Gremlin-ATL: A Scalable Model Transformation Framework

Gwendal Daniel
AtlanMod Team
Inria, IMT Atlantique, LS2N
gwendal.daniel@inria.fr

Frédéric Jouault
TRAME Team
Groupe ESEO
frederic.jouault@eseo.fr

Gerson Sunyé
AtlanMod Team
Inria, IMT Atlantique, LS2N
gerson.sunye@inria.fr

Jordi Cabot
ICREA
UOC
jordi.cabot@icrea.cat

***Abstract*—Industrial use of Model Driven Engineering techniques has emphasized the need for efficiently store, access, and transform very large models. While scalable persistence frameworks, typically based on some kind of NoSQL database, have been proposed to solve the model storage issue, the same level of performance improvement has not been achieved for the model transformation problem. Existing model transformation tools (such as the well-known ATL) often require the input models to be loaded in memory prior to the start of the transformation and are not optimized to benefit from lazy-loading mechanisms, mainly due to their dependency on current low-level APIs offered by the most popular modeling frameworks nowadays.**

In this paper we present Gremlin-ATL, a scalable and efficient model-to-model transformation framework that translates ATL transformations into Gremlin, a query language supported by several NoSQL databases. With Gremlin-ATL, the transformation is computed within the database itself, bypassing the modeling framework limitations and improving its performance both in terms of execution time and memory consumption. Tool support is available online.

***Index Terms*—ATL; Gremlin; OCL Scalability; Persistence Framework; model transformation; NoSQL**

I. Introduction

Models are used in various engineering fields as abstract views helping designers and developers understand, manipulate, and transform complex systems. They can be manually constructed using high-level modeling tools, such as civil engineering models [1], or automatically generated in model-reverse engineering processes such as software evolution tasks [2] or schema discovery from existing documents [3].

Models are then typically used in Model Driven Engineering (MDE) processes that rely intensively on model transformation engines to implement model manipulation operations like view extraction, formal verification or code-generation [4].

With the growing accessibility of big data (such as national open data programs [5]) as well as the progressive adoption of MDE techniques in the industry [6], [7], the volume and diversity of data to model has grown to such an extent that the scalability of existing technical solutions to store, query, and transform these models has become a major issue [8].

For example, reverse engineering tools such as MoDisco [2] rely on MDE technologies to extract a set of high-level models representing an existing code base. These models are then operated by model transformations to create a set of artifacts providing a better understanding of the system, such as UML [9] class diagrams, code documentation, or quality metrics. However, these tools typically face scalability issues when the input code base increases because the underlying modeling frameworks are not designed to store and transform large models efficiently.

Scalable modeling storage systems [10]–[12] have been proposed to tackle this issue, focusing on providing a solution to store and manipulate large models in a constrained memory environment with minimal performance impact. Relational and NoSQL databases are used to store models, and existing solutions rely on a *lazy-loading* mechanism to optimize memory consumption by loading only the accessed objects from the database.

While these systems have 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, transform, and update models. These APIs are focused on manipulating individual model elements and do not offer support for generic queries and transformations.

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. As shown in Figure 1, this is particularly true in the context of model transformations, which heavily rely on high-level model navigation queries (such as the `allInstances()` operation returning all instances of a given type) to retrieve input elements to transform and create the corresponding output elements. This mismatch between high-level modeling languages and low-level APIs generates a lot of fragmented queries that cannot be optimized and computed efficiently by the database [13].

To overcome this situation, we propose Gremlin-ATL, an alternative transformation approach. Instead of translating high-level model transformation specifications into a sequence of inefficient API calls, we translate them into database queries and execute them directly where the model resides, i.e. in the database store. This approach is based on a *Model Mapping* that allows to access an existing database using modeling

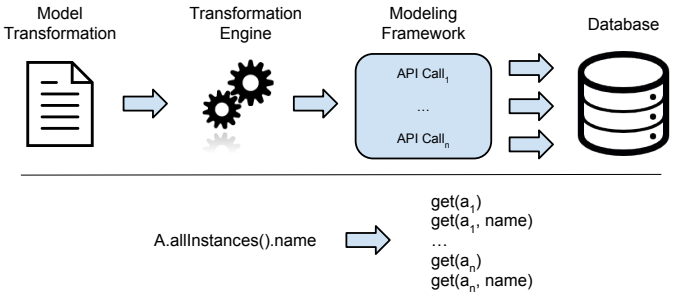


Fig. 1. Model Transformation Engine and Modeling Framework Integration

primitives, and on a *Transformation Helper* that lets modelers tune the transformation execution to fit their performance needs.

The rest of this paper is structured as follow: Section II introduces the input transformation language and the output database query language of our approach and presents a running example that will be used through the paper. Section III presents Gremlin-ATL and its key components, Section IV present how a transformation is executed from a user point of view. Sections V and VI present our prototype and the benchmarks used to evaluate our solution. Section VII shows an application example where Gremlin-ATL is used to specify data extraction rules between two data sources. Finally, Section VIII presents the related work and Section IX summarizes the key points of the paper, draws conclusion, and presents our future work.

II. Background

In this section we introduce the key features of ATL [4], the model transformation language we use as the input of our framework, and Gremlin [14], a multi-database graph traversal query language we use as our output language. We also present along the section a running example that is used through this article to illustrate the different steps of our approach.

A. ATL Transformation Language

In MDE, models are key elements in any software engineering activity. Models are manipulated and refined using model-to-model transformations, and the final software artifacts are (partially) generated with a model-to-text transformation. Each model conforms to a *metamodel* that describes its structure and the possible relationships between model elements and their properties.

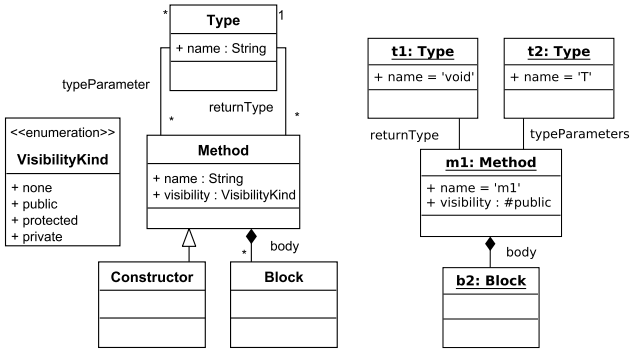
As an example, Figure 2(a) shows a simple metamodel representing *Types*, *Methods*, and *Blocks*. *Methods* are defined by a *name*, a *visibility*, and contain a set of *Blocks* representing their *body*. A *Method* is associated with a *return Type*, and can have *type parameters*. A *Constructor* is defined as a subclass of *Method*. An instance of this metamodel is shown in Figure 2(b).

