolved to *Operation*.

B. Finding Optimisable Queries

The second step of the approach is to find potential opportunities for speeding up queries using indices. Our query optimiser operates over programs that consume EMF models. The indexing approach is done only for EMF models as for some types of models (e.g. Simulink) it is possible to look up elements by feature values using built-in indices maintained by the modelling tool. The first step of the process is to find potential indices by visiting the entire program. The static analyser detects where the user is retrieving all instances of a type, filtered by a specific property or attribute, then only for such properties will indices be created. This approach works by detecting expressions in the form of *Class.all.operation(...)* to optimise. It currently supports filtering operations like *select*, *selectOne* and *exists*, while it can be extended to support other first order operations as well, as discussed in the Further Work Section VI. As all Epsilon languages are built on top of EOL, certain expressions in an Epsilon program may be executed just once such as Line 9 in Listing 4. For such expressions the overhead of the computation of indices would not pay off if that index is to only be used once. To tackle this issue, we need to find the expressions that are likely to be executed multiple times in a script. We use Algorithm 1 to carry out call graph analysis and identify such expressions. *OptimiseBlock* method is a recursive method and calls *OptimiseStatement* for every atomic statement. Finally *OptimiseStatement* method checks if the statement is optimisable or not and then added to a list of potential indices.

The condition expression in the detected first-order expressions, which can be executed multiple times, can have logical operators. The condition expression abstract syntax graph is decomposed into each logical operand and then indexed

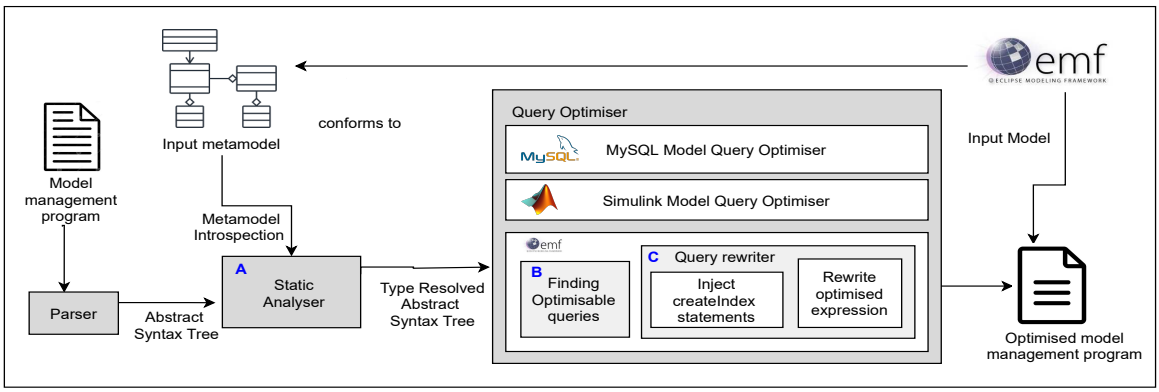

Fig. 2. Architecture of the Query Optimisation Approach

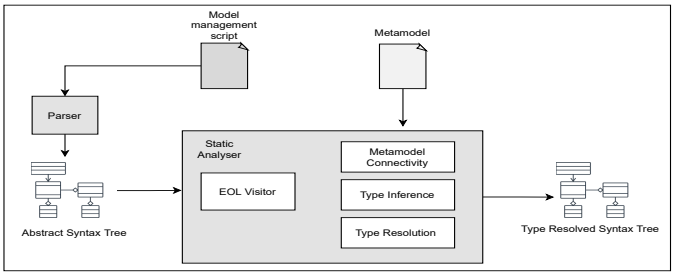

Fig. 3. The structure of the static analyser

separately based on the type of logical operator. Indexing for "and" and "or" logical operator conditions is shown in Table I.

