Mogwaï: a Framework to Handle Complex Queries on Large Models

Gwendal Daniel
AtlanMod Team
Inria, Mines Nantes & Lina
Gwendal.Daniel@inria.fr

Gerson Sunyé
AtlanMod Team
Inria, Mines Nantes & Lina
Gerson.Sunye@inria.fr

Jordi Cabot
ICREA
UOC
Jordi.Cabot@icrea.cat

***Abstract*—While Model Driven Engineering is gaining more industrial interest, scalability issues when managing large models have become a major problem in current modeling frameworks. Scalable model persistence has been achieved by using NoSQL backends for model storage, but existing modeling framework APIs have not evolved accordingly, limiting NoSQL query performance benefits. In this paper we present the Mogwaï, a scalable and efficient model query framework based on a direct translation of OCL queries to Gremlin, a query language supported by several NoSQL databases. Generated Gremlin expressions are computed inside the database itself, bypassing limitations of existing framework APIs and improving overall performance, as confirmed by our experimental results showing an improvement of execution time up to a factor of 20 and a reduction of the memory overhead up to a factor of 75 for large models.**

***Index Terms*—Model Query, OCL, Gremlin, Scalability, NoSQL.**

I. Introduction

Model queries are a key concept in Model Driven Engineering (MDE). They constitute the basis for several modeling activities, such as model validation [1], derived features computation [2], constraint specification [3], or model transformation [4].

With the progressive adoption of MDE techniques in the industry [5], [6], existing tools have to increasingly deal with large models, and the scalability of existing technical solutions to store, edit collaboratively, transform, and query models has become a major issue [7], [8]. Large models typically appear in various engineering fields, such as civil engineering [9], automotive industry [10], product lines [11], and can be generated in model-driven reverse engineering processes [12], such as software modernization.

In the last decade, the Eclipse Modeling Framework (EMF) [13] has become the *de-facto* standard framework for building modeling tools, offering a strong foundation to implement model storage, querying, and persisting functionalities. The popularity of EMF is attested by the large number of available EMF-based tools on the Eclipse marketplace [14], coming from both industry and academia. Therefore, most of the research works aimed at improving modeling scalability target this framework. Nevertheless, EMF was first designed to handle simple modeling activities, and its default serialization mechanism – XMI [15] – has shown clear limitations to handle very large models [16], [17]. Furthermore, XML-based serialization has two important drawbacks: (i) it favors readability instead of compactness and (ii) XMI files have to be entirely parsed to obtain a navigational model of their contents. The first reduces performance of I/O access operations, while the second increases the memory consumption to load and query a model, and limits the use of proxies and partial loading to inter-document relationships. In addition, XMI implementations do not provide advanced features such as transactions or collaborative edition, and large monolithic model files are challenging to integrate in existing versioning systems [18].

CDO [19] was designed to address those issues by providing a client-server repository structure to handle large model in a collaborative environment. CDO supports transactions and provides a *lazy-loading* mechanism, which allows the manipulation of large models in a reduced amount of memory by loading only accessed objects. Recently, the increasing popularity of NoSQL databases has led to a new generation of persistence frameworks that store models in scalable and schema-less databases. Morsa [16], [20] is one of the first approaches that uses NoSQL databases to handle very large models. It relies on a client-server architecture based on MongoDB and aims to manage scalability issues using document-oriented database facilities. NeoEMF [21] is another persistence framework initially designed to take advantage of graph databases to represent models [17], [22]. It has been extended to a multi-backend solution supporting graph and key-value stores and can be configured with application-level caches to limit database accesses.

While this evolution of model persistence backends has improved the support for managing large models, they are just a partial solution to the scalability problem in current modeling frameworks. In its core, all frameworks are based on the use of low-level model handling APIs. These APIs are then used by most other MDE tools in the framework ecosystem to query and update models. Since these APIs are focused on manipulating individual model elements and do not offer support for generic queries, all kinds of queries required by model-based tools must be translated into a sequence of API calls for individual accesses. This is clearly inefficient when combined with persistence frameworks 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 (e.g. to chain the sequence of fine-grained API calls required to obtain the final result).

To overcome this situation, we propose the Mogwaï, an efficient and scalable query framework for large models. The Mogwaï framework translates model queries written in OCL into expressions of a graph traversal language, Gremlin, which are directly used to query models stored in a NoSQL backend. We argue that this approach is more efficient and scalable than existing solutions relying on low-level APIs. To evaluate our solution, we perform a set of queries extracted from MoDisco [12] software modernization use-cases and compare the results against existing frameworks based on EMF API.

The paper is organized as follows: Section II introduces Gremlin, a language to query multiple NoSQL databases, Section III presents the architecture of our tool. Section IV and V introduces the transformation process from OCL expressions to Gremlin and the prototype we have developed. Section VI describes the benchmarks used to evaluate our solution and the results. Finally, Section VII presents related works and Section VIII summarizes the key points of the paper, draws conclusions, and presents our future work.

II. The Gremlin Query Language

A. Motivation

NoSQL databases are an efficient option to store large models [17], [20]. Nevertheless, their diversity in terms of structure and supported features make them hard to unify under a standard query language to be used as a generic solution for our approach.

Blueprints [23] is an interface designed to unify NoSQL database access under a common API. Initially developed for graph stores, it has been implemented by a large number of databases such as Neo4j, OrientDB, and MongoDB. Blueprints is, to our knowledge, the only interface unifying several NoSQL databases[1].

Blueprints is the base of the Tinkerpop stack: a set of tools to store, serialize, manipulate, and query graph databases. Gremlin [24] is the query language designed to query Blueprints databases. It relies on a lazy data-flow framework and is able to navigate, transform, or filter a graph. It can express graph traversals finely and shows positive performance results when compared to Cypher, the pattern matching language used to query the Neo4j graph database [25].

Therefore, we choose Gremlin as our target language as it is the most mature and generic solution nowadays to query a wider variety of NoSQL databases.

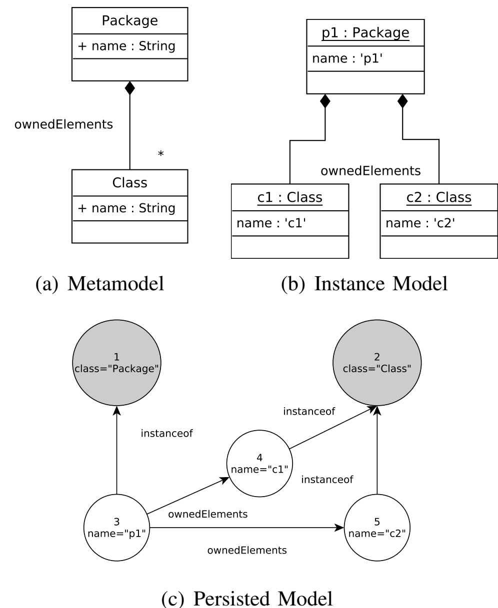
B. Language description

Gremlin is a Groovy domain-specific language built on top of *Pipes*, a data-flow framework based on process graphs. A process graph is composed of vertices representing computational units and communication edges which can be combined to create a complex processing. In the Gremlin terminology, these complex processing are called *traversals*, and are composed of a chain of simple computational units named *steps*. Gremlin defines four types of steps:

- **Transform steps:** functions mapping inputs of a given type to outputs of another type. They constitute the core of Gremlin: they provide access to adjacent vertices, incoming and outgoing edges, and properties. In addition to built-in navigation steps, Gremlin defines a generic *transformation* step that applies a function to its input and returns the computed results.
- **Filter steps:** functions to select or reject input elements w.r.t. a given condition. They are used to check property existence, compare values, remove duplicated results, or retain particular objects in a traversal.
- **Branch steps:** functions to split the computation into several parallelized sub-traversals and merge their results.
- **Side-effect steps:** functions returning their input values and applying side-effect operations (edge or vertex creation, property update, variable definition or assignation).

