

Establishing a Bridge from Graph-based Modeling Languages to Ontology Languages ^{*}

Tobias Walter¹, Hannes Schwarz¹, Yuan Ren²

¹ University of Koblenz-Landau, {walter,hschwarz}@uni-koblenz.de

² University of Aberdeen, y.ren@abdn.ac.uk

Abstract. In today’s software industry, the pursuit of Model-Driven Development has become a serious option. Depending on their purpose, models can be represented in different languages with different strengths and weaknesses. Recently, ontologies with the predominant Web Ontology Language OWL are more and more recognized as being able to adequately complement modelware, i.e. the more traditional modeling languages such as MOF/UML or the TGraph approach. This paper compares the TGraph approach as representative for modelware with OWL and bridges both technologies by introducing a transformation from TGraphs to OWL. Subsequently, the advantages of both technologies are highlighted, showing how to benefit from the presented bridging approach.

1 Introduction

Today Model-Driven Development (MDD) plays a key role in building software systems. A variety of different modeling languages may be used to develop large software systems. Each language focuses on different views and problems of the system [1] and offers other features and advantages. Thus, it is often required to rely on different modeling languages for the description of a single system. However, to represent the same models in different languages, a proper *bridging* between the languages is essential.

In this paper we consider two modeling languages to be bridged: *TGraphs* combined with the metamodeling language *grUML* and the query language *GReQL* [2], together constituting the TGraph approach, and *OWL 2* [3].

The TGraph approach belongs to the so-called *modelware* which comprises “traditional” modeling languages, with the Meta Object Facility (MOF) [4] including the Object Constraint Language (OCL) [5] by the Object Management Group as its probably most popular representative. Essential MOF (EMOF) is a less complex subset of MOF which dominates practical usage, especially with respect to tool-building. Nevertheless, although the TGraph approach can basically be compared with EMOF and OCL, we have chosen it for the purposes of this paper, for it offers advantages with respect to expressivity and formal background. TGraphs have been used for various applications, e.g. the development of meta-case tools and software reengineering [2].

TGraphs are a very general kind of directed graphs whose vertices and edges are typed, attributed, and ordered. They provide all needed features for representing languages and models as graphs in a tool. An efficient implementation of the TGraph

^{*} supported by EU STReP-216691 MOST

approach, the JGraLab Java API³ enables the easy use of TGraphs in practice. TGraphs conform to metamodels, called *schemas* in TGraph terminology. Schemas are specified in grUML [6] and can incorporate constraints formulated in GReQL, the Graph Repository Query Language [7]. The main features setting the TGraph approach apart from EMOF are its formally defined semantics, its treatment of edges as first-class objects, and the various orderings imposed on graph elements.

OWL 2 is a W3C recommendation with a comprehensive set of constructs for concept definitions [3]. OWL 2 is an implementation of a description logic [8]. Description logics constitute a family of logics for concept definitions that allow for joint as well as for separate sound and complete reasoning at the model and at the instance level.

The motivation for this work comes from the different advantages and features provided by modelware and ontologies: while usually, development is mainly conducted using modelware languages, developers are more and more interested in representing selected models as ontologies to be able to benefit from their exclusive features. For example, reasoning services could be applied to detect whether a schema's constraints are contradicting, resulting in the schema to be uninstantiable. In this paper, we will therefore restrict ourselves to the description of a transformation-based bridge from modelware to ontologies, or more precisely, from TGraphs to OWL.

The paper is structured as follows: in section 2, we introduce both relevant modeling languages, i.e. TGraphs, grUML, and GReQL, as well as the Web Ontology Language. Section 3 presents the transformation of TGraph schemas plus their constraints to an OWL ontology. In section 4, we show the different advantages provided by the modeling approaches and we give an idea of the benefits of transforming TGraphs to an ontology representation. In section 5, we will examine some related modeling approaches, including MOF and OCL. Finally, section 6 concludes the paper.

2 Modeling Approaches and Languages

In the following we introduce TGraphs together with grUML and GReQL in section 2.1 and OWL 2 in section 2.2. In section 2.3, we compare the modeling concepts provided by the two approaches.

2.1 The TGraph Approach

TGraphs are a versatile kind of graphs whose edges are first class citizens, i.e. they are distinguished graph elements with similar properties as vertices. TGraphs are *directed*, their edges and vertices are *typed* and *attributed*, and for each vertex the incident edges are *ordered*. In addition, the vertices and edges of the graph are ordered globally. Every graph is an instance of a *graph schema* which defines the types of edges and vertices, associates them with attributes, and structures them in generalization hierarchies.

grUML. The metamodeling language *grUML* used for modeling schemas corresponds to a subset of UML class diagrams [6]. With regard to the OMG metamodeling architecture [4], graph schemas are defined on the M2 layer. Their instances, i.e. TGraphs,

³ <http://jgralab.uni-koblenz.de>

lie on the M1 layer, whereas the metaschema of grUML itself, prescribing the structure of graph schemas, is defined on the M3 layer.

