UMLtoGraphDB: Mapping Conceptual Schemas to Graph Databases

Conference Paper *in* Lecture Notes in Computer Science · November 2016

DOI: 10.1007/978-3-319-46397-1_33

3 authors:

Gwendal Daniel
Universitat Oberta de Catalunya
10 PUBLICATIONS **35** CITATIONS
SEE PROFILE

Gerson Sunyé
University of Nantes
78 PUBLICATIONS **980** CITATIONS
SEE PROFILE

Jordi Cabot
Catalan Institution for Research and Advanced …
252 PUBLICATIONS **2,737** CITATIONS
SEE PROFILE

UMLtoGraphDB: Mapping Conceptual Schemas to Graph Databases

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

[1] AtlanMod Team
Inria, Mines Nantes & Lina
{gwendal.daniel,gerson.sunye}@inria.fr
[2] ICREA
jordi.cabot@icrea.cat
[3] Internet Interdisciplinary Institute, UOC

Abstract. The need to store and manipulate large volume of (unstructured) data has led to the development of several NoSQL databases for better scalability. Graph databases are a particular kind of NoSQL databases that have proven their efficiency to store and query highly interconnected data, and have become a promising solution for multiple applications. While the mapping of conceptual schemas to relational databases is a well-studied field of research, there are only few solutions that target conceptual modeling for NoSQL databases and even less focusing on graph databases. This is specially true when dealing with the mapping of business rules and constraints in the conceptual schema. In this article we describe a mapping from UML/OCL conceptual schemas to Blueprints, an abstraction layer on top of a variety of graph databases, and Gremlin, a graph traversal language, via an intermediate Graph metamodel. Tool support is fully available.

Keywords: Database Design, UML, OCL, NoSQL, Graph Database, Gremlin

1 Introduction

NoSQL databases have become a promising solution to enhance scalability, availability, and query performance of data intensive applications. They often rely on a *schemaless* infrastructure, meaning that their schemas are implicitly defined by the stored data and not formally described. This approach offers great flexibility since it is possible to use different representations of a same concept (non-uniform data), but client applications still need to know (at least partially) how conceptual elements are stored in the database in order to access and manipulate them. Acquiring this implicit knowledge of the underlying schema can be an important issue, for example in data integration processes, where each data source has to be inspected to find its underlying structure [13].

Graph databases are a particular type of NoSQL databases that represent data as a set of vertices linked together by edges where both vertices and edges can be labeled with a number of property values. Graph databases often provide advanced and expressive query languages that are particularly optimized to compute traversals of highly interconnected data. Recently, the graph database ecosystem is gaining popularity in several

engineering fields such as social network [11] or data provenance [1] analysis, and the leading graph database vendor Neo4j[4] is used in production by several companies [16].

In order to take full benefit of NoSQL solutions, designers must be able to integrate them in current code-generation architectures to use them as target persistence backend for their conceptual schemas. Unfortunately, while several solutions provide transformations from ER and UML models to relational database schemas, the same is not true for NoSQL databases as discussed in detail in the related work. Moreover, NoSQL databases present an additional challenge: data consistency is a big problem since the vast majority of NoSQL approaches lack any advanced mechanism for integrity constraint checking [21].

To overcome this situation, we propose the UMLtoGraphDB framework, that translates conceptual schemas expressed using the Unified Modeling Language (UML) [24] into a graph representation, and generates database-level queries from business rules and invariants defined using the Object Constraint Language (OCL) [23]. The framework relies on a new GraphDB metamodel, as an intermediate representation to facilitate the integration of several kinds of graph databases. Enforcement of (both OCL and structural) constraints is delegated to an intermediate software component (middleware) in charge of maintaining the underlying database consistent with the conceptual schema. External applications can then use this middleware to safely access the database. This is illustrated in Figure 1.

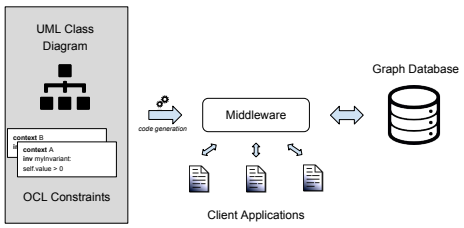

Fig. 1. Conceptual Model to Graph database

The rest of the paper is structured as follows: Section 2 presents the UMLtoGraphDB framework and its core components, Section 3 introduces the GraphDB metamodel and details the model-to-model transformation which creates an instance of it from a UML model. Section 4 presents the transformation that creates graph database queries from OCL expressions, and Section 5 introduces the code generator. Finally, Section 6 describes our tool support, Section 7 presents the related works and Section 8 ends up with the conclusions and future work.

