UNIVERSITY *of York*

This is a repository copy of *Towards Scalable Validation of Low-Code System Models: Mapping EVL to VIATRA Patterns*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/179960/

Version: Accepted Version

White Rose
university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Towards Scalable Validation of Low-Code System Models: Mapping EVL to VIATRA Patterns

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

Benedek Horváth
IncQuery Labs cPlc.
Budapest, Hungary
Johannes Kepler University Linz
Linz, Austria
benedek.horvath@incquerylabs.com

Dimitris Kolovos,
Konstantinos Barmpis
Department of Computer Science
University of York
York, UK
firstname.lastname@york.ac.uk

Ákos Horváth
IncQuery Labs cPlc.
Budapest, Hungary
akos.horvath@incquerylabs.com

Abstract—**Adoption of low-code engineering in complex enterprise applications also increases the size of the underlying models. In such cases, the increasing complexity of the applications and the growing size of the underlying artefacts, various scalability challenges might arise for low-code platforms. Task-specific programming languages, such as OCL and EOL, are tailored to manage the underlying models. Existing model management languages have significant performance impact when it comes to complex queries operating over large-scale models reaching magnitudes of millions of elements in size. We propose an approach for automatically mapping expressions in Epsilon validation programs to VIATRA graph patterns to make the validation of large-scale low-code system models scalable by leveraging the incremental execution engine of VIATRA. Finally, we evaluate the performance of the proposed approach on large Java models of the Eclipse source code. Our results show performance speed-up up to 1481x compared to the sequential execution in Epsilon.**

Index Terms—**static analysis, model querying, scalability, graph patterns**

I. INTRODUCTION

Model-driven engineering (MDE) is a software engineering methodology that considers models as first-class citizens of the software development process. MDE typically includes several tasks such as model validation, transformation, code generation. These tasks usually include a common set of queries over certain model elements. While executing these queries over large-scale models (in order of million of elements), there is a significant cost of computation, taxing available resources.

Low-code platforms commonly use MDE concepts to produce software, thus reducing or eliminating the need for hand-written code. There are a number of low-code platforms already available such as ZAppDev [1], Mendix [2], OutSystems [3]. In many low-code platforms users can develop applications from which platform-specific source code, e.g., Java, can be generated. As such, low-code platforms need to manage a larger number and size of software models, often on cloud resources, with scalability and efficiency becoming increasingly important. Although cloud resources can scale elastically on-demand, but they may imply large financial costs. Therefore, we propose an approach that can be used not only in cloud environments.

In this paper, we propose a method to improve the performance (primarily by reducing the execution time) of model validations by mapping OCL-like expressions embedded in Epsilon validation constraints to graph patterns. We have implemented a prototype demonstrating our solution by mapping Epsilon Validation Language (EVL) to VIATRA graph patterns. The implementation of the aforementioned tool is open-source and available on GitHub [4]. Moreover, we measured the performance of the solution on validation rules from the Findbugs validation suite [5].

The rest of the paper is structured as follows: Section II introduces Epsilon and VIATRA frameworks followed by a motivating example. Section III presents the overall EVL to VIATRA pattern mapping approach. Section IV presents initial performance results. Section V reviews the state-of-the-art related to our research. Section VI concludes the paper and suggests possible future extensions.

II. BACKGROUND

In this section, we introduce the Epsilon and VIATRA frameworks, followed by the motivating example of the paper.

A. Epsilon

Epsilon [6] is a family of task-specific languages for performing a number of model management tasks, such as validation (Epsilon Validation Language - EVL), model-to-model transformation (Epsilon Transformation Language - ETL) and pattern matching (Epsilon Pattern Language - EPL). All these languages are built on the top of a base language, the Epsilon Object Language (EOL), providing imperative constructs such as loops, conditionals and operations (both built-in and user-defined). EOL is inspired by OCL, a widely used language that has a similar syntax. All languages of Epsilon support

managing models from a number of modeling technologies (and their respective persistence formats), through a uniform interface, the Epsilon Model Connectivity (EMC) layer. The architecture of Epsilon is illustrated in Figure 1.
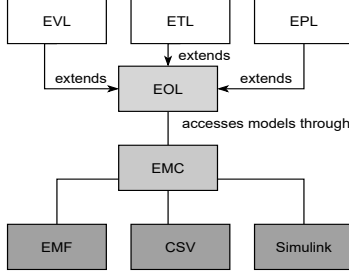

Fig. 1. Epsilon architecture

B. VIATRA

VIATRA is an open-source model query, validation and transformation framework supporting the efficient evaluation of model queries on top of EMF models [7], [8]. The core language of VIATRA is the Viatra Query Language (VQL), which allows the definition of model queries as incremental graph patterns. A graph pattern is a graph-like structure consisting of conditions (nodes and edges) to be matched against a large instance model.

