Scalable Queries and Model Transformations with the Mogwaï Tool

Gwendal Daniel, Gerson Sunyé, Jordi Cabot

Scalable Queries and Model Transformations with the Mogwaï Tool

Gwendal Daniel[1], Gerson Sunyé[2], and Jordi Cabot[3,1]

[1] Universitat Oberta de Catalunya
gdaniel@uoc.edu
[2] LS2N, Université de Nantes
gerson.sunye@ls2n.fr
[3] ICREA
jordi.cabot@icrea.cat

Abstract. Scalability of modeling frameworks has become a major issue hampering MDE adoption in the industry. Specifically, scalable model persistence, as well as efficient query and transformation engines, are two of the key challenges that need to be addressed to enable the support for very large models in current applications. In this paper we demonstrate Mogwaï, a tool designed to efficiently compute queries and transformations (expressed in OCL and ATL) over models stored in NoSQL databases. Mogwaï relies on a translational approach that maps constructs of the supported input languages to Gremlin, a generic NoSQL query language, and a model to datastore mapping allowing to compute the generated query on top of several datastores. The produced queries are computed on the database side, benefiting of all its optimizations, improving the execution time and reducing the memory footprint compared to standard solutions. The Mogwaï tool is released as a set of open source Eclipse plugins and is fully available online.

Keywords: MDE, Scalability, OCL, ATL, Model Query, Model Transformation

1 Introduction

Existing empirical assessments from industrial companies adopting MDE [14] point to the limited support for managing large models as one of the factors limiting the success of MDE in industrial MDE processes. Indeed, existing modeling solutions were primary designed to handle simple, human-based modeling activities, and existing technical solutions are not designed to handle large models (potentially generated using model driven reverse engineering techniques [2]) commonly used nowadays. In particular, several studies have reported the scalability issues of the Eclipse Modeling framework (the *de-facto* standard framework for building modeling tools in the Eclipse community) and its default serialization mechanism XMI.

These limitations have led to the creation of several scalable model persistence frameworks built on top of different types of databases [1, 4, 7] combined with advanced mechanisms such as application-level caches [13] and *lazy-loading*. While this new generation of model persistence techniques has globally improved the support for

managing large models, they are partial solutions to the scalability problem in current modeling frameworks. In its core, all frameworks are based on the use of low-level model handling APIs that are focused on manipulating individual model elements and do not provide support for generic model query and transformation computation. This approach is clearly inefficient because (i) the API granularity is too fine-grained 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 that are needed by the modeling API.

To overcome this situation, we developed Mogwaï, a scalable query and transformation framework for large models. Mogwaï consists of a translation component that maps model queries and transformations (expressed in OCL [11] and ATL [8]) into expressions of a graph traversal language, Gremlin [12], a multi-database graph traversal query language we use as our output language. Generated queries are then directly computed on the database side, bypassing the standard modeling API. This avoids the above mentioned problems and significantly improves the overall performance.

This paper complements our existing work [3, 5] by introducing its modular architecture, additional tool implementation details, and includes a unified *ModelDatastore* component that allows to access multiple datastores transparently for both query and transformation computations.

The rest of the paper is organized as follows: Section 2 gives an overview of the Mogwaï infrastructure, Section 3 presents the architecture of our tool and its query processing engine, and Section 4 presents the tool implementation. Finally, Section 5 summarizes the key points of the paper.

2 Framework Overview

Figure 1 shows an overview of the Mogwaï framework that creates Gremlin scripts from input model queries and transformations. An initial *Model Query* or *Transformation* is parsed and sent to a *Translation Engine*, that selects the translation to apply and performs a systematic mapping of the input expressions' to Gremlin constructs. These constructs are then assembled into a *Gremlin Script* that is sent to the database for computation.

