Softw Syst Model (2014) 13:301–321
DOI 10.1007/s10270-012-0250-3

THEME SECTION PAPER

GReTL: an extensible, operational, graph-based transformation language

Jürgen Ebert · Tassilo Horn

Received: 4 April 2011 / Revised: 6 March 2012 / Accepted: 13 April 2012 / Published online: 8 May 2012
© Springer-Verlag 2012

Abstract This article introduces the graph-based transformation language GReTL. GReTL is operational, and transformations are either specified in plain Java using the GReTL API or in a simple domain-specific language. GReTL follows the conception of incrementally constructing the target metamodel together with the target graph. When creating a new metamodel element, a set-based semantic expression is specified that describes the set of instances that have to be created in the target graph. This expression is defined as a query on the source graph. GReTL is a kernel language consisting of a minimal set of operations, but it is designed for being extensible. Custom higher-level operations can be built on top of the kernel operations easily. After a description of the foundations of GReTL, its most important elements are introduced along with a transformation example in the field of metamodel integration. Insights into the design of the GReTL API are given, and a convenience copy operation is implemented to demonstrate GReTL's extensibility.

Keywords Model transformation · Graph transformation · Metamodel merging

1 Introduction

With the advent of *model-driven development* (MDD [1–3]), models and modeling languages have gained considerable attention. Models are viewed as the key software development artifacts which are created, analyzed, refactored, versioned, and maintained like code, the latter being just viewed as yet another model.

Models are written in modeling languages which are defined by *metamodels*. They are instances of these metamodels in the sense that models of a given modeling language have to conform to the language's metamodel in their abstract syntax.

Modeling languages are being developed and adapted to optimally fit to given domains, which lead to lots of domain-specific languages (DSLs [4]), each of which is defined by its own metamodel, concrete syntax, and semantics. With this new trend, the area of *metamodel engineering* has emerged, which treats metamodels as first class entities and deals with their design, refactoring, versioning, and maintenance.

Several graph and model transformation languages provide a well-established means for transforming models and are widely used for automating transformation chains in the course of MDD. Consequently, metamodel engineering generates the need for metamodel transformation languages, which support the transformation of metamodels including the migration of their instance models.

In this paper, we introduce the *Graph Repository Transformation Language* (GReTL) [5], an operational graph-based transformation language that allows the transformation of a source metamodel to a target metamodel by a sequence of create-operations, thereby also migrating existing source instances into corresponding target instances. In this respect, GReTL combines metamodeling with model transformation. If the target metamodel already exists, GReTL can also be used to transform only the instances.

In essence, GReTL is a kernel language defined by a *Java API* that provides a minimal set of elementary transformation operations for constructing metamodels and conforming

Communicated by Dr. Andy Schürr and Arend Rensink.

J. Ebert · T. Horn (✉)
Institute for Software Technology, University Koblenz-Landau, Koblenz, Germany
e-mail: horn@uni-koblenz.de

J. Ebert
e-mail: ebert@uni-koblenz.de

models in parallel. These API operations are used as *transformation objects* referencing the relevant context information. Since the corresponding transformation classes follow the command pattern [6], they can be manipulated and combined to higher-level transformations easily. Thus, it is possible and even encouraged to develop packages of suitable transformations for given (domain-specific) languages by extending and combining the set of transformation operations provided by the API. An example of such an extended transformation operation is illustrated in Sect. 4.3.

The usage of GReTL as a Java API enables exploitation of all Java facilities for the definition and structuring of complicated transformation tasks. Thus, Java can be used to perform extensive intermediate computations to derive powerful comprehensive transformations for larger tasks. For simpler cases, also a concrete DSL for notating GReTL transformations is provided, where transformations are composed as sequences of single transformation operations.

GReTL was developed for the *TGraph technological space* [7,8], where typed, attributed, and ordered directed graphs are used to represent models. In TGraphs, edges are first class elements. They are typed and attributed and can be traversed in both directions. Moreover, there is an order between all vertices and edges in a TGraph, and for each vertex, there is a local order between all incident edges. This fact entails useful properties of TGraphs and makes them amenable for describing global interrelationships between model elements.

GReTL is by no means a rule-based graph transformation language with *match–replace* semantics. GReTL transformations are usually performed out-place, they are guaranteed to terminate, and running a transformation twice on the same source model produces identical target models.

GReTL's conception is to construct a new target metamodel and thereby specify the target graph in terms of extensions (i.e., instance sets) of the new target metamodel's constituents. Since GReTL builds on the formally defined graph query language GReQL [9] for defining these extensions, it is mathematically well founded.

The usage of GReTL as a transformation language and its capabilities are explained along a typical task from metamodel engineering, namely *metamodel merging*. Given two different but related domain-specific languages from the telecommunication domain, we show how their metamodels may be merged into one using GReTL operations, thereby also fusing corresponding pairs of instances of both languages. Applications like this occur when models developed concurrently by different partners in different DSLs have to be merged to enable further cooperative work.

Although GReTL is still under development, the kernel of the language described in this paper can be assumed to be stable. First experiences with GReTL in a architecture migration project[1] have shown that the approach is applicable in practice for transforming graphs consisting of several millions of elements. Furthermore, the live contests of the Transformation Tool Contest (TTC) 2010[2] and 2011[3] could be won. In 2011, the GReTL solution for the TTC reengineering case [10,11] submitted by the second author has won, and GReTL also scored as the second-best solution after the reference solution for the TTC compiler optimization case [12,13]. The solutions can be reproduced online using the excellent SHARE research cloud [14,15].

The remainder of this article is organized as follows: Sect. 2 describes the TGraph technical space with its compositional semantics and gives a first sketch of the conception behind GReTL. In Sect. 3, the use of the GReTL DSL is explained along a non-trivial example of merging two domain-specific modeling languages. Section 4 introduces the GReTL Java framework and its use, describes the execution semantics of GReTL, and exemplifies its extensibility by defining a custom higher-level transformation operation. Section 5 compares GReTL to other transformation languages and Sect. 6 concludes the article.

2 Foundations

GReTL is the transformation language of the *TGraph* technological space [8], where models are represented using a very general and expressive kind of graphs. This section introduces the concept of TGraphs, their schemas, and the relation between schemas and their corresponding graphs.

2.1 Overview

GReTL is defined on typed, attributed, and ordered directed graphs (*TGraphs*). In this article, the ordering is ignored.

A TGraph metamodel (*grUML schema*) specifies sets of conforming TGraphs (a graph class) by defining the abstract syntax of graph instances. grUML (*graph UML*) is a large subset of UML class diagrams[4] comprising only elements that can be given a graph semantics [16].

Figure 1 depicts a TGraph that conforms to the grUML schema in Fig. 2. This graph is used as one of the source graphs for the example transformation discussed in Sect. 3. It is the abstract syntax graph of a model written in BEDSL (Fig. 4 on page 307), a domain-specific modeling language which will be explained in more detail, later.

[1] http://www.soamig.de.

[2] http://planet-research20.org/ttc2010.

[3] http://planet-research20.org/ttc2011.

[4] The readers are expected to have basic knowledge about UML class diagrams.

