Efficient validation of large models using the Mogwaï tool

Gwendal Daniel[1]

Internet Interdisciplinary Institute (IN3)
Universitat Oberta de Catalunya (UOC)
gdaniel@uoc.edu

Abstract. Scalable model persistence frameworks have been proposed to handle large (potentially generated) models involved in current industrial processes. They usually rely on databases to store and access the underlying models, and provide a lazy-loading strategy that aims to reduce the memory footprint of model navigation and manipulation. Dedicated query and transformation solutions have been proposed to further improve performances by generating native database queries leveraging the backend's advanced capabilities. However, existing solutions are not designed to specifically target the validation of a set of constraints over large models. They usually rely on low-level modeling APIs to retrieve model elements to validate, limiting the benefits of computing native database queries. In this paper we present an extension of the Mogwaï query engine that aims to handle large model validation efficiently. We show how model constraints are pre-processed and translated into database queries, and how the validation of the model can benefit from the underlying database optimizations. Our approach is released as a set of open source Eclipse plugins and is fully available online.

1 Introduction

The limited support for managing large models has been one of the key reported factors hampering the adoption of MDE in industrial processes [15]. Indeed, existing modeling solutions were primary designed to assist simple, human-based modeling activities, and are struggling to deal with large models (potentially generated using model driven reverse engineering techniques [3]) involved in current industrial workflows. As an example, several studies have reported scalability issues of the Eclipse Modeling Framework [9, 12] (the *de-facto* standard modeling solution in the Eclipse ecosystem) and its default serialization mechanism, XMI.

These limitations have led to the creation of several scalable model persistence frameworks relying on databases to store and access large models [5, 8, 12]. They usually provide advanced mechanisms such as application-level caches and *lazy-loading* approaches, allowing to balance the memory consumption and the application performances. This new generation of modeling frameworks has been enhanced with database-specific query and transformation mechanisms [1, 6], bypassing the modeling API limitations, and leveraging the database capabilities to further improve the performances of existing modeling workflows.

While these techniques have clearly enhanced the performance of model queries and transformation over large models, there is only a few works that specifically address the validation of large models. Indeed, validating a model usually requires to first find the elements to validate, and evaluate, for each one of them, the set of associated constraints. While the specific constraint evaluation can be addressed with existing querying solution, the element retrieval step is usually performed at the modeling framework API level, that have shown clear limitations when combined with current model persistence frameworks [5].

In this article, we present our work on extending the Mogwaï query framework [6] to provide advanced support for efficient model validation. In particular, we present an approach that maps context-specific constraints into global queries that can be translated into efficient database operations, improving the execution time and memory consumption of model validation processes.

The rest of the paper is organized as follows: Section 2 gives an overview of the Mogwaï framework and how we have extended it to improve the support of model validation. Section 3 introduces the tool support, and Section 4 presents our first results on using our approach to evaluate constraints on a reverse engineering case study. Finally, Section 5 summarizes the key points of the paper and presents our future work.

2 Framework Overview

2.1 Mogwaï

Existing model query frameworks are typically based on the low-level APIs provided by the modeling framework to access and evaluate queries over a model [7]. Queries are translated into a sequence of API calls, which are then performed one after another on the persistence layer. While this architecture allows to easily integrate query solutions into existing tool chains, this low-level design is clearly inefficient when combined with persistence framework because (i) the API granularity is too fine to benefit from the advanced query capabilities of the backend, and (ii) an important time and memory overhead is necessary to construct navigable intermediate objects needed to interact with the API.

The Mogwaï framework tackles these limitations by providing an alternative query solution that translates modeling queries (expressed in OCL [11]) into database queries. The framework relies on a model-to-model transformation that generates queries expressed using the Gremlin traversal language [13], a high-level NoSQL query language that targets multiple databases.

Figure 1 presents an overview of the framework: a designer defines a set of OCL queries over the model[1], that constitute the input of the *Mogwaï Translation Engine*. This translation engine embeds a model-to-model transformation defining a systematic mapping of the input OCL expressions to their equivalent Gremlin constructs. The resulting query fragments are then assembled into a *Gremlin Script*, that is sent to the database for computation. The resulting database elements are finally returned to the

[1] The framework also supports model transformations that are not discussed in this article.

query framework, that takes care of their reification into standard modeling elements that are returned to the modeler.