In addition, the *step* interface provides a set of built-in methods to access meta information: number of objects in a step, output existence, or first element in a step. These methods can be called inside a traversal to control its execution or check conditions on particular elements in a step.

Gremlin allows the definition of custom steps, functions, and variables to handle query results. For example, it is possible to assign the result of a traversal to a variable and use it in another traversal, or define a custom step to handle a particular processing.



(a) Metamodel (b) Instance Model

(c) Persisted Model

Fig. 1. Sample Metamodel and Model

[1]Implementation list is available at https://github.com/tinkerpop/blueprints

As an example, Figure 1(a) shows a simple metamodel representing *Packages* and *Classes*. *Packages* are named containers owning *Classes* through their *ownedElements* reference. An instance of this metamodel is shown in Figure 1(b) and its graph database representation is shown in Figure 1(c). Grey vertices represents *Package* and *Class* metaclasses and are linked to their instance through *instanceof* edges. The package `p1` is linked to classes `c1` and `c2` using *ownedElements* edges.

In what follows, we describe some simple Gremlin examples based on this model. A Gremlin traversal begins with a *Start* step, that gives access to graph level informations such as indexes, vertex and edge lookups, and property based queries. For example, the traversal below performs a query on the *classes* index that returns the vertices indexed with the name *Package*, representing the *Package* class in the Figure 1(a). In our example, this class matches vertex 1.

```
g.idx("classes")[[name:"Package"]]; // → v(1)
```

The most common steps are *transform* steps, that allow navigation in a graph. The steps *outE(rel)* and *inE(rel)* navigate from input vertices to their outgoing and incoming edges, respectively, using the relationship *rel* as filter. *inV* and *outV* are their opposite: they compute head and tail vertices of an edge. For example, the following traversal returns all the vertices that are related to the vertex 3 by the relationship *ownedElements*. The *Start* step `g.v(3)` is a vertex lookup that returns the vertex with the id 3.

```
g.v(3).outE("ownedElements").inV; // → [v(4),v(5)]
```

Filter steps are used to select or reject a subset of input elements given a condition. They are used to filter vertices given a property value, remove duplicate elements in the traversal, or get the elements of a previous step. For example, the following traversal returns all the vertices related to vertex 3 by the relationship *ownedElements* that have a property *name* with a size longer than 1 character.

```
g.v(3).outE("ownedElements").inV
  .has("name").filter{it.name.length > 1}; // → [v(4),v(5)
      ]
```

Branch steps are particular steps used to split a traversal into sub queries, and merge their results. As an example, the following traversal collects all the *id* and *name* properties for the vertices related to vertex 3 by the relationship *ownedElements*. The computation is split using the *copySplit* step and merged in the parent traversal using *exhaustMerge*.

```
g.v(3).outE("ownedElements").inV.copySplit(
  _().name,_().id).exhaustMerge();
```

Finally, *side-effect* steps modify a graph, compute a value, or assign variables in a traversal. They are used to fill collections with step results, update properties, or create elements. For example, it is possible to store the result of the previous traversal in a *table* using the *Fill* step.

```
def table = [];
g.v(3).outE("ownedElements").inV.
  has("name").filter{it.name.length > 10}.fill(table); //
      → [v(4),v(5)]
```

III. The Mogwaï Framework

The Mogwaï framework is our proposal for handling complex queries on large models. As discussed above, we will assume that those large models are stored in a NoSQL backend with Gremlin support. On the modeling side we will also assume that queries are expressed in OCL (Object Constraint Language), the OMG standard for complementing graphical languages with textual descriptions of invariants, operation contracts, derivation rules, and query expressions.

More precisely, the Mogwaï approach relies on a model-to-model transformation that generates Gremlin traversals from OCL queries which are then directly computed by any Blueprints database. The results of the query are then translated back to the modeling framework resulting in the set of modeling objects that satisfies the query expression.

Figure 2 shows the overall query process of (a) the Mogwaï framework and compares it with (b) standard EMF[2] API based approaches.

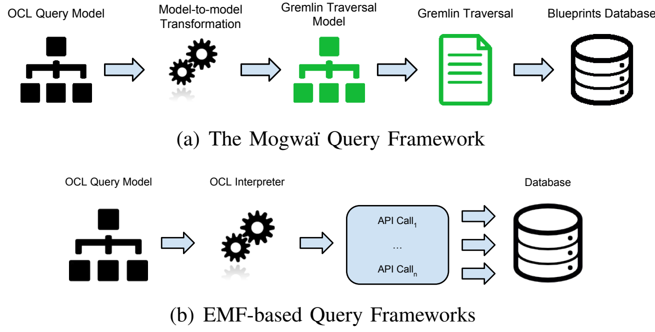
An initial textual OCL expression is parsed to transform it into an OCL model conforming to the OCL metamodel. This model constitute the input of a model-to-model transformation that generates the corresponding Gremlin model. The Gremlin model is then expressed as a text string conforming to the Gremlin grammar and sent to the Blueprints database for its execution.

The main difference with existing query frameworks is that the Mogwaï framework does not rely on the EMF API to perform a query. In general, API based query frameworks translate OCL queries into a sequence of low-level API calls, which are then performed one after the other on the database. While this approach has the benefit to be compatible with every EMF-based application, 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, execution time of the EMF-based solutions strongly depends on the number of intermediate objects reified from the database (which depends on the complexity of the query but also on the size of the model, bigger models will need a larger number of reified objects to represent the intermediate steps) while for the Mogwaï framework, execution time does not depend on the number of intermediate objects, making it more scalable over large models.

Once the Gremlin traversal has been executed on the database side, the results are returned to the framework that reifies those results into the corresponding model elements. With this architecture, it is possible to plug our solution on top of various persistence frameworks and use it in multiple contexts.

To sum up, the transformation process generates a single Gremlin traversal from an OCL query and runs it over the database. This solution provides two benefits: (i) delegation of the query computation to the database, taking full advantage

[2]We focus the explanation on the EMF framework but results are generalizable to all other modeling frameworks we are familiar with.



(a) The Mogwaï Query Framework

(b) EMF-based Query Frameworks

Fig. 2. Comparison of OCL execution

of the built-in caches, indexes, and query optimizers; and (ii) single execution compared to fragmented queries with the EMF API, removing intermediate object reification.

IV. OCL to Gremlin Transformation

A. Mapping of OCL expressions

To illustrate the different phases of the transformation, we introduce a running example: Listing 1 shows a simple query (on a model conforming to Figure 1(a)) that selects the *Packages* instances which contains at least one element through the *ownedElements* reference. The transformation process that generates the Gremlin traversal in Listing 2 relies on the mappings shown in Table I to translate individual OCL expressions into Gremlin steps. In this Section we detail how the different steps of the traversal are generated using this mapping. In the next Section we present how the input OCL syntax tree is processed and generated steps are linked together to produce the complete Gremlin query.

```
def sampleSelect: res : Set(Package) =
    Package.allInstances()→select(e | e.ownedElements→
        isEmpty())
```

Listing 1. Sample OCL Query

```
var packageV = g.idx("classes")[[name:Package]];
packageV.inE("instanceof").outV.filter{e=it;
  e.outE("ownedElements").inV.toList().isEmpty()};
```

Listing 2. Generated Gremlin Textual Traversal

We have divided the supported OCL expressions into four categories based on Gremlin step types: transformations, collection operations, iterators, and general expressions. Note that other types of OCL expressions that are not explicitly listed in the table can be first expressed in terms of those that appear there [26] and therefore be also covered by our mapping.