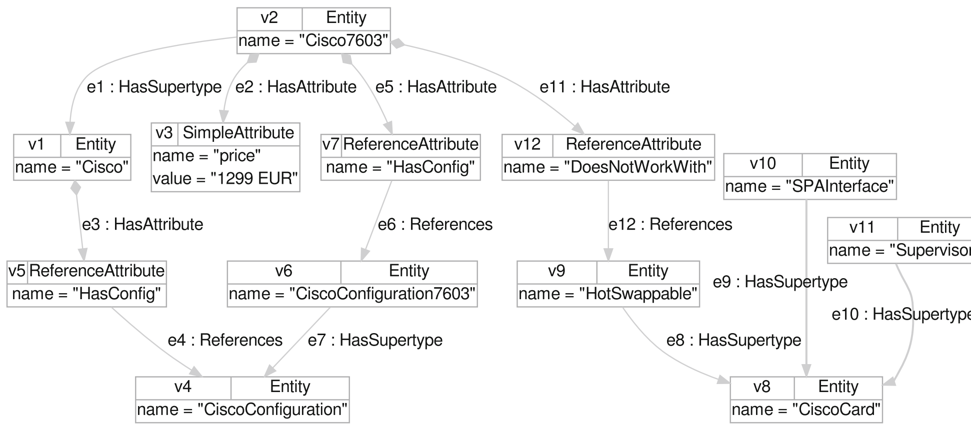

Fig. 1 A TGraph conforming to the schema in Fig. 2

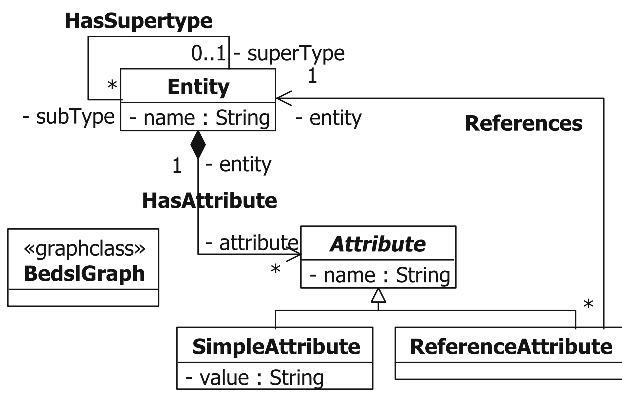

Fig. 2 A schema (source schema) conforming to the metaschema in Fig. 3

In Fig. 1, the TGraph properties (except ordering) are clearly visible. All elements have a type, the edges are directed, and the vertices are attributed. For example, vertex v1 has the type Entity, and its name attribute is set to the string “Cisco”. It is connected to vertex v5 of type ReferenceAttribute via the edge e3 of type HasAttribute. The name attribute of v5 is set to “HasConfig”. In this example, there are no attributes on the edges.

Figure 2 shows the schema defining the graph class BedslGraph. The vertex types (*VertexClasses*) are modeled as UML classes, and edge types (*EdgeClasses*) are modeled as associations. The incidences between associations and classes prescribe the relation between the edge types and the types of their start and end vertices for instance graphs. An additional UML class with a <<graphclass>> stereotype specifies the type of the graph, BedslGraph. Classes and associations may contain attributes specified in the usual UML style, e.g., by using association classes for attributed associations. They prescribe the attribute names and the value domains for instance elements. Supported attribute types (*Domains*) include all common basic types (booleans, numbers, strings, etc.), enumerations and records (also specified in the schema), and homogenous collections of arbitrary other domains (lists, sets, maps). Both vertex and edge classes may specialize other vertex or edge classes, respectively.

2.2 Definitions

Unordered TGraphs consist of five *components*: (i) a non-empty finite set V of vertices, (ii) a finite set E of edges, (iii) an incidence function ϕ assigning a start and an end vertex to each edge, (iv) a type function $type$ assigning a type to each vertex and edge, and (v) an attribute function $value$ assigning to each vertex and edge a set of attribute-value pairs (i.e., a finite partial function from attribute identifiers to values), according to the following definition:

Definition 2.1 (*Unordered TGraph*) Let

- $Vertex$ be a universe of **vertices**,
- $Edge$ be a universe of **edges**,
- $TypeId$ be a universe of **type identifiers**,
- $AttrId$ be a universe of **attribute identifiers**, and
- $Value$ be a universe of **attribute values**.

Then, $G = (V, E, \phi, type, value)$ is an **unordered TGraph**, iff

(i) $V \subseteq Vertex$ is a non-empty finite **vertex set**,
(ii) $E \subseteq Edge$ is a finite **edge set**,
(iii) $\phi : E \to V \times V$ is an **incidence function** assigning a start and target vertex to each edge,
(iv) $type : V \cup E \to TypeId$ is a **type function**, and
(v) $value : V \cup E \to (AttrId \nrightarrow Value)$ is an **attribute function**, where $\forall x, y \in V \cup E$: $type(x) = type(y) \implies \text{dom}\ (value(x)) = \text{dom}\ (value(y))$.

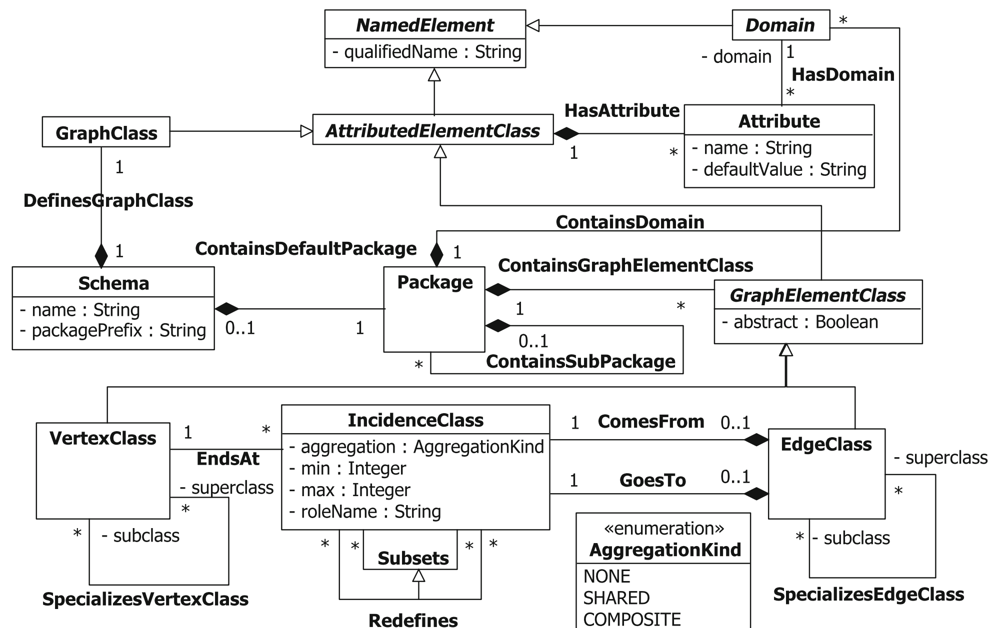

Fig. 3 grUML metaschema, describing all schemas such as in Fig. 2

2.3 Schemas and their constituents

The structure of grUML schemas is defined by the grUML metaschema (Fig. 3), which contains four important, non-abstract *constituents*[5] that define a schema S: (1) VertexClasses, (2) EdgeClasses, (3) Attributes, and (4) specialization hierarchies. Since an EdgeClass uniquely determines two IncidenceClasses and an Attribute uniquely determines a Domain, the latter are subsumed as parts of the former. All other elements of the metaschema are not essential for conforming schemas.