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

```
1 var a : Class = Class.all.first();
2 a.printName();
3
4 operation Message printName() {
5   return self.name.println("Message Name:");;
6 }
7 operation Class printName() {
8   return self.name.println("Class Name:");
9 }
10 operation Any printName(){
11   return self.name.println("Name:");
12 }
```

Listing 3. Context type polymorphism example

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

We pass the type resolved AST to a call graph generator component, which generates the input program's call graph. An example of a call graph generated from Listing 4 is shown in Figure 4. Call graphs for EOL programs can be visualised on the fly using Graphviz[2] through Picto [9].

```
1 model UML driver EMF {
2   nsuri = "http://www.eclipse.org/uml2
3   /5.0.0/UML"
4 };
5 printMessagesofReplySort();
6 getClassByName("A").println("Class A:");
7 operation getClassByName(name: String) {
8   Class.all.select(a|a.name = name);
9 }
10 operation printMessagesofReplySort() {
11   for(n in Message.all.select
12   (m|m.messageSort = MessageSort#reply)) {
13     getMessageByName(n.name).println();
14   }
15 }
16 operation getMessageByName(name: String) {
17   Message.all.select(a|a.name = name);
18 }
```

Listing 4. Example of call graph program input

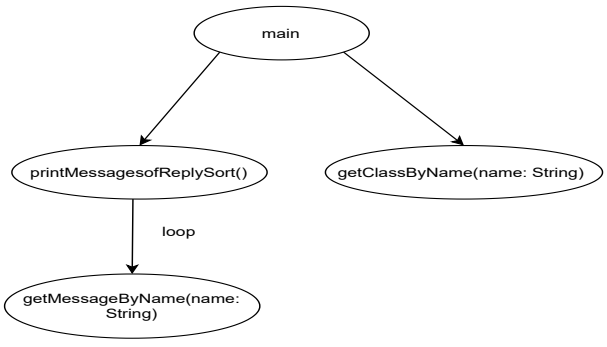

Fig. 4. Generated Call Graph (from Listing 4)

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

TABLE I
INDEXING AND QUERY REWRITING FOR LOGICAL OPERATORS

Input EOL Expression	Rewritten EOL Expression
UML!Class.all.select(c | c.name = ”ClassA” or c.visibility = ”public”)	UML.findByIndex(”Class”, ”name”,”ClassA”).includingAll(UML.findByIndex(”Class”,”visibility”,”public”))
UML!Class.all.select(c | c.name = ”ClassA” and c.visibility = ”public”)	UML.findByIndex(”Class”, ”name”,”ClassA”).select(c | c.visibility = ”public”)
UML!Class.all.select(c | c.name = ”ClassA” and c.name = ”ClassB”)	UML.findByIndex(”Class”, ”name”,”ClassA”).includingAll(UML.findByIndex(”Class”,”name”,”ClassB”))
UML!Class.all.select(c | isPublic(c) and c.name = ”ClassA”)	UML!Class.all.select(c | isPublic(c) and c.name = ”ClassA”)
UML!Class.all.select(c | c.visibility = ”Public” and not c.returnType = null)	UML.findByIndex(”Class”, ”visibility”,”Public”).select(c | not c.returnType = null)

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

C. Query Rewriting

After collecting potential indices i.e., class-feature pairs, by analysing the input program in the first phase, the final phase is to rewrite the program. The program is traversed again to find expressions which can leverage the created in-memory indices. This rewriting is performed behind the scenes: it does not alter the original program nor is it visible to the user (unless they wish to see it in which case there is a dedicated eclipse view for this, detailed below). Rewriting includes two main tasks: i) Injecting *createIndex* statements for creating in-memory indices ii) Rewriting the relevant expressions to *findByIndex* statements, where these indices are used. The respective syntax of *createIndex()* and *findByIndex()* statements is showcased in Listings 5 and 6.