ATL [4] is a declarative rule-based model transformation language that operates at the metamodel level. Transformations written in ATL are organized in modules, that are used to group transformation rules and define libraries. The language defines three types of rules: (i) matched rules that are declarative and automatically executed, (ii) lazy rules that have to be invoked explicitly from another rule, and (iii) called rules which contain imperative code[1]. ATL does not assume any order between matched rules, and keeps a set of *trace links* between the source and target models that are used to resolve elements, and set target values that have not been transformed yet.

The language also defines helpers expressed in OCL [15] that are used to compute information in a specific context, provide global functions, and runtime attributes computed on the fly. OCL helpers can be invoked multiple times in a transformation, and are a good solution to modularize similar navigation chains and condition checking.

Finally, ATL programs are themselves described as models that conform to the ATL metamodel. This feature allows to define high-order transformations, that take an ATL transformation as their input and manipulate it to check invariant properties that should hold in the output model, infer type, or refine the transformation. Our approach relies on this reflective feature to translate ATL transformations into efficient database queries.

Note that in this paper we focus on ATL as our input language, but our approach can be adapted to other rule-based transformation languages, notably the QVT [16] standard.



(a) Metamodel (b) Instance Model

Fig. 2. Example Metamodel and Model

Listing 1 shows a simple ATL transformation based on the metamodel shown in Figure 2(a). The rule *MethodToMethodUnit* (line 10) matches all the *Method* elements from the input model that do not contain a *typeParameter* and creates the corresponding *MethodUnit* in the target model. The attributes and references of the created element are set using *binding specifications*: the first one (line 13) checks if the source element is a *Method* or a *Constructor* and sets the *kind* attribute accordingly. The second binding (line 15) sets the value of the *export* attribute by calling the OCL helper *getVisibility* on the source element. Finally, the *codeElement* reference is set with the *Block* elements contained in the *body* reference of the source *Method*. Note that an additional rule has to be specified

[1]Imperative constructs are not discussed in this paper

to map *Block* instances to their corresponding output elements, that will be resolved by the ATL engine using its *trace links* mechanism.

```
-- returns a String representing the visibility of a
     Method
helper context java!Method def: getVisibility():String=
  let result : VisibilityKind = self.visibility in
    if result.oclIsUndefined() then
      "unknown"
    else
      result.toString()
    endif;
-- Transforms a Java Method into a KDM Method unit
rule MethodToMethodUnit {
  from src : java!Method(src.typeParameters.isEmpty())
  to tgt : kdm!MethodUnit(
    kind <- if (src.oclIsKindOf(java!Constructor))
              then 'constructor' else 'method' endif,
    export <- src.getVisibility(),
    codeElement <- src.body->asSet() }
```

Listing 1. Simplified Java2KDM Rule Example

B. Gremlin Query Language

NoSQL databases are an efficient option to store large models [17], [18]. 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 [19] 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[2].

Blueprints is the base of the Tinkerpop stack: a set of tools to store, serialize, manipulate, and query graph databases. Gremlin [14] 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.

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 Gremlin terminology, these complex processing vertices are called *traversals*, and are composed of a chain of simple computational units named *steps*. Gremlin defines three types of steps: (i) *transform steps* that compute values from their inputs, (ii) *filter steps* that select or rejects elements w.r.t a given condition, and (iii) *side-effect steps* that compute side-effect operations such as vertex creation or property updates. In addition, the *step* interface provides a set of built-in methods to access meta information, such as the number of objects in a step, output existence, or the 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.

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

Fig. 3. Persisted Model

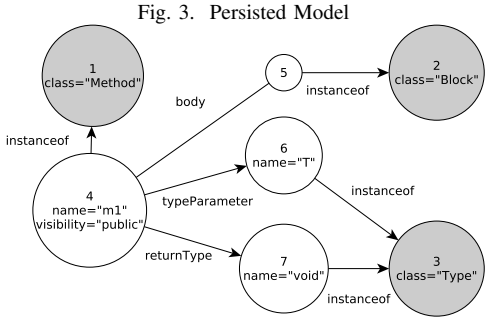


Figure 3 presents a possible graph database representation of the model shown in Figure 2(b): grey vertices represent *Method*, *Type*, and *Block* metaclasses that are linked to their instance through *instanceof* edges. The *Method* `m1` is linked to a *Block* instance through the *body* edge, and the *Types* `void` and `T` using *returnType* and *typeParameters* edges, respectively. Finally, `m1` contains a property `visibility` that holds a string representation of the *VisibilityKind* enumeration. This graph mapping of metamodel instances is based on the one used in NeoEMF/Graph [20].

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 information 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 *Method*, representing the *Method* class in the Figure 2(a). In our example, this class matches vertex 1.

```
g.idx("classes")[[name:"Method"]]; // -> v(1)
```

The most common steps are *transform* steps, which 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 4 by the relationship *typeParameters*. The *Start* step `g.v(4)` is a vertex lookup that returns the vertex with the id 4.

```
g.v(4).outE("typeParameters").inV; // -> [v(6)]
```

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 4 by the relationship *typeParameters* that have a property *name* containing at least one character.

```
g.v(4).outE("typeParameters").inV
.has("name").filter{it.name.length > 0}; // -> [v(6)]
```

