
Proceedings Paper:
Ali, Qurat ul ain, Kolovos, Dimitris and Barmpis, Konstantinos (2020) Efficiently Querying Large-Scale Heterogeneous Models. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. MODELS '20. ACM, New York, NY, USA.

https://doi.org/10.1145/3417990.3420207

Efficiently Querying Large-Scale Heterogeneous Models

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

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

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

ABSTRACT

With the increase in the complexity of software systems, the size and the complexity of underlying models also increases proportionally. In a low-code system, models can be stored in different backend technologies and can be represented in various formats. Tailored high-level query languages are used to query such heterogeneous models, but typically this has a significant impact on performance. Our main aim is to propose optimization strategies that can help to query large models in various formats efficiently. In this paper, we present an approach based on compile-time static analysis and specific query optimizers/translators to improve the performance of complex queries over large-scale heterogeneous models. The proposed approach aims to bring efficiency in terms of query execution time and memory footprint, when compared to the naive query execution for low-code platforms.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**.

KEYWORDS

Model-Driven Engineering, Scalability, Model Querying, Static Analysis

ACM Reference Format:
Qurat ul ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. 2020. Efficiently Querying Large-Scale Heterogeneous Models. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3417990.3420207

1 INTRODUCTION

Low-code platforms use model-driven engineering (MDE) [8] processes such as domain specific languages and code generation to develop applications. MDE is a promising software engineering methodology that considers models as first-class artefacts of the software development process, further raising the level of abstraction beyond programming languages and frameworks. MDE has been shown to provide benefits over traditional software engineering processes, not only in tackling complexity, but also in terms of increased productivity [14, 16]. Though there are several low-code platforms available like OutSystems, Mendix, Google AppMaker [1] and ZAppDev [4], there are still open challenges limiting the broader adoption of low-code platforms in the industry. One of the main challenges is model-driven environments, including low-code platforms, is scalability [9, 24]. Scalability issues can be further categorized as follows [19]:

- Scalable Domain-Specific Languages: Ability to design and construct large models and domain-specific languages.
- Scalable Querying and Transformation: Ability to efficiently query and transform very large models (millions of model elements).
- Scalable Collaborative Modelling: Ability to collaboratively work on the same models by different modelers.
- Scalable Model Persistence: Ability to store large models efficiently with a low memory footprint.

This paper contributes to tackling the challenge of scalable querying of large-scale heterogeneous models for low-code platforms. We propose an architecture that helps to optimize certain classes of queries for models stored in different backend technologies. In this paper, the implementation of the proposed architecture is discussed on top of the Epsilon framework [17]. We assume that the reader is already familiar with 3-level meta-modeling architectures [3].

The remainder of this paper is organized as follow: Section 2, presents a motivational example and identifies the performance challenges involved. Section 3 presents an architecture for query optimisation over heterogeneous models. Section 4 discusses the existing work in the field of model query optimisation. Section 5 concludes the paper and presents future direction for this work.

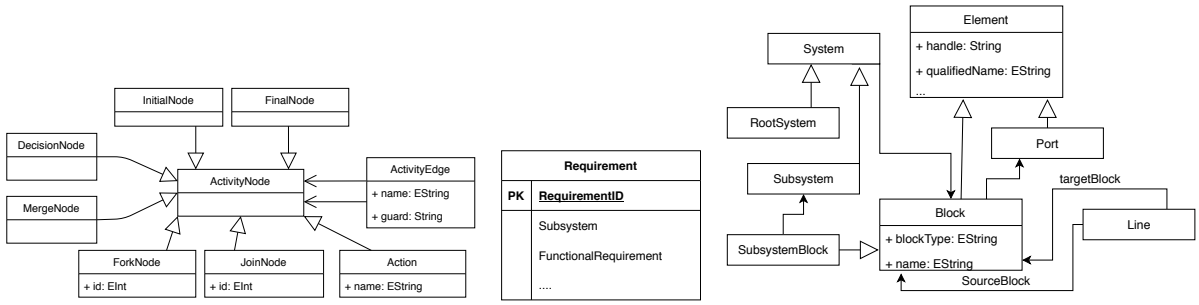
2 MOTIVATION

In MDE, for some domains, there is a need to handle very large models (VLMs) [25], for example, the models of the Automotive Open System Architecture (AUTOSAR) [13], having models containing millions of elements. Other areas with elements in the order of millions include Building Information Modelling and reverse-engineered code from complex systems [25]. While executing complex and computationally expensive queries over such large models, there is a significant performance cost in terms of execution time [21]. Low-code platform is an instance of a generic model-driven platform. In a model-driven 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 1(a) and 1(c). 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 1(b).