The first group, transformation expressions, returns a computed value from their input. Expressions that navigate the elements in the model are mapped to navigation steps: `Type` access is translated into an index query returning the vertex representing the type, assuming the type exists. In the example, the *Package* type is mapped to the index call `g.idx("classes")[[name:Package]]`, and the result vertex is stored in a dedicated variable to reduce database accesses. The `AllInstances` collection is mapped to a traversal returning adjacent vertices on the Type vertex having an *instanceof* outgoing edge (`inE("instanceof").outV`). Reference and attribute `collect` operations are respectively mapped to an adjacent vertex collection on the reference name and a property step accessing the attribute. Type conformance is checked by comparing the adjacent *instanceof* vertex with the type one using a generic *transform step*. Finally, attribute and reference `collect` from type casting are mapped as regular `collect` operation, because each vertex in the database contains its inherited attributes and edges.

The second group, operations on collections, needs a particular mapping because Gremlin step content is unmodifiable and cannot support collection modifications natively. `Union`, `intersection` and `set subtraction` expressions are mapped to the *fill* step, which puts the result of the traversal into a variable. We have extended Gremlin by adding *union*, *intersection*, and *subtract* methods that compute the result of those operations from the variables storing the traversed elements. `Including` operation is translated to a *gather* step, that collects all the objects and process the gathered list with a closure that adds the element to includes. The list is then transformed back to a step input by using the *scatter* step. `Excluding` operations can be achieved by using the *except* step, that removes from the traversal all the elements in its argument. A transformation of the step content into a Groovy collection is done to handle `includes` and `excludes` operations, which are then mapped to a containment checking. Finally, functions returning the size and the first element of a collection are mapped to *count()* and *first()* step methods. Note that there is no specific method to check if a collection is empty in Gremlin but this can be achieved by calling a Groovy collection transformation.

Iterator expressions are OCL operations that check a condition over a collection, and return either the filtered collection or a boolean value. `Select` is mapped to a *filter step* with the translation of the condition as its body. In the example the body of the `select` operation contains an implicit `collect` on the reference *ownedElements* and a collection operation `isEmpty()`, that are respectively mapped to `outE("ownedElements").inV` and `toList().isEmpty()`. `Reject` is mapped the same way with a negation of its condition. `Exists` and `forAll` mapping follow the same schema: a *filter* step with the condition or its negation is generated and the number of results is analyzed. Finally, general operations (comparisons, boolean operations, variable declaration, and literals) are simply mapped to their Groovy equivalent.

The mapping presented in this Section produces all the Gremlin *steps* of the result traversal. In the next section, we detail the processing of the input OCL expression and how these *steps* are linked to produce the complete Gremlin query shown in Listing 2.

B. Transformation Process

1) OCL Metamodel: The input of the transformation is an OCL model representing the abstract syntax tree of the

TABLE I
OCL TO GREMLIN MAPPING

OCL expression	Gremlin step
Type	g.idx('classes')[[name:'Type']][a]
allInstances()	inE('instanceof').outV
collect(attribute)	attribute
attribute (implicit collection)	attribute
collect(reference)	outE('reference').inV
reference (implicit collection)	o.outE('reference').inV
oclIsTypeOf(C)	o.outE('instanceof').inV.transform{it.next() == C}
oclAsType(C).attribute	attribute
oclAsType(C).reference	outE('reference').inV
col_1 →union(col_2)	col_1.fill(var_1); col_2.fill(var_2); union(var_1, var_2);
col_1 →intersection(col_2)	col_1.fill(var_1); col_2.fill(var2); intersection(var_1, var_2);
$col_1 - col_2$ (Set subtraction)	col_1.fill(var_1); col_2.fill(var_2); subtract(var_1, var_2);
including(object)	gather{it << object;}.scatter;
excluding(object)	except([object]);
includes(object)	toList().contains(object)
excludes(object)	!(toList().contains(object))
size()	count()
first()	first()
isEmpty()	toList().isEmpty()
select(condition)	c.filter{condition}
reject(condition)	c.filter{!(condition)}
exists(expression)	filter{condition}.hasNext()
forAll(expression)	!(filter{!condition}.hasNext())
$=, >, >=, <, <=, <>$	$==, >, >=, <, <=, !=$
$+, -, /, \%, *$	$+, -, /, \%, *$
and,or,not	&&,||,!
variable	variable
literals	*literals*

[a]Results of index queries are stored in dedicated variables to optimize database accesses

OCL query to perform. Figure 3 presents a simplified excerpt of the OCL metamodel[3]. In the OCL, a *Constraint* is a named top-level container that contains a *specification* described in an *ExpressionInOCL* element. This expression is composed of an *OCLExpression* representing its *body*, a *context variable* (self), and may define *result* and *parameter variables*. An *OCLExpression* can be a type access (*TypeExp*), a variable access or definition (*VariableExp*), or an abstract call expression (*CallExp*). *CallExp* are divided into three subclasses: *OperationCallExp* representing OCL operations, *PropertyCallExp* representing property navigations (attribute and reference accesses), and *IteratorExp* representing iteration loops over collections. These *IteratorExp* elements define an iterator *Variable*, and contains a *body OCLExpression* representing the expression to apply on each element of their input. Finally, expressions can be chained by the *CallExp source* reference representing the element the call apply on, or by being an *argument* of an *OperationCallExp*. In the OCL metamodel we use, all the operations are encapsulated into *OperationCallExp* elements. The actual identifier of the operation is contained in the *name* attribute.

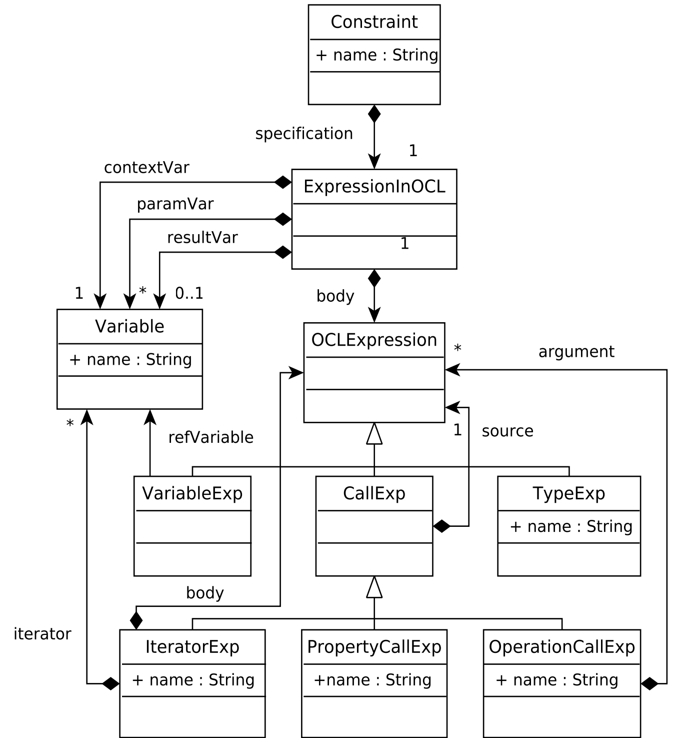


Fig. 3. Extract of OCL Metamodel

Figure 4 shows the instance of the OCL metamodel representing the abstract syntax tree for the sample query presented in Listing 1. The top level element *Constraint* `sampleSelect` contains the *context variable* `self` of the

[3]The complete OCL metamodel we use is available at http://tinyurl.com/hof89by

query, its *result variable* (`res`), and an *ExpressionInOCL* element representing the query itself. Each expression in the OCL metamodel is linked to its *source* expression, in charge of computing the object/s on which the next expression will be applied. In the example the *ExpressionInOCL body* contains the root expression in the *source* tree of the query (the *select* in this case). This *select* iterator has the *allInstances* operation as its source, which has itself a *source* reference on the *TypeExp* `Package` (meaning that we iterate over the whole population of the `Package` class). It also defines an iterator *variable* `e` and a *body* tree (representing the expression to evaluate over each element of the *source* collection) starting with the *isEmpty* operation that is the root of the expression. This operation is applied on the result of the `ownedElements` property navigation, which has a *source* reference to a *VariableExp* expression that refers to the iterator `e`.