We showed in our previous work [6] that this translation of OCL expressions into database queries can dramatically improve the performance of existing applications. Specifically, using a translation-based approach provides the following benefits: (i) it reduces the query fragmentation compared to standard solutions based on low-level model handling APIs, (ii) the generated query can fully benefit from the backend capabilities (such as embedded caches and indexes), and (iii) the approach does not reify intermediate objects, reducing the memory consumption.

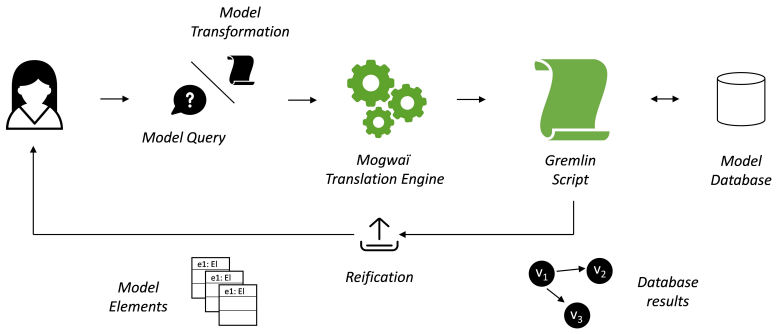

Fig. 1. Mogwaï Approach

2.2 Validating Models with Mogwaï

Existing modeling frameworks usually implement model validation as a two-step process: the framework first retrieves all the elements to validate (i.e. that are concerned by at least one constraint) and checks, for each elements, whether its associated constraints are satisfied.

While the evaluation of the constraints themselves can be performed by efficient query frameworks like Mogwaï, the initial model element retrieval usually relies on the modeling framework API, that is not aligned with current model persistence frameworks [5] storing large models. This API mismatch generates fragmented queries on the database, increasing the memory consumption and execution time of validation operations, and reducing the benefit of using advanced query frameworks. This element retrieval issue has been specifically reported in the context of EMF-based applications [14].

To tackle this limitation, we propose an extension of our approach that generates a single database query combining both the element retrieval and the constraint computation. To do so, we rely on existing OCL rewriting techniques [2] to pre-process the input constraints and produce optimized OCL queries that can be efficiently translated and computed by the model database.

To illustrate our approach, we introduce in Listing 1.1 an example OCL constraint *validClient*. This constraint is *validated* when the provided *Client*'s *hasPaid* attribute is true and all its associated ordered *Products* have a positive *price*.

Listing 1.1. Sample OCL Query

```
context Client inv validClient:
  self.hasPaid implies self.orders.products
    ->forAll(p | p.price > 0)
```

Our solution relies on an in-place transformation of the input OCL constraint that generates a model query returning all the model elements violating the base constraint. This mapping is systematically applied by using the constraint *Context* as the input of the generated query, followed by an `allInstances()` operation that retrieves all the instances of the *Context* type. This `allInstances()` operation is followed by a `select` iterator that contains the negation of the original constraint body. A similar query rewriting approach has been proposed in [2] to build database views from model constraints. Listing 1.2 shows the example OCL query after this pre-processing step.

Listing 1.2. Pre-Processed OCL Query

```
Client.allInstances()->select(c | !(c.paid implies
  c.orders.products->forAll(p.price > 0)))
```

The generated query is then translated by the Mogwaï framework into a Gremlin traversal, following the operation mapping and expression composition presented in our previous work [6]. In particular, this mapping provides an efficient implementation of the `allInstances` operation by using database indexes and caches, improving the performance of the element retrieval. The resulting traversal is sent to the model database for computation, bypassing the standard modeling API, and benefiting from the database query optimizations and built-in capabilities.

3 Tool Support

The Mogwaï tool is implemented as a set of open-source Eclipse plugins released under the EPL license. Source code and benchmark materials are fully available in the project's GitHub repository[2]. An Eclipse update site containing the latest stable version of the framework is also available online.