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

Listing 5. Syntax of CreateIndex Statement

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

Listing 6. Syntax of FindByIndex Statement

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

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

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

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

Rewriting is performed behind-the-scenes, before the execution of the program. The original lines and column coordinates of ASTs are maintained, so that if exceptions occur at runtime, they are reported at the correct location in the original program. If the user wishes to visualise this automated program rewriting, we have implemented a query rewriting view shown in Figure 5, which displays the rewritten program of the EOL or EVL file in the currently active editor.

Fig. 5. Screenshot of the Query Rewriting View

Algorithm 1 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)
        end if
    end for
end procedure
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
        end for
    end if
    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
                        end if
                    end if
                end if
            end if
        end if
    end if
end procedure
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
        end if
    OPTIMISEBLOCK(body of op)
    end if
end for
```

IV. Evaluation

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

A. Experiment Setup

TABLE II
Specifications of java models used for benchmarking

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

The execution-time performance of the proposed approach to optimise EVL programs over large-scale EMF models has been evaluated. Since Epsilon supports parallel execution [10] for EVL, the proposed approach is compared with the parallel mode of EVL execution. The first experiment evaluates the constraints using EVL without optimisation with parallel mode enabled. The second evaluates the use of our rewriting strategy (with an extension of EMF EMC driver with two additional *createIndex* and *findByIndex* methods discussed in Section III-C) also in the parallel mode of EVL. In the rest of the section, the first approach is referred to as EVL– since it executes the EVL programs in a naive parallel mode, while the second one is referred to as EVL-QR – since it makes use of the query rewriting strategy, on the top of the EVL engine in parallel mode. 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.

Constraints and Models: For evaluating the query optimisation approach, the validation constraints that we used were introduced in [10] and are based on the Findbugs [11] project, a static analysis tool that reports a large number of "code smells" in Java code. Our EVL script (*Java_findBugs*) consists of 31 constraints over 17 contexts, and 11 operations. We execute the *Java_findBugs* script over a set of large models reverse-engineered from the Java source code of Eclipse projects [12] using MoDisco [13]. We opted for the Modisco Java metamodel, as it is both complex enough and relatively familiar to Java programmers. Also, such reverse engineered models are commonly used to evaluate the scalability of MDE tools [14], [15]. The models that we used vary from approximately 100k to over 4 million elements, as illustrated in Table II.

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

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

B. Results

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

Static analysis and query rewriting works at the metamodel level and does not require any information from models themselves. Static analysis and program rewriting took less than 50ms for all the experiments, and therefore the overhead incurred can be seen as negligible, with respect to the overall execution times observed for these experiments. Also, this computation time is independent of model size, due to the fact that the whole process of query optimisation only uses metamodel introspection. Time for static analysis and query rewriting depends on two major factors: the size of the program under consideration and the size of the underlying model's metamodel. To investigate the most computationally expensive constraints, we measured the distribution of overall execution times of the validation program. We divided this program into two parts and reported execution times for the first constraint as *FindBugs_First* and then execution times of the remainder of the constraints as *FindBugs_Rest* in Table IV. Out of 31 constraints in the *Java_FindBugs* script, the first constraint named *allImportsAreUsed* is the most expensive one, as illustrated in Table IV. *allImportsAreUsed* being very demanding, takes 99% of the execution time and it contains an expression that is optimisable using our technique. Due to