Finally, *side-effect* steps modify a graph, compute a value, or assign variables in a traversal. They are used to fill collections

with traversal 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("typeParameters").inV.has("name")
.filter{it.name.length > 0}.fill(table); // -> [v(6)]
```

III. Gremlin-ATL Framework

In this section we present Gremlin-ATL, our proposal for handling complex model-to-model transformations by taking advantage of the features of the database backends where the model resides. We first introduce an overview of the framework and its query translation process. We then present the ATLtoGremlin transformation that maps ATL transformations into Gremlin traversals. Finally, we present the auxiliary components used within the Gremlin query to specify the database on which to compute the query on and the specific configuration to use.

A. Overview

Figure 4 presents an overview of the translation used in Gremlin-ATL to create *Gremlin Traversals* from *ATL Transformations*. An input transformation is parsed into an *ATL Transformation Model* conforming to the *ATL metamodel*. This model constitutes the input of our *ATLtoGremlin* high-order transformation that creates an output *Gremlin Traversal* representing the query to compute.

The *ATLtoGremlin* transformation uses two generic libraries to produce the output query: (i) a *Model Mapping Definition* allowing to access a database as a model by mapping its implicit schema to modeling primitives, and (ii) a *Transformation Helper Definition* used to tune the transformation algorithm according to memory and execution time requirements.

The generated traversal can be returned to the modeler or directly computed in a Gremlin engine that manages the underlying database. Note that our approach also allows to compute directly the generated query in a preset NeoEMF database, as explained in Section V.

In the following we detail the *ATLtoGremlin* transformation, the *Model Mapping* and *Transformation Helper Definition* used in the generated traversal. A dynamic view of our approach is provided in Section IV.

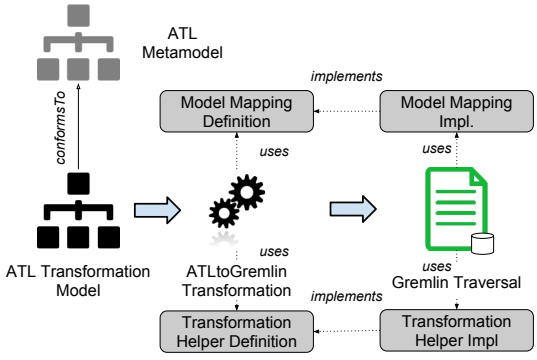


Fig. 4. Overview of the Mogwaï-ATL Framework

B. ATLtoGremlin Transformation

1) Transformation Mapping: Table I shows the mapping used by Gremlin-ATL to translate ATL constructs into Gremlin steps. An ATL *Module* is translated into a Gremlin script, that represents the top-level container storing the entire query to execute.

Matched Rule Definitions inside the ATL module are mapped to a sequence of steps that access all the elements of the type of the rule. For example, the matched rule definition *MethodToMethodUnit* in Listing 1 is translated into the Gremlin expression `g.allOfKind("Method")` that searches in the input graph all the elements representing *Method* instances. *Abstract Rule Definitions* are not translated, because they are called only when a specialized rule is matched. *Lazy rule definitions* are translated into function definitions named with the rule's identifier and containing their translated body.

Matched rule bodies are mapped to a transform step containing the translated expressions representing rule's out pattern and bindings. This transform step is followed by an iterate step that tells the Gremlin engine to execute the query. *Abstract rule bodies* are directly mapped without creating a transform step, and generated Gremlin steps are added to the ones of the translated bodies of the corresponding *specialized rules*. This approach flattens the inheritance hierarchy of a transformation by duplicating parent code in each concrete sub-rule.

Rule Guards defining the set of elements matched by a rule are translated into a Gremlin filter step containing the translated condition to verify. For example, the guard of the rule *MethodToMethodUnit* is translated into the following Gremlin expression that first navigates the reference *typeParameters* and searches if it contains at least one value: `filter{!(src.getRef("typeParameters").hasNext())}`.

Rules' body can contain two types of ATL constructs: *out patterns* representing the target element to create, and *attribute/reference bindings* describing the attribute and references to set on the created element. *Out patterns* are mapped to a variable definition storing the result of the *createElement* function which creates the new instance and the associated trace links. This instruction is followed by a *resolveTraces* call that tells the engine to resolve the potential trace links associated to the created element. In our example it generates the sequence `tHelper.createElement("MethodUnit", src)` that creates a new *MethodUnit* instance in the target model and associates it with the *src* element from the source model. *Attribute* and *Reference Bindings* are respectively translated into the mapping operation *setAtt* and a Transformation Helper's *link* call.

Our mapping translates *helper definitions* into global methods, which define a *self* parameter representing the context of the helper and a list of optional parameters. This global function is dynamically added to the *Object* metaclass to allow method-like invocation, improving query readability. *Global helper definitions* are also mapped to global methods, but do not define a *self* parameter. Finally, *Global Variables* are translated into unscoped Gremlin variables.

ATL embeds its own specification of OCL, which is used to navigate the source elements to find the objects to transform, express the guard condition of the transformation rules, and define helpers' body. We have adapted the mapping defined in the Mogwaï framework [21] to fit the OCL metamodel embedded in ATL. In addition, we integrated our *Model Mapping Definition* component in the translation in order to provide a generic translation based on an explicit mapping. The current version of Gremlin-ATL supports an important part of the OCL constructs, allowing to express complex navigation queries over models. As an example, the inline *if* construct used in *MethodToMethodUnit* to check whether the source element represents a constructor can be translated into the equivalent Gremlin ternary operator: `src.isKindOf("Constructor")? "constructor": "method"`.

TABLE I
ATL TO GREMLIN MAPPING

ATL expression	Gremlin step
module	Gremlin Script
matched_rule definition	g.allOfType(type)
abstract_rule definition	not mapped
lazy_rule definition	def name(type) { body }
matched_rule body	transform{ body }.iterate()
abstract_rule body	body[3]
specialized_rule body	transform{ body ∪ parent.body }.iterate()
rule_guard(condition)	filter{condition}
out_pattern(srcEl, tgtEl)	var out = thelper .createElement(tgtEl.type, srcEl) tHelper.resolveTraces(srcEl,tgtEl);
attribute_binding	e1.setAtt(exp)
reference_binding	e1.link(exp)
helper_definition	def name(var self, params){ expression } Object.metaClass.name = { (params) -> name(delegate, params)}
obj.helper(params)	obj.helper(params)
global_helper_definition	def name(params) { expression }
global_helper_computation	name(params);
global_variable	def name = expression
OCL_Expression	Mogwaï[4]

2) Operation Composition: The above mappings explain how ATL constructs are mapped individually into the corresponding Gremlin steps. In this section we present an overview of the algorithm used to compose these generated steps into a complete query. Listing 2 shows the final Gremlin output for the transformation example shown in Listing 1.

In order to generate a complete Gremlin query, our transformation has to navigate the input ATL model and link the generated elements together. First, the transformation searches all the *helper* definitions (including global ones) and translates them according to the mapping shown in Table I. The generated functions are added to the Gremlin script container, making them visible for the translated ATL rules. This first step generates the lines 1 to 8 for the helpers *getVisibility*. Note that this initial phase also generates the function that registers contextual helpers to the *Object* metaclass (lines 10-13), allowing method-like invocation in generated expressions.

Lazy_rule definitions are then translated into global functions, and added to the Gremlin script container. *Out pattern* and *bindings* contained in the *Lazy_rule body* are translated into their Gremlin equivalent following the mapping presented in the previous section and appended in the generated function body.