The Mogwaï *Translation Engine* relies on a *Model Datastore Definition* to produce the output gremlin script. This generic library provides an abstraction layer that decouples the computation from the low-level database access by adding modeling primitives to manipulate natively the data representing the model. As a result, the generated *Gremlin Script* is not tailored to a specific data store, and can be parametrized with a *Model Datastore Implementation*, that wraps the concrete *Model Datastore* to use (i.e. the backend storing the model). This architecture, originally defined for the Gremlin-ATL engine [3], has been integrated in the OCL engine [5] to allow to query models stored in multiple types of data storage solutions, and can be easily extended to support additional backends.

Internally, the framework defines two model-to-model transformations: *OCL2Gremlin*, that handles model queries expressed using the OCL language [11], and *ATL2Gremlin*, that translates model transformations expressed in ATL [8]. Note that the modular archi-

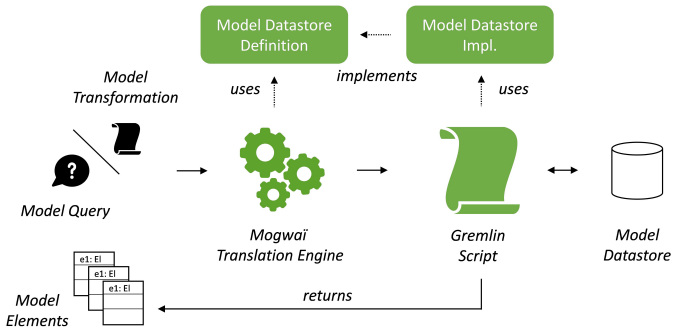

Fig. 1. Mogwaï Infrastructure

tecture of the framework allows to define additional translations to support alternative query and transformation solutions such as EOL [9] or QVT [10].

The generated scripts can be returned to the modeler and used as stored procedures to execute in the future, or directly computed with a specific implementation of the *Model Datastore* library. Finally, the returned elements from the computation (if any) are reified into regular model elements thanks to the *Model Datastore* implementation.

Compared to existing query frameworks, Mogwaï does not rely on the default modeling API to compute model queries and transformations. In general, API based frameworks translate queries and transformations into a sequence of low-level API calls, which are then performed one after another on the persistence layer. While this approach has the benefit to be compatible with every API-based applications, it does not take full advantage of the database structure and query optimizations. Furthermore, each object fetched from the database has to be reified to be navigable, even if it is not going to be part of the end result. Therefore, the execution time and memory consumption of the API-based solutions strongly depends on the number of intermediate objects fetched from the database.

3 Architecture

Figure 2 describes the internal structure of the Mogwaï framework. The *QueryProcessor* is the core of the engine: it provides the *process* method, that takes as its input a *MogwaiQuery* and a set of *ModelDatastores*, and returns a *QueryResult* containing the result of the computation and additional monitoring information (such as the computed *MogwaiQuery* and the raw query execution time).

The *QueryProcessor* relies on an internal *GremlinScriptRunner* that provides utility methods to setup a Gremlin environment and execute queries. Note that the tool provides two implementations of the abstract *QueryProcessor*: the first one, *ATLProcessor*, is dedicated to ATL transformation computation, and the second one, *OCLProcessor*, handles OCL queries. Both processors rely on an internal model to model transformation responsible for mapping the constructs of the input languages to Gremlin. Note that this modular architecture could be easily extended to support alternative query and transformation languages.