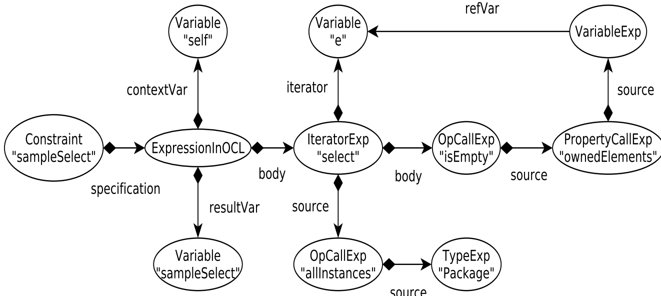


Fig. 4. OCL Query Syntax Tree

2) Gremlin Metamodel: The output of our model-to-model transformation is a Gremlin model. Since Gremlin does not have a metamodel-based representation of its grammar we propose our own Gremlin metamodel. As Gremlin is a Groovy based language, it could have been possible to reuse the Java or Groovy metamodels but they are too large for our needs and miss an easy way to define the concept of step, a core concept specific to Gremlin.

Figure 5 presents the Gremlin metamodel we use in our approach. In this metamodel, a *GremlinScript* is defined by a set of *Instructions* that can be *TraversalElements*, *VariableDeclarations*, or *Expressions*. Supported *Expressions* are unary and binary comparisons, boolean operations, and *Literals*. *UnaryExpressions* and *BinaryExpressions* contain respectively one and two inner *Instructions*. A *TraversalElement* is a single computation step in a Gremlin traversal. It can be either a Gremlin *Step*, a *VariableAccess*, or a *MethodCall*. *TraversalElements* are organized through a composite pattern: each *Step* has a *next* containment reference that links to the next *TraversalElement* in the chain. In the Gremlin terminology, such a chain of computation is called a *traversal*. *Step* elements are the core concept in the Gremlin language. We represent each step presented in Section II and the ones defined in the Gremlin documentation[4] as subclasses of the *Step* class. *Steps* subclasses can contain attributes, like *InEStep* or *OutEStep*, that contain the *label* of the edge to navigate. *EdgesStep* and *VerticesStep* correspond to edge and vertex lookup (`g.E()` and `g.V()`). Finally, *FilterSteps* are particular *Steps* that contain a reference to a *Closure*, that is defined by a set of *Instructions* that are applied on each filtered element.

For the sake of readability we only put the key concepts in this metamodel excerpt. In particular, we omit an important number of *Steps* and *MethodCalls*, as well as the concrete subclasses of supported unary and binary expressions. A complete definition of the metamodel is provided in the project repository[5].

Figure 6 presents the instance of the Gremlin metamodel corresponding to the traversal shown in Listing 2. The top-level *GremlinScript* contains two *instructions*. The first one is a *VariableDeclaration* that defines the variable `packageV`. The *value* of this variable is defined by a Gremlin traversal composed of a *Start* step (the initial access to the graph), an *IndexCall* representing the index query returning the vertex representing the metaclass *Package*, and a *NextCall* that unroll the step content and returns the vertex. The second *instruction* is a *VariableAccess*, representing the access to the variable defined in the previous instruction. This access is the begining of a second traversal composed of navigation steps (*InE*, *OutV*), and a *Filter*. This last step contains a *Closure*, representing the boolean condition of the *Filter*. This *Closure* is composed of two *instructions*: a *VariableDeclaration* that is mapped to the closure iterator, and a *VariableAccess* followed by a navigation, a *ToList* cast, and a Groovy *IsEmpty* check. As shown in the example, it is possible to compose *Steps* and regular Groovy *MethodCalls* in order to create a traversal.

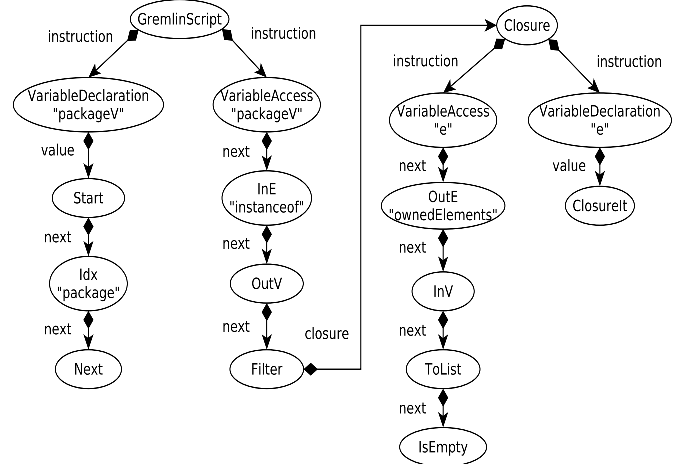


Fig. 6. Generated Gremlin Syntax Tree

3) Transformation Execution: To create a complete Gremlin traversal, the Mogwaï framework needs to process the AST model representing the syntax tree of the OCL query. In this Section we present how the input OCL query is navigated and how the different elements produced by the mappings