These *four constituents* have the following properties:

1. A VertexClass has a qualified name and may be abstract. With reference to Fig. 2, Entity is a non-abstract VertexClass where the qualifiedName is set to "Entity", and Attribute is an abstract VertexClass with qualifiedName "Attribute".
2. An EdgeClass also has a qualified name and may be abstract. It owns exactly two IncidenceClasses holding the association end properties (multiplicities, role name, aggregation kind) with respect to the VertexClass it ends at. The aggregation kind may be NONE, SHARED, or COMPOSITE. In Fig. 2, References is an EdgeClass between the vertex classes ReferenceAttribute and Entity where the endpoints are IncidenceClass objects whose min and max multiplicity and roleName attributes are set accordingly.
3. An Attribute has a name, an optional default value, and exactly one associated Domain, and it belongs to exactly one vertex or edge class. In Fig. 2, value is an Attribute of the VertexClass with the qualifiedName "SimpleAttribute" and with name set to "value" and no defaultValue.
4. There are two separate (acyclic) specialization hierarchies defined by SpecializesVertexClass and SpecializesEdgeClass. In Fig. 2, the specializations from Attribute to SimpleAttribute and ReferenceAttribute are both of type SpecializesVertexClass, and there are no specializations among edge classes.

With these constituents, a schema describes the available vertex types (as VertexClasses) and edge types (as EdgeClasses), and for each edge type, the valid start and end vertex types including their association end properties are defined. Furthermore, the schema specifies the attributes of all element classes, i.e., vertex or edge classes. The specialization hierarchies imply (transitive) inheritance of attributes and allowed incidences from superclasses to subclasses.

The implementation of the TGraph approach (*JGraLab*) guarantees the conformance of a graph to its schema. Likewise, the implementation guarantees the conformance of a schema to the grUML metaschema.

2.4 Defining a graph

There may be an infinite number of instance graphs *conforming* to a given grUML schema S. But, assuming S is the target schema of a model transformation, *one specific*

[5] Some metaschema elements, e.g., GraphClass, Schema, and Package, are used for structuring grUML diagrams and are not discussed further in this paper.

instance graph has to be created as target graph for a given source graph.

To describe such a specific TGraph G conforming to S, all five components (i)–(v) of G (according to Def. 2.1) have to be defined. We do this in a *compositional* way by defining the *extensions* (instance sets) of the four respective schema constituents (1)–(3) of S respecting (4).

Let $VertexClass$ and $EdgeClass$ be the sets of vertex and edge classes defined by a schema S, let $subtypes$ be a reflexive function that returns all subclasses for a given class, and let G be any graph conforming to S.

Then, the *extensions* in G of the constituents of S are simple mathematical objects, namely sets and functions:

1. There is a set $V_c \subseteq V$ for every vertex class $c \in VertexClass$.
2. There is a set $E_r \subseteq E$ for every edge class $r \in EdgeClass$.
   Assuming that r connects a vertex class c to a vertex class c', there is a function $\phi_r : E_r \to V \times V$, which assigns a tuple (u, w) to every $e \in E_r$ with $type(u) \in subtypes(c)$ and a $type(w) \in subtypes(c')$.
3. There is a function $val_A : \cup_{c' \in subtypes(c)} V_{c'} \to Value$ for every $A \in Attribute$ of $c \in VertexClass \cup EdgeClass$.
   If the attribute A has the domain D and assuming that D denotes a set $T_D \subset Value$, val_A assigns only values from T_D.

For example, the extension of the vertex class ReferenceAttribute in the graph in Fig. 1 is $V_{ReferenceAttribute} = \{v5, v7, v12\}$, the extension of the edge class HasSupertype is $E_{HasSupertype} = \{e1, e7, e8, e9, e10\}$, and the extension of the value attribute of the SimpleAttribute vertex class is $val_{SimpleAttribute.value} = \{v3 \mapsto \text{“1299 EUR”}\}$.

Given these sets and functions, the corresponding TGraph is uniquely determined by $G = (V, E, \phi, type, value)$ with

$$(i)\ V = \bigcup_{c \in VertexClass} V_c, \quad (ii)\ E = \bigcup_{r \in EdgeClass} E_r,$$
$$(iii)\ \phi = \bigcup_{r \in EdgeClass} \phi_r, \ (iv)\ type : V \cup E \to TypeId$$
$$\text{with } \forall v \in V : type(v) = c \iff v \in V_c$$
$$\wedge \forall e \in E : type(e) = r \iff e \in E_r$$
$$(v)\ value : V \cup E \to (AttrId \nrightarrow Value)$$
$$\text{with } \forall x \in V \cup E : value(x)(A) = t \iff val_A(x) = t$$

These definitions are compatible with the model-theoretic semantics of grUML, since the set of all TGraphs conforming to a schema S equals the set of TGraphs composable from extensions of the schema constituents [16].

2.5 Specifying extensions

Archetypes and images In GReTL, the extensions of vertex classes, edge classes, and attributes are specified by so-called *semantic expressions* which are assigned to the create-operations of the respective metamodel constituents as parameters, e.g., a vertex class with three vertices may be described by the expression $\{1, 2, 3\}$.

The semantic expressions define *archetype sets* for vertex and edge classes. For every member of such a set (an *archetype*), a new element is created in the target graph. Thus, every newly created vertex or edge is an *image* of exactly one archetype. It has to be stressed that there is no restriction on what may be chosen as archetype. Although mostly source graph elements are used, archetypes may as well be primitive values (numbers, strings) or composite values thereof (tuples, sets, lists). This also enables the use of GReTL to build new graphs without any source graph at all.

More precisely, the semantic expressions define

1. for each set V_c a set of vertex archetypes $\overline{v}$,
2. for each set E_r a set of triples $(\overline{e}, \overline{u}, \overline{w})$ denoting an edge archetype $\overline{e}$ together with two vertex archetypes $\overline{u}$ and $\overline{w}$ of its start and end vertex,
3. for each attribute function val_A a function assigning a value t to an archetype $\overline{x}$, with the constraints that t has to conform the domain (attribute type) of A, and the image of $\overline{x}$ is a target graph element whose type defines or inherits the attribute A.

Thus, any target graph element is the *image* of exactly one *archetype*.

Maps During the creation of the images for a target schema element class X, two fine-grained functions (img_X from archetypes to images and its inverse function arch_X from images to archetypes) are constructed implicitly. The functions for the element class X also include the mappings of all functions corresponding to an element class Y, iff Y is a specialization of X. Therefore, we can also say that an archetype is an arbitrary object that unambiguously identifies some target graph element within its type hierarchy.

These functions are implemented according to the `java.util.Map` interface. These maps may be accessed by other transformation operations in their semantic expressions. They may also be persisted to keep the information for *traceability* purposes.

Because the functions img_X and arch_X are viewed as maps, they are accessed using methods compatible with the Map interface, i.e., containsKey returns true, iff the map contains an entry for the specified key, and keySet returns a set view of the keys contained in the map.

GReQL In GReTL operations, the semantic expressions are described using the *Graph Repository Query Language GReQL* [9]. GReQL is a sophisticated, dynamically typed graph query language based on set theory and predicate logics, giving access to all TGraph properties and to the schema information.