Figure 1 depicts a sample activity diagram in concrete syntax. It represents the visualization of a TGraph describing the diagram's abstract syntax which in turn conforms to the grUML schema in figure 2. Here, action nodes (e.g. Receive Order, Fill Order) are used to model concrete actions within an activity. Object nodes (e.g. Invoice) can be used to model the flow of values or objects. Control nodes (e.g. the initial node before Receive Order, the fork and join nodes around Ship Order, and the final node after Close Order) are used to coordinate the flows between other nodes. Activity diagrams can contain two types of edges: object flows and control flows. Object flows model the flow of values to or from object nodes. Control flows specify the sequencing of nodes.

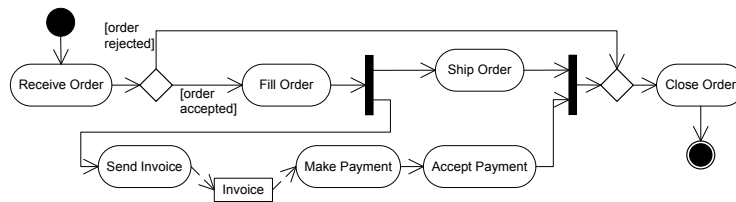


Fig. 1. Example of Activity Diagram

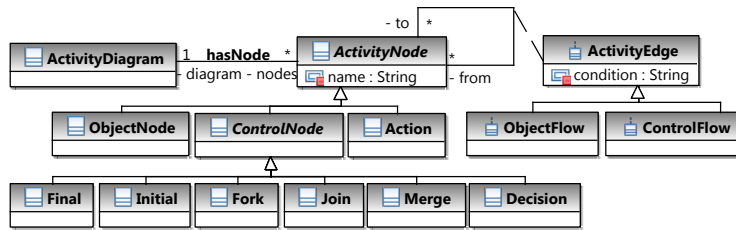


Fig. 2. Schema of Activity Diagram

GReQL. The *Graph Repository Query Language*, is a declarative, expression-based query language for TGraphs [7]. It can be used to extract information from TGraphs, for example attributes of vertices and edges or complete structures inside of graphs. Typical GReQL queries are the so-called *FWR expressions* and *quantified expressions*.

FWR expressions consist of the three clauses from, with and report. The from-clause declares variables for concerned elements (e.g. vertices and edges) in the graph. The variables' domains can be taken from the types defined in the graph schema. The optional with-clause summarizes predicates which have to be fulfilled by the variables. These predicates can include powerful graph-oriented expressions such as regular path expressions. Finally, the report-clause determines the result structure of the query.

Universally and existentially quantified expressions test whether all or some graph elements fulfill specific conditions and return a boolean value, respectively.

For the purposes of this work, we consider GReQL as language for formulating constraints complementing grUML schemas. Since only expressions returning a boolean value qualify as constraints, the bridging approach will exclusively address such expressions, with quantified expressions being the most important ones. Examples for GReQL expressions can be taken from section 3.

2.2 OWL 2

In general, ontologies are used to define sets of concepts that describe domain knowledge and allow for specifying classes by rich, precise logical definitions. Among various existing ontology languages, we use the W3C standard OWL in its currently second version (OWL 2) [3] in this paper. OWL actually stands for a family of languages with increasing expressiveness. Furthermore, OWL inherits the modularisation mechanism of XML, namely namespaces and imports (text inclusion).

The difference between OWL and modeling languages such as grUML is the capability to describe classes in many different ways and to handle incomplete knowledge. These OWL features increase the expressiveness of the metamodeling language, making OWL a suitable language to formally define classes of modeling languages. OWL 2 is axiom-based and thus provides different constructs to restrict classes or properties.

The full OWL 2 metamodel, which provides different kinds of expressions and axioms, can be found in [3]. In the following we use the textual *Functional-Style Syntax* as a concrete syntax for OWL. Its terminal symbols directly refer to classes of the metamodel. The model-theoretic semantics of OWL is presented in [9].

Example. In the following we give an example of an OWL ontology describing a small excerpt of the schema in figure 2:

```
Declaration(Class(ActivityDiagram))
Declaration(Class(ActivityNode))
Declaration(Class(ObjectNode))

SubClassOf(ObjectNode ActivityNode)

Declaration(ObjectProperty(HasNode))
ObjectPropertyDomain(HasNode ActivityDiagram)
ObjectPropertyRange(HasNode ActivityNode)
```

Here three classes (ActivityDiagram, ActivityNode, ObjectNode) are declared. ObjectNode is a subclass of ActivityNode. The object property HasNode is declared which connects the classes ActivityDiagram and ActivityNode via a domain- and a range-axiom.

Open World vs. Closed World Assumption. While the semantics of grUML-based modeling adopts the closed world assumption (CWA), description logics adopt open world assumption (OWA) by default. The closed-world assumption states that the elements in the model are known and unchanging. The open world assumption assumes incomplete information as default and allows for validating incomplete models which are still in the design phase. However, research in the field of combining DL and logic programming [10] provides solutions to support DL-based reasoning with closed world assumption as well [11].