[4]http://tinyurl.com/j9hloxr

[5]http://tinyurl.com/peuyu32

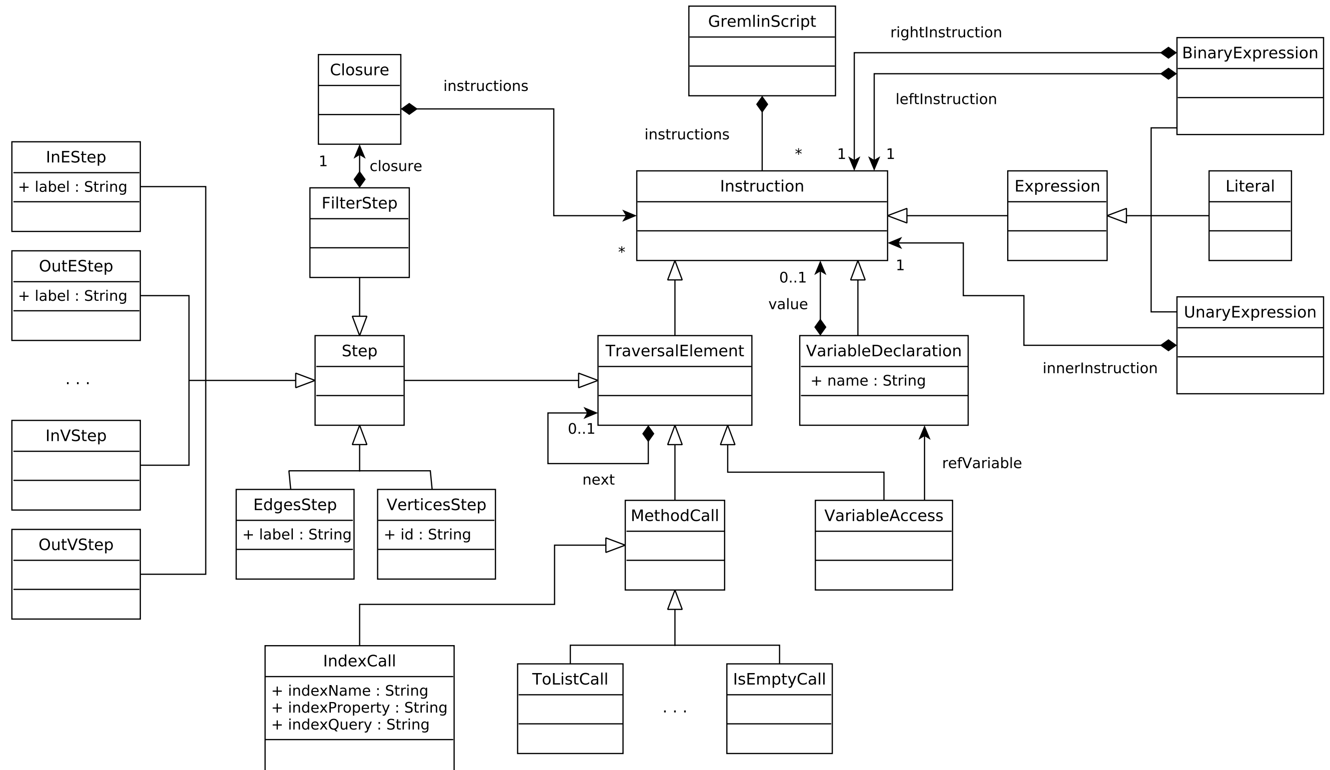


Fig. 5. Extract of Gremlin Metamodel

presented in Table I are assembled to create the final Gremlin script. For the sake of clarity, we provide an overview of the transformation in Figure 7, which presents how an input *OCL Query Model* (1) is processed to produce the output *Gremlin Traversal Model* (9).

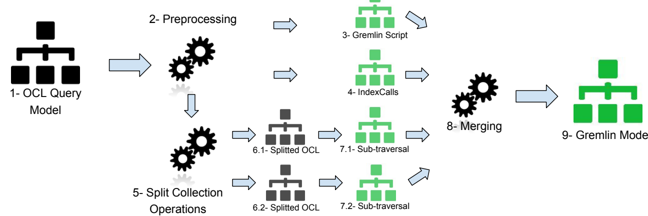


Fig. 7. Overview of the Transformation

The transformation starts by processing (2) the top-level OCL *Constraint* and generates the corresponding *Gremlin-Script* element (3). The input model is then inspected to find if the context variable `self` is accessed in the *Constraint* body. If it is the case, a *VariableDeclaration* element is created. The *value* of the created variable is not set at the model level, it will be binded by the framework before executing the query (see Section V). The same processing is performed to generate *VariableDeclarations* from *parameter variables*.

Once this is done, the transformation collects all the *Type-Exp* element in the input model, and generates corresponding traversals containing an *IndexCall* (4) that returns the vertex from the database index representing the accessed type. The results of these calls are stored in *VariableDeclaration* elements. These variables are created to improve the execution performance of the generated script by caching index results and limit database access. During this step, a mapping between generated variables and *TypeExp* elements is computed. This mapping is then reused in the transformation to transform every *TypeExp* into a *VariableAccess* element.

In the Gremlin language, it is not possible to merge natively two traversals that do not have the same start *step*. Furthermore, Groovy Collection API does not provide methods to merge or subtract two collections and return the updated collection (methods such as `addAll` and `removeAll` return a boolean value, and thus can not be used as the input of the next computation step). A consequence of this limitation is that it is not possible to express in a single traversal *union*, *intersection*, and *set subtraction* operations. To handle these expressions, it is necessary to split the input OCL query (5) into several traversals representing each part of the operation. To handle that, the transformation collects all the root elements of each part (source and argument) of *union*, *intersection*, and *set subtraction* operations in the input model, and creates *VariableDeclarations* to store the result of the subexpressions. Each root element corresponds to an OCL expression (6.1, 6.2) that will be translated into a single traversal (7.1, 7.2). During this step, helper functions that compute the results of these OCL operations are also generated.

The elements created in the different steps of the transformation are then merged (8) inside the *GremlinScript* to produce the output *Gremlin Traversal Model* (9).

To better illustrate how the transformation works, we discuss now how the Mogwaï framework transforms the OCL expression in Listing 1 to the final Gremlin expression shown in Figure 6 (abstract syntax tree) and Listing 2 (final textual expression).

As an initial step, the transformation has to preprocess the OCL model to first find the access to context and parameter variables. In our example, the input OCL expression does not contain references to those variables, and no *VariableDeclaration* are created. Then, the transformation collects the *TypeExp* `Package` and creates the corresponding *VariableDeclaration* `packageV`. The *value* of the created variable is defined by the traversal composed of a *StartStep*, an *IndexCall*, and a *Next* method call, representing the query performed on the `class` index returning the vertex corresponding to the metaclass `Package`.

Then, the transformation collects *union*, *intersection*, and *set subtraction* operations and computes their *source* and *argument* root elements. In our example, there is no such operation, and this phase simply returns the root element of the entire OCL expression.

Once this is done, we can start with the actual transformation of the OCL expressions computed in the pre-processing phase. To generate the first *step* of a traversal, the root expression in the *source* chain is retrieved and transformed according to Table I. In the example, the type access *TypeExp* is transformed into a *VariableAccess* (the one defined during the pre-processing phase). Next elements in the traversal are generated by processing the *source* containment tree in a postorder traversal where transformed OCL nodes are mapped and linked to the previous generated *step* using the *next* reference. In the example, this processing generates the Gremlin nodes *inE('instanceof)'* and *outV* corresponding to the *allInstance* expression.

Iterator operations need a particular processing: their *body* has to be transformed as well. In the example, the *select* iterator is transformed into a *filter* step containing a *closure* that represents its *body*. The *body* expression is parsed starting from the root element and generated *steps* are linked together. In Figure 6, *body* expression is mapped to *variableAccess*, *outE('ownedElements')* and *inV*, *toList* and *isEmpty*, corresponding respectively to the iterator access, *collect(ownedElements)*, and *isEmpty* OCL expressions. The iterator *Variable* generates a *VariableDeclaration* instruction. The name of the iterator *Variable* is assigned to the generated *VariableDeclaration*, and its *value* contains the closure *it* value, that represents the current element processed. This variable shadowing is necessary to avoid *it* erasement in nested iterators.

Finally, if the OCL expression ends with an *Union*, *intersection*, or *set subtraction* operation, or if it is the last one in the *argument* expression, the transformation generates a *Fill* step that ends the traversal and puts the results in the dedicated variable defined in the initial step. Then, if the result of the operation is the source of another OCL expression, the transformation generates another traversal that starts with a *MethodCall* element representing the call to the helper function generated in the initial step.

To better illustrate this particular processing, we present the transformation process of the simple OCL expression shown in Listing 3.

```
p1.ownedElements.name→union(
  p2.ownedElements.name)→size()
```

Listing 3. Sample Union OCL Query

This expression collects the names of the elements contained in the packages `p1` and `p2`, merge them using an *union* operation, and returns the size of the computed collection. As stated before, the transformation starts by processing the input model in order to find source and argument root elements of the *union* operation. In this example, this processing returns the *VariableExp* `p1` and `p2`. The transformation then generates two *VariableDeclarations* named `union1` and `union2` to store the results of the subexpressions `p1.ownedElements.name` and `p2.ownedElements.name` (line 1–2 in Listing 4). Then each subexpression is translated according to the process presented before, and the resulting traversals are affected to the generated *VariableDeclaration* using a *Fill* step (lines 3–4). In addition, the transformation generates the helper function `union(col1,col2)` that performs the union of two collections (lines 5–7). Finally, the transformation processes the `size` call, that has the result of the union call as its *source*. A *MethodCall* is generated that represents a call to the helper function `union`, and constitute the input of the last traversal that computes the size of the collection (line 8).

```
1 var union1;
2 var union2;
3 p1.outE('ownedElements').inV.name.fill(union1);
4 p2.outE('ownedElements').inV.name.fill(union2);
5 def union(col1,col2) {
6     // union helper body
7 }
8 union(union1,union2).count();
```

Listing 4. Sample Union Traversal

Once the traversal model has been generated, it is then parsed to produce the textual Gremlin query that is finally processed by our tool as described in the next section.