Considering these metamodels and table in the figure, constraints can be written in the Epsilon Validation Language (EVL) [18] as shown in Listing 1. EVL is the validation language of the Epsilon platform, built on-top of the OCL-based Epsilon Object Language

MODELS '20 Companion, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-8135-2/20/10…$15.00
https://doi.org/10.1145/3417990.3420207



(a) UML Activity Diagram Metamodel (b) Requirement Table (c) Excerpt of Simulink Metamodel

Figure 1: Metamodels and Excerpt of Requirements table used in Listing 1

(EOL), which is used to evaluate constraints on the models. In Listing 1, 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 constraints also check that requirements refer to valid subsystem names (Line 16-21) and 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 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.

```
1  context UML!Activity {
2   constraint SubsystemCounterpart {
3    check: Simulink!`simulink/Ports & Subsystems/Subsystem`
4    .allInstances.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
13    .exists(r|r.subsystem = self.name)
14   }
15 }
16 context Requirements!Requirement {
17  constraint ValidSubsystem {
18   check : Simulink!`simulink/Ports & Subsystems/Subsystem`
19   .allInstances.exists(s|s.name = self.subsystem)
20  }
21 }
```

Listing 1: EVL constraint before optimisation

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 *findBlocks* method as shown in EVL Listing 2. This constraint is semantically equivalent to the one shown in Listing 1 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 2, to improve performance.

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

Listing 2: EVL constraint after optimisation

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 1) but still benefit from technology-specific optimisations.

3 PROPOSED APPROACH

In this section, we present a framework for query optimisation over heterogenous models in a low-code platform. 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 2. 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.

3.1 Compile-time Static Analysis

We will use Epsilon for the implementation of our approach, as Epsilon supports different modelling technologies in a modular way through its Epsilon Model Connectivity (EMC) abstraction layer. Epsilon is divided into two main parts a) Task-Specific i.e. Epsilon family of languages b) Technology specific drivers i.e. EMC. Epsilon is a model agnostic technology. It seamlessly accesses and manages several underlying model persistence technologies (EMF, MySQL, Spreadsheets etc.). Epsilon supports these technologies through several EMC drivers which is extensible, any new models persisted in other technology can be implemented and added to the EMC layer as a driver. We implemented static analysis in Epsilon to pre-populate a field known as *ResolvedType* for every expression in the program at compile-time. For instance, consider the following:

```
1 Simulink!Block.all().collect(f|f.name="Commonly Used Blocks")
```

When an EOL program containing this expression is compiled, the static analyser will populate the value of resolved type. In this case, the *ResolvedType* of *Simulink!Block.allInstances()* will be *Collection<Block>*. Then static analyser set the *ResolvedType* of expression *Simulink!Block.allI().collect(f|f.name="Commonly Used Blocks"))* to *Collection<String>* .The compile-time static analysis will yield type resolved AST (Abstract Syntax Tree), which will be used by the query optimizer. For the whole process of query optimization, let us consider the running example, as shown in Listing 3. 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 model declaration system is shown in Listing 3. 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.

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

Listing 3: Syntax of Model Declaration Statement

Static analysis of the program in Listing 3 would yield the type resolved abstract syntax graph, as shown in Figure 4. It should be noted that all the fields of resolved type are now populated using static analysis.

3.2 Query Translation

The query optimisation block will have specific optimizers for each back-end technology such as MySQL or Simulink. 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.