Knowledge Base and Reasoning. A DL knowledge base is established by a set of assertions and a set of terminological axioms, e.g. concept definitions and constraints, and can be structured into TBox (Terminological Box) and ABox (Assertional Box). The TBox is used to specify concepts which denote sets of individuals and roles defining binary relations between individuals. In the ABox, concrete knowledge is asserted by defining individuals of concepts and linking them using the roles defined in the TBox. OWL 2 is a language for modeling both TBox and ABox. Based on the DL knowledge base, standard reasoners (e.g. Pellet [11]) can provide reasoning services such as consistency checking, satisfiability checking, and subsumption checking. Consistency checking proves that the ABox is consistent with regard to its TBox, satisfiability checking verifies that a given concept can be instantiated, and subsumption checking determines if a given concept subsumes (is super-concept of) another concept.

2.3 Structural Comparison

Before we go into the details of transforming TGraph concepts to OWL, we conduct a first comparison of the different language constructs. An overview is given by table 1.

grUML schemas are used to describe the structure of conforming TGraphs. They contain vertex classes to define sets of vertices and edge classes to define sets of edges. The elements in graph schemas are instantiated to build a graph. Ontologies merge both type and instance definitions and allow for using types in instance definitions and vice versa. To separate the model elements which are provided by a graph schema, grUML allows for the definition of packages. This is not supported in OWL. There, only one ontology document can be modeled which cannot be further separated hierarchically.

grUML vertex classes and edge classes specify types for vertices and edges. OWL provides *class expressions* to define types of *individuals* and *object property expressions* to define possible relations between two individuals. The OWL correspondence for attributes are *data property expressions*. Since edge classes can possess attributes, but OWL does not provide a similar concept for object property expressions, we compare an edge class to a class expression together with two object property expressions connecting to the class expressions representing the start and end vertex classes. The existence of an object property between two individuals is specified by a so-called *object property assertion*. Similarly, *data property assertions* assign values to individuals.

grUML	OWL
Graph Schema, Graph	Ontology
Package	—
Vertex Class	Class Expression
Edge Class	Class Expression, two Object Property Expressions
Attribute	Data Property Expression
Vertex	Individual
Edge	Individual, two Object Property Assertions
Attribute Value	Data Property Assertion

Table 1. Comparison of language elements

3 Transformation from the TGraph Approach to OWL

In the following we define a transformation of TGraphs, their schemas and constraints to an OWL representation. While sections 3.1 and 3.2 are concerned with the transformation of pure TGraph and grUML concepts, sections 3.3 to 3.5 deal with GReQL constraints. The mapping of TGraphs to OWL is complete with the exception of some GReQL expressions which cannot be translated to OWL, for example those comparing attribute values or using different functions of the GReQL library (see also section 4).

3.1 General Modeling of TGraphs and Schemas in OWL

In the following we define how vertex classes and edge classes of a graph schema are transformed into OWL class expressions and object properties.

1. Each vertex class (e.g. `ActivityNode`) is represented as an OWL class:

```
Declaration(Class(ActivityNode))
```

2. All vertex classes (e.g. `ActivityNode`) are subsumed by a super concept `Vertex`:

```
Declaration(Class(Vertex))
SubClassOf(ActivityNode Vertex)
```

3. Each edge class (e.g. `HasNode`) in the graph schema which relates two vertex classes (e.g. `ActivityDiagram` and `ActivityNode`) is represented by an OWL class. Furthermore, there exist two object properties (e.g. `fromHasNode` and `toHasNode`) which define the source and target vertices of an edge, respectively. In addition, a separate object property (e.g. `HasNodeProperty`) is declared as the chain of these two object properties, connecting the source and target vertices of the edge (e.g. `InverseObjectProperty(fromHasNode)` and `toHasNode`):

```
Declaration(Class(HasNode))

Declaration(ObjectProperty(toHasNode))
ObjectPropertyDomain(toHasNode ActivityDiagram)
ObjectPropertyRange(toHasNode HasNode)

Declaration(ObjectProperty(fromHasNode))
ObjectPropertyDomain(fromHasNode ActivityNode)
ObjectPropertyRange(fromHasNode HasNode)

Declaration(ObjectProperty(HasNodeProperty))
SubObjectPropertyOf(SubObjectPropertyChain(InverseObjectProperty(fromHasNode) toHasNode)
HasNodeProperty)
```

4. All edge classes are subsumed by a super concept `TopEdge`. Furthermore there exist two object properties `to` and `from`, which define source and target vertex of an edge. `toInverse` and `fromInverse` are their inverse object properties, respectively:

```
Declaration(Class(TopEdge))
SubClassOf(HasNode TopEdge)

Declaration(ObjectProperty(to))
ObjectPropertyDomain(to Vertex)
ObjectPropertyRange(to TopEdge)
```

Declaration(ObjectProperty(from))
ObjectPropertyDomain(from Vertex)
ObjectPropertyRange(from TopEdge)