The framework relies on Eclipse MDT OCL [7] to parse the input queries, and the produced OCL models constitute the input of a set of 70 ATL [10] transformation rules and helpers implementing the mapping and the transformation process presented in detail in our previous work [6]. We enhance this mapping with an additional in-place ATL transformation refining the input OCL constraint to produce the optimized query introduced in Section 2.2.

We have also extended the Mogwaï API to provide support for model validation. Specifically, we added a *validate* method to the *MogwaiResource* class, complementing the existing *query* and *transform* ones targeting respectively OCL queries and ATL transformations computations. This additional method takes care of pre-processing the input OCL constraints, transform it into a Gremlin traversal to execute on the database, and reifies the database results to display model-level information to the designer. The

[2] `https://github.com/atlanmod/Mogwai`

framework also provides a new *QueryResult* type: *MogwaiValidationResult*, that provides utility methods to check which constraints are violated, what are the model element involved, and retrieve query execution information.

4 Evaluation

In this section we evaluate the performance of the Mogwaï validation framework by comparing the performance of validating OCL constraints in three configurations: the standard MDT OCL toolkit implementation [7] relying on the EMF API, the *standard* Mogwaï query computation coupled with API-based element retrieval, and the Mogwaï *validation* implementation including the OCL pre-processing presented above. The computed queries are evaluated over a model stored in the NeoEMF persistence framework [5] relying on a graph database.

The experiments are executed on a computer running Windows 10 64 bits. Relevant hardware elements are: an Intel Core I7 processor (2.9 Ghz), 16GB of DDR3 SDRAM (1600Mhz), and a SSD hard-disk. Experiments are executed on Eclipse 4.7.2 (Oxygen) running Java SE Runtime Environment 1.8.

4.1 Benchmark Presentation

The experiments are run over a large model representing a Java project that has been automatically constructed by the MoDisco Java Discoverer [3]. The model contains around 80000 elements representing a Java Eclipse plugin at a low level of abstraction.

We selected two constraints to validate over this model. The first one, *NotEmptyClassName*, performs a simple navigation from an input *ClassDeclaration* and checks that its name is not empty. The second one, *ValidJavadocTags*, preforms a multi-level navigation from an input *CompilationUnit* a checks that all the Javadoc tags contained in its comments are valid (i.e. not empty and corresponding to standard Javadoc tags).

We manually introduced a set of errors in the benchmarked model to ensure that all the approaches produce the same results. The resulting model contains 10 elements (out of 166) violating *NotEmptyClassName*, and 30 elements (out of 132) violating *ValidJavadocTags*.

4.2 Results

Table 1 and 2 show the execution time (in milliseconds) and memory consumption (in mega bytes) of the validation of each constraint. Each column stores the results of a specific configuration. Note that the presented results are average values obtained by running 50 time each constraint.

The execution time results for the *MDT OCL* and *EMF allInstances + Mogwaï* configurations are detailed between parenthesis: the first value corresponds to the time required to compute the *allInstances* operation to retrieve all the model elements to validate, and the second one corresponds to the execution time of the constraint over all the retrieved model elements.

Table 1. Model Validation Execution Time (ms)

Query	MDT OCL	Standard allInstances + Mogwaï	Pre-processing + Mogwaï
NotEmptyClassName	6436 (6177 + 259)	8667 (6221 + 2446)	3310
ValidJavadocTags	8287 (6063 + 2224)	8151 (6064 + 2087)	5597

Table 2. Model Validation Memory Consumption (MB))

Query	MDT OCL	Standard allInstances + Mogwaï	Pre-processing + Mogwaï
NotEmptyClassName	84	80	9
ValidJavadocTags	116	92	36

4.3 Discussion

The results from Table 1 and 2 shows that retrieving the elements to validate in the model is costly for *MDT OCL* and *Standard allInstances + Mogwaï*. This can be explained by the granularity of the EMF API, that does not allow to retrieve efficiently all the instances of a given type [14], and thus requires to fully traverse the model to retrieve the elements to validate.