2 UMLtoGraphDB Approach

UMLtoGraphDB is aligned with the OMG's MDA standard [22], proposing a structured methodology to systems development that promotes the separation between a specifi-

[4] http://neo4j.com/

cation defined in a platform independent way (Platform Independent Model, PIM), and the refinement of that specification adapted to the technical constraints of the implementation platform (Platform Specific Model, PSM). A model-to-model transformation (M2M) generates PSM models from PIMs while a model-to-text transformation typically takes care of producing the final code out of the PSM models. This PIM-to-PSM phased architecture brings two important benefits: (i) the PIM level focuses on the specification of the structure and functions, raising the level of abstraction and postponing technical details to the PSM level. (ii) Multiple PSMs can be generated from one PIM, improving portability and reusability. Moreover, using an intermediate PSM model instead of a direct PIM-to-code approach allows designers to tune the generation when needed and simplify the transformations by reducing the semantic gap between their input and output artefacts.

In our scenario, the initial UML and OCL models would conform to the PIM level. UMLtoGraphDB takes care of generating the PSM and code from them. Figure 2 presents the different component of the UMLtoGraphDB framework (light-grey box).

In particular, **Class2GraphDB** (1) is the first M2M of the UMLtoGraphDB framework. It is in charge of the creation of a low-level graph representation (PSM) from the input UML class diagram (PIM). The output of the Class2GraphDB transformation is a **GraphDB Model** (2), conforming to the GraphDB metamodel (Section 3). This metamodel is defined at the PSM level, and describes data structures in terms of graph primitives, such as *vertices* or *edges*. The **OCL2Gremlin** transformation (3) is the second M2M in the UMLtoGraphDB framework. It is in charge of the translation of the OCL constraints, queries, and business rules defined at the PIM level into graph-level queries. It produces a **Gremlin Model**, conforming to the Gremlin language metamodel that complements the previous GraphDB one.

The last step in MDA processes is a PSM-to-code transformation, which generates the software artifacts (database schema, code, configuration files . . .) in the target platform. In our approach, this final step is handled by the **Graph2Code** (5) transformation (Section 5) that processes the generated GraphDB and Gremlin models to create a set of Java Classes wrapping the structure of the database, the associated constraints, and the business rules. These Java classes compose the **Middleware** layer (6) presented in Figure 1, and contain the generated code to access the physical **Graph Database** (7).

To illustrate the different transformation steps of our framework we introduce as a running example the conceptual schema presented in Figure 3 representing a simple excerpt of an e-commerce application. This schema is specified using the UML notation, and describes *Client*, *Orders*, and *Products* concepts. A *Client* is an abstract class defined by a name and an address. *PrivateCustomers* and *CorporateCustomers* are subclasses of *Client*. They contain respectively a *cardNumber* and a *contractRef* attribute. *Clients* own *Orders*, that are defined by a *reference*, a *shipmentDate*, and a *deliveryDate*. In addition, an *Order* maintains a *paid* attribute, that is set to true if the *Order* has been paid. *Products* are defined by their *name*, *price*, and a textual *description* and are linked to *Orders* through the *OrderLine* association class, which records the *quantity* and the *price* of each *Product* in a given *Order*.

In addition, the conceptual data model defines three textual OCL constraints (presented in Listing 1), which represent basic business rules. The first one checks that the