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

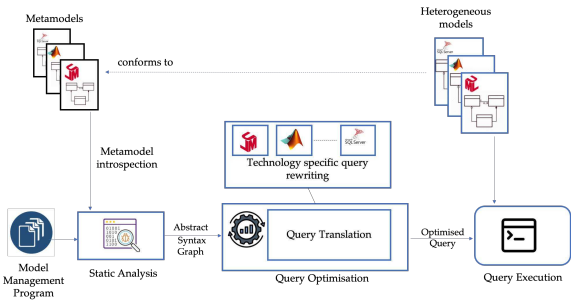


Figure 2: Proposed query optimisation architecture

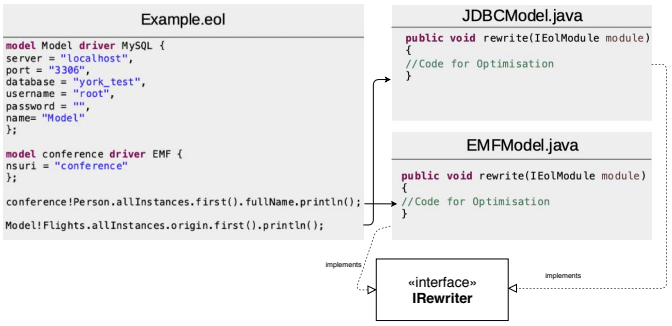


Figure 3: IRewriter interface architecture

example of this approach is shown in Figure 3. 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 3. 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 atleast one instance in the collection that satisfies the given condition. In translation process *Requirements!Requirement.allInstances.exists(f|f.subsystem=self.name)* would be translated to *Requirement.runSql(select * from Requirement where subsystem = '+ self.name + '").size() > 0*

4 RELATED WORK

We can classify related model querying approaches into two main categories. (i) Native Querying (ii) Backend Independent Querying. Native querying is efficient as it is tailored for the specific backend persistence technology: the native language of the model backend is used. 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 [6]: i) Query conciseness: Native queries can sometimes be wordy, difficult to understand. ii) Query abstraction level: Native queries are technology-specific if backend technology is changed, often requiring considerable effort to change queries.

Another common way is the use of high-level languages that abstract over model representations and persistence formats. ATL (Atlas Transformation Language) [15], OCL (Object Constraint Language) [28], 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.

In [20], 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 [11, 12] authors have proposed ways to generate SQL from OCL expressions. In the Hawk model index [7], an approach has been introduced based on derived features. Authors suggest to precompute such features and cache them in the model index itself. 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 [27] 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: 1) Check if the program makes multiple calls to *allInstances()* 2) If yes, then precompute all *allInstances()* collections. Cache all the precomputed collections in one pass.

In [21], 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 [10], 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 backend technology for large models. To address scalability challenges in MDE, one solution is through the use of distributed systems. Pagan et al. [23] propose an efficient query language: MorsaQL (Morsa Query Language) for the MORSA repository [22] – 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.

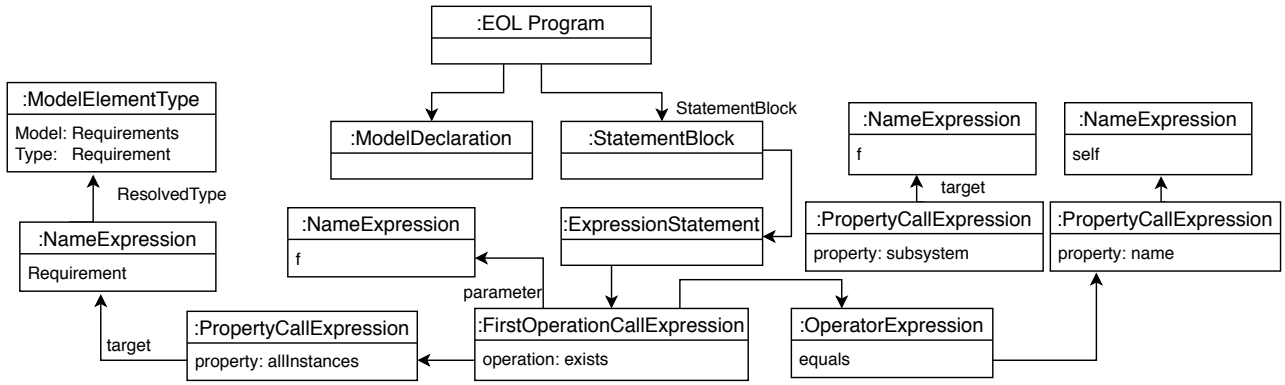


Figure 4: The Abstract Syntax Graph of Program in Listing 3

5 CONCLUSIONS AND FUTURE WORK

In this paper, we have discussed how in a low-code platform, models stored in various backend 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 backend technology's specific optimizations.

In the future, we plan to extend this static analysis and query rewriting for various other formats to evaluate the performance over large models extracted by reverse engineering existing Java code, i.e., Grabats [2] models. We also plan to evaluate the proposed approach on synthetic and publicly available (e.g., on GitHub) Simulink and Building Information Models (BIMs)[5].

ACKNOWLEDGMENTS

This paper disseminates results from the Lowcomote Training Network, that received funding from the European Union's Horizon 2020 Research and Innovation Programme under the Marie Skłodowska-Curie grant agreement n° 813884.