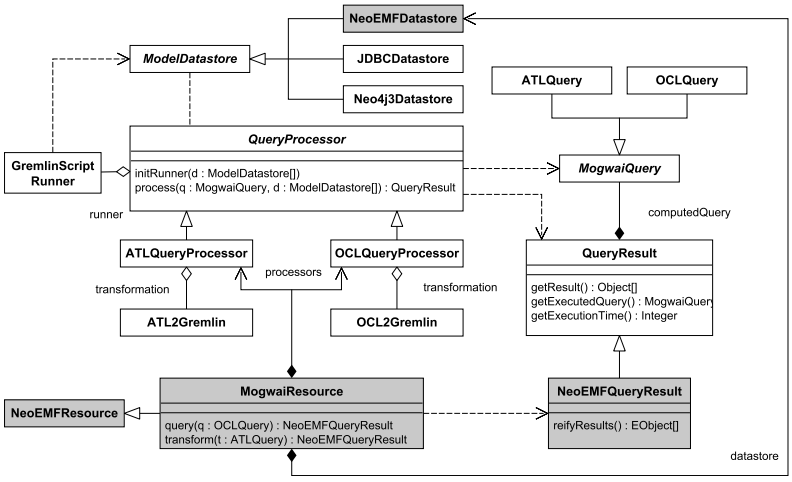

Fig. 2. Mogwaï Internal Structure

The Mogwaï's architecture is not tailored to a specific backend or model persistence technology, and can be used on top of any *ModelDatastore* implementation. However, the tool also provides an advanced integration into the NeoEMF platform (light-grey boxes) that speeds-up query computation and improves the tool's integration with existing EMF-based application by returning EMF-compatible objects. The *MogwaiResource* class extends the NeoEMF one with a simple API defining the `query` and `transform` methods. This resource embeds a set of *QueryProcessors* as well as a preset *ModelDatastore* implementation targeting the native API of the database storing the model. Query and transformations executed through the *MogwaiResource* return *NeoEMFQueryResults*, that contain database records that can be reified into navigable EMF elements if needed. Note that resulting model elements are created only from the results of the Gremlin script execution, removing the memory overhead implied by intermediate objects created during EMF-based computations.

4 Implementation

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[4]. An Eclipse update site containing the last stable version of the framework is also available online[5].

The OCL engine relies on Eclipse MDT OCL [6] to parse the input queries, and the produced OCL models constitute the input of a set of 70 ATL [8] transformation rules and helpers implementing the mapping and the transformation process presented in detail in our previous work [5].

The ATL engine presents a similar architecture, and relies on the ATL parser to create a model from the transformation to compute. This transformation model is then

[4] `https://github.com/atlanmod/Mogwai`
[5] `atlanmod.github.io/Mogwai`

sent to a high-order transformation mapping ATL constructs to Gremlin [3] (represented as a set of 80 rules and helpers).

5 Conclusion

We have showcased Mogwaï, a tool that generates Gremlin scripts from model queries and transformations in order to maximize the benefits of using a NoSQL backend to store and manipulate large models. Gremlin scripts are created using a set of model-to-model transformations, and are parametrized with a specific *Model Datastore*, enabling their computation over a variety of backends compatible with the Gremlin language. The Mogwaï approach allows to bypass the existing modeling framework's API, improving the performance of query and transformation computations both in terms of execution time and memory consumption [3, 5]. The tool development roadmap for Mogwaï includes adding support for more types of NoSQL backends, like document-oriented and column databases by providing the necessary translations from ATL and OCL to their native languages.

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. 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.
3. G. Daniel, F. Jouault, G. Sunyé, and J. Cabot. Gremlin-ATL: a scalable model transformation framework. In *Proceedings of the 32nd ASE Conference*, pages 462–472. IEEE, 2017.
4. 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.
5. 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.
6. Eclipse Foundation. MDT OCL, 2018. URL: `www.eclipse.org/modeling/mdt/?project=ocl`.
7. Eclipse Foundation. The CDO Model Repository (CDO), 2018. URL: `http://www.eclipse.org/cdo/`.
8. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1):31 – 39, 2008.
9. D. S. Kolovos, R. F. Paige, and F. A. Polack. The Epsilon Object Language (EOL). In *Proceedings of the 2nd ECMDA-FA Conference*, pages 128–142. Springer, 2006.
10. OMG. QVT Specification, 2017. URL: `http://www.omg.org/spec/QVT`.
11. OMG. OCL Specification, 2018. URL: `www.omg.org/spec/OCL`.
12. Tinkerpop. The Gremlin Language, 2018. URL: `www.gremlin.tinkerpop.com`.
13. 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*, 98:80–99, 2015.
14. J. Whittle, J. Hutchinson, and M. Rouncefield. The State of Practice in Model-Driven Engineering. *IEEE software*, 31(3):79–85, 2014.

Appendix A Demonstration Overview

The demonstration presents two typical use cases where the Mogwaï framework significantly improves the execution time and memory consumption of an application computing OCL queries and ATL transformations on top of large models.