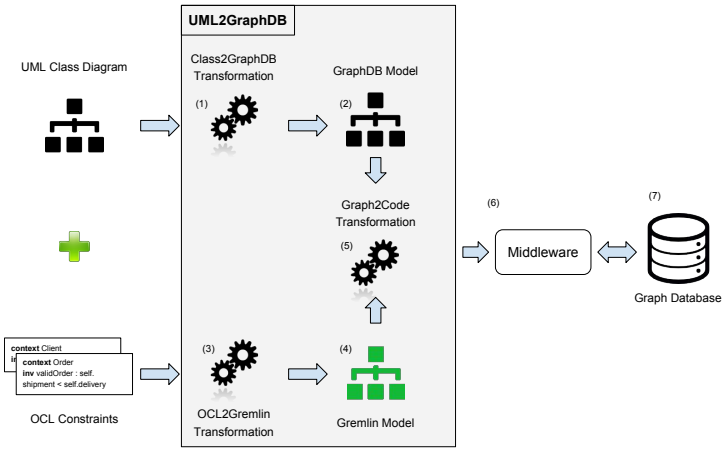

Fig. 2. Overview of the UMLtoGraphDB Infrastructure

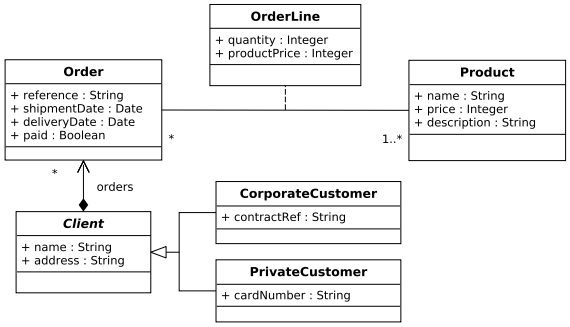

Fig. 3. Class Diagram of a Simple e-commerce Application

price of a *Product* is always positive, the second one verifies that the *shipmentDate* of an *Order* precedes its *deliveryDate*, and the last one ensures a *Client* has less than three unpaid *Orders*.

```
context Product inv validPrice: self.price > 0
context Order inv validOrder: self.shipmentDate < self.deliveryDate
context Client inv maxUnpaidOrders:
    self.orders→select(o | not o.paid)→size() < 3
```

Listing 1. Textual Constraints

3 Mapping UML Class Diagram to GraphDB

In this section we present the Class2Graph transformation, which is the initial step in the approach presented in Figure 2. We first introduce the GraphDB metamodel and then, we focus on the transformation itself.

3.1 GraphDB Metamodel

The GraphDB metamodel defines the possible structure of all GraphDB models. It is compliant with the Blueprints [26] specification, which is an interface designed to unify NoSQL database access under a common API. Initially developed for graph stores, Blueprints has been implemented by a large number of databases such as Neo4j, OrientDB, and MongoDB. The Blueprints API is, to our knowledge, the only interface unifying several NoSQL databases[5]. Blueprints is the base of the Tinkerpop stack: a set of tools to store, serialize, manipulate, and query graph databases. Among other features, it provides Gremlin [27], a traversal query language designed to query Blueprints databases.

Figure 4 presents the GraphDB metamodel. A *GraphSpecification* element represents the top-level container that owns all the objects. It has a *baseDB* attribute, that defines the concrete database to instantiate under the Blueprints API. In our prototype, the baseDB can be either Neo4j or OrientDB, two well known graph databases. *GraphSpecification* contains all the *VertexDefinitions* and *EdgeDefinitions* through the associations *vertices* and *edges*.

A *VertexDefinition* can be *unique*, meaning that there is only one vertex in the database that conforms to it. *VertexDefinitions* and *EdgeDefinitions* can be linked together using *outEdges* and *inEdges* associations, meaning respectively that a *VertexDefinition* has outgoing edges and incoming edges. In addition, *VertexDefinition* and *EdgeDefinition* are both subtypes of *GraphElement*, which can define a set of *labels* that describe the type of the element, and a set of *PropertiesDefinition* through its *properties* reference. In graph databases, properties are represented by a *key* (the name of the property) and a *Type*. In the first version of this metamodel we define four primitive types: *Object*, *Integer*, *String*, and *Boolean*.

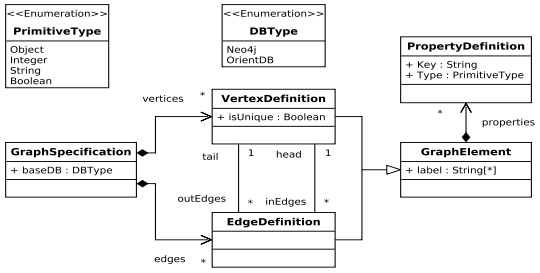

Fig. 4. GraphDB Metamodel

3.2 Class2GraphDB Transformation

Intuitively, the transformation consists of mapping UML *Classes* to *VertexDefinitions*, *Associations* to *EdgeDefinitions*, and *AssociationClasses* to new *VertexDefinitions* con-

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

nected to the ones representing the involved classes. The mapping also creates *PropertyDefinitions* for each *Attribute* in the input model, and add them to the corresponding mapped element.

Note that GraphDB has no construct to represent explicitly inheritance, and thus, the mapping has to deal with inherited attributes and associations. To handle them, the translation finds all the attributes and associations in the parent hierarchy of each class, and adds them to the mapped *VertexDefinition*. While this creates duplicated elements in the GraphDB model, it is the more direct representation to facilitate queries on the GraphDB model. In the following, we describe this transformation in more detail.

A class diagram *CD* is defined as a tuple $CD = (Cl, As, Ac, I)$, where *Cl* is the set of classes, *As* is the set of associations, *Ac* is the set of association classes, and *I* the set of pairs of classes such as $(c1, c2)$ represents the fact that $c1$ is a direct or indirect subclass of $c2$. Note that the first version of UMLtoGraphDB transforms only a subset of the class diagram, for example enumerations and interfaces supports are planned as future work.