Declaration(ObjectProperty(toInverse))
Declaration(ObjectProperty(fromInverse))
InverseObjectProperties(toInverse to)
InverseObjectProperties(fromInverse from)

5. All OWL classes representing edge classes and their object properties to and from are represented by one single object property `topEdgeProperty`, which is an object property chain (a sequence of object properties) of `fromInverse` and `to`. Furthermore, `topEdgeProperty` has a transitive super object property `transitiveTopEdgeProperty`:

Declaration(ObjectProperty(topEdgeProperty))
ObjectPropertyDomain(topEdgeProperty Vertex)
ObjectPropertyRange(topEdgeProperty Vertex)
SubObjectPropertyOf(SubObjectPropertyChain(fromInverse to) topEdgeProperty)

SubObjectPropertyOf(topEdgeProperty transitiveTopEdgeProperty)
TransitiveObjectProperty(transitiveTopEdgeProperty)

6. Each attribute (e.g. `name`) is transformed to a data property in OWL and bound to the class it belongs to via a domain axiom. The datatype of the attribute (e.g. `string`) is specified by a range axiom:

Declaration(DataProperty(name))
DataPropertyDomain(name ActivityNode)
DataPropertyRange(name xsd:string)

7. Specialization relationships relating vertex classes and edge classes, respectively, are transformed to OWL subclass axioms. For example, the edge class `ObjectFlow` is subclass of `ActivityEdge` and the vertex class `Action` is subclass of `ActivityNode`:

SubClassOf(ObjectFlow ActivityEdge)
SubClassOf(Action ActivityNode)

8. TGraphs themselves are transformed to a set of assertions. The example below shows an excerpt of the OWL representation of the diagram in figure 1. Note the transformation of edges to named individuals (`ControlFlow_7` in this example).

ClassAssertion(MakePayment Action)
ClassAssertion(AcceptPayment Action)
ClassAssertion(ControlFlow_7 ControlFlow)
ObjectPropertyAssertion(fromControlFlow MakePayment ControlFlow_7)
ObjectPropertyAssertion(toControlFlow AcceptPayment ControlFlow_7)
DataPropertyAssertion(name MakePayment "Make Payment")
DataPropertyAssertion(name AcceptPayment "Accept Payment")

3.2 Cardinality Expressions

In grUML cardinalities are directly defined by annotating edge classes in graph schemas (see figure 2 for an example). In OWL, cardinalities are realized by the class expressions `ObjectMinCardinality`, `ObjectMaxCardinality`, and `ObjectExactCardinality`. They describe those individuals which are connected via an object property to at least, at most, and exactly a given number of individuals of a specified class expression, respectively.

In the following we restrict the concept `ActivityNode` by defining a superclass containing those individuals of `ActivityNode` which are linked via the inverse of object property `HasNodeProperty` with exactly 1 individual of `ActivityDiagram`.

```
SubClassOf(ActivityNode ObjectExactCardinality(1 InverseObjectProperty(HasNodeProperty) ActivityDiagram))
```

3.3 Quantified Expressions

While GReQL constraints are logics-based (thus, for a given instance the evaluation of a constraint returns true or false), their OWL equivalence, so-called class expressions, contain those individuals which fulfill the constraint.

In the following, we show how to transform GReQL's quantified expressions to OWL. Basically, quantifications specify whether all (universal quantification) or at least one (existential quantification) element of a given set of elements must fulfill a given condition. Universal quantification in GReQL is realized by using the `forall` keyword. The following expression defines that all vertices `n` of vertex class `ActivityNode` must fulfill the condition after `@`, i.e. the name-attribute may not be the empty string.

```
forall n:V{ActivityNode} @ not(n.name = "")
```

Existential quantification is realized by using the `exists` keyword. The constraint below defines that for each vertex `f` of vertex class `Final`, there exists at least one vertex `a` of vertex class `ActivityNode` which is connected to `f` via an `ActivityEdge`.

```
forall f:V{Final} @ exists a:V{ActivityNode} @ a -->{ActivityEdge} f
```

In OWL, the `DataAllValuesFrom` class expression allows for universal quantification over a data property. It contains those individuals that are connected by a data property expression *only* to a given data value. Therefore, the above universally quantified expression can be represented by the following subsumption axiom. The negation is realized by an `DataComplementOf` construct.

```
SubClassOf(ActivityNode DataAllValuesFrom(name DataComplementOf("")))
```

The existential quantification over an object property in OWL is realized by an `ObjectSomeValuesFrom` class expression. It describes those individuals that are connected via an object property to at least one instance of a given class expression. In the following we restrict the concept `Final` by defining a superclass containing those individuals which are linked to individuals of concept `ActivityNode` via the object property `from`.

```
SubClassOf(Final ObjectSomeValuesFrom(InverseObjectProperty(ActivityEdgeProperty) ActivityNode))
```

3.4 Boolean Connectives

Boolean connectives are logical operators that connect logical expressions returning a boolean value. GReQL provides the operators `and`, `or` and `not` for boolean connectives. They have two (in the case of `not` one) input parameters and return a boolean value.