First, we briefly introduce the `sample` model, that is a real-world model obtained by applying model driven reverse engineering techniques on an existing code base. The manipulated model contains around 80 000 elements representing a Java application. Figure 3 shows an excerpt of the metamodel describing the `sample`. Note that the complete metamodel can be found in the MoDisco repository[6].

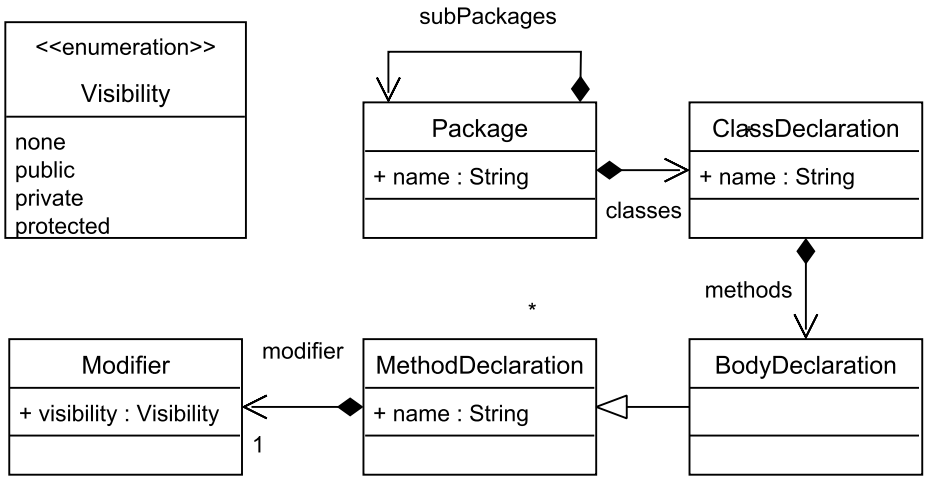

Fig. 3. Excerpt of the MoDisco Java Metamodel

Then, we present a set of OCL queries to compute against this model. An example of such query is provided in Listing 1.1, that describes the `protectedMethod` query, which finds in the model all the *MethodDeclarations* that have a protected *Modifier*. We show in the demonstration how to initialize and configure the Mogwaï engine to translate and run the query from an existing Java application (Figure 4). In parallel, we show how the query is computed using the regular Eclipse MDT-OCL interpreter embedded with Eclipse, and emphasize the execution time and memory consumption differences.

[6] `http://git.eclipse.org/c/modisco`

Listing 1.1. Sample OCL Query

```
context ClassDeclaration
def: protectedMethods : Sequence(BodyDeclaration) =
ClassDeclaration.allInstances()->
 collect(bodyDeclarations)->
  select(each | each.oclIsTypeOf(MethodDeclaration))->
   select(each | not(each.modifier.oclIsUndefined()))->
 select(each | each.modifier.visibility = VisibilityKind::protected)->
 asSequence()
```

MogwaiProtectedMethods.java

```
1  package fr.inria.atlanmod.mogwai.demo.query.mogwai;
2
3  import java.io.File;
23
24 public class MogwaiProtectedMethods {
25
26     public static void main(String[] args) throws IOException {
27         EPackage.Registry.INSTANCE.put(JavaPackage.eNS_URI, JavaPackage.eINSTANCE);
28         JavaPackage.eINSTANCE.eClass();
29
30         PersistenceBackendFactoryRegistry.register(
31                 MogwaiURI.MOGWAI_SCHEME,
32                 BlueprintsPersistenceBackendFactory.getInstance());
33
34         ResourceSet rSet = new ResourceSetImpl();
35         rSet.getResourceFactoryRegistry().getProtocolToFactoryMap()
36             .put(MogwaiURI.MOGWAI_SCHEME, MogwaiResourceFactory.getInstance());
37
38         Resource resource = rSet.createResource(
39                 MogwaiURI.createMogwaiURI(new File("models/myModel.graphdb")));
40         resource.load(Collections.emptyMap());
41         MogwaiResource mogwaiResource = (MogwaiResource)resource;
42
43         MogwaiQuery query = OCLQueryBuilder.newBuilder()
44                 .fromURI(URI.createURI("ocl/protectedMethods.ocl"))
45                 .build();
46         startQuery();
47         QueryResult result = mogwaiResource.query(query);
48         endQuery();
49         printResults(result);
50         mogwaiResource.close();
51     }
```