```
1  // getVisibility() helper
2  def getVisibility(var vertex) {
3    var result = vertex.getAtt("visibility");
4    if(result == null)
5      return "unknown";
6    else
7      return result;
8  }
9
10 // Add getVisibility to Vertex method list
11 Vertex.metaClass.getVisibility =
12   { -> getVisibility(delegate) }
13
14 // MethodToMethodUnit
15 g.allOfKind(Method).filter{
16   def src = it;
17   !(src.getRef("typeParameters").hasNext())
18 }.transform{
19   def src = it;
20   var tgt = tHelper.createElement("MethodUnit", src);
21   tHelper.resolveTraces(src, tgt);
22   tgt.setAtt("kind", src.isKindOf("Constructor") ? "
         constructor" : "method");
23   tgt.setAtt("export", src.getVisibility());
24   tgt.setRef("codeElement", src.getRef("body").toList
         () as Set);
25 }.iterate();
```

Listing 2. Generated Gremlin Traversal

Once this initial step has been performed the transformation searches all the *matched rules definitions* and creates the associated Gremlin instructions. If the rule defines a guard the generated *filter* step is directly linked after the *allOfKind* operation in order to select the elements that satisfy the guard's condition. Then, the *transform step* (and the associated *iterate* call) corresponding to the *matched rule body* is created and added at the end of the traversal. In our example this operation generates lines 15 to 18.

Out pattern elements contained in the *matched rule body* are retrieved and the corresponding Gremlin instructions (lines 19 to 24) are added to the *transform* step closure. Finally, Rule's *bindings* are transformed following the mapping and added in the closure's instructions. Note that helper calls inside binding's expressions are also translated into the corresponding method calls during this operation.

C. Auxiliary Components

1) Model Mapping: The *Model Mapping* library defines the basic modeling operations that can be computed by a given database storing a model. It provides a simple API allowing designers to express the implicit schema of their database with modeling primitives. *Model Mapping* can be manually implemented for a given database, or automatically extracted using schema inference techniques such as NoSQLDataEngineering [22]. This mapping is used within the generated Gremlin query to access all the elements of a given type, retrieve element's attribute values, or navigate references.

[3]The body of abstract rules is duplicated in the transform step of all its sub-rules

[4]OCL expression are translated by an improved version of the Mogwaï framework

Table II summarizes the mapping operations used in Gremlin-ATL and groups them into two categories: **metamodel-level** and **model-level operations**. The first group provides high-level operations that operate at the metamodel level, such as retrieving the type of an element, type conformance checks, new instances creation, and retrieve all the elements conforming to a type. The second group provides methods that compute element-based navigation, such as retrieving a referenced element, computing the parent/children of an element, and access its attributes. Finally, these model-level methods allow to update and delete existing references and attributes.

Note that the *ATLtoGremlin* transformation only relies on the definition of these operations to generate a Gremlin query, and is not tailored to a specific *Model Mapping* implementation.

TABLE II
GRAPH MAPPING API

Operation	Description
allOfType(type)	Returns all the strict instances of the given *type*
allOfKind(type)	Returns all the instances of the given type or one of its sub-types
getType(el)	Returns the type of the element *el*
isTypeOf(el, type)	Computes whether *el* is a strict instance of *type*
isKindOf(el, type)	Computes whether *el* is an instance of *type* or one of its sub-types
newInstance(type)	Creates a new instance of the given *type*
getParent(el)	Returns the element corresponding to the parent of *el*
getChildren(el)	Returns the elements corresponding to the children of *el*
getRef(from, ref)	Returns the elements connected to *from* with the reference *ref*
setRef(from, ref, to)	Creates a reference *ref* between *from* and *to*
delRef(from, ref, to)	Deletes the reference *ref* between *from* and *to*
getAtt(from, att)	Returns the value of the attribute *att* contained in the element *from*
setAtt(from, att, v)	Set the value of the attribute *att* of the element *from* to the given value *v*
delAtt(from, att)	Deletes the attribute *att* contained in the element *from*

2) Transformation Helper: The second component used by the *ATLtoGremlin* transformation is a *Transformation Helper* library that provides transformation-related operations that can be called within the generated Gremlin traversal. It is based on the ATL execution engine (presented in Section II-A), and provides a set of methods wrapping a *Model Mapping* with transformation specific operations. Note that this library aims to be generic and can be used directly in ad hoc Gremlin scripts to express transformation rules.

Specifically, our *TransformationHelper* provides the following interface:

- **createElement(type, source)**: creates a new instance of the given *type* mapped to the provided *source* element.
- **link(from, ref, to)**: creates a link between *from* and *to*.
- **resolveTraces(source, target)**: resolves the trace links connected to *source* with the *target*.
- **getTarget(source, bindingName)**: retrieves the target element mapped to *source*. If multiple target elements are created from a single source an optional. *bindingName* can be specified to tell the engine which one to choose.
- **isResolvable(el)**: computes whether an element can be resolved.
- **isTarget(el)**: computes whether an element is in the target model.
- **isSource(el)**: computes whether an element is in the source model.

The two first operations are wrappers around mapping operations that add model transformation specific behavior to the mapping. The `createElement(type, sourceElement)` operation delegates to the mapping operation `newInstance(type)`, and adds a trace link between the created element and the source one. `link(from, ref, to)` delegates to the mapping operation *setRef* to create a regular reference between from and to or a trace link if *to* is not yet part of the target model.

The five last operations define transformation specific behaviors, such as retrieving the target element from an input one, resolve trace links, or compute whether an element is part of the source of the target model. Note that Gremlin-ATL provides two implementations (one in memory and one in a database) of this library that can be directly plugged to compute a transformation.

IV. RUNTIME FRAMEWORK

The Gremlin script generated by the *ATLtoGremlin* transformation is generic and relies on the *Model Mapping* and *Transformation Helper* interfaces. In order to execute it on an existing database, the framework needs to bind these abstract interfaces to their concrete implementations designed to access a specific database.

Figure 5 shows our framework from a user point of view: the user provides an ATL query to the *Query Translation* component (1), which compiles the query into a Gremlin script and returns it to the user (2). The generated script is then sent to the *Query Execution* API with the concrete implementation of the *Model Mapping* and a *Transformation Helper* (3) the user wants to use during the execution. The internal Gremlin engine finally computes the transformation using these implementations (4) and returns the result (such as execution traces, or transformation errors if any) to the user (5).

This architecture allows to pre-compile transformations into Gremlin scripts and execute them later with specific implementations of the *Model Mapping*. In addition, generated scripts can be executed multiple times on different mappings without recompilation. The framework can also process and compile new input transformations on the fly, generating the corresponding Gremlin scripts dynamically.

As an example, we can define a possible *Model Mapping* implementation for the persisted model of Figure 3 that would compute the `allOfType(Type)` operation with an access to the vertex representing the metaclass *Type* and a backward navigation of its incoming *instanceof* edges. The `getAtt` and `getRef` operations can be translated into property access