A GraphDB diagram *GD* is defined as a tuple $GD = (V, E, P)$, where *V* is set of vertex definitions, and *E* the set of edge definitions, and *P* the set of property definitions that compose the graph.

- **R1:** each class $c \in Cl, not\ c.isAbstract$ is mapped to a vertex definition $v \in V$, where $v.label = c.name \cup c_{parents}.name$, with $c_{parents} \subset Cl$ and $\forall p \in c_{parents}, (c, p) \in I$.
- **R2:** each attribute $a \in (c \cup c_{parents}).attributes$ is mapped to a property definition p, where $p.key = a.name$, $p.type = a.type$, and added to the property list of its mapped container v such as $p \in v.properties$.
- **R3:** each association $as \in As$ between two classes $c_1, c_2 \in Cl$ is mapped to an edge definition $e \in E$, where $e.label = as.name$, $e.tail = v_1$, and $e.head = v_2$, where v_1 and v_2 are the *VertexDefinitions* representing c_1 and c_2. Note that $e.tail$ and $e.head$ values are set according to the direction of the association. If the association is not directed, a second edge definitions $e_{opposite}$ is created, where $e_{opposite}.label = as.name$, $e_{opposite}.tail = v_2$, and $e_{opposite}.head = v_1$, representing the second possible direction of the association. Aggregation associations are mapped the same way, but their semantic is handled differently in the generated code. In order to support inherited associations, *EdgeDefinitions* are also created to represent associations involving the parents of c.
- **R4:** each association $as \in As$ between multiple classes $c_1 ... c_n \in Cl$ is mapped to a vertex definition v_{asso} such as $v_{asso}.label = as.name$ and a set of *EdgeDefinitions* $e_i.tail = v_i$ and $e_i.head = v_{asso}$, associating the created vertex definition to the ones representing $c_1 ... c_n$.
- **R5:** each association class $ac \in Ac$ between classes $c_1 ... c_n$ is mapped like an association between multiple classes using a vertex definition v_{ac} such as $v_{ac}.label = ac.name$. As for a regular class, v_{ac} contains the properties corresponding to the attributes $ac.attributes$, and a set of *EdgeDefinitions* $e_i \in E$ where $e_i.tail = v_i$ and $e_i.head = v_{ac}$.

To better illustrate this mapping, we now describe how the GraphDB model shown in Figure 5 is created from the example presented in Figure 3. Note that for the sake

of readability we only show an excerpt of the created GraphDB model. To begin with, all the classes are translated into *VertexDefinition* instances following *R1*. This process generates the elements `v1`, `v2`, `v3`, and `v4`, with the labels *(Client, PrivateCustomer)*, *(Client,CorporateCustomer)*, *Order*, and *Product*. Then, *R2* is applied to transform attributes into *PropertyDefinitions*. For example, the attribute *name* of the class *Client* is mapped to the *PropertyDefinition* `p1`, which defines a key *name* and a type *String*. These *PropertyDefinition* elements are linked to their containing *VertexDefinition* using the *properties* association. Once this first step has been done, *R3* is applied on the association *orders*, mapping it to the *EdgeDefinitions* `e1` and `e2`, containing the name of the association. *VertexDefinitions* representing *PrivateCustomer* and *CorporateCustomer* classes are then linked to the one representing *Order*, respectively with `e1` and `e2`. Since the association *orders* is directed, the transformation puts `v1` and `v2` as the tail of the edge, and `v3` as its head. Then, the association class *OrderLine* is transformed by *R5* to the *VertexDefinition* `v5`, and its attributes *productPrice* and *quantity* are transformed into the *PropertyDefinitions* `p6` and `p7`. Finally, two *EdgeDefinitions* (`e3` and `e4`) are also created to link the *VertexDefinition* `v3` and `v4` to it.