The following GReQL constraint requires that each activity diagram must have at least one initial node *and* at least one final node:

```
forall ad:V{ActivityDiagram} @ (exists i:V{Initial} @ ad -->{HasNode} i) and (exists f:V{Final} @ ad -->{HasNode} f)
```

For negating some boolean expression in GReQL the not-operator is used. The following constraint requires that all initial nodes do *not* have incoming ActivityEdges:

```
forall i:V{Initial} @ not(exists n:V{ActivityNode} @ n -->{ActivityEdge} i)
```

Boolean connectives in OWL are provided by ObjectIntersectionOf (and), ObjectUnionOf (or), and ObjectComplementOf (not) class expressions. They provide the standard set-theoretic operations on class expressions.

To state that an activity diagram contains at least one start and at least one final state, both restrictions are formulated as class expressions and connected by ObjectIntersectionOf to form a new class expression which becomes superclass of ActivityDiagram:

```
SubClassOf(ActivityDiagram ObjectIntersectionOf(ObjectSomeValuesFrom(HasNodeProperty Initial)
ObjectSomeValuesFrom(HasNodeProperty Final)))
```

The negation in OWL is realized by the ObjectComplementOf class expression. To state that an initial node has no incoming transitions we describe the concept of incoming transitions by an ObjectSomeValuesFrom class expression and negate it:

```
SubClassOf(Initial ObjectComplementOf(ObjectSomeValuesFrom(InverseObjectProperty(ActivityEdgeProperty)
ActivityNode)))
```

3.5 Path Descriptions

In the following we examine the constructs available for path descriptions in GReQL and transform them into corresponding OWL constructs.

Simple Path Description. In GReQL a simple path description consists of an edge symbol ($-->$ (outgoing), $<--$ (incoming), $<-->$ (direction not important)) and optionally an edge type restriction in curly braces. In the following we define that each Action has some outgoing edge ($-->$) and some incoming edge ($<--$), that all vertices in the graph must be incident with some incoming or outgoing edge ($<-->$), and that each ObjectNode has some incoming edge of type ObjectFlow ($-->\{ObjectFlow\}$).

```
forall v:V{Action} @ exists w:V @ v --> w
forall v:V{Action} @ exists w:V @ v <-- w
forall v:V @ exists w:V @ v <--> w
forall w:V{ObjectNode} @ exists v:V @ w <-->\{ObjectFlow\} v
```

In OWL we define super classes which are used to restrict a given class. Action is restricted by an ObjectSomeValuesFrom class expression, which defines that there is some outgoing edge from each Action to some further vertex via topEdgeProperty. Each Action has some incoming edge via the inverse of topEdgeProperty. An ObjectUnionOf class expression defines that there is some incoming *or* outgoing edge for each Action. To define that each ObjectNode has some incoming edge of type ObjectFlow an ObjectSomeValuesFrom class expression is used to define that there exists some edge of type ObjectFlow, which comes from some Vertex and goes to the given ObjectNode.

```

SubClassOf(Action ObjectSomeValuesFrom(topEdgeProperty Vertex))
SubClassOf(Action ObjectSomeValuesFrom(InverseObjectProperty(topEdgeProperty) Vertex))
SubClassOf(Vertex ObjectUnionOf(ObjectSomeValuesFrom(InverseObjectProperty(topEdgeProperty) Vertex)
    ObjectSomeValuesFrom(topEdgeProperty Vertex)))
SubClassOf(ObjectNode ObjectSomeValuesFrom(objectFlowProperty Vertex)))

```

Edge Path Description. An edge path description $--exp->$ matches exactly one edge, given as expression exp . The following constraint requires the edge e of edge class `ControlFlow` between initial vertex and some other vertex:

```
forall i:V{Initial} @ exists e:E{ControlFlow} @ exists v:V @ i --e-> v
```

In OWL, edge path descriptions are represented by class expressions. In the example below, there is a `ObjectOneOf` class expression containing the individual e of type `ControlFlow`. An `ObjectSomeValuesFrom` expression defines that each `Initial` vertex has some outgoing (fromInverse) edge of type `ObjectOneOf(e)` which goes to some `Vertex`.

```

ClassAssertion(e ControlFlow)
SubClassOf(Initial ObjectSomeValuesFrom(fromInverse ObjectIntersectionOf(ObjectSomeValuesFrom(to Vertex)
    ObjectOneOf(e))))

```

Goal- and Start-restricted Path Description. In GReQL the start and end vertices of a path description can be restricted. A vertex class expression which restricts the start or end vertex is separated from the path description with a `&`. The following restrictions define that for each `ObjectNode`, there is an outgoing and an incoming edge to and from a vertex of type `Action` or `ObjectNode`.

```
forall o:V{ObjectNode} @ exists v:V @ o --> &{Action, ObjectNode} v
forall o:V{ObjectNode} @ exists v:V @ v {Action, ObjectNode}& --> o
```