V. Tool Support

A prototype implementation of the Mogwaï framework is provided as part of NeoEMF [17], a NoSQL persistence framework built on top of the EMF. It is implemented as an extension of the framework and supports query translation, execution, and result reification from Blueprint's persisted models. The framework presents a simple query API, that accepts a textual OCL expression or an URI to an OCL file containing the expression to transform. In addition, the query API accepts input values that represents *self* and parameter variables.

Initial OCL queries are parsed using Eclipse MDT OCL [27] and the output OCL models constitute the input of a set of 70

ATL [4] transformation rules and helpers implementing the mapping presented in Table I and the associated transformation process (Section IV-B3). As an example, Listing 5 shows the transformation rule that creates a *filter* step from an OCL *select* operation. The *next* step is computed by the `getContainer` helper, which returns the parent of the element in the source tree. The *instructions* of the *closure* are contained in an ordered set, to ensure the instruction defining the iterator variable (rule *var2def*) is generated before the body instructions. Finally, the *select* body is generated, using the helper `getFirstInstruction` that returns the root element in a source tree.

```
rule select2filter {
    from
    s : OCL!IteratorExp (s.getOpName() = 'select')
    to
    f : Gremlin!FilterStep (
        closure ← cl,
        next ← select.getContainer()),
    cl : Gremlin!Closure(
        instructions ← OrderedSet{}
            .append(thisModule.var2def(select.iterator.
                first()))
            .append(select.body.getFirstInstruction()))
}
```

Listing 5. Select to Filter ATL Transformation Rule

Once the Gremlin model is generated by the transformation, it is expressed using its textual concrete syntax and input values corresponding to context and parameter variables are binded to the ones defined during the transformation. The resulting script is sent to an embedded Gremlin engine, which executes the traversal on the database and returns the result back to NeoEMF that reifies it to create a navigable EMF model. The reification process is done once the query has been entirely executed, and the constructed model only contains the query result objects, removing the memory overhead implied by created objects from intermediate steps of the traversal.

Finally, it is also possible to provide input elements to the Mogwaï framework to check invariants, compute a value, or navigate the model from them.

VI. Evaluation

In this section, we evaluate the performance of the Mogwaï framework to query EMF models, in terms of memory footprint and execution time. Results are compared against performance of different querying APIs/strategies (EMF-Query, standard Eclipse OCL, IncQuery, and Mogwaï) on top of the NeoEMF/Graph backend with Neo4j.

A complementary comparison of the increased level of performance due to the uses of NoSQL solutions over SQL ones has been done in previous work [21].

Experiments are executed on a computer running Fedora 20 64 bits. Relevant hardware elements are: an Intel Core I7 processor (2.7 GHz), 16 GB of DDR3 SDRAM (1600 MHz) and a SSD hard-disk. Experiments are executed on Eclipse 4.4.1 (Luna) running Java SE Runtime Environment 1.7. To run our queries, we set the virtual machine parameters `-server` and `-XX:+UseConcMarkSweepGC` that are recommanded in Neo4j documentation.

TABLE II
Overview of the experimental sets

Plug-in	# LOC	XMI Size	# Elements
org.eclipse.gmt.modisco.java	22 074	20.2 MB	80 664
org.eclipse.jdt.core	328 568	420.6 MB	1 557 006

A. Benchmark presentation

The experiments are run over two large models automatically generated using the MoDisco Java Discoverer. MoDisco is a reverse engineering tool able to obtain complete (low-level) models from Java code. The two example models are the result of applying Modisco on two Eclipse Java plug-ins: the MoDisco plug-in itself and the Eclipse Java Development Tools (JDT) core plug-in. Table II shows the details of the experimental sets in terms of number of line of code (LOC) in the plug-in, resulting XMI file size and number of model elements.

To compare the scalability of the different approaches, we perform several queries on the previous models. To simulate a realistic setting, these queries are taken from typical MoDisco software modernization use cases. Queries retrieve:[6]

- **InvisibleMethods**: collects the set of *private* and *protected* methods of a Java project.
- **Grabats09**: returns the set of static methods returning their containing class (singleton patterns).
- **ThrownExceptions**: returns the list of exceptions thrown in each package of the plug-in.
- **TextElementInJavadoc**: returns the textual contents of the *Javadoc* tags in comments of the input *Model* element.
- **EmptyTextInJavadoc**: returns the empty textual contents of the *Javadoc* tags in comments of the input *Model* element.

The first three queries start with an `allInstances` call, which is an important bottleneck for EMF API based query frameworks [28]. The fourth perform a partial navigation from the input *Model* element, and returns all the textual contents in the it javadoc comments. The last query navigates the model the same way, but only returns empty comments. Table III shows the number of intermediate objects loaded using EMF API (#Interm.) and the size of the result set (#Res.) to give an idea of the query complexity.

Correctness of the translation has been checked by comparing manually the results of the Mogwaï framework against results of the Eclipse OCL interpreter. In addition, we provide several test suites in the project repository[7] that check the validity of the translation of single OCL expression and multiple expression composition.

All the queries are executed under two memory configurations: the first one is a large virtual machine of 8 GB and the second is a small-one of 250 MB. This allows us to compare the different approaches both in normal and stressed memory conditions.

[6]Query benchmarks can be found at http://tinyurl.com/nhpf6pq
[7]http://tinyurl.com/jrj66og

TABLE III
NUMBER OF LOADED OBJECTS AND RESULT SIZE FOR MODISCO AND JDT

	MoDisco		JDT	
	#Interm.	**#Res.**	**#Interm.**	**#Res.**
InvisibleMethods	80 664	134	1 557 006	3927
Grabats09	80 664	0	1 557 006	92
ThrownExceptions	80 664	0	1 557 006	1155
TextElementInJavadoc	28 505	12 359	136 753	54 201
EmptyTextInJavadoc	28 505	0	136 753	0

B. Results

Tables IV (MoDisco) and V (JDT) present the average results of 100 executions of the presented queries with EMF-Query, Eclipse OCL interpreter, IncQuery (detailed in Section VII), and the Mogwaï framework on a NeoEMF/Graph database. The correctness of query results has been validated by comparing the results of the different frameworks with the ones of the queries executed with the OCL interpreter.

Left columns of the tables present the time to perform the queries while right columns focus on the memory consumption implied by the query computation. All the tables present the results for large and small virtual machine configurations. Note that we were not able to express *TextElementInJavadoc* and *EmptyTextInJavadoc* queries as EMF-Query code.

C. Discussion

The Mogwaï framework outperforms the other query frameworks executed over NeoEMF/Graph both in terms of memory consumption and execution time. Results of `allInstances` based queries (1–3) show that the difference in terms of execution time is up to 20 times better than the Eclipse OCL interpreter and the EMF-Query framework, plus up to 75 times better in terms of memory consumption. This improvement is explained by (i) the absence of created intermediate objects that consume time and memory and (ii) the use of indexes and query optimizations on the database side, avoiding a complete traversal of the model elements.

Instead, if the query traverses a small subset of the model and does not use database indexes (queries 4 and 5) the benefits of using the Mogwaï framework are reduced. The result of the fourth query (where an important part of the intermediate elements are needed anyway since they are part of the result set) confirms this observation. Fourth and fifth query results also show that an important memory consumption is implied by the reification of the result elements. This overhead also impacts query execution time: execution and result reification of the fourth query over the *JDT* model is around 1 s longer than for the fifth query.

Note that the comparison only considers a single execution of each query over non-loaded models. In cases where the query is executed many times over a slightly different version of the same model, an incremental approach like the one provided by IncQuery could be a very interesting complement to our approach by using the Mogwaï framework to perform initialization queries of the incremental engine and then let the incremental engine take over from there.