VIATRA provides two engines to evaluate the graph patterns on EMF models. The first one is the incremental engine implementing the RETE algorithm [9]. This engine computes the pattern matches and caches them, therefore enabling incremental re-evaluation of patterns. The second one is the local search engine that employs efficient search plans to compute and collect pattern matches [10]. Both engines benefit from the base index that caches the base relations and objects in the model by type [11].
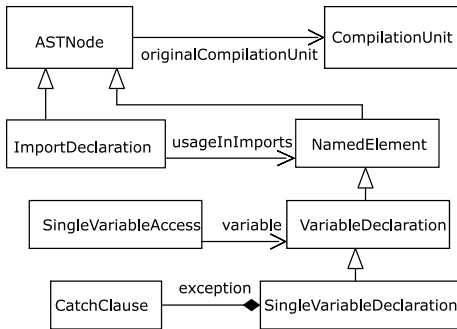
C. Motivating Example

Fig. 2. An excerpt of Java metamodel

In this paper, we use the Java metamodel as a motivating example, because it is sufficiently complex while also being well-known and understandable by a MDE and low-code audience. An excerpt of the metamodel is depicted in Figure 2. We can specify several model validation constraints in EVL for Java models conforming to the metamodel.

```
1 model Java driver ViatraEMF {
2   nsuri = "http://www.eclipse.org/MoDisco/
3   Java/0.2.incubation/java"
4 };
5
6 context Java!ImportDeclaration {
7   constraint allImportsAreUsed {
8     check: Java!NamedElement.all.exists(
9     ne|ne.originalCompilationUnit =
10       self.originalCompilationUnit and
11         ne.usagesInImports = self)
12   }
13 }
14
15 context Java!VariableDeclaration {
16   constraint variableIsUsed {
17     check: Java!SingleVariableAccess.all
18       .exists(sva|sva.variable = self)
19   }
20 }
21
22 context Java!CatchClause {
23   constraint exceptionIsUsed {
24     check: Java!SingleVariableAccess.all
25       .exists(sva|sva.variable =
26         self.exception)
27   }
28 }
```

Listing 1. Example EVL script before optimisation

Listing 1 shows three constraints called *allImportsAreUsed*, *variableIsUsed* and *exceptionIsUsed*, respectively. The *allImportsAreUsed* constraint checks that every *ImportDeclaration* in the model is used by at least one *NamedElement*. The *variableIsUsed* constraint checks that every *VariableDeclaration* represented in the model is accessed at least once. Finally, *exceptionIsUsed* similarly checks that every exception variable in *CatchClause* is accessed at least once.

Now, if we consider evaluating these constraints over a large Java model containing elements in the order of hundreds of thousands, then naively executing these constraints would be computationally expensive. If we assume the number of *ImportsDeclaration* and *NamedElement* to be M and N respectively, then the complexity of the *allImportsAreUsed* constraint would be $O(M*N)$. Similarly, the complexity of evaluating *variableIsUsed* over M number of *VariableDeclaration* and N number of *SingleVariableAccess* would be $O(M*N)$.

In this paper, we propose a translation of these computationally expensive expressions to VIATRA patterns as depicted in Listing 2. Although this optimization does not reduce the computational complexity of the problem, but in many practical cases it provides a shorter evaluation time due to the incremental query engine (see Section IV).

```
1 import "http://www.eclipse.org/MoDisco/
2 Java/0.2.incubation/java"
3
```
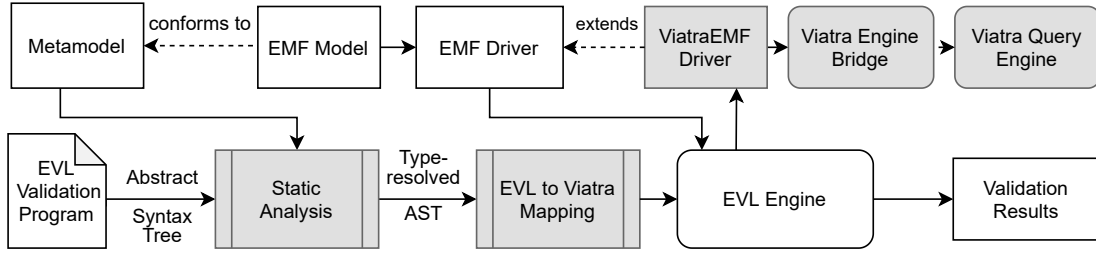
Fig. 3. EVL to VIATRA mapping architecture

```
4 pattern Java2(sva: SingleVariableAccess,
5 self0: VariableDeclaration){
6   SingleVariableAccess.variable(
7   sva, self0);
8 }
```