An important kind of composite expressions in GReQL are *from-with-report comprehensions* as used in the following example query:

```
1 from e: V{Entity}, a: V{Attribute}
2 with e.name = "Cisco7603"
3  and e -->{HasSupertype}* -->{HasAttribute} a
4 reportSet a, a.name end
```

This query delivers a set of tuples, where the first component is an Attribute vertex and the second component is the value of that Attribute's name. The reported attributes are restricted to those contained by an Entity named "Cisco7603" and all its supertypes, i.e., those attributes which are reachable by traversing a path consisting of an arbitrary number of HasSupertype edges in forward direction followed by exactly one HasAttribute edge. When run on the source graph of Fig. 1, the query returns the tuples (v3, "price"), (v5, "HasConfig"), (v7, "HasConfig"), and (v12, "DoesNotWorkWith").

This query also highlights one of GReQL's most powerful features, namely *regular path expressions* [9] which allow for describing complex correlations between vertices using regular operators (sequences, options, alternatives, and iterations). In the example query, the $*$ denotes the transitive closure with respect to edges of type HasSupertype in the outgoing direction with respect to the Entity vertex bound to e.

2.6 Creating the target graph

The conception of GReTL is to create the new target schema operationally, thereby specifying the extensions of each schema element in order to define the target graph. For that purpose, GReTL supplies a set of elementary *create-operations*, one for each kind of constituent of a grUML schema:

(1) CreateVertexClass, (2) CreateEdgeClass, and (3) CreateAttribute.

These operations create the constituents of the target schema and their extensions, the latter being given as a set- or function-valued GReQL semantic expression, respectively. Besides that, the hierarchy information has to be given by additional operations:

(4) AddSuperClass or AddSubClass.

For example, to create a new vertex class Property in the target schema, with one Property vertex per Attribute vertex computed above, one could write the following statement using the GReTL DSL:

```
1 CreateVertexClass Property
2 <== #bedsl# from e: V{Entity}, a: V{Attribute}
3            with e.name = "Cisco7603"
4                 and e -->{HasSupertype}* -->{HasAttribute} a
5            reportSet a end;
```

Here, line 1 describes the necessary syntactic parameters (the name *Property*) to create a vertex class according to Fig. 3, and lines 2–5 contain the semantic GReQL expression which has to be evaluated on the graph with alias bedsl.

Using the Java API, the same effect would be achieved by the creation and execution of a CreateVertexClass object:

```
1 VertexClass property =
2   new CreateVertexClass(context, "Property",
3     "#bedsl#                                        "
4   + "from e: V{Entity}, a: V{Attribute}             "
5   + "with e.name = 'Cisco7603'                      "
6   + "      and e -->{HasSupertype}* -->{HasAttribute} a "
7   + "reportSet a end                                ").execute();
```

Here, the (single) syntactic parameter follows the context parameter in line 2, and the semantic expression is given as string as the last parameter in lines 3–7. Both forms of operations, including their parameters and conventions, are introduced in more detail in the following sections.

3 GReTL for schema merges

As stated in Sect. 1, the usage of models in software development can be considered a standard. During the development of a system, a multitude of models are created on different layers of abstraction and focusing on different aspects. Today, domain-specific modeling languages are widely used for code generation purposes and to restrict the available concepts to those that are supported by a domain.

In this section, the elementary GReTL transformation operations are introduced in the context of a schema merging scenario. Two graphs conforming to two different DSLs (defined by their schemas) are given, and the transformation creates an integrated schema and merges the elements of these source graphs into one target graph.

3.1 A schema merge use case

Comarch[6] is a Polish IT company that is specialized in innovative solutions for the telecommunication industries and uses different domain-specific modeling languages during

[6] http://www.comarch.com.

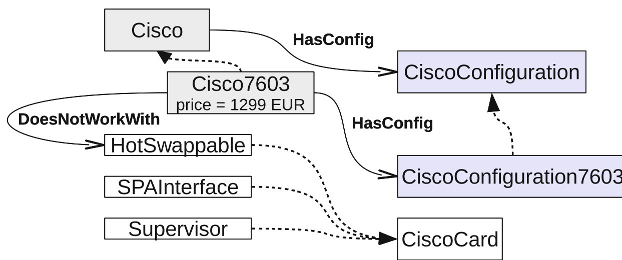

Fig. 4 The source BEDSL model of Fig. 1 in concrete syntax

their development process. Two of them will be considered in a simplified form in this section [17].

BEDSL The *Business Entities Domain-Specific Language* (BEDSL) is a platform-independent modeling language, abstracting from any specific technology and focusing on representing business objects, like entities their attributes and relationships, in a generic manner.

The description is done on the instance level, but the entities are only placeholders of objects in the context of product management and decision support. A simple BEDSL model in a *concrete syntax* is shown in Fig. 4.

The BEDSL schema describing the *abstract syntax* has already been depicted in Fig. 2 on page 303: An Entity has a name and may have an arbitrary number of Attributes. Each Attribute has a name, and it may either be a SimpleAttribute carrying a value or a ReferenceAttribute referencing another Entity. At last, an Entity may be specialized, but only single inheritance is supported. Figure 1 on page 303 shows the abstract syntax graph corresponding to the model of Fig. 4. It is one of the two source graphs used by the example integration transformation in the following.

Figure 4 models an Entity *Cisco* with one subtype *Cisco7603*. The *Cisco* entity has a ReferenceAttribute *HasConfig*. It specifies, that a *Cisco* entity has a *CiscoConfiguration*. Likewise, the entity *Cisco7603* has a *CiscoConfiguration7603*, which is a subtype of *CiscoConfiguration*. For the *Cisco* subtype *Cisco7603*, its price of 1299 EUR is given by the SimpleAttribute *price*, and an additional ReferenceAttribute states that this entity does not work with *HotSwappable CiscoCards*. The Entity *CiscoCard* has three subtypes: *HotSwappable*, *SPAInterface*, and *Supervisor*.

PDDSL The other domain-specific language used is the *Physical Device Domain-Specific Language* (PDDSL). It is used to describe possible connections between physical device elements for a concrete customer in the context of concrete planning of network infrastructures. Here, the description is also on the instance level, but concrete sellable devices are addressed. An example model in a tree-like representation is depicted in Fig. 5.

The PDDSL schema is shown in Fig. 6.

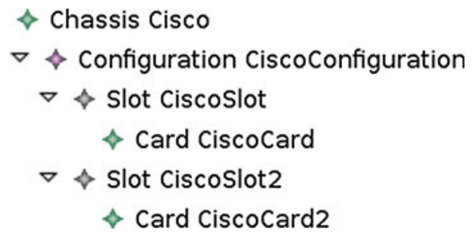

Fig. 5 A sample PDDSL model presented as tree

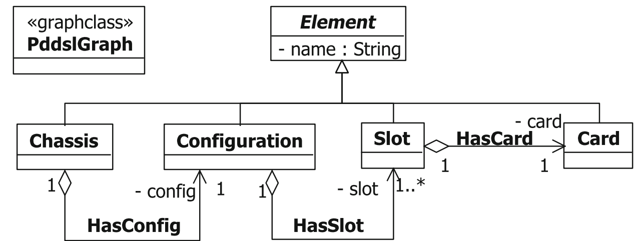

Fig. 6 The PDDSL schema

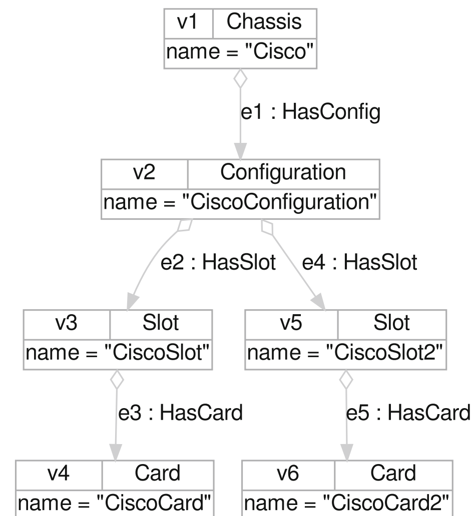

Fig. 7 The PDDSL source graph

Each PDDSL Element has a name. A Chassis has a Configuration, which may have one or many Slots. Each Slot contains exactly one Card.