The execution time of the *Standard AllInstances + Mogwaï* shows that the Mogwaï framework can improve the computation of complex queries over large models (around 6% for *ValidJavadocTags*), but is less efficient than the *MDT OCL* to compute simple queries such as single attribute navigations. This can be explained by the nature of the *NotEmptyClassName* query, that performs a single atomic access to an element of the model, a costly operation in graph databases, that are rather designed to efficiently handle complex navigations. In addition, the time required to translate the input expression and start the underlying Gremlin traversal engine adds an execution time overhead that is not compensated by the performance improvements for simple queries. The *Standard AllInstances + Mogwaï* implementation reduces the memory consumption with respect to *MDT OCL* for the query computation part. The analysis of the memory content revealed that most of the remaining elements where loaded by the initial *allInstances* computation, confirming our previous results [6].

Finally, the *Pre-processing + Mogwaï* approach shows positive results both in terms of execution time and memory consumption compared to the other solutions. This is can be explained by (i) the translation of the element retrieval operation, that allows to benefit from the database optimizations (in particular the indexes), and (ii) the execution of a single query to validate the entire model, that limits the number of intermediate result elements and allows the query engine to fully optimize the query.

These results are encouraging and show that rewriting model validation operations to improve the generated database queries has a significant impact on the performances. Still, a deeper study is required to evaluate the benefits of the approach over a more significant set of input constraints.

5 Conclusion

In this paper we presented our work on improving the Mogwaï framework to enhance the performance of model validation operations. We showed how query rewriting techniques can be applied to create query variants that are efficiently translated by the Mogwaï framework. We evaluate this strategy against the standard Eclipse-based solution, and showed positive results in terms of execution time and memory consumption.

As future work, we plan to complement our study by analyzing the performance impact of generated Gremlin scripts from equivalent OCL expressions within the constraints themselves [4] (e.g. by translating *implies* operators to *if-then-else* blocks). This information could be used to further enhance our OCL pre-processing component and create OCL queries that can be translated into more efficient database queries. Another ongoing work is the merging of similar constraint navigation paths into generic database queries that could be reused and combined, reducing the database overhead implied by redundant query computation.

References

1. K. Barmpis and D. Kolovos. Hawk: Towards a Scalable Model Indexing Architecture. In *Proceedings of the 1st BigMDE Workshop*, pages 6–9. ACM, 2013.
2. M. Brambilla and J. Cabot. Constraint Tuning and Management for Web Applications. In *Proceedings of the 6th ICWE Conference*, pages 345–352, 2006.
3. H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot. MoDisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012 – 1032, 2014.
4. J. Cabot and E. Teniente. Transformation Techniques for OCL Constraints. *Science of Computer Programming*, 68(3):179 – 195, 2007.
5. G. Daniel, G. Sunyé, A. Benelallam, M. Tisi, Y. Vernageau, A. Gómez, and J. Cabot. Neoemf: a multi-database model persistence framework for very large models. *Science of Computer Programming*, 149:9–14, 2017.
6. G. Daniel, G. Sunyé, and J. Cabot. Mogwaï: a framework to handle complex queries on large models. In *Proceedings of the 10th RCIS Conference*, pages 225–237. IEEE, 2016.
7. Eclipse Foundation. MDT OCL, 2018. URL: `www.eclipse.org/modeling/mdt/?project=ocl`.
8. Eclipse Foundation. The CDO Model Repository (CDO), 2018. URL: `http://www.eclipse.org/cdo/`.
9. A. Gómez, G. Sunyé, M. Tisi, and J. Cabot. Map-Based Transparent Persistence for Very Large Models. In *Proceedings of the 18th FASE Conference*, pages 19–34. Springer, 2015.
10. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1):31 – 39, 2008.
11. OMG. OCL Specification, 2018. URL: `www.omg.org/spec/OCL`.
12. J. E. Pagán, J. S. Cuadrado, and J. G. Molina. Morsa: A Scalable Approach for Persisting and Accessing Large Models. In *Proceedings of the 14th MoDELS Conference*, pages 77–92. Springer, 2011.
13. Tinkerpop. The Gremlin Language, 2018. URL: `www.gremlin.tinkerpop.com`.
14. R. Wei and D. S. Kolovos. An Efficient Computation Strategy for allInstances(). *Proceedings of the 3rd BigMDE Workshop*, pages 32–41, 2015.
15. J. Whittle, J. Hutchinson, and M. Rouncefield. The State of Practice in Model-Driven Engineering. *IEEE software*, 31(3):79–85, 2014.