Fig. 4. Running OCL Queries with Mogwaï

Then, we introduce a simple model-to-model transformation defined with the ATL language to compute on top of the `sample` model (Listing 1.2). This transformation extracts all the *ClassDeclaration* instances from the input model and maps them to the *Table* construct of the output metamodel, and sets a unique *key* that allows to identify a *ClassDeclaration* instance. A second rule is responsible of transforming each *MethodDeclaration* into a *Column* representing the number of calls to the method.

In the demonstration, we show the required steps to initialize the Mogwaï engine with the transformation and compute it against the database storing the model (Fig-

ure 5). In addition, we show how the output model can be stored in another data representation using a different *Model Mapping Implementation*. We also compare the execution time of computing the transformation using the regular ATL engine with Mogwaï, and show that using our approach can bring significant improvements in terms of execution time.

Listing 1.2. Sample ATL Transformation

```
module Class2Relational;
create OUT : RelationalMM from IN : Java;

rule Class2Table {
  from
    c : Java!ClassDeclaration
  to
    out : RelationalMM!Table (
      name ← c.name,
      col ← Sequence{key}->union(c.bodyDeclarations
        ->select(b | b.oclIsTypeOf(Java!MethodDeclaration))),
      key ← key
    ),
    key : RelationalMM!Column (
      name ← 'objectId',
      type ← keyType
    ),
    keyType : RelationalMM!Type (
      name ← 'Integer'
    )
}

rule Method2Column {
  from
    m : Java!MethodDeclaration
  to
    out : RelationalMM!Column (
      name ← m.name + 'CallCount',
      type ← type
    ),
    type : RelationalMM!Type (
      name ← 'Integer'
    )
}
```

Finally, the key points of the tool will be summarized and some remarks on the integration into existing modeling application will be provided. All the presented examples and models will be publicly available on the Mogwai GitHub repository. In addition, a video summarizing the key points of the demonstration is available online at `https://youtu.be/_nTBPJMVRQY`.

MogwaiClassToRelational.java

```java
package fr.inria.atlanmod.mogwai.demo.transformation.mogwai;

import java.io.File;

public class MogwaiClassToRelational {

    public static void main(String[] args) throws IOException {
        EPackage.Registry.INSTANCE.put(JavaPackage.eNS_URI, JavaPackage.eINSTANCE);
        JavaPackage.eINSTANCE.eClass();

        PersistenceBackendFactoryRegistry.register(
                MogwaiURI.MOGWAI_SCHEME,
                BlueprintsPersistenceBackendFactory.getInstance());

        ResourceSet rSet = new ResourceSetImpl();
        rSet.getResourceFactoryRegistry().getProtocolToFactoryMap()
            .put(MogwaiURI.MOGWAI_SCHEME, MogwaiResourceFactory.getInstance());

        Resource resource = rSet.createResource(
                MogwaiURI.createMogwaiURI(new File("models/myModel.graphdb")));
        resource.load(Collections.emptyMap());
        MogwaiResource mogwaiResource = (MogwaiResource)resource;

        MogwaiQuery query = ATLQueryBuilder.newBuilder()
                .fromURI(URI.createURI("atl/Class2Relational.atl"))
                .sourcePackage(JavaPackage.eINSTANCE)
                .targetPackage(ClassDiagramPackage.eINSTANCE)
                .build();

        Map<String, Object> qOptions = new HashMap<>();
        qOptions.put(GremlinScriptRunner.PRINT_SCRIPT_OPTION, true);

        startQuery();
        QueryResult result = mogwaiResource.transform(query, qOptions);
        endQuery();

        mogwaiResource.close();
    }
```

Fig. 5. Running ATL Queries with Mogwaï