Figure 7 shows the abstract syntax graph corresponding to the PDDSL model of Fig. 5. In this graph, the Chassis *Cisco* has a Configuration *CiscoConfiguration*, which has two Slots, *CiscoSlot* and *CiscoSlot2*. The former has one Card *CiscoCard*, and the latter has another card *CiscoCard2*.

The two modeling languages describe different aspects of the same things on different levels of abstraction. While the BEDSL model describes relationships between business entities in terms of arbitrary reference attributes and allows for arbitrary simple attributes, the PDDSL model specifies how concrete elements may be plugged together to gain a working configuration. In general, different people with different

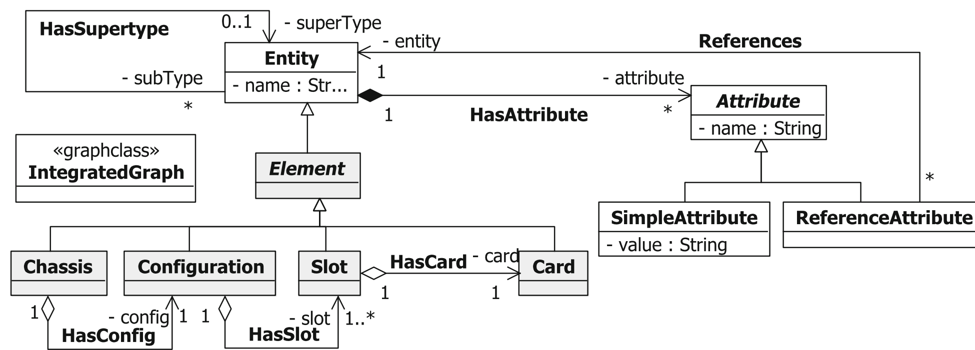

Fig. 8 The target schema the transformation should construct

technical backgrounds and different viewpoints are responsible for creating the models.

3.1.1 Merging BEDSL and PDDSL

In the following, we derive a GReTL integration transformation that merges the two schemas and graphs to gather a complete view on all relevant artifacts of the domain without duplicating or losing information. The intention behind this merging is to achieve an amalgamation of both views, the product management view and the installation planning view.

A target schema which fulfills this intention is visualized in Fig. 8. The transformation exemplified in this section will construct this target schema on its own. The grUML diagram given here for explanation purposes was generated from the target schema constructed by the transformation, and not the other way round.

In the target schema, the PDDSL schema (printed with gray background color) is integrated unchanged, except for leaving out the name attribute of Element which is inherited from Entity here. The rationale for this decision is that PDDSL is the more specific schema, i.e., the different kinds of network devices and their relationships should preferably be modeled with PDDSL instead of the more lax BEDSL. This also adds the requirement that the transformation should identify entities in the BEDSL graph that actually model network devices and represent them using one of the PDDSL types in the target graph. In this example, we assume that entities whose name equals some PDDSL element actually represent the same thing, but in general, this identification may be much more sophisticated.

The BEDSL models may also contain entities that are unrelated to network devices and cannot be represented appropriately as an instance of Element's subclasses; thus, the Entity vertex class is also present in the target schema. Because entities that the transformation identified as a network device, and thus are represented using a subclass of Element in the target graph, might have taken part in HasSupertype relationships or were connected to Attributes, Element must be a specialization of Entity in order not to lose this information. One could argue that the abstract vertex class Element is not needed and Chassis, Configuration, Slot, and Card should specialize Entity directly. This is a justified claim, but by keeping Element one can distinguish network devices from other entities simply by testing if it is an Element instance.

3.2 The schema merge transformation

As described in Sect. 2, the conception of GReTL is to specify the transformation's target schema in conjunction with the extensions of the individual schema element instances which have to be created in the target graph. This means that GReTL combines modeling with transformation. The general guidance to define GReTL transformations is to think about metamodeling the target domain in terms of a class diagram. Before an association (i.e., an edge class) can be defined using CreateEdgeClass, its connecting vertex classes have to be created using CreateVertexClass. Before adding attributes to the classes and associations using CreateAttribute, it makes sense to define the type hierarchies using AddSubClass, because specialized classes and associations inherit their parents' attributes.

All elementary transformation operations in the schema merge transformation example are expressed in GReTL's simple DSL. The use of the Java API in order to extend the language with a higher level operation will be discussed in Sect. 4.

Defining the transformation Any transformation starts by declaring its name.

```
1  transformation BedslPddslMerge;
```

Declaring the source graphs The two source graphs are loaded from files, and to each of them a unique alias is

assigned that can be used throughout the transformation to refer to it.

```
2 AddSourceGraph #pddsl# "pddsl-graph.tg";
3 AddSourceGraph #bedsl# "bedsl-graph.tg";
```

Usually, the setting of source graphs is done externally by a GReTLRunner class that provides a convenient command line interface and allows for batch processing transformations.

Creating a vertex class and vertices The first class created is the abstract vertex class Element.

```
4 CreateAbstractVertexClass Element;
```

Since abstract classes do not have instances, the CreateAbstractVertexClass operation does not have a semantic expression.

The next operation creates the vertex class Chassis originating from the PDDSL schema.

```
5 CreateVertexClass Chassis
6 <== #pddsl# from c : V{Chassis} reportSet c.name end;
```

The semantic expression is evaluated on the PDDSL graph as indicated by the pddsl alias and evaluates to the set of chassis names. These string values are used as the *archetypes* of new chassis vertices in the target graph. For each archetype, a new chassis vertex (its *image*) is created in the target graph. The functions from archetypes to images and the inverse functions are automatically saved by the transformation framework. In all following semantic expressions, these functions are accessible via the maps img_Chassis and arch_Chassis. In later operation calls, we can then access any target graph chassis vertex using the function img_Chassis applied to a source model chassis name.

Creating vertices When looking at the PDDSL graph in Fig. 7, the target graph now consists of a single chassis vertex, which is the image of the string "Cisco", the value of the name attribute of vertex v1. Note that "Cisco" is also the name of vertex v1 in the BEDSL model, and thus can be identified as a chassis. Furthermore, the BEDSL graph's *Cisco7603* entity v2 in Fig. 1 can be considered a chassis too, because it is a subtype of the entity v1, for which we know the type from the PDDSL graph.