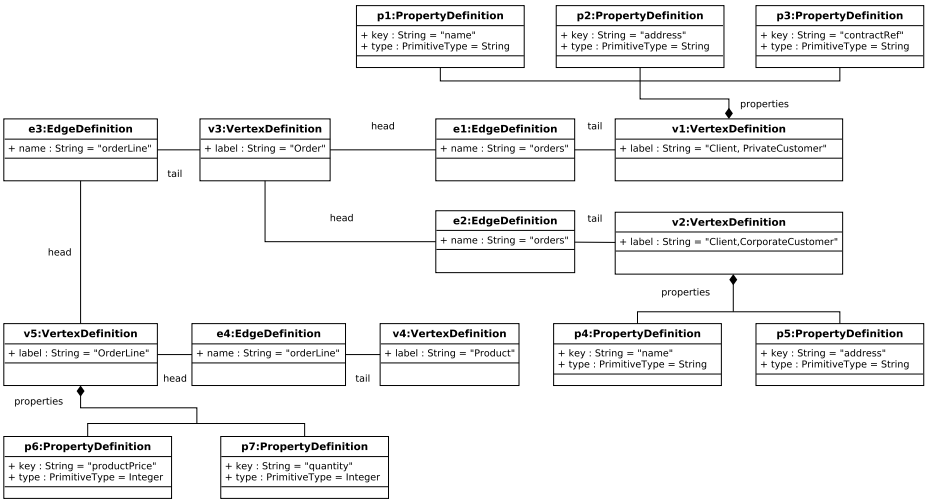

Fig. 5. Excerpt of the Mapped GraphDB Model

These mapping rules have also been specified in ATL [14], which is a domain-specific language for defining model-to-model transformations aligned with the QVT standard [15]. ATL provides both declarative (rule-based) and imperative constructs for transforming and manipulating models. As an example, Listing 2 shows the ATL transformation rule that maps a UML *Class* to a *VertexDefinition*. It is applied for each non-abstract *Class* element, excepted *AssociationClasses*, which have a particular mapping, as explained in Section 3. The rule creates a *VertexDefinition* element, and sets its label attribute with the *name* of each *Class* in its parent hierarchy. The set of parent *Classes* is computed by the helper `getParentClassHierarchy`, which returns a sequence

containing all the parents of the current *Class*. Finally, *VertexDefinition properties* are set, by getting all the attributes from the parent hierarchy, and are transformed by the abstract *lazy rule* `GenericAttribute2Property`. The full ATL transformation is available in the project repository[6].

```
rule Class2VertexDefinition {
  from
    class : UML!Class (not(class.oclIsTypeOf(UML!AssociationClass)) and not(class.
        abstract))
  to
    vertex : Graph!VertexDefinition (
    labels ← class.getParentClassHierarchy()→collect(cc | cc.name)
    -- Generate a property for each Attribute in the class hierarchy
    properties ← class.getParentClassHierarchy()
      →collect(cc | cc.attribute)
      →collect(att | thisModule.GenericAttribute2Property(att))
    )
}
```

Listing 2. Class2VertexDefinition ATL Transformation Rule

4 Translating OCL Expressions to Gremlin

Once the GraphDB model has been created, another transformation is performed to translate the OCL expressions defined in the conceptual schema into a Gremlin query model. The mapping presented in this Section is adapted from the one presented in [8] dedicated to OCL query evaluation on NeoEMF, a scalable model persistence framework designed to store models into graph databases [2]. In this Section, we present the Gremlin language and describe how OCL expressions are transformed into Gremlin queries according to the UML to GraphDB mapping.

4.1 The Gremlin Query Language

Gremlin is a Groovy domain-specific language built over *Pipes*, a data-flow framework on top of *Blueprints*. We have chosen Gremlin as the target query language for UML-toGraphDB due to its adoption in several graph databases.

Gremlin is based on the concept of process graphs. A process graph is composed of vertices representing computational units and communication edges which can be combined to create a complex processing flow. In the Gremlin terminology, these complex processing flows are called *traversals*, and are composed of a chain of simple computational units named *steps*. Gremlin defines four types of steps: **Transform steps** that map inputs of a given type to outputs of another type, **Filter steps**, selecting or rejecting input elements according to a given condition, **Branch steps**, which split the computation into several parallel sub-traversals, and **side-effect steps** that perform operations like edge or vertex creation, property update, or variable definition or assignment.

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.

[6] https://github.com/atlanmod/UML2NoSQL

4.2 OCL2Gremlin Transformation

Table 1 presents the mapping between OCL expressions and Gremlin concepts. Supported OCL expressions are divided into four categories based on Gremlin step types: transformations, collection operations, iterators, and general expressions. Note that due to lack of space we only present a subset of the OCL expressions which are supported by our approach. A complete version of this mapping is available in previous work [8].

Table 1. OCL to Gremlin mapping

OCL expression	**Gremlin step**
Type	"Type.name"
C.allInstances()	g.V().hasLabel("C.name")
collect(attribute)	property(attribute)
collect(reference)	outE('reference').inV
oclIsTypeOf(C)	o.hasLabel("C.name")
col_1 →union(col_2)	col_1.fill(var_1); col_2.fill(var_2); union(var_1, var_2);
including(object)	gather{it << object;}.scatter;
excluding(object)	except([object]);
size()	count()
isEmpty()	toList().isEmpty()
select(condition)	c.filter{condition}
reject(condition)	c.filter{!(condition)}
exists(expression)	filter{condition}.hasNext()
$=, >, >=, <, <=, <>$	$==, >, >=, <, <=, !=$
$+, -, /, \%, *$	$+, -, /, \%, *$
and,or,not	&&,‖,!
variable	variable
literals	*literals*