To summarize these results, the Mogwaï framework is an interesting solution to perform complex queries over large models. Using query translation approach, gains in terms of execution time and memory consumption are positive, but the results also show that the overhead implied by the transformation engine may not be worthwhile when dealing with relatively small models or simple queries.

The main disadvantage of the Mogwaï framework concerns its integration to an EMF environment. To benefit from the Mogwaï, other Eclipse plug-ins need to be explicitly instructed to use it. Integration with the Mogwaï framework is straighforward but must be explicitly done. Instead, other solutions based on the standard EMF API provide benefits in a transparent manner to all tools using that API.

VII. RELATED WORK

There are several frameworks to query models, specially targeting the EMF framework (including one or more of the EMF backends mentioned in Section I). The main ones are Eclipse MDT OCL [27], EMF-Query [29] and IncQuery [30].

Eclipse MDT OCL provides an execution environment to evaluate OCL invariants and queries over models. It relies on the EMF API to navigate the model, and stores `allInstances` results in a cache to speed up their computation. EMF-Query is a framework that provides an abstraction layer on top of the EMF API to query a model. It includes a set of tools to ease the definition of queries and manipulate results. Compared to the Mogwaï framework, these two solutions are strongly dependent on the EMF API, providing on the one hand an easy integration in existing EMF applications, but on the other hand they are unable to benefit from all performance advantages of NoSQL databases due to this API dependency.

EMF-IncQuery [30] is an incremental pattern matcher framework to query EMF models. It bypasses API limitations using a persistence-independent index mechanism to improve model access performance. It is based on an adaptation of a RETE algorithm, and query results are cached and incrementally updated using the EMF notification mechanism to improve performance. While EMF-IncQuery shows great execution time performances [1] when repeating a query multiple times on a model, the results presented in this article show mitigated performances for single evaluation of queries. This is not the case for our framework. Caches and indexes must be build for each query, implying a non-negligible memory overhead compared to the Mogwaï framework. In addition, the initialization of the index needs a complete resource traversal, based on EMF API, which can be costly for *lazy-loading* persistence frameworks.

Alternatively, other approaches that target the translation of OCL expressions to other languages/technologies [31] are also relevant to our work. For example, Heidenreichin et al. [32] propose a solution to automatically build a database from a UML representation of an application, and translate the OCL invariants into database constraints. A similar approach was proposed by Brambilla et al. [33] in the field of web applications. In that case, queries are translated into triggers

TABLE IV
QUERY FRAMEWORK RESULTS ON MODISCO MODEL (LARGE VM / SMALL VM)

	Execution Time (s)				**Memory Consumption** (MB)			
	EMF-Query	**OCL**	**IncQuery**	**Mogwaï**	**EMF-Query**	**OCL**	**IncQuery**	**Mogwaï**
InvisibleMethods	9/9	10/10	20/20	4/4	25/21	23/23	34/34	4/4
Grabats09	9/9	11/10	20/20	4/4	24/24	19/23	39/41	4/4
ThrownExceptions	11/11	10/10	20/19	4/4	23/24	19/19	30/26	7/6
TextElementInJavadoc	X	6/6	20/20	5/5	X	10/9	55/54	8/8
EmptyTextInJavadoc	X	7/7	22/21	4/4	X	10/9	58/53	7/7

TABLE V
QUERY FRAMEWORK RESULTS ON JDT MODEL (LARGE VM / SMALL VM)

	Execution Time (s)				**Memory Consumption** (MB)			
	EMF-Query	**OCL**	**IncQuery**	**Mogwaï**	**EMF-Query**	**OCL**	**IncQuery**	**Mogwaï**
InvisibleMethods	133/169	151/153	326/662	8/9	392/116	393/93	550/162	6/6
Grabats09	131/157	154/158	332/2418	7/7	388/120	389/127	616/228	5/5
ThrownExceptions	171/198	151/157	324/457	6/6	388/120	386/92	486/81	7/7
TextElementInJavadoc	X	19/20	321/595	10/10	X	39/41	570/171	25/23
EmptyTextInJavadoc	X	20/21	322/592	9/9	X	42/40	568/173	7/6

or views. Nevertheless, in all these scenarios the goal is to use OCL for code-generation purposes as part of a data validation component. Similar generative approaches exist also for other pairs of query and target languages [34]. Once generated, there is no link between the code and the models and therefore it cannot be used to speed up the model queries. In addition, all these approaches perform the translation at compilation-time, whereas the Mogwaï framework translates OCL queries to Gremlin at runtime.

De Carlos et al. [35], [36] present the Model Query Translator (MQT), an approach similar to the Mogwaï framework that translates EOL [37] queries into SQL. MQT uses a metamodel-agnostic database schema to store models, and it extends EOL to produce optimized SQL queries executed on the database side. Our translation approach is different because it relies on a model-to-model transformation to produce Gremlin traversals from OCL queries, allowing runtime execution of the transformation as well as preparation of the traversals at compilation time. In addition, graph-based navigation of models removes the overhead implied by complex joins, and the Gremlin language is expressive enough[8] to translate the entire OCL.

Beyond the EMF world, proprietary meta-modeling tools provide specific query languages. This is the case of MetaEdit+ [38] from MetaCase, a commercial tool that supports the development of domain-specific languages, which provides a proprietary query language. This is also the case of ConceptBase [39], a deductive object manager for conceptual modeling and meta-modeling, which provides O-Telos, a query language for deductive databases.

Efficient model queries can also be linked to live models and Models@Run.Time [40], which aims to create adaptive software that keeps a model representation of the running system during the execution. In this environment, models become decisional artifacts that are queried during the execution to take decisions, compute metrics, or retrieve information. In this context, time and memory consumption are critical aspects, since the decisions (i. e., the queries) have to be taken as quickly as possible in a stressed and concurrent environment. The results presented in this article show that the Mogwaï framework can be an interesting candidate to handle these queries both in term of memory consumption and time performance.

[8]Gremlin is written using the Groovy programming language, which is a dynamic imperative language for the Java platform

VIII. CONCLUSIONS AND FUTURE WORK

In this paper we presented the Mogwaï, a framework that generates Gremlin traversals from OCL queries in order to maximize the benefits of using a NoSQL backend to store large models. OCL queries are translated using model-to-model transformation into Gremlin traversals that are then computed on the database side, reducing the overhead implied by the modeling API and the reification of intermediate objects. We also presented a prototype integrated in NeoEMF/Graph, a scalable model persistence framework that stores models into graph databases. Our experiments have shown that the Mogwaï framework outperforms existing solutions in terms of memory consumption (up to a factor of 75) and execution time (up to a factor of 20) to perform complex queries over large models.

Model transformations intensively rely on model queries to match candidate elements to transform and navigate source model. Integrating the Mogwaï framework in model transformation engines to compute these queries directly on the database could reduce drastically execution time and memory consumption implied by the transformation of large models. Another possible solution to enhance transformation engines would be to translate the transformation itself into database queries. This approach would allow to benefit of the Mogwaï framework improvements for model queries as well as element creation, deletion, or update.

As future work, we plan to study the definition of Gremlin's custom steps and to optimize collection operations to produce more readable traversals. Moreover, while the Gremlin language defines update operations, these modifications cannot

be expressed using standard OCL, which is a side-effect free language. We plan to combine our OCL support with imperative constructs [41] allowing the efficient execution of complex update operations as well. We also plan to study the impact of semantically-equivalent OCL expressions [26] on generated traversals. With this information, it could be possible to improve the quality of the traversals by first applying an automatic refactoring on the OCL side.

Finally, we would like to study the integration of the Mogwaï framework into model persistence solutions that do not rely on a Gremlin compatible database. For instance, we plan to adapt existing work on EOL to SQL translation [35] to test our model-to-model transformation based approach over SQL databases.