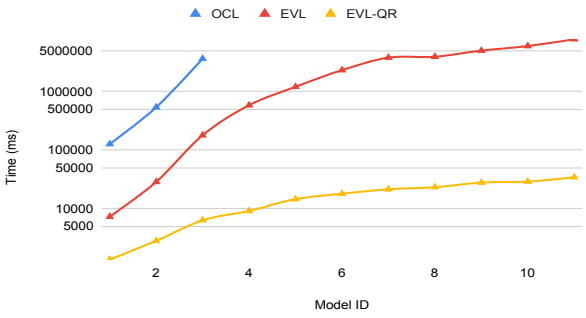

Fig. 6. Comparison of OCL, EVL and EVL QR

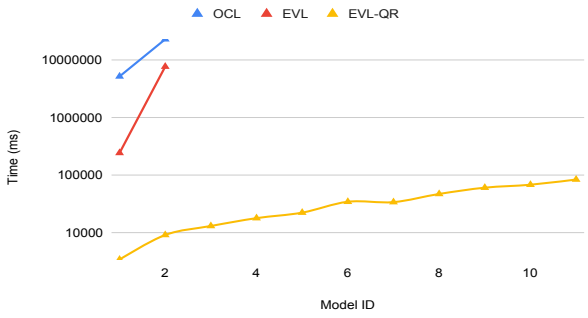

Fig. 7. Comparison of OCL, EVL and EVL QR without eOpposites

TABLE III
EXECUTION TIME IN SECONDS

Model ID	1	2	3	4	5	6	7	8	9	10	11
OCL	126.9	538.7	3649.5	TO	TO	TO	TO	TO	TO	TO	TO
EVL	7.29	28.6	179.3	583.5	1206.7	2322.6	3797.4	3934.9	5003.6	5987.3	7826.9
EVL-QR	1.3	2.8	6.3	9.1	14.5	17.9	21.3	23.2	27.8	28.9	34.3
Speedup vs EVL	5.6	10.21	28.46	64.1	83.22	129.75	178.28	169.60	179.98	207.17	228.18
Speedup vs OCL	97.61	192.39	579.28	-	-	-	-	-	-	-	-
OCL (No eOpp)	5256	23263	TO	TO	TO	TO	TO	TO	TO	TO	TO
EVL (No eOpp)	24579	777997	TO	TO	TO	TO	TO	TO	TO	TO	TO
EVL-QR (No eOpp)	3.44	9.22	13.20	18.07	22.53	34.81	34.32	47.58	61.154	68.84	84.77
Speedup vs EVL	7145	84564	-	-	-	-	-	-	-	-	
Speedup vs OCL	1527.9	2522.9	-	-	-	-	-	-	-	-	

this, in the case of EVL Query rewriting we see a significant improvement in performance, by just creating one index.

TABLE IV
DISTRIBUTION OF EXECUTION TIME IN FINDBUGS SCRIPT

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

Observing the comparison graph shown in Figure 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. The results also suggest that the larger the model size, the more the performance gain is in terms of execution time. This can be explained by the fact that in smaller models, the overhead of creating indices is proportionally more, than for larger models.

We also performed experiments on the same validation constraints after removing the opposite references from the Java metamodel. The reason for removing opposites is to create more room for optimisations as sometimes adding opposite references is not an option, such as for standardised metamodels (See Section II). The original Java metamodel has 173 references in total out of which 48 have opposite

references. We removed opposites on the following criteria: if a reference is containment, we remove its opposite reference. if a reference is non-containment, then we remove one of the pair of opposites based on alphabetical order. In total, we removed 23 opposite references, which leaves 150 references in the modified metamodel. Model migration was then carried out to update the original models to conform to the new metamodel without opposites using Flock [16]. EVL validation constraints and OCL constraints were also updated to not make use of the removed opposite references. For example, *VariableDeclaration* class had an opposite reference to *SingleVariableAccess*. The *variableIsUsed* constraint is written originally as:

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

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

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

This experiment has as goal to measure the performance of query optimisation when having opposites (to speed up certain classes of queries) is not possible. The graph shown in Figure 7 illustrates the comparison between EVL and EVL-QR with no opposite references in the model. Validation with EVL is so computationally expensive in this case, that it timed out(TO) even for models with around 500K elements. Utilising EVL-QR, shows a performance gain of over 84564x in comparison with EVL for the experiments that were completed. EVL-QR provides a performance gain of over 2522x while comparing with OCL. When there are opposite references, there is still more room for creating in-memory indices and thus reducing the execution time overall, as some queries may have to keep navigating through the entire model to find matching elements (that could have otherwise been navigable through an opposite reference).

C. Threats to validity

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

V. Related Work

This section, summarises existing work within the scope of this article in two main categories: First, it lists existing tools in MDE that provide static analysis facilities; and second it discusses model query optimisation strategies.

AnATLyzer [17] 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.

Another tool [18] 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 [19], where a pseudo-type *OCLSelf* is introduced to infer the type of built-in operations such as *oclAsSet()* and *oclType()*. Willink [20] 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.

AnATLzyer, is used in [21] to develop an A2L compiler for parallel execution of ATL transformations. It uses static analysis to generate efficient code at the transformation level which results in improved performance.