The following operation call creates a target graph chassis vertex for any BEDSL entity standing in a subtype relationship to another entity for which we know it is a chassis. The CreateVertices operation only works on the instance level and requires the existence of the given vertex class Chassis which was created by the previous operation.

```
7  CreateVertices Chassis
8  <== #bedsl# from e : V{Entity}, se : e <--{HasSupertype}+
9              with containsKey(img_Chassis, e.name)
10               and not containsKey(img_Chassis, se.name)
11             reportSet se.name end;
```

The variable e iterates over all entities, and se iterates over e's subtypes. The predicate in line 9 restricts e to those entities already identified as chassis by the last operation, and the predicate in line 10 ensures that se's name has no image yet. For all bindings of se, for which the predicates in the with-part hold, the value of the name attribute is chosen as an archetype.

With respect to the BEDSL example graph (Fig. 1), the only archetype contained in the result set is the string "Cisco7603" for which a new target graph chassis is created. Again, the mappings are added to img_Chassis and arch_Chassis. This means that the names of BEDSL entities that we can identify as being a chassis because of their specialization relationships also resolve into a target graph chassis vertex when applied to the img_Chassis function.

Further operations The exact same idiom of a CreateVertexClass operation followed by a CreateVertices operation, where the former creates a vertex class in the target schema and creates vertices for archetypes in the PDDSL graph and the latter creates vertices for BEDSL entities related by subtype relationships is applied for Configuration, Slot, and Card, as well.

The next operation registers the four target schema vertex classes Chassis, Configuration, Slot, and Card as specializations of Element.

```
12 AddSubClasses Element Chassis Configuration Slot Card;
```

This operation has no direct effect on the instance level. However, this specialization relationship specifies that the image and archetype functions of Element are the union of the respective functions of its subclasses. This requires that in each type hierarchy, all archetypes must be disjoint and the GReTL framework will throw an exception if a transformation violates this contract.

Creating an edge class and edges The next operation copies the HasConfig edge class into the target schema. Instances have to start at a Chassis vertex and end at a Configuration vertex. Any chassis must have exactly one configuration and vice versa. The new edge class is defined with aggregation semantics, where the chassis is the whole and the configuration is the part. This schema information is added in textual form to the respective syntactic part in line 14 of the create-operation:

```
13 CreateEdgeClass HasConfig
14 from Chassis (1,1) to Configuration (1,1) role config aggregation shared
15 <== #pddsl# from e: E{HasConfig}
16             reportSet t, t[0], t[1] end
17             where t:= tup(startVertex(e).name, endVertex(e).name);
```

The semantic expression for the CreateEdgeClass operation has to result in a set of triples. The first component in each triple is the archetype of a new edge that will be created in the target graph as its image. The second and third components are the archetypes of the start and the end vertex, respectively. Here, they are the names of the original PDDSL source graph chassis and configuration vertices. Internally, the CreateEdgeClass operation looks up the corresponding images in img_Chassis and img_Configuration and creates new edges connecting them.

Creating edges The BEDSL graph might also contain similar has-configuration relationships modeled as ReferenceAttributes named *HasConfig*. The next operation identifies those and creates further HasConfig target graph edges.

```
18 CreateEdges HasConfig
19 <== #bedsl# from ra: V{ReferenceAttribute}
20             with ra.name = "HasConfig"
21               and not containsKey(img_HasConfig, t)
22             reportSet t, t[0], t[1] end
23             where t:= tup(theElement(<>--{HasAttribute} ra).name,
24                           theElement(ra -->{References}).name);
```

The variable ra iterates over all ReferenceAttributes. The predicate in line 20 restricts those to reference attributes with name "HasConfig", and the predicate in line 21 enforces the bijection property of img_HasConfig. If the previous operation already created a HasConfig edge for an archetype tuple (*Chassis-name*, *Configuration-name*) and thus this tuple is in the domain of the img_HasConfig function, then for the tuple (*Entity1-name*, *Entity2-name*), consisting of the name of the entity containing the reference attribute ra and the name of the entity referenced by ra, no new HasConfig edge must be created.

Further operations The same idiom of one CreateEdgeClass followed by a CreateEdges operation call is applied in the same manner for the edge classes HasSlot and HasCard with the respective multiplicities.

Up to this point, the *target schema* equals the PDDSL schema (without attributes) and the *target graph* contains all elements of the PDDSL graph and additional Chassis, Configuration, Slot, and Card vertices transformed from BEDSL entities which could be identified as belonging to one of those four types because of subtype relationships in the BEDSL graph.

Besides those entities identified by their names, the BEDSL graph may also contain entities that do *not* belong to one of these four types. The next operation creates the Entity vertex class in the target schema and vertices for these leftovers in the target graph.

```
25 CreateVertexClass Entity
26 <== #bedsl# from e : V{Entity}
27             with not containsKey(img_Element, e.name)
28             reportSet e.name end;
```

As archetypes, the semantic expression selects the names of all entities in the BEDSL graph that are not already used as an archetype of some target graph Element vertex.

Next, we make Entity the superclass of Element.

```
29 AddSuperClass Element Entity;
```

Creating an attribute and setting attribute values Having created the vertex and edge classes, the attributes of these classes may be defined and the attribute values of their elements can be assigned.

The following operation creates the name attribute of type String for the Entity class. The semantic expression has to define a function that assigns a value to all Entity archetypes. For each of those archetypes, the corresponding image in the target graph (an Entity vertex) gets the attribute value assigned.

```
30 CreateAttribute Entity.name : String
31 <== #bedsl# from en : keySet(img_Entity)
32             reportMap en -> en end;
```

Because Entity is the top-level class of the created vertex class hierarchy, this operation sets the names of all entities, chassis, configurations, slots, and cards in one go. Since we chose the name attribute values as archetypes, the map assigns each archetype to itself.

Further operations Now, we create the edge class HasSupertype originating from the BEDSL schema.

```
33 CreateEdgeClass HasSupertype
34 from Entity role subType to Entity (0,1) role superType
35 <== #bedsl# from e : E{HasSupertype}
36             reportSet e, startVertex(e).name, endVertex(e).name end;
```

Again, for each HasSupertype instance in the BEDSL source graph, a HasSupertype edge is created in the target graph, starting at the image of the original start entity's name and ending at the image of the original end entity's name.

The next three operation calls create the abstract Attribute vertex class in the target schema, and one of its subclasses, namely SimpleAttribute.

```
37 CreateAbstractVertexClass Attribute;
38 CreateVertexClass SimpleAttribute
39 <== #bedsl# V{SimpleAttribute};
40 CreateAttribute SimpleAttribute.value : String
41 <== #bedsl# from sa : V{SimpleAttribute}
42             reportMap sa -> sa.value end;
```

The SimpleAttributes are copied from the BEDSL source graph by creating a target graph SimpleAttribute for any simple attribute in the source graph and setting the value attribute according to the archetype values.

The ReferenceAttribute vertex class including its corresponding edge classes are the last parts missing in the merged schema.

```
43 CreateVertexClass ReferenceAttribute
44 <== #bedsl# from ra : V{ReferenceAttribute}
45           with not(ra.name =~ 'Has(Card|Slot|Config)')
46           reportSet ra end;
```

As archetypes for new reference attribute vertices in the target graph, the given semantic expression selects those BEDSL source graph reference attributes whose name does *not* match the regular expression "Has(Card|Slot|Config)", because for those, we have already created edges of the corresponding PDDSL types. The =~ operator tests if the string given as first operand matches the regular expression given as second operand. Because GReQL is implemented in Java, the regular expressions are represented as strings that are passed to the java.util.regex.Pattern.compile() factory method. The API documentation of this class defines the exact syntax of regular expressions.