These mappings are systematically applied on the input OCL expression, following a postorder traversal of the OCL Abstract Syntax Tree. As an example, Listing 3 shows the Gremlin queries generated from the OCL constraints of the running example (Section 2). The `v` variable represents the vertex that is being currently checked, and the following steps are created using the mapping. Note that generated expressions are queries that return a boolean value. These queries are embedded in checking methods during the generation phase (Section 5).

```
v.property("price") > 0; // validPrice
v.property("shipmentDate") < self.property("deliveryDate"); // validOrder
v.outE("orders").inV.filter{it.property("paid")==false}
    .count() < 3; // maxUnpaidOrders
```

Listing 3. Generated Gremlin Queries

5 Code Generation

Our code-generator relies on the Blueprints API for interacting with the graph database in a vendor neutral way. We first briefly review this API and then we show how we leverage it to enforce that any application aiming to query/store data through the created middleware does it so according to the its initial UML/OCL conceptual schema.

5.1 Blueprints API

The Blueprints API is composed of a set of Java classes to manipulate graph databases in a generic way. These classes are wrappers for database-level elements, such as *vertices* and *edges*, providing methods to access, update, and delete them. A Blueprints database is instantiated using a `GraphFactory`, that takes a configuration file containing the properties of the databases (type of the underlying graph engine, allocated memory ...) and creates the corresponding graph store.

The Blueprints `Vertex` class provides the methods `addEdge(String label, Vertex otherEnd)` and `removeEdge(otherEnd)` that allow to connect/disconnect two vertices by creating/deleting an edge between the current vertex and *otherEnd* with the given *label*. Blueprints also defines the vertex method `property(String key)`, that retrieve the value of the vertex property defined by the given key. In addition, the Blueprints API provides the `traversal()` method, that allows to send Gremlin traversals to the database and return the subgraph resulting from that query.

A complete reference of the Blueprints API is available in [26] .

5.2 Graph2Code Transformation

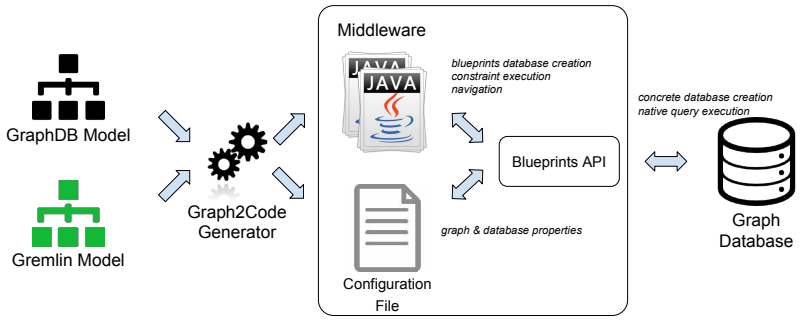

Fig. 6. Generated Infrastructure

The final step in our UMLtoGraphDB process is the database and code artifacts generation. Figure 6 presents the infrastructure generated by the Graph2Code transformation. In short, the generator processes the GraphDB model to retrieve all the *VertexDefinition* elements and, for each one, it creates a corresponding Java class with the relevant

getters and *setters* for its attributes (derived from the properties definitions linked to the vertex) and associations (derived from the input/output edges of the vertex).

Listing 4 presents an excerpt of the Java class generated from the *Client* element. Note that this class extends *BlueprintsBean*, which is a generic class that we provide as part of the UMLtoGraphDB infrastructure. *BlueprintsBean* provides auxiliary methods to connect the class with the Graph database via the Blueprints API and facilitates the creation and management of graph elements.

Once this basic Java class structure is completed, the generator starts processing the *Gremlin Model* to create additional methods. Each method is in charge of checking one of the OCL constraints (or queries) in the conceptual schema. As usual, checking methods return a boolean value (*false* if the constraint is violated). As an example, Listing 4 includes the method `checkMaxUnpaidOrder` executing the Gremlin traversal mapped from the OCL expression self.orders→select(o | not o.paid)→size() < 3 (this mapping is detailed in Section 4). The generated expression follows the syntax variant of the Gremlin internal DSL and not the Groovy-based syntax, both versions can be generated by our infrastructure. Note that the task of calling the generated constraint-checking method is responsibility of the client application. Automatic and incremental checking of these constraints is left for future work.