In Hawk [22], a derived attributes approach includes pre-computing certain expensive features and caching them in the model index. Results have shown a decrease in execution time by using derived attributes, but it has certain shortcomings as well. Firstly, it adds an overhead of computing these derived attributes, which increases the model insertion time containing derived attributes, as well as the overhead of updating the values of these features when the model changes. However, these attributes are defined by the user and to the best of our knowledge, there is no automatic detection of optimisation opportunities through static analysis such as the one proposed in this paper.

Another approach presented in [23] is to execute calls to *allInstances()* queries efficiently. This approach is based on greedy computation instead of on-demand computation. It checks if the program makes multiple calls to *allInstances()*, then precomputed all collections and caches them in one pass. The approach just works on *allInstances()* calls.

In [24], the authors present how combining three optimisation techniques (parallelisation, lazy evaluation, and short-circuiting) can significantly increase the performance of queries over large models. It requires the use of the parallel variant of EOL, which can be automated through static analysis, as is the case in our proposed approach. We use parallelisation proposed in [24] as a comparison baseline for our approach. In [25], a tool called Mogwai is proposed for efficient and scalable querying. Mogwai maps OCL and ATL expressions to Gremlin scripts – a query language for NoSQL databases. This leverages the optimisations implemented by the underlying database technology.

VI. Conclusions and Further Work

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

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

Acknowledgments

The authors are grateful to Sorour Jahanbin and Dr. Simos Gerasimou for their collaboration in static analysis of Epsilon. This research is supported by the Lowcomote, funded by the EU's H2020 Research and Innovation Programme under the Marie Skłodowska- Curie GA n° 813884.

References

[1] A. Bucchiarone, J. Cabot, R. F. Paige, and A. Pierantonio, "Grand challenges in model-driven engineering: an analysis of the state of the research," *Software and Systems Modeling*, vol. 19, no. 1, pp. 5–13, 2020.

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

[3] Q. u. a. Ali, D. Kolovos, and K. Barmpis, "Efficiently querying large-scale heterogeneous models," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, ser. MODELS '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3417990.3420207

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

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

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

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

[8] R. Wei and D. S. Kolovos, "Automated analysis, validation and suboptimal code detection in model management programs," in *Proceedings of the 2nd Workshop on Scalability in Model Driven Engineering co-located with the Software Technologies: Applications and Foundations Conference, BigMDE@STAF2014, York, UK, July 24, 2014*, ser. CEUR Workshop Proceedings, D. S. Kolovos, D. D. Ruscio, N. D. Matragkas, J. de Lara, I. Ráth, and M. Tisi, Eds., vol. 1206. CEUR-WS.org, 2014, pp. 48–57.

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

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

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

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

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

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

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

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

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

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

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

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

[21] J. Sanchez Cuadrado, L. Burgueno, M. Wimmer, and A. Vallecillo, "Efficient execution of atl model transformations using static analysis and parallelism," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

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

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

[24] S. Madani and D. K. R. F. Paige, "Towards optimisation of model queries: A parallel execution approach," *Journal of Object Technology*, vol. 18, no. 2, pp. 3:1–21, 2019, the 15th European Conference on Modelling Foundations and Applications.

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