To restrict start or end vertex of an edge in OWL, the start or end is described by a class expression. To define that for each `ObjectNode` there is some edge that ends at an `ObjectNode` or `Action`, we create an `ObjectUnionOf` class expression that describes the restricted end. We define an `ObjectSomeValuesFrom` class expression to define the existence of an edge going to `ObjectNode` or `Action`. To define that for each `ObjectNode` there is some incoming edge coming from an `ObjectNode` or `Action` vertex we define the restricted start vertex by an `ObjectUnionOf` class expression. The incoming edge is described by the inverse of `topEdgeProperty`. An `ObjectSomeValuesFrom` expression defines that there exists an incoming edge from some `ObjectNode` or `Action` vertex.

```

SubClassOf(ObjectNode ObjectSomeValuesFrom(topEdgeProperty ObjectUnionOf(ObjectNode Action)))
SubClassOf(ObjectNode ObjectSomeValuesFrom(InverseObjectProperty(topEdgeProperty) ObjectUnionOf(
    ObjectNode Action)))

```

Sequential Path Description. GReQL supports the concatenation of path descriptions to sequential path descriptions. The following constraint specifies that all `Initial` vertices are connected v by a path of two outgoing edges.

```
forall i:V{Initial} @ exists v:V @ i -->--> v
```

In OWL, sequential path descriptions are represented by nested class expressions. We define a super class for `Initial` that consists of two nested `ObjectSomeValuesFrom` class expressions. It defines that from each individual of type `Initial` there exists some individual of `Vertex` reachable via a sequence of two `topEdgeProperty` object properties:

```
SubClassOf(Initial ObjectSomeValuesFrom(topEdgeProperty ObjectSomeValuesFrom(topEdgeProperty Vertex)))
```

Exponentiated Path Description. Exponentiated path descriptions are defined by some path description followed by a given natural number. The following example states that for each initial vertex there is an outgoing path of length 2 to some vertex `v`:

```
forall i:V{Initial} @ exists v:V @ i -->^2 v
```

In OWL, exponentiated path descriptions are defined by nested class expressions. To restrict the OWL class `Initial` we define the same superclass as above consisting of two nested `ObjectSomeValuesFrom` class expressions. It defines that from each individual of `Initial` there is some other individual of type `Vertex` reachable via 2 `topEdgeProperty`s:

```
SubClassOf(Initial ObjectSomeValuesFrom(topEdgeProperty ObjectSomeValuesFrom(topEdgeProperty Vertex)))
```

Optional Path Description. In GReQL a path description can be marked as optional by surrounding it with brackets. Here we want to define that each `ObjectNode` has a path of length one or two to some vertex:

```
forall o:V{ObjectNode} @ exists v:V @ o -->[-->] v
```

In OWL optional path expressions can be defined by using an `ObjectUnionOf` expression to represent the optional path. In the following each individual of `ObjectNode` is connected via `topEdgeProperty` with some further individual of type `Vertex`. There can optionally be some further `topEdgeProperty` to another individual of type `Vertex`:

```
SubClassOf(ObjectNode ObjectSomeValuesFrom(topEdgeProperty ObjectUnionOf(ObjectSomeValuesFrom(topEdgeProperty Vertex) Vertex)))
```

Alternative Path Description. In GReQL it is possible to define paths as alternatives by separating them with a pipe. In the following we define that each `ObjectNode` is connected to some `Action` or to some `ObjectNode`:

```
forall o:V{ObjectNode} @ exists v:V @ o --> &{Action} | --> &{ObjectNode} v
```

In OWL alternative path expressions are expressed by using the `ObjectUnionOf` expression. The following constraint defines that each individual of `ObjectNode` has some edge to an `ObjectNode` individual, or alternatively some edge to an `Action` individual:

```
SubClassOf(ObjectNode ObjectUnionOf(ObjectSomeValuesFrom(topEdgeProperty ObjectNode) ObjectSomeValuesFrom(topEdgeProperty Action)))
```

Iterated Path Description. GReQL supports iteration in path description by the use of Kleene operators * and +. In the following we define that there exists a path from every Initial vertex *i* to some Final vertex *f* with *i* and *f* not being identical:

```
forall i:V{Initial} @ exists f:V{Final} @ i -->+ f
```

In OWL iterated path expressions are realized by transitive object properties. In the following we define that each individual of Initial is connected via transitiveTopEdgeProperty with some individual of Final. Since transitiveTopEdgeProperty is transitive the path between Initial individuals and Final individuals can be of arbitrary length.

```
SubClassOf(Initial ObjectSomeValuesFrom(transitiveTopEdgeProperty ObjectSomeValuesFrom(
transitiveTopEdgeProperty Final)))
```

If the iterated path description has edges of specific types, a new transitive object property must be created which is super object property of the chain of the object properties representing the path.

4 Advantages of the TGraph Approach and OWL

In this section we want to present different benefits of both modeling approaches, thus allowing to judge which representation is to be chosen for which purpose.