The next two operations declare SimpleAttribute and ReferenceAttribute as subclasses of the abstract Attribute vertex class, and the name attribute declared for all Attributes is set.

```
47 AddSubClasses Attribute SimpleAttribute ReferenceAttribute;
48 CreateAttribute Attribute.name : String
49 <== #bedsl# from a : keySet(img_Attribute)
50           reportMap a -> a.name end;
```

The semantic expression results in a function assigning to each Attribute archetype the value of its name attribute. Thus, all target graph Attribute vertices get assigned the name of their source graph counterparts.

Both simple and reference attributes have in common that they are contained by exactly one entity. Thus, in the following operation call, the containment is specified by declaring the edge class' HasAttribute target vertex class Attribute as composite in this relationship.

```
51 CreateEdgeClass HasAttribute
52 from Entity to Attribute role attributes aggregation composite
53 <== #bedsl# from a : keySet(img_Attribute)
54           reportSet a, theElement(-->{HasAttribute} a).name, a end;
```

The semantic expression results in a set of triples. The Attribute archetypes are used as archetypes for the new edges. The target graph edges start at the image of the name of the given attribute a's container (some BEDSL source graph Entity) and end at the image of a.

Finally, the edge class References from ReferenceAttribute to Entity is created.

```
55 CreateEdgeClass References
56 from ReferenceAttribute to Entity (1,1) role entity
57 <== #bedsl# from ra : keySet(img_ReferenceAttribute)
58           reportSet ra, ra, theElement(ra -->{References}).name end;
```

The ReferenceAttribute archetypes are used as archetypes for the new edges. The edges start at the image of ra and end at the image of the name value of the source graph entity referenced by ra.

Summary The transformation's final merged target graph is shown in Fig. 9.

Summarizing on the *schema level*, the BEDSL and PDDSL schemas have been merged resulting in the merged schema shown in Fig. 8. The essence of the schema merge is the declaration of the abstract Element vertex class originating from the PDDSL schema as a subclass of the Entity class from the BEDSL schema. As a result, the possibility of attributing elements is opened up also for PDDSL elements.

At the *instance level*, the transformation does several actions.

1. The more specific types from the PDDSL graph are preferred. For example, there was a "Cisco" entity in the BEDSL graph (v1 in Fig. 1), and a "Cisco" chassis in the PDDSL graph (v1 in Fig. 7). Because both represent the same thing, the two elements result in exactly one Chassis vertex v1.
2. The PDDSL types of elements that occur only as entities in the BEDSL graph have been inferred by inspecting the HasSupertype relationships. For example, the BEDSL entities "HotSwappable", "SPAInterface", and "Supervisor" (v9, v10, and v11 in Fig. 1) have no counterpart in the PDDSL graph (Fig. 7). However, their supertype "CiscoCard" (v8) is known to be a Card there. Thus, in the target graph, those three vertices are also Card instances.
3. All information added as attributes of BEDSL entities has been transferred into the target graph. For example, the price of the source entity "Cisco6703" (v2 in Fig. 1) is the price of the corresponding chassis v2 in the target graph. Additionally, the information that this entity is known not to work correctly with hot-swappable devices (v9 in Fig. 1) is manifested in the target reference attribute v13.

The target graph contains all information provided by the two source graphs without any duplication. To achieve this, the key points were the use of set- or function-valued semantic expressions and the fact that archetypes can be chosen arbitrarily. Since we used the string values assigned to the name attributes of entities and elements as archetypes here,

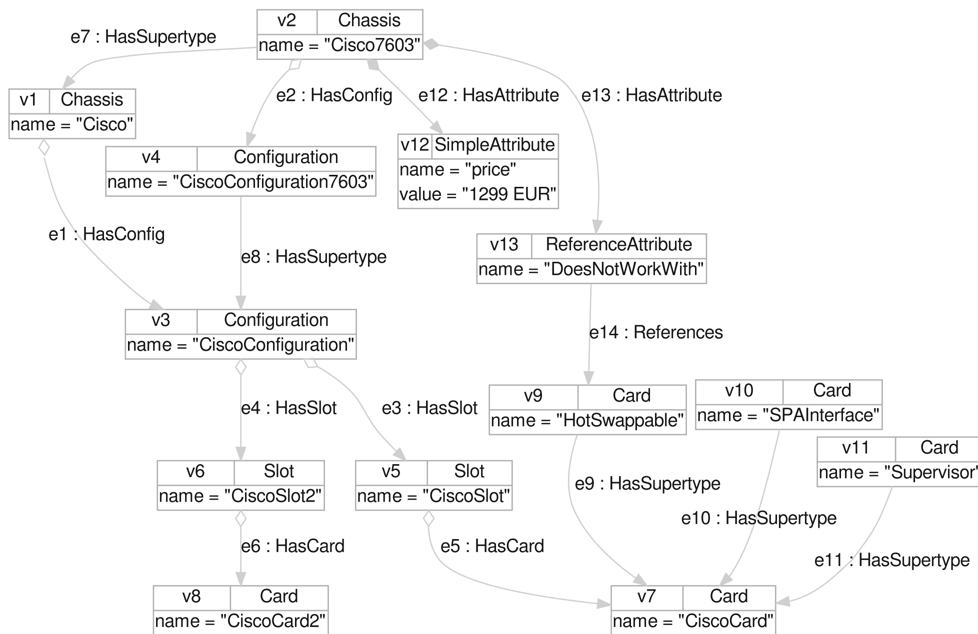

Fig. 9 The final target graph

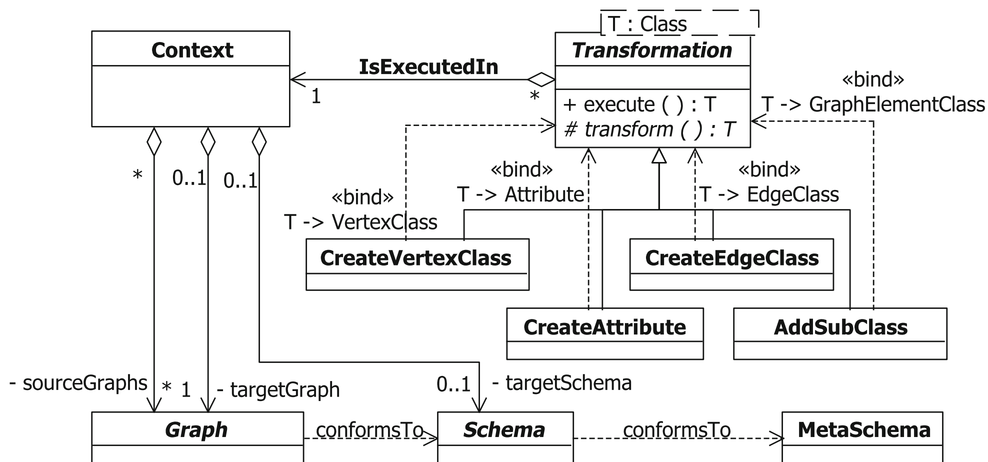

Fig. 10 Transformations as objects

we simulated the *key* concept known from QVT Relations [18] without the need of a new language construct.