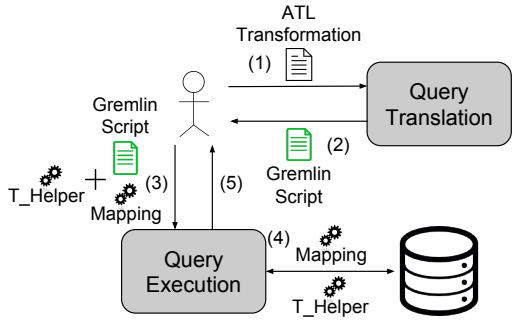


Fig. 5. User Point of View

and edge navigation, respectively, and the `newInstance` operation can be computed by adding a new vertex in the database with an outgoing edge linked to the vertex representing its metaclass. Note that Gremlin-ATL embeds preset *Model Mapping* implementations for accessing NeoEMF, Neo4j, and relational databases.

Once the *Model Mapping* interface is bounded to its implementation, the modeler can choose the *Transformation Helper* he needs to compute the transformation. The choice of a specific implementation is a trade-off between execution time and memory consumption: a *Transformation Helper* storing transformation information (such as trace links) in memory will be efficient when computing transformations on top of relatively small models, but will not scale to larger ones. Conversely, an implementation relying on a dedicated database can scale to larger models but would not be optimal when using small models. Gremlin-ATL defines two default implementations (one in memory and one in database) that can be directly plugged to compute a transformation and extended to provide further performances.

V. Tool Support

Gremlin-ATL is released as a set of open source Eclipse plugins publicly available on GitHub.[5]

Input ATL transformations are parsed using the standard ATL parser, which creates a model representing the abstract syntax of the transformation. The generated model is sent to the *ATLtoGremlin* ATL transformation, which contains around 120 rules and helpers implementing the translation presented in Section III. Once the Gremlin model has been generated, it is transformed into a textual Gremlin query using a model-to-text transformation. A final step binds the *Model Mapping* and the *Transformation Helper* definitions to their concrete implementation and the resulting script is then sent to a Gremlin script engine, which is responsible of computing the query on top of the database. The updated database is finally saved with the transformed content.

Additionally, a preconfigured implementation of Gremlin-ATL is bundled with the NeoEMF model persistence framework that extends the standard *Resource* API with transformation operations. ATL transformations are computed using the Gremlin-ATL engine transparently, on top of a preset NeoEMF *Model Mapping* implementation. The resulting model is compatible with NeoEMF and its content can be reified and manipulated using the standard modeling API if needed. This transparent integration allows the use of Gremlin-ATL for critical model transformations, while keeping the rest of the application code unchanged. We believe that this approach can ease the integration of Gremlin-ATL into existing modeling applications that have to compute complex transformations on the top of large models.

The current implementation embeds a set of predefined *Model Mappings* and *Transformation Helpers*. Both can easily be adapted to specify how you want transformation-related operations to be computed. This is done by extending a set of abstract classes with limited implementation effort. As an example, the model mapping used to access Neo4j databases storing NeoEMF models defines 23 methods (around 250 LOC), and the in-database *Transformation Helper* defines 9 methods (around 100 LOC) implementing the transformation operations and storing intermediate results in the database.

VI. Evaluation

In this section we evaluate the performance of the Gremlin-ATL framework by comparing the execution performance of the same set of ATL transformations using the standard ATL engine and the Gremlin-ATL framework. Note that we do not consider alternative transformation frameworks in this evaluation, because they are either based on the same low-level modeling API as ATL (such as QVT [16]), or are not designed to optimize the same transformation scenario (such as Viatra [23]).

Our evaluation aims to address the following research questions: (i) is Gremlin-ATL faster than standard ATL when applied to large models stored in scalable persistence frameworks? and (ii) does our approach scales better in terms of memory consumption?

In the following we benchmark our approach with two transformations, a toy one created on purpose for this analysis and a more complex one taken from an industrial project involving a reverse engineering scenario. Transformations are evaluated on a set of models of increasing size, stored in Neo4j using the NeoEMF mapping. Transformed models are also stored in NeoEMF using the same mapping. Experiments are executed on a computer running Fedora 20 64 bits. Relevant hardware elements are: an Intel Core I7 processor (2.7GHz), 16GB of DDR3 SDRAM (1600MHz) and a SSD hard-disk. Experiments are executed on Eclipse 4.6.0 (Neon) running Java SE Runtime Environment 1.8.

A. Benchmark Presentation

Experiments are run over five models (sets 1 to 5) of increasing sizes, automatically constructed by the MoDisco [2] Java Discoverer, a reverse engineering tool that extracts low-level models from Java code. The first two models are generated from internal projects containing few dozens of classes, and the larger ones are computed from the MoDisco framework

[5]https://github.com/atlanmod/Mogwai

itself, and the Java Development Tool (JDT) plugins embedded in Eclipse. Table III presents the size of the input models in terms of number of elements and XMI file size. These models are migrated to NeoEMF/Graph before executing the benchmark to enable scalable access to their contents.

TABLE III
BENCHMARKED MODELS

Model	# Elements	XMI Size (MB)
set1	659	0.1
set2	6756	1.7
set3	80665	20.2
set4	1557007	420.6
set5	3609354	983.7

In order to evaluate our approach, we perform two transformations that take as input the Java models conforming to MoDisco's Java metamodel, and translate them into models conforming to the Knowledge Discovery Model (KDM) [24] that is a pivot metamodel used to represent software artifact independently of their platform.

The *AbstractTypeDeclaration2DataType* transformation matches all the *AbstractTypeDeclaration* elements (declared classes, interfaces, enumerations, etc.) of the input model and create the corresponding KDM *DataType*. It is composed of a single rule that matches all the subtypes of *AbstractTypeDeclaration*. The second benchmarked transformation is a subset of the *Java2KDM* transformation. It is extracted from an existing industrial transformation that takes a low-level Java model and creates the corresponding abstract KDM model. The transformations are run in a 512MB JVM with the arguments `-server` and `UseConcMarkSweepGC` that are recommended by Neo4j.

B. Results

Tables IV and V present the results of executing the transformations on top of the input model sets. First columns contain the name of the input model of the transformation, second and third columns present the execution time and the memory consumption of ATL engine and Gremlin-ATL, respectively. Execution times are expressed in milliseconds and memory consumption in megabytes.

The correctness of the output models are checked by comparing the results of our approach with the ones generated by running the ATL transformation on the original input XMI file using a large Java virtual machine able to handle it. The comparison is performed using EMF Compare [25].

TABLE IV
ABSTRACTTYPEDECLARATION2DATATYPE RESULTS

Model	Execution Time (ms)		Memory Consumption (MB)	
	ATL	Gremlin-ATL	ATL	Gremlin-ATL
set1	1633	710	2.0	8.3
set2	3505	1139	3.2	10.0
set3	11480	1649	17.6	11.7
set4	67204	3427	99.3	23.0
set5	*OOM*[6]	11843	*OOM*	100.0

TABLE V
JAVA2KDM RESULTS

Model	Execution Time (ms)		Memory Consumption (MB)	
	ATL	Gremlin-ATL	ATL	Gremlin-ATL
set1	1735	1341	3.2	11.2
set2	4874	2469	11.0	12.8
set3	33407	4321	45.2	23.2
set4	5156798	38402	504.5	52.0
set5	*OOM*	129908	*OOM*	96.0

C. Discussion

The results presented in Tables IV and V show that Gremlin-ATL is a good candidate to compute model transformations on top of large models stored in NeoEMF/Graph. Our framework is faster than the standard ATL engine for the two benchmarked transformations, outperforming it both in terms of execution time and memory consumption for the larger models.