4.1 Using the TGraph Approach

Relying on grUML and GReQL as modeling and constraint languages offer various advantages not exhibited by the OWL representation, detailed in the following.

Simple syntax. As it is apparent from the examples in section 3, in most cases the GReQL syntax is more concise. This especially applies to regular path expressions whose OWL representations involve intricate nested expressions. Consequently, complex GReQL constraints should be more easily graspable than their OWL counterparts.

Expressions over Attribute Values. GReQL offers the possibility to include expressions over attribute values in constraints, incorporating comparative and arithmetic operations, for instance. This goes beyond the mere assurance that a literal has a specific value or not, which is possible in OWL (exemplified in section 3.3).

Function Library. GReQL provides a function library comprising a variety of functions covering different aspects. The library is extensible, i.e. missing functionality can be added by users. The predefined functions cover aspects such as logics, arithmetics, string manipulation, operations on collections, as well as path and graph analysis. In the following example, the first constraint uses regular expression matching to ensure that the name of activity nodes does not contain the substring “Activity”. The second constraint, consisting of a single function call, imposes the acyclicity of the graph.

```
forall n:V{ActivityNode} @ not reMatch(n.name, ".*Activity.*")
isAcyclic()
```

Efficiency. Some axioms for object property expressions, e.g. symmetry, transitivity, and the existence of equivalent or inverse properties potentially result in the population of ontologies with an abundance of additional object properties during the reasoning process, i.e. the ontology is complemented by properties inferred from previously existing properties. In contrast, the automaton-driven evaluation of regular path expressions by the GReQL evaluator included in the JGraLab API does not require such space-consuming materialization. As explained in [7], the implementation approach also guarantees a time efficient evaluation of constraints and queries. The practical usability of GReQL has been shown in various real-life applications and projects [2].

4.2 Using OWL

The use of OWL provides the following advantages:

Reasoning on Schema Layer. Designers creating graph schemas are possibly interested in computing the vertex classes and edge classes which are not satisfiable, i.e. classes which cannot be instantiated without the graph becoming inconsistent. The following GReQL constraint leads to an unsatisfiable vertex class because it simultaneously forbids and requires that Final vertices have a successor.

```
forall f:V{Final} @ exists v:V @ not(f --> v) and (f --> v)
```

A knowledge base consisting of an OWL ontology provides reasoning on the schema layer. Here we are able to validate the schema and check its satisfiability. The result of this check is an OWL class, e.g. Final.

Extending the above functionality, we can further check the consistency of the combination of various constraints. Assume that the following constraints are given:

```
forall f:V{Final} @ exists v:V @ v --> f
exists f:V{Final} @ not (exists v:V @ f <-> v)
```

According to the transformation pattern we have presented in the last section, the following axioms will be created:

```
SubClassOf(Final ObjectSomeValuesFrom(InverseObjectProperty(topEdgeProperty) Vertex))
EquivalentClasses(FreeStandingFinal ObjectIntersectionOf(Final ObjectComplementOf(ObjectUnionOf(
  ObjectSomeValuesFrom(InverseObjectProperty(topEdgeProperty) Vertex) ObjectSomeValuesFrom(
  topEdgeProperty Vertex))))))
```

With ontology reasoning it can be derived that class FreeStandingFinal is unsatisfiable. Therefore, we know that the two constraints are contradicting.

Reasoning and Expressions covering two layers. OWL allows for defining expressions covering the schema- and instance-layer. The following expressions define that there must be a specific initial node – initialDataInput – preceding each object node. Further, each object node has some incoming and some outgoing object flow edge:

```
Declaration(Individual(initialDataInput))
ClassAssertion(initialDataInput Initial)
EquivalentClasses(ObjectNode ObjectIntersectionOf(ObjectSomeValuesFrom(InverseObjectProperty(fromObjectFlow)
ObjectFlow) ObjectSomeValuesFrom(InverseObjectProperty(toObjectFlow) ObjectFlow) ObjectHasValue(
InverseObjectProperty(transitiveTopEdgeProperty) initialDataInput)))
```

A DL knowledge base allows for joint reasoning on both schema and graph (model), e.g. for classifying individuals to find their possible types. Below, the individuals initialDataInput of type Initial, object1 which has no type and final of type Final are declared.

```
Declaration(Individual(initialDataInput))
Declaration(Individual(object1))
Declaration(Individual(final))
ClassAssertion(initialDataInput Initial)
ClassAssertion(final Final)
ObjectPropertyAssertion(objectFlowProperty initialDataInput object1)
ObjectPropertyAssertion(objectFlowProperty object1 final)
```

Using a reasoner we are able to classify the object1 individual to its possible type. The result is the class ObjectNode, since object1 fulfills all the restrictions defined above.