4 GReTL as an extensible language

As mentioned in the introduction and in Sect. 2.6, GReTL is implemented as a Java API, which can easily be extended. This section introduces the overall design of the GReTL transformation framework, describes the runtime execution of GReTL transformations, and presents an example of GReTL's extensibility.

4.1 GReTL core design

The structure of the GReTL transformation framework is depicted in Fig. 10.

Transformation is the top-level class of this framework providing a very slim public interface consisting only of the method execute() for executing a transformation. Additionally, it declares an abstract, protected transform() method. This method has to be overridden by concrete subclasses where the transformation's behavior is to be implemented. The Transformation class is defined with a type parameter T, which specifies the return type of its transform() and execute() methods.

The elementary transformation operations CreateVertexClass, CreateEdgeClass, CreateAttribute, and AddSubClass used in the GReTL DSL example transformation in Sect. 3 are such concrete Transformation subclasses. The operations CreateAbstractVertexClass and AddSubClasses are not shown here, because they are only convenience operations built on top of CreateVertexClass and AddSubClass, respectively. Furthermore, even CreateVertexClass and CreateEdgeClass are in fact composites: they perform the target schema element construction themselves, but create and invoke CreateVertices and CreateEdges transformation objects for the instantiation of vertices and edges in the target graph.

Every transformation is executed in some Context. The context is an object that keeps track of the transformation's state, holds the source graphs, and manages the creation of the target schema and the target graph. Since all state information is managed by the transformation's Context, a transformation itself is stateless.

To invoke such a transformation, a transformation object is instantiated and executed using the execute() method. The constructor gets as its first parameter the context object, followed by the schema-relevant (syntactical) parameters and the instance-relevant semantic expression. As an example, the following statement depicts the creation of the *Chassis* vertex class and its extension and is completely equivalent to the lines 5 and 6 of the example transformation in Sect. 3.2.:

```
1  VertexClass chassis = new CreateVertexClass(context, "Chassis",
2          "#pddsl# from c: V{Chassis} reportSet c.name end").execute();
```

Besides the elementary transformations, also larger composite transformations may be implemented directly in Java code. As an example, the whole transformation described in Sect. 3 could been implemented by a class extending Transformation.

```
1  public class BedslPddslMerge extends Transformation<Graph> {
2    public BedslPddslMerge(Context c) { super(c); }
3
4    @Override protected Graph transform() {
5      VertexClass element = new CreateAbstractVertexClass(context,
6          "Element").execute();
7      VertexClass chassis = new CreateVertexClass(context, "Chassis",
8          "#pddsl# from c: V{Chassis} reportSet c.name end").execute();
9      // ...
10     // ... other operation calls follow here ...
11     // ...
12     return context.getTargetGraph();
13   }
14
15   public static void main(String... a) {
16     Context c = new Context("bedsl_pddsl.MergedSchema", "MergedGraph");
17     c.addSourceGraph("bedsl", BedslSchema.instance().loadGraph(a[0]));
18     c.addSourceGraph("pddsl", PddslSchema.instance().loadGraph(a[1]));
19     GraphIO.saveGraphToFile(a[2], new BedslPddslMerge(c).execute());
20   }
21 }
```

Here, the sequence of operations is specified in the transform() method (lines 4–13). The type parameter of the Transformation class is set to Graph, and the transform() method returns the target graph.

For convenience, also a main() method (lines 15–20) is shown, which gets the source BEDSL and PDDSL graph files as its first and second parameter and the file name for the target graph as its third parameter. It creates a context object, which receives the qualified name of the schema to be created and that schema's graph class name. Then, the two source graphs are assigned to the context with their aliases. Finally, an instance of the BedslPddslMerge transformation is created and executed, and the resulting target graph is saved to a file.

Compared to the GReTL DSL, there are only a few differences:

1. All transformations are instantiated using the new keyword.
2. The context is passed explicitly as a parameter.
3. The names of the new schema elements and the semantic expressions are provided as Java strings.
4. The transformations are executed explicitly.

Note that the composite BedslPddslMerge transformation is technically not different from the elementary transformations like CreateVertexClass, because they share the same interface inherited from Transformation. The difference is only conceptual: BedslPddslMerge performs a complete transformation use case with a fixed number of source graphs conforming to fixed schemas and producing a target graph conforming to a complete target schema it creates on its own. In contrast, CreateVertexClass performs only one single step in such a use case and is completely generic.

Executing GReTL transformations The GReTL user should be able to specify a transformation in the DSL, Java, or even a mix of both. Therefore, the DSL interpreter can be invoked from within a transformation.

This interpreter for the GReTL DSL (class ExecuteTransformation) simply parses the file, instantiates transformation objects using reflection, and executes them. The interpreter is a transformation itself, which makes it possible to execute transformations from within other transformations, passing the context of the calling transformation to the called transformation.

```
1  new ExecuteTransformation(context,
2    new File("~/My Transforms/Example.gretl")).execute();
```

Since all transformation classes of the API are also available in the DSL, the possibility of calling DSL transformations from within other DSL transformation is given as well.

```
1  ExecuteTransformation "~/My Transforms/Example.gretl";
```

For executing a transformation written in the GReTL DSL from the command line, a GReTLRunner class exists which requires all context information (source graphs, name of target schema/graph class) by command line options. This allows for executing complex chains of transformations in terms of batch processing.

4.2 Execution semantics

As already discussed in Sect. 2.6, GReTL's conception is to create the target schema incrementally, while specifying the extensions of the new schema elements in order to define the target graph. Thus, the execution of GReTL transformations distinguishes two phases:

- The first phase (*Schema* phase) creates the target schema constituents.
- The second phase (*Graph* phase) creates the extensions of the target schema constituents in the target graph.

The current phase is encoded in the transformation's Context object. The transform() method is called twice, once for the *Schema* phase and the other for the *Graph* phase. The elementary transformation operations' transform() methods act differently depending on the current phase.

- In the *Schema* phase, the elementary transformations create the target schema constituents according to the (syntactical) schema information given as their first parameters. The semantic expressions are *not* evaluated. After this phase, the target schema is fully defined. All vertex and edge classes exist, all attributes are assigned to their classes, and the specialization relationships between vertex and edge classes are established.
- The *Graph* phase starts with generating Java code for the target schema created in the previous phase. This generation is initiated by the switch between the phases. The generated code is compiled in memory, and the target graph is instantiated as an instance of the graph class specified by the schema.
  Then, the elementary transformation operations retrieve the schema element they have created in the *Schema* phase by the qualified name provided to all operations as parameter. The other parameters specifying schema properties are ignored. The semantic expressions are evaluated on the respective source graph, and the creation of vertices and edges and the assignment of attribute values are performed.

This two-phase execution is fully transparent to users if the transform() method only contains invocations of elementary transformation operations. If the transform() method of a composite transformation should also contain other code besides pure transformation invocations, it should be considered that the method will be run twice, e.g., expensive calculations should run only in the phase where the result of the calculations are needed by checking the value of context.getPhase().

It should be noted that the *Schema* phase is skipped if the target schema already exists. For example, when transforming a set of graphs in a sequence, then only the first execution creates the target schema and all following executions simply reuse it.

4.3 Extension operation: copying vertex classes

The encapsulation of transformations in transformation classes allows defining composite transformations which deliver higher level transformation services.

As an example, imagine a CopyVertexClass operation, which can be used to copy a source schema vertex class into the target schema, where all attributes defined for the source vertex class shall be copied as well. At the instance level, this operation