Results from Tables IV and V show that the complexity of the transformation has a significant impact on ATL's performances. This is particularly visible when the transformations are evaluated on the large models: *Java2KDM* is 2.9 times slower than *AbstractType2DataType* on *set3*, and up to 76 times on *set4*. This difference is explained by the large number of low-level modeling API calls that are generated by *Java2KDM* in order to retrieve the matching elements, computes rules' conditions, and helpers.

Gremlin-ATL's execution time is less impacted by the transformation complexity, because the generated Gremlin query is entirely computed by the database, bypassing the modeling API. The database engine optimizes the query to detect access patterns and cache elements efficiently, and allows to benefit from the database indexes to retrieve elements efficiently. Results on *set4* show that Gremlin-ATL's approach is faster than ATL by a factor of 19 and 134 for *AbstractType2DataType* and *Java2KDM*, respectively.

The results also show that ATL's performance on *set4* and *set5* are tightly coupled to the memory consumption. Indeed, in our experiments we measured that most of the execution time was spent in garbage collection operations. This high memory consumption is caused by the in-memory nature of the engine, that keeps in memory all the matched elements as well as the trace links between source and target models. When the input model grows, this in-memory information grows accordingly, triggering the garbage collector, which blocks the application until enough memory has been freed. In addition, the intensive usage of the low-level model handling API increases the memory overhead, by reifying intermediate modeling elements that also stay in the memory.

Conversely, Gremlin-ATL does not require these complex structures, because trace links are stored within the database itself, and can be removed from the memory if needed. This implementation avoids garbage collection pauses, and allows to scale to very large models. Our approach operates on

[6]The application threw an OutOfMemory error after two hours

the optimized database structures, and thus does not have to reify modeling elements, improving the memory consumption. Looking at the *Java2KDM* transformation, this strategy reduces the memory consumption by a factor of 10 for *set4*.

ATL requires less memory than Gremlin-ATL to compute the transformations on top of the small models (*sets 1* and *2*). This difference is caused by the initialization of the Gremlin engine (and its underlying Groovy interpreter) used to evaluate the generated scripts. However, this extra memory-consumption has a fixed size and does not depend on the evaluated transformation nor on the transformed model.

Note that the results only show execution time and memory consumption related to the computation of the transformations, we did not take into account the time and memory required to generate an executable file that can be interpreted by the ATL engine nor the time needed to produce the Gremlin script to compute, because this extra cost is fixed for a given transformation and does not depend on the model size. In addition, Gremlin-ATL allows to pre-compile and to cache existing Gremlin queries in order to limit script generation.

As a summary, our experiments report that using Gremlin-ATL to compute a well-known model transformation outperforms the standard ATL engine when applied on top of large models stored in current model persistence frameworks. Still, additional experiments could reinforce these conclusions. We plan to extend this preliminary evaluation by reusing transformations available in the ATL Zoo[7].

VII. An Application to Data Migration

We have seen how Gremlin-ATL can improve performance of transformations for large models. In this section, we show how Gremlin-ATL can also be useful in the context of a data migration process when the schema of the data is unavailable (the common scenario in many NoSQL databases that are schemaless). To do so, we implement a simple migration operation on top of two data sources, and emphasize that using ATL as a migration language shortens the amount of required code to write for the migration with a limited impact on the application performance.

In this example, the input of our process is the open source ICIJ Offshore Leaks Database[8] that contains the results of the Panama papers investigations stored in a Neo4j database. The output of the process is a simple H2[9] relational database containing the migrated data.

The input database contains one million elements divided into four categories represented as node's labels: *Officers*, *Entities*, *Addresses*, and *Intermediates*. Connections between elements are represented by labeled edges, for example the shareholder of an entity is represented by an edge with the label `SHAREHOLDER_OF` between the corresponding *Officer* and *Entity* nodes. Other edges are used to represent the different roles of an *Officer*, the connections between *Entities* and *Intermediates*, and their *Addresses*. Finally, node's properties are used to represent additional information, such as the different fields of an *Address* record.

The operation to perform is a simple data extraction task that retrieves from the input database all the *Officer* elements that are shareholders of a company. Results are stored in a relational table containing the *Officer* names and the name of the corresponding *Entity*.

We define a Neo4j mapping allowing to navigate the input database as a model: `allOfType(type)` operations are computed by searching for the nodes labeled by *type*, and `newElement(type)` instructions are mapped to a node creation with a label representing the type. We also define `getRef(from, ref)` navigation operations as edge navigations, and `getAtt(from, att)` property accesses on the node representing *from*. Note that this mapping does not contain any information specific to the ICIJ Offshore Leaks Database, and can be reused on top of any Neo4j database using the same data representation.

To let the framework store the results in the desired relational database we defined an additional mapping that is able to access a relational database and serialize model elements in tables representing their types. Attributes are mapped to table columns, and references to foreign keys. Note that due to the lack of space we do not detail this mapping, but the complete definition is available on the project repository.

Listing 3 shows the ATL transformation rule that expresses our operation: it matches all the *Officer* entities that corresponds to *Entity* shareholders in the input database and creates a corresponding *CompanyShareholder* record in the output model containing the names of the *Officer* and the related *Entity*. Listing 4 presents the corresponding Java code that uses the Neo4j and JDBC APIs to manipulate the databases.

When the transformation is computed, source and target mappings are used to execute modeling operations defined in the ATL transformation. For example, the navigation of the `SHAREHOLDER_OF` reference is delegated to the Neo4j mapping, while the creation of the *CompanyShareholder* instance is computed by the relational mapping that creates a new table *CompanyShareholder* if it does not exist and inserts a new record in it containing the provided information.

As illustrated by the provided examples, the ATL program is around two times shorter than the corresponding Java implementation. In addition, the ATL transformation only relies on ATL language constructs, and does not contain any explicit database specific operation. We believe that using ATL as a common language to perform data migration is interesting when modelers have to integrate data from heterogeneous sources, and do not have the expertise to write efficient queries both in the input and output database languages.

The execution of this data migration operation creates 296041 records in the output database. The Java implementation is computed within 10342 ms, and the Gremlin-ATL one in 11006 ms. This execution time increase of 6.4% is caused by the underlying Gremlin engine that adds a small execution overhead, and our modeling layer that wraps the native APIs. Still, we think that the gains in terms of code

[7]https://www.eclipse.org/atl/atlTransformations/

[8]Available online at https://offshoreleaks.icij.org/pages/database

[9]http://www.h2database.com

readability and maintainability can balance this slight overhead increase and constitute an interesting solution to integrate data from multiple sources.