REFERENCES

[1] [n.d.]. *Google App Maker.* https://developers.google.com/appmaker
[2] [n.d.]. *Grabats2009: 5th Int. Workshop on Graph-Based Tools (2012).* http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/
[3] [n.d.]. *Object Management Group. Meta Object Facility.* http://www.omg.org/mof/
[4] [n.d.]. *zAppDev - Cloud Application Development Suite.* http://www.zappdev.com
[5] Giovanna Acampa, Fabiana Forte, and Pierfrancesco De Paola. 2020. BIM models and evaluations. In *Values and Functions for Future Cities*. Springer, 351–363.
[6] Konstantinos Barmpis and Dimitrios S Kolovos. 2014. Evaluation of contemporary graph databases for efficient persistence of large-scale models. *Journal of Object Technology* 13, 3 (2014), 3–1.
[7] Konstantinos Barmpis and Dimitrios S Kolovos. 2014. Towards scalable querying of large-scale models. In *European Conference on Modelling Foundations and Applications*. Springer, 35–50.
[8] Jean Bézivin. 2005. Model driven engineering: An emerging technical space. In *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, 36–64.
[9] Antonio Bucchiarone, Jordi Cabot, Richard F Paige, and Alfonso Pierantonio. 2020. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling* 19, 1 (2020), 5–13.
[10] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. 2018. Scalable queries and model transformations with the mogwai tool. In *International Conference on Theory and Practice of Model Transformations*. Springer, 175–183.
[11] Marina Egea and Carolina Dania. 2019. SQL-PL4OCL: an automatic code generator from OCL to SQL procedural language. *Software & Systems Modeling* 18, 1 (2019), 769–791.
[12] Marina Egea, Carolina Dania, and Manuel Clavel. 2010. MySQL4OCL: A stored procedure-based MySQL code generator for OCL. *Electronic Communications of the EASST* 36 (2010).
[13] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. 2009. AUTOSAR–A Worldwide Standard is on the Road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, Vol. 62. 5.
[14] Ari Jaaksi. 2002. Developing mobile browsers in a product line. *IEEE software* 19, 4 (2002), 73–80.
[15] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. 2006. ATL: a QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 719–720.
[16] Juha Kärnä, Juha-Pekka Tolvanen, and Steven Kelly. 2009. Evaluating the use of domain-specific modeling in practice. In *Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling*.
[17] Dimitris Kolovos, Louis Rose, Antonio Garcia-Dominguez, and Richard Paige. 2010. *The Epsilon Book. Eclipse.*
[18] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2009. On the evolution of OCL for capturing structural constraints in modelling languages. In *Rigorous Methods for Software Construction and Analysis*. Springer, 204–218.
[19] Dimitrios S Kolovos, Louis M Rose, Nicholas Matragkas, Richard F Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan De Lara, István Ráth, Dániel Varró, Massimo Tisi, et al. 2013. A research roadmap towards achieving scalability in model driven engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*. 1–10.
[20] Dimitrios S Kolovos, Ran Wei, and Konstantinos Barmpis. 2013. An approach for efficient querying of large relational datasets with ocl-based languages. In *XM 2013-Extreme Modeling Workshop*, Vol. 48.
[21] Sina Madani, Dimitris Kolovos, and Richard F Paige. 2019. Towards optimisation of model queries: a parallel execution approach. *Journal of Object Technology* 18, 2 (2019).
[22] Javier Espinazo Pagán, Jesúss Sánchez Cuadrado, and Jesús García Molina. 2011. Morsa: A scalable approach for persisting and accessing large models. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 77–92.
[23] Javier Espinazo Pagán and Jesús García Molina. 2014. Querying large models efficiently. *Information and Software Technology* 56, 6 (2014), 586–622.
[24] JR Rymer, C Richardson, C Mines, D Jones, D Whittaker, J Miller, and I McPherson. [n.d.]. Low-Code Platforms Deliver Customer-Facing Apps Fast, But Will They Scale Up. *Forrester Research* ([n. d.]).
[25] Massimo Tisi, Salvador Martínez, and Hassene Choura. 2013. Parallel execution of ATL transformation rules. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 656–672.
[26] Massimo Tisi, Jean-Marie Mottu, Dimitrios S. Kolovos, Juan De Lara, Esther M Guerra, Davide Di Ruscio, Alfonso Pierantonio, and Manuel Wimmer. 2019. Lowcomote: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms. In *STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations (STAF 2019)*.
[27] Ran Wei and Dimitrios S Kolovos. 2015. An Efficient Computation Strategy for allInstances ().. In *BigMDE@ STAF*. 32–41.
[28] Edward Daniel Willink. 2011. Aligning ocl with uml. *Electronic Communications of the EASST* 44 (2011).