Finally, the Graph2Code generator creates a *Configuration File* that contains the graph and database properties, and is used by the Blueprints API to instantiate the concrete graph engine.

```
public class Client extends BlueprintsBean {
  public String getName() {
    return (String)this.vertex.property("name").value();
  }
  public String getAddress() {
    return (String)this.vertex.property("address").value();
  }
  public void setName(String newName) {
    this.vertex.property("name",newName);
  }
  public void setAddress(String newAddress) {
    this.vertex.property("address",newAddress);
  }
  public void addOrder(Order order) {
    this.vertex.addEdge("orders", order.getVertex());
  }
  public void removeOrder(Order order) {
    this.vertex.removeEdge(order.getVertex());
  }
  public boolean checkMaxUnpaidOrders() {
    return this.graph.traversal().V(this.vertex).outE("orders")
      .inV().filter(v -> v.get().<Boolean>property("paid").value())
        .count().is(P.lt(3)).hasNext();
  }
}
```

Listing 4. Generated Client Java Class

6 Tool Support

UMLtoGraphDB has been implemented as a collection of open-source Eclipse plugins, available on Github[7]. UMLtoGraphDB takes as input the UML and OCL files (defined, for instance, using Eclipse-based UML editors such as Papyrus[8]), that are then translated, respectively, by the Class2GraphDB and OCL2Gremlin ATL transformations seen before. These transformations add up to a total of 110 rules and helper functions.

The code-generator is implemented using the XTend programming language [3]. Even if this language was initially designed as a template-based language for generation tasks it has now evolved to a more general programming language that provides syntactic sugar, lambda expressions and other useful extensions on top of Java. The generator takes the GraphDB and Gremlin models and processes them as described in Section 5.

The time needed by the entire transformation chain to produce the Java code from the input UML and OCL specifications is in the order of a few seconds for the several examples we have tested. A precise analysis of the scalability of the transformation performance according to the size of the input for very large conceptual model is left for future work.

7 Related Work

Mapping conceptual schemas to relational databases is a well-studied field of research [19]. A few works also cover schemas that include (OCL) constraints. For example, Demuth and Hussman [9] propose a mapping from UML (augmented with OCL constraints) to SQL that covers most of OCL and implement it via a code generator [10] that automates the process. Brambilla et al. [4] propose a methodology to implement integrity constraints into relational databases recommending alternative implementations based on performance parameters. While these approaches are well-suited for relational databases, they all rely on the generation of database constraints. In a NoSQL environment, and especially for graph databases, there is a lack of support for built-in constraint constructs, and data validation must be delegated to the application layer as UMLtoGraphDB does.

Li et al. proposed an approach to transform UML class diagrams into a HBase data model [18], by mapping classes to tables, attributes to columns, and providing transformation rules for associations, compositions, and generalization. Still, it is only applicable to column-based datastores, and does not support the definition of custom OCL constraints and business rules.

More specific to NoSQL databases, the NoSQL Schema Evaluator [20] generates query implementation plans from a conceptual schema and workload definition. For now, the approach is limited to Cassandra, but authors intend to adapt it to different data models, such as key-values and document stores. However, this solution does not take into account constraints specified in the conceptual model. Sevilla et al. [25] presented

[7] https://github.com/atlanmod/UML2NoSQL

[8] https://eclipse.org/papyrus/

a tool to infer versioned schemas from NoSQL databases. The resulting model is then used to automatically generate a viewer and validator for the schema but they do not aim to provide support for a full-fledged application nor consider the addition of constraints on the reversed schema. Bugiotti et al. [5] propose a database design methodology for NoSQL databases. It relies on NoAM, an abstract data model that aims to represent NoSQL systems in a system-independent way. NoAM models can be implemented in several NoSQL databases, including key-value stores, document databases, and extensible record stores. Instead, we focus on generating NoSQL databases from higher-level UML models, and thus, designers do not need to learn a new language/platform. Nevertheless, NoAM could be integrated in our approach if we manage to extend it with constraint support. In that case, NoAM could be seen as a PSM derived from UML models and OCL constraints, and can be used to implement non-graph databases, which are not supported by our approach for now.

8 Conclusion and Future Work

In this article we have presented the UMLtoGraphDB framework, a MDA-based approach to implement (UML) conceptual schemas in graph databases, including the generation of the code required to check the OCL constraints defined in the schema. Our approach is specified as a chain of model transformations that use a new intermediate GraphDB metamodel. This metamodel can also be regarded as a kind of UML profile (and could be easily reexpressed as such) for graph databases.

As future work, we plan to provide refactoring operations on top of the GraphDB model to allow designers to tune the data representation according to specific needs, such as query execution performance or memory consumption. We also plan to extend our approach to cover reverse engineering scenarios, by adapting existing work on schema extraction from relational databases [7] to graph databases. Another ongoing work pursues adapting our framework to cover multiple database types. More precisely, we aim to support conceptual schema fragmentation between several databases (even mixing NoSQL and SQL ones). This requires a mechanism to evaluate constraints over several persistence solutions and query languages. Apache Drill [12] or Hibernate OGM [17] could be reused for this.