Listing 2. variableIsUsed's check block translated to a VQL pattern

III. PROPOSED APPROACH

In this section, we will discuss an approach for the efficient execution of model validation programs using static analysis and automatic program rewriting. The main goal of the proposed approach is to improve performance by reducing execution time. This approach is illustrated in Figure 3.

A. Static Analysis

In the first step, the EVL validation program is parsed to extract its Abstract Syntax Tree (AST). Then the static analyser produces a type-resolved AST in which the resolved types are populated using metamodel introspection and type inference. As EVL extends EOL, the core language of Epsilon, EVL's static analyser also extends EOL's static analyser. It essentially includes additional support for analysing expressions inside constraints, pre and post blocks. The pre and post blocks of an EVL program consist of EOL expressions that are executed before and after evaluating the constraints, respectively.

B. EVL to VIATRA Mapping

In the second step, we traverse the type-resolved AST sequentially and detect the expressions that can be optimized. These expressions are in the form of first-order operations operating over *allInstances* of a model element. In a check block of a constraint, each expression is evaluated over all elements of the context, thus first-order operations can be computationally expensive for large models in this case, see complexity measures at the end of Section II-C. Therefore, we translate these operations to VIATRA patterns as follows:

First, the NsUri of an EPackage is extracted from the model declaration statement (lines 2-3 in Listing 1) and mapped to import statement in VQL (lines 1-2 in Listing 2). After that, the body of the first-operations are translated to graph patterns. Names of the patterns should be unique, due to VQL naming conventions. Therefore, they are generated based on

the model name concatenated with a sequence number, see line 4 in Listing 2.

In the operation body, we translate operator expressions consisting of property call expressions to graph patterns in VIATRA. Property call expressions define navigations in EOL, e.g., *sva.variable* in line 18 of Listing 1 reads the *variable* field of the *sva* object. In operator expressions, the name of the first operand is the name of the property on which the navigation should happen. The operator defines the comparison basis, and the value of the second operand is the value to be compared against. It can be either a literal value or a reference to the single-valued result of another EOL expression. In the latter case, the value is received as an additional pattern parameter. Model navigations are represented by declarative graph patterns in VQL, where the starting node is the type of the model element, and the edge is the property used in the property call expression. If the operator is an inequality operator, then an additional check constraint is generated in VQL whose body contains the inequality comparison. As an example, let's consider the EOL expression *sva.variable = self* in line 18 of Listing 1, that is translated to *SingleVariableAccess.variable(sva, self0)* in lines 6-7 of Listing 2 with *self0* being an additional pattern parameter in VQL (line 5 of Listing 2). In our prototype implementation [4], only conjunctions of operator expressions are supported.

As shown in Table I, most first-order operations are translated to only one VQL pattern. However, in some cases, such as in the *reject* and *forAll* operations, we need to generate an additional pattern. In those cases, we look for the matches of the negated expression, therefore the main pattern contains a negative invocation of the second pattern (neg find). Besides, the type of the first-order operation defines the method to be called on the Matcher API of VIATRA with some additional parameters used for comparing the result, e.g., countMatches == 0.