REFERENCES

[1] G. Bergmann, A. Horváth, I. Ráth, and D. Varró, "Incremental evaluation of model queries over emf models: A tutorial on emf-incquery," in *Proc. of the 7th ECMFA*, Berlin, Heidelberg, 2011, pp. 389–390.

[2] I. Ráth, A. Hegedüs, and D. Varró, "Derived features for emf by integrating advanced model queries," in *Proc. of the 8th ECMFA*, Kgs. Lyngby, Denmark, 2012, pp. 102–117.

[3] OMG, "OCL Specification," 2015, URL: http://www.omg.org/spec/OCL. [Online]. Available: http://www.omg.org/spec/OCL

[4] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "Atl: A model transformation tool," *SCP*, vol. 72, no. 1–2, pp. 31 – 39, 2008, special Issue on Second issue of experimental software and toolkits (EST).

[5] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-driven engineering practices in industry," in *Proc of the 33rd ICSE*. IEEE, 2011, pp. 633–642.

[6] P. Mohagheghi, M. A. Fernandez, J. A. Martell, M. Fritzsche, and W. Gilani, "Mde adoption in industry: challenges and success criteria," in *Proc. of Workshops at MoDELS 2008*. Springer, 2009, pp. 54–59.

[7] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi *et al.*, "A research roadmap towards achieving scalability in model driven engineering," in *Proc. of BigMDE*. ACM, 2013, pp. 1–10.

[8] J. Warmer and A. Kleppe, "Building a flexible software factory using partial domain specific models," in *Proc. of the 6th OOPSLA Workshop on Domain-Specific Modeling*. University of Jyvaskyla, 2006, pp. 15–22.

[9] S. Azhar, "Building information modeling (bim): Trends, benefits, risks, and challenges for the aec industry," *Leadership and Management in Engineering*, vol. 11, no. 3, pp. 241–252, 2011.

[10] G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös, "Incremental evaluation of model queries over emf models," in *Proc. of the 13th MoDELS Conference*. Springer, 2010, pp. 76–90.

[11] R. Pohjonen and J.-P. Tolvanen, "Automated production of family members: Lessons learned," in *Proc. of PLEES*, 2002, pp. 49–57.

[12] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot, "Modisco: A model driven reverse engineering framework," *IST*, pp. 1012 – 1032, 2014.

[13] The Eclipse Foundation, "The Eclipse Modeling Framework," 2015. [Online]. Available: https://www.eclipse.org/modeling/emf

[14] Eclipse Foundation, "Eclipse Marketplace - Modeling Tools," 2015. [Online]. Available: http://marketplace.eclipse.org/

[15] OMG, "OMG MOF 2 XMI Mapping Specification version 2.4.1," Object Management Group, August 2011. [Online]. Available: http://www.omg.org/spec/XMI/2.4.1/

[16] J. E. Pagán and J. G. Molina, "Querying large models efficiently," *IST*, pp. 586–622, 2014.

[17] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay, "Neo4EMF, a Scalable Persistence Layer for EMF Models," in *Proc. of the 10th ECMFA*, York, United Kingdom, 2014, pp. 230–241.

[18] K. Barmpis and D. Kolovos, "Hawk: Towards a scalable model indexing architecture," in *Proc. of BigMDE'13*. New York, NY, USA: ACM, 2013, pp. 6:1–6:9.

[19] Eclipse Foundation, "The CDO Model Repository (CDO)," 2015. [Online]. Available: http://www.eclipse.org/cdo/

[20] J. E. Pagán, J. S. Cuadrado, and J. G. Molina, "Morsa: A scalable approach for persisting and accessing large models," in *Proc. of the 14th MoDELS Conference*, Wellington, New Zealand, 2011, pp. 77–92.

[21] A. Gómez, G. Sunyé, M. Tisi, and J. Cabot, "Map-based transparent persistence for very large models," in *Proc. of the 18th FASE Conference*. London, United Kingdom: Springer, 2015, pp. 19–34.

[22] G. Daniel, G. Sunyé, A. Benelallam, and M. Tisi, "Improving memory efficiency for processing large-scale models," in *Proc. of BigMDE'14*, York, United Kingdom, 2014, pp. 31–39.

[23] Tinkerpop, "Blueprints API," 2015. [Online]. Available: www.blueprints.tinkerpop.com

[24] ——, "The Gremlin Language," 2015. [Online]. Available: www.gremlin.tinkerpop.com

[25] F. Holzschuher and R. Peinl, "Performance of graph query languages: Comparison of cypher, gremlin and native access in neo4j," in *Proc. of the Joint EDBT/ICDT 2013 Workshops*, New York, NY, USA, 2013, pp. 195–204.

[26] J. Cabot and E. Teniente, "Transformation techniques for ocl constraints," *SCP*, vol. 68, no. 3, pp. 179 – 195, 2007, special Issue on Model Transformation.

[27] The Eclipse Project, "MDT OCL." [Online]. Available: www.eclipse.org/modeling/mdt/?project=ocl

[28] R. Wei and D. S. Kolovos, "An efficient computation strategy for allinstances ()," *BigMDE 2015*, p. 32, 2015.

[29] The Eclipse Foundation, "EMF Query," 2015. [Online]. Available: https://projects.eclipse.org/projects/modeling.emf.query

[30] G. Bergmann, Á. Horváth, I. Ráth, and D. Varró, "Efficient model transformations by combining pattern matching strategies," in *Proc. of the 2nd ICMT*, Zurich, Switzerland, 2009, pp. 20–34.

[31] J. Cabot and E. Teniente, "Constraint support in mda tools: A survey," in *Proc. of the 2nd ECMDA-FA*, 2006, vol. 4066, pp. 256–267.

[32] F. Heidenreich, C. Wende, and B. Demuth, "A framework for generating query language code from ocl invariants," *Electronic Communications of the EASST*, vol. 9, pp. 1–10, 2007.

[33] M. Brambilla and J. Cabot, "Constraint tuning and management for web applications," in *Proc. of the 6th ICWE*, New York, 2006, pp. 345–352.

[34] T. Halpin, M. Curland, K. Stirewalt, N. Viswanath, M. McGill, and S. Beck, "Mapping orm to datalog: An overview," in *On the Move to Meaningful Internet Systems: OTM 2010 Workshops*. Springer, 2010, pp. 504–513.

[35] X. D. Carlos, G. Sagardui, and S. Trujillo, "Mqt, an approach for runtime query translation: From EOL to SQL," in *Proc. of OCL 2014 co-located with MoDELS 2014*, Valencia, Spain, 2014, pp. 13–22.

[36] X. De Carlos, G. Sagardui, A. Murguzur, S. Trujillo, and X. Mendialdua, "Model query translator: A model-level query approach for large-scale models," in *2015 3rd International Conference on MODELSWARD*, Feb 2015, pp. 62–73.

[37] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The epsilon object language (eol)," in *Proc. of the 2nd ECMDA-FA*. Springer, 2006, pp. 128–142.

[38] J. Tolvanen and S. Kelly, "Metaedit+: defining and using integrated domain-specific modeling languages," in *Proc. of the 24th OOPSLA Conference*. ACM, 2009, pp. 819–820.

[39] M. Jarke, M. A. Jeusfeld, H. W. Nissen, C. Quix, and M. Staudt, *Object Databases: 2nd ICOODB Conference*. Berlin, Heidelberg: Springer, 2010, ch. Metamodelling with Datalog and Classes: ConceptBase at the Age of 21, pp. 95–112. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14681-7_6

[40] B. Morin, O. Barais, J. Jezequel, F. Fleurey, and A. Solberg, "Models@ run. time to support dynamic adaptation," *Computer*, vol. 42, no. 10, pp. 44–51, 2009.

[41] F. Büttner and M. Gogolla, "On ocl-based imperative languages," *SCP*, vol. 92, Part B, pp. 162-178, 2014.