```
rule shareholder2relationalOfficer {
  from s : IN!Officer (not(s.name.oclIsUndefined()
           and s.SHAREHOLDER_OF->isEmpty())
  to t : OUT!CompanyShareholder(
      name <- s.name,
      company <- s.SHAREHOLDER_OF.name) }
```

Listing 3. Company Shareholder Migration ATL Rule

```
Connection c = ds.getConnection();
c.createStatement().execute("create␣table␣if␣not␣exists
 ␣CompanyShareholder␣(id␣integer␣not␣null␣
 auto_increment,␣name␣varchar(200),␣company␣varchar
 (200),␣primary␣key␣(id));");
c.commit();
try (Transaction tx = graphdb.beginTx()) {
  try (ResourceIterator<Node> officers = graphdb.
    findNodes(Label.label("Officer"))) {
    while (officers.hasNext()) {
      Node off = officers.next();
      Iterable<Relationship> rels = off.
        getRelationships(Direction.OUTGOING,
        RelationshipType.withName("SHAREHOLDER_OF"));
      for (Relationship r : rels) {
        Node head = r.getEndNode();
        if(off.hasProperty("name") && head.hasProperty(
          "name")) {
          String officerName = (String)off.getProperty(
            "name");
          String companyName = (String)head.getProperty
            ("name");
          c.createStatement().execute("insert␣into␣
            CompanyShareholder␣values␣(NULL,␣'" +
            officerName +"',␣'" + companyName + "');");
    }}}
    officers.close();}}
```

Listing 4. Company Shareholder Migration Java

VIII. Related Work

Several solutions have proposed to parallelize and distribute model transformations to improve the efficiency and scalability of existing transformation engines. ATL-MR [26] is a map-reduce based implementation of the ATL engine that computes transformations on top of models stored in HBase. The tool benefits from the distributed nature of the database to compute ATL rules in parallel, improving the overall execution time. Parallel-ATL [27] is an implementation of the ATL engine able to benefit from a multicore environment by splitting the execution into several workers that access a global shared model asynchronously. The LinTra [28] platform is another solution that relies on the Linda coordination model to enable concurrency and distribution of model transformations.

Compared to these approaches, Gremlin-ATL does not require a parallel infrastructure to optimize the transformation but instead relies on the scalability mechanisms of the database engine, one of the strong points and initial motivations for the apparition of NoSQL databases. For example, the distributed Neo4j database provides a Gremlin endpoint that is able to execute efficiently traversals on graph databases stored in a cluster, by splitting the computation according to the data localization. The Gremlin language itself provides native parallization constructs (e. g., the *split-merge* step) that could be used to further improve transformation execution performance.

The Viatra project [23] is an event-driven and reactive model transformation platform that relies on an incremental pattern matching language to access and transform models. Viatra receives model update notifications and uses its incremental engine to re-compute queries and transformation in an efficient way at the cost of a higher memory consumption. Viatra and Gremlin-ATL work best in different scenarios. Viatra is very efficient when a set of query/transformations are executed multiple times on a model, while Gremlin-ATL is designed to efficiently perform single transformation computations.

Transforming large models stored in databases is also related to the schema matching and data migration problems targeted in the database community [29]. Several approaches such as COMA [30] or Cupid [31] have been proposed to detect equivalent constructs in data schemas (using schema information or instances analysis), and integrated into data migration frameworks [32] to semi-automate data migration between heterogeneous sources. While Gremlin-ATL could benefit from these approaches (e. g., by automatically constructing *Model Mappings*), they usually focus on the schema matching phase and not on the efficient execution of the data migration processes based on those mappings.

Our approach can also be combined with proposals on efficient graph transformation computation, such as the approach proposed by Krause et al. that parallelizes graph transformations using the Bulk Synchronous Parallel (BSP) framework Apache Giraph [33]. The Gremlin language provides a connector to enable Giraph computation on top of graph databases[10], that could be used to link both approaches.

IX. Conclusion

In this article we presented Gremlin-ATL, a framework that computes rule-based transformations by reexpressing them as database queries written in the Gremlin graph traversal language. Our evaluation shows that using Gremlin-ATL to transform models significantly improves the performance both in terms of execution time and memory consumption.