Finally, the body of the first-order operation is replaced by an operation call expression, encapsulating a Run Viatra Call Parameters object that contains: (i) the generated VQL patterns, (i) the name of the main VQL pattern that is used to collect the pattern matches, (iii) the name and the (iv) parameter of the method to be called on VIATRA's Matcher API, and (v) the EOL expressions representing the extra parameters of the patterns. The values of these expressions will be bound

EOL expression	VQL Pattern	Matcher API call

Java!NamedElement.all.select(u|u.name="main")
Java!NamedElement.all.selectOne(u|u.name="main")
Java!NamedElement.all.exists(u|u.name="main")
Java!NamedElement.all.one(u|u.name="main")
Java!NamedElement.all.none(u|u.name="main")
Java!NamedElement.all.count(u|u.name="main")
Java!NamedElement.all.nMatch(u|u.name="main",2)
Java!NamedElement.all.atLeastNMatch(u|u.name="main",1)
Java!NamedElement.all.atMostNMatch(u|u.name="main",1)
	pattern Java1(namedElement: NamedElement) {
	    NamedElement.name(namedElement, "main");
	}
		allMatches
		oneArbitraryMatch
		hasMatch
		countMatches == 1
		countMatches == 0
		countMatches
		countMatches == 2
		countMatches >= 1
		countMatches <= 1

Java!NamedElement.all.reject(u|u.name="main")
Java!NamedElement.all.forAll(u|u.name="main")
	pattern Java1(namedElement: NamedElement) {
	    neg find Java1internal(namedElement);
	}
	pattern Java1internal(namedElement: NamedElement) {
	    NamedElement.name(namedElement, "main");
	}
		allMatches
		hasMatch == false

at runtime to the corresponding parameters of the patterns.

C. Collecting Validation Results

After the translation of optimizable EOL expressions to VQL patterns, the EVL engine iterates through the EVL program and evaluates the expressions. If it finds a translated EOL expression, then the *runViatra* method of the ViatraEMF driver is called, which calls the VIATRA Engine Bridge that prepares query specifications from the textual VQL patterns, obtains a matcher for the main specification and invokes the corresponding method with the appropriate parameters on the matcher. Finally, the found matches are returned to the EVL engine, which combines them with the matches from the unoptimized expressions and returns the validation results.

IV. EVALUATION

In order to measure the query execution time and memory use of the proposed approach we adopted three validation constraints (Listing 1) from the Findbugs validation suite [5] and evaluated them on the Java MoDisco EMF model of the Eclipse source code [5]. We compared the incremental (RETE) and local search (LS) engines of VIATRA with the sequential EVL engine. In the query evaluation phase, the models are already loaded in memory, and the EOL expressions are already translated to VQL. The query rewriting took 9 ms for all queries. The measurements were conducted on a machine with Windows 10, Intel i7-9750H CPU @ 2.60GHz, 32 GB RAM, Java HotSpot[TM] 64-Bit Server VM 13.0.1+9 (with 16 GB max heap size).

A. Analysis of the Results

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

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

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

1481x between the RETE and the sequential EVL engine, in the case of models with 500k elements.

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

B. Threats to Validity

Internal validity: The results reported in this paper are computed on programs containing first-order operation calls on all instances of model elements. If there are no such optimisable operations and queries in the program, then the execution time is the same as in sequential EVL.

External validity: Opposed to the local search engine of VIATRA, the sequential Epsilon engine does not consider opposite edges in the metamodel when creating the search

plan. Therefore, to have comparable results we used the Java metamodel without these edges. Otherwise the local search engine would have performed similar to the RETE engine in both cases, due to the simplicity of the patterns.

Technical limitations: Although Viatra has a non-EMF-based adoption in MPS [12], and the wide ranges of EMC drivers enable Epsilon to be used with different modeling sources, but there is no EMC driver for MPS yet. Therefore, our solution is limited to the EMF technical space.

V. RELATED WORK

This section discusses relevant literature in the field of model querying. In particular, we will discuss the use of query translation approaches for optimization purposes.

In our previous work, we used static analysis for enabling the translation from EOL to SQL [13]. A solution for efficient querying large-scale databases is presented in [14], where OCL queries are translated to SQL at runtime. Heidenreich et al. proposed an approach for translating from OCL to multiple query languages like SQL and XQuery using model-to-text transformations [15]. These solutions work on relational database backends.

Mogwai [16] is a tool to efficiently query large-scale models persisted in NoSQL backends. It translates the scripts written in OCL and ATL to Gremlin, a native query language for NoSQL databases. Sanchez et al. proposed an approach for translating OCL queries to MATLAB commands for efficiently querying large Simulink models [17]. Bergmann et al. presented a mapping strategy from OCL to graph-based patterns [18]. In their approach, they map a subset of OCL expressions to EMF-IncQuery graph patterns.