Open World Assumption. The open world assumption assumes incomplete information as default and allows for validating incomplete models. With regard to quantified expressions, a reasoner by default assumes that a given individual is linked with other individuals at least or only of a given type. Although an individual is not linked with a given number (cardinality) of other individuals a reasoner would assume by default that cardinality restrictions are fulfilled by assumed individuals in the domain. With regard to path expressions, a reasoner assumes incomplete or incorrect paths (e.g. with violated goal- or start-restriction or a wrong length of a sequence of edges) as correct.

5 Related Languages

Similar to grUML and GReQL, MOF and OCL [4, 5] provide constructs to define cardinalities, quantified expressions and boolean connectives. Regarding regular path expressions, OCL is limited. Although the navigation of links between two objects is possible and alternative and optional paths can be expressed by boolean connectives, OCL does not allow for the iteration of paths. Conversely, GReQL does not support OCL's generic iterate expression and is unable to define contexts for constraints. In [12], a bridge between the MOF-based UML and OWL is presented. In principle, the authors transform UML models to an intermediate representation based on the so-called *Ontology Definition Metamodel* and further on to OWL. The reverse direction is also possible.

Alloy is a structural modelling language based on first-order logic supporting structural and behavioral constraints. In [13], a transformation from OWL to Alloy is presented. The Alloy Analyzer tool can provide standard reasoning services such as consistency, satisfiability and subsumption checking. In [14], the ontology is transformed to a Z-specification to prevent automatic assumption of implicit facts to be true. Thus for open world reasoning, a DL-based reasoner must be used.

6 Conclusion

In this paper we presented a transformation of TGraphs and their schemas together with GReQL constraints to OWL 2 ontologies. We show how to bridge both modeling approaches to combine their benefits. The approach features the transformation of graph

and schema elements as well as GReQL expressions such as quantified and regular path expressions to OWL. Further, we have identified individual advantages of both modeling approaches to illustrate the usefulness of the bridging approach. While the TGraph approach provides an efficient implementation and simple usage, OWL 2 allows for strong expressiveness and reasoning support.

The approach has in part been implemented and applied in the *ReDSeeDS* project⁴, where requirements captured as TGraphs, albeit without any GReQL constraints, are transformed to OWL in order to use reasoning services to determine their similarity [15]. Future work includes the validation of the whole approach, i.e. including the transformation of GReQL constraints, on the basis of a case study.

References

1. Mellor, S., Clark, A., Futagami, T.: Model-driven development. *IEEE Software* **20**(5) (2003) 14–18
2. Ebert, J., Riediger, V., Winter, A.: Graph Technology in Reverse Engineering, The TGraph Approach. In: 10th Workshop Software Reengineering (WSR 2008). (2008) 67–81
3. Motik, B., Patel-Schneider, P.F., Parsia, B.: OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. <http://www.w3.org/TR/owl2-syntax> (2009)
4. Object Management Group: Meta Object Facility (MOF) Core Specification, OMG Available Specification, Version 2.0. (2006) <http://www.omg.org/spec/MOF/2.0/PDF>.
5. Object Management Group: Object Constraint Language, Version 2.2. (2010) <http://www.omg.org/spec/OCL/2.2/PDF>.
6. Bildhauer, D., Horn, T., Riediger, V., Schwarz, H., Strauss, S.: grUML - A UML based Modeling Language for TGraphs. Technical report, University of Koblenz-Landau (2010) To appear in *Arbeitsberichte Informatik*.
7. Ebert, J., Bildhauer, D.: Reverse Engineering Using Graph Queries. In: *Graph Transformations and Model Driven Engineering*. Springer (2010) . To appear.
8. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.: *The Description Logic Handbook*. Cambridge University Press New York, NY, USA (2007)
9. Motik, B., Patel-Schneider, P.F., Grau, B.C.: OWL 2 Web Ontology Language Direct Semantics. <http://www.w3.org/TR/owl2-direct-semantics> (2009)
10. Motik, B., Horrocks, I., Rosati, R., Sattler, U.: Can OWL and logic programming live together happily ever after? In: *Proc. ISWC-2006*. (2006) 501–514
11. Parsia, B., Sirin, E.: Pellet: An OWL DL Reasoner. In: *Proc. of the 2004 International Workshop on Description Logics (DL2004)*. CEUR Workshop Proceedings (2004)
12. Gašević, D.V., Djurić, D.O., Devedžić, V.B.: Bridging MDA and OWL Ontologies. *Journal of Web Engineering* **4**(2) (2005) 118–143
13. Wang, H., Dong, J., Sun, J., Sun, J.: Reasoning support for Semantic Web ontology family languages using Alloy. *Multiagent and Grid Systems* **2**(4) (2006) 455–471
14. Dong, J.S., Lee, C.H., Lee, H.B., Li, Y.F., Wang, H.: A combined approach to checking web ontologies. In: *Proc. of the 13th International World Wide Web Conference*. (2004) 714–722
15. Wolter, K., Krebs, T., Bildhauer, D., Nick, M., Hotz, L.: Software Case Similarity Measure. Project Deliverable D4.2, ReDSeeDS Project (2007)

⁴ <http://www.redseeds.eu>