As future work we plan to extend our approach into a family of mappings adapted to different NoSQL backends in order to provide a set of NoSQL modeling frameworks able to both store and manipulate models “natively”. We also plan to complete our mapping with the ATL constructs that are not yet supported and study translations from other transformation languages, which could reuse many of the patterns defined for ATL. Finally, we want to explore how Gremlin-ATL can be used in other scenarios involving the query, evolution and integration of unstructured data. Manipulating such data requires first to discover its implicit schema/s. This schema can be naturally expressed as an explicit model and therefore be a natural fit for our framework that could then be used to facilitate its manipulation.

X. Aknowledgement

This work has been partially funded by the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No. 737494 (MegaM@Rt2 project) and the Spanish government (TIN2016-75944-R project).

[10]http://tinkerpop.apache.org/providers.html

REFERENCES

[1] 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.

[2] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot, "MoDisco: A model driven reverse engineering framework," *Information and Software Technology*, vol. 56, no. 8, pp. 1012 – 1032, 2014.

[3] J. L. C. Izquierdo and J. Cabot, "JSONDiscoverer: Visualizing the schema lurking behind JSON documents," *Knowledge-Based Systems*, vol. 103, pp. 52–55, 2016.

[4] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 1, pp. 31 – 39, 2008.

[5] N. Huijboom and T. Van den Broek, "Open data: an international comparison of strategies," *European journal of ePractice*, vol. 12, no. 1, pp. 4–16, 2011.

[6] J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure," *Science of Computer Programming*, vol. 89, pp. 144–161, 2014.

[7] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE software*, vol. 31, no. 3, pp. 79–85, 2014.

[8] 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 *Proceedings of BigMDE'13*. ACM, 2013, pp. 1–10.

[9] OMG, "UML Specification," 2017. [Online]. Available: www.omg.org/spec/UML

[10] Eclipse Foundation, "The CDO Model Repository (CDO)," 2017. [Online]. Available: http://www.eclipse.org/cdo/

[11] 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," in *Proceedings of the MoDELS 2016 Demo and Poster Sessions*. CEUR, 2016, pp. 1–7.

[12] J. E. Pagán, J. S. Cuadrado, and J. G. Molina, "A repository for scalable model management," *Software & Systems Modeling*, vol. 14, no. 1, pp. 219–239, 2015.

[13] R. Wei and D. S. Kolovos, "An efficient computation strategy for allinstances()," *Proceedings of BigMDE'15*, pp. 32–41, 2015.

[14] Tinkerpop, "The Gremlin Language," 2017. [Online]. Available: www.gremlin.tinkerpop.com

[15] OMG, "OCL Specification," 2017. [Online]. Available: www.omg.org/spec/OCL

[16] ——, "QVT Specification," 2017. [Online]. Available: http://www.omg.org/spec/QVT

[17] 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*. Springer, 2011, pp. 77–92.

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

[19] Tinkerpop, "Blueprints API," 2017. [Online]. Available: www.blueprints.tinkerpop.com

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

[21] G. Daniel, G. Sunyé, and J. Cabot, "Mogwaï: a framework to handle complex queries on large models," in *Proceedings of the 10th RCIS Conference*. IEEE, 2016, pp. 225–237.

[22] D. S. Ruiz, S. F. Morales, and J. G. Molina, "Inferring versioned schemas from NoSQL databases and its applications," in *Proceedings of the 34th ER Conference*. Springer, 2015, pp. 467–480.

[23] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró, "Viatra-visual automated transformations for formal verification and validation of UML models," in *Proceedings of the 17th ASE Conference*. IEEE, 2002, pp. 267–270.

[24] OMG, "Knowledge Discovery Metamodel (KDM)," 2017. [Online]. Available: http://www.omg.org/technology/kdm/

[25] C. Brun and A. Pierantonio, "Model differences in the eclipse modeling framework," *UPGRADE, The European Journal for the Informatics Professional*, vol. 9, no. 2, pp. 29–34, 2008.

[26] A. Benelallam, A. Gómez, M. Tisi, and J. Cabot, "Distributed model-to-model transformation with ATL on MapReduce," in *Proceedings of the 8th SLE Conference*. ACM, 2015, pp. 37–48.

[27] M. Tisi, S. Martinez, and H. Choura, "Parallel execution of ATL transformation rules," in *Proceedings of the 16th MoDELS Conference*. Springer, 2013, pp. 656–672.

[28] L. Burgueño, M. Wimmer, and A. Vallecillo, "A linda-based platform for the parallel execution of out-place model transformations," *Information and Software Technology*, vol. 79, pp. 17–35, 2016.

[29] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *The VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.

[30] H.-H. Do and E. Rahm, "Coma: a system for flexible combination of schema matching approaches," in *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 2002, pp. 610–621.

[31] J. Madhavan, P. A. Bernstein, and E. Rahm, "Generic schema matching with cupid," in *The VLDB Journal*, vol. 1, 2001, pp. 49–58.

[32] C. Drumm, M. Schmitt, H.-H. Do, and E. Rahm, "Quickmig: automatic schema matching for data migration projects," in *Proceedings of the 16th CIKM conference*. ACM, 2007, pp. 107–116.

[33] C. Krause, M. Tichy, and H. Giese, "Implementing graph transformations in the bulk synchronous parallel model." in *Proceedings of the 17th FASE Conference*, vol. 14, 2014, pp. 325–339.