The novelty of the approach proposed in this paper is adding partial incrementality by just translating a part of EVL program detected through static analysis. Only the expensive expressions are translated to corresponding VIATRA patterns. This is achieved as a trade-off between execution time and memory consumption.

VI. CONCLUSION AND FUTURE WORK

We have proposed an architecture for automatically mapping certain expensive expressions from an EVL validation program to VQL patterns. Mapping takes the benefit of static type information extracted by the static analyzer from the EMF models. The translated VQL patterns executed by the RETE engine outperform the sequential execution of EVL validation. We have argued that this sort of partial translation can help model validation scale well for large-scale low-code system models, without explicitly relying on the elastically scalable computational resources in the cloud. Therefore, the approach can be used in other deployment scenarios as well.

This work can be extended in further iterations to cover the mapping of more complex expressions, e.g., navigating multi-valued references. Besides rewriting queries to a different language, another way to improve the performance is to use the parallel EVL engine [5] or to cache all instances of every type in the model. Comparing the performance of these approaches with the RETE engine is also an interesting future direction.

REFERENCES

[1] "Zappdev," http://zappdev.com/, last accessed on 02-Sep-2021.

[2] "Mendix," https://mendix.com, last accessed on 02-Sep-2021.

[3] "OutSystems," https://outsystems.com, last accessed on 02-Sep-2021.

[4] "Epsilon Validation Language to Viatra Patterns," https://github.com/lowcomote/evl-viatra-prototype.git.

[5] S. Madani, D. S. Kolovos, and R. F. Paige, "Parallel model validation with Epsilon," in *Proceedings of the 14th European Conference on Modelling Foundations and Applications, ECMFA@STAF 2018*, ser. LNCS, vol. 10890. Springer, 2018, pp. 115–131.

[6] "Epsilon," https://www.eclipse.org/epsilon/, last accessed on 02-Sep-2021.

[7] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2008.

[8] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi, "Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework," *Software and Systems Modeling*, vol. 15, no. 3, pp. 609–629, 2016.

[9] C. Forgy, "Rete: A fast algorithm for the many patterns/many objects match problem," *Artif. Intell.*, vol. 19, no. 1, pp. 17–37, 1982.

[10] M. Búr, Z. Ujhelyi, Á. Horváth, and D. Varró, "Local search-based pattern matching features in EMF-IncQuery," in *Proceedings of the 8th International Conference on Graph Transformations.*, ser. LNCS, vol. 9151. Springer, 2015, pp. 275–282.

[11] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró, "EMF-IncQuery: An integrated development environment for live model queries," *Science of Computer Programming*, vol. 98, pp. 80–99, 2015.

[12] T. Szabó, S. Erdweg, and M. Voelter, "Inca: a DSL for the definition of incremental program analyses," in *Proceedings of the 31st International Conference on Automated Software Engineering, ASE 2016*. ACM, 2016, pp. 320–331.

[13] 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.

[14] D. S. Kolovos, R. Wei, and K. Barmpis, "An approach for efficient querying of large relational datasets with OCL based languages," in *In Proceedings of the Workshop on Extreme Modeling co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems, MODELS 2013*, ser. CEUR Workshop Proceedings, vol. 1089. CEUR-WS.org, 2013, pp. 46–54.

[15] F. Heidenreich, C. Wende, and B. Demuth, "A framework for generating query language code from OCL invariants," *Electronic Communications of the EASST*, vol. 9, 2007.

[16] G. Daniel, G. Sunyé, and J. Cabot, "Scalable queries and model transformations with the mogwaï tool," in *Proceedings of the 11th International Conference on Theory and Practice of Model Transformation, ICMT@STAF 2018*, ser. LNCS, vol. 10888. Springer, 2018, pp. 175–183.

[17] B. Sanchez, A. Zolotas, H. Hoyos Rodriguez, D. Kolovos, and R. Paige, "On-the-fly translation and execution of OCL-like queries on simulink models," in *Proceedings of the 22nd International Conference on Model Driven Engineering Languages and Systems, MODELS*, 2019, pp. 205–215.

[18] G. Bergmann, "Translating OCL to graph patterns," in *Proceedings of the 17th International Conference on Model-Driven Engineering Languages and Systems, MODELS 2014*, ser. LNCS, vol. 8767. Springer, 2014, pp. 670–686.