Finally, we plan to reuse existing work on the integration of incremental constraint checking [6] as part of the code-generation phase so that the scalable performance of the graph database is not hampered by the constraint evaluation phase.

References

1. Manish Kumar Anand, Shawn Bowers, and Bertram Ludäscher. Techniques for efficiently querying scientific workflow provenance graphs. In *EDBT*, volume 10, pages 287–298, 2010.
2. Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. Neo4EMF, a Scalable Persistence Layer for EMF Models. In *Proc. of the 10th ECMFA*, pages 230–241. Springer, 2014.
3. Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.

4. Marco Brambilla and Jordi Cabot. Constraint tuning and management for web applications. In *Proc. of the 6th ICWE Conference*, pages 345–352. ACM, 2006.
5. Francesca Bugiotti, Luca Cabibbo, Paolo Atzeni, and Riccardo Torlone. Database design for nosql systems. In *Proc. of the 33rd ER Conference*, pages 223–231. Springer, 2014.
6. Jordi Cabot and Ernest Teniente. Incremental integrity checking of UML/OCL conceptual schemas. *JSS*, 82(9):1459–1478, 2009.
7. Roger HL Chiang, Terence M Barron, and Veda C Storey. Reverse engineering of relational databases: Extraction of an EER model from a relational database. *Data & Knowledge Engineering*, 12(2):107–142, 1994.
8. Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. Mogwaï: a framework to handle complex queries on large models. In *Proc. of the 10th RCIS Conference [To appear]*. IEEE, 2016. Available Online at http://tinyurl.com/zx6cfam.
9. Birgit Demuth and Heinrich Hussmann. Using UML/OCL constraints for relational database design. In *«UML»'99—The Unified Modeling Language*, pages 598–613. Springer, 1999.
10. Birgit Demuth, Heinrich Hußmann, and Sten Loecher. OCL as a specification language for business rules in database applications. In *≪ UML≫ 2001—The Unified Modeling Language. pages=104–117, year=2001, publisher=Springer*.
11. Wenfei Fan. Graph pattern matching revised for social network analysis. In *Proc. of the 15th ICDT*, pages 8–21. ACM, 2012.
12. Michael Hausenblas and Jacques Nadeau. Apache drill: interactive ad-hoc analysis at scale. *Big Data*, 1(2):100–104, 2013.
13. Javier Luis Cánovas Izquierdo and Jordi Cabot. Discovering implicit schemas in json data. In *Web Engineering*, pages 68–83. Springer, 2013.
14. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *SCP*, 72(1–2):31 – 39, 2008.
15. Frédéric Jouault and Ivan Kurtev. On the architectural alignment of ATL and QVT. In *Proc. of the 21st SAC conference*, pages 1188–1195. ACM, 2006.
16. Mahesh Lal. *Neo4j Graph Data Modeling*. Packt Publishing Ltd, 2015.
17. Anghel Leonard. *Pro Hibernate and MongoDB*. Apress, 2013.
18. Yan Li, Ping Gu, and Chao Zhang. Transforming UML class diagrams into HBase based on meta-model. In *Proc. of the 4th ISEEE conference*, volume 2, pages 720–724. IEEE, 2014.
19. Esperanza Marcos, Belén Vela, and José María Cavero. A methodological approach for object-relational database design using UML. *SoSyM*, 2(1):59–72, 2003.
20. Michael J Mior, Kenneth Salem, Ashraf Aboulnaga, and Rui Liu. NoSE: Schema design for NoSQL applications. In *[Accepted] at the 32nd ICDE Conference*. IEEE, 2016. Available Online at http://tinyurl.com/hqoxddx.
21. Lior Okman, Nurit Gal-Oz, Yaron Gonen, Ehud Gudes, and Jenny Abramov. Security issues in NoSQL databases. In *Proc. of the 10th TrustCom Conference*, pages 541–547. IEEE, 2011.
22. OMG. MDA Specifications, 2016. URL: http://www.omg.org/mda/specs.htm.
23. OMG. OCL Specification, 2016. URL: www.omg.org/spec/OCL.
24. OMG. UML Specification, 2016. URL: www.omg.org/spec/UML.
25. Diego Sevilla Ruiz, Severino Feliciano Morales, and Jesús García Molina. Inferring versioned schemas from NoSQL databases and its applications. In *Proc. of the 34th ER Conference*, pages 467–480. Springer, 2015.
26. Tinkerpop. Blueprints API, 2016. URL: blueprints.tinkerpop.com.
27. Tinkerpop. The Gremlin Language, 2016. URL: gremlin.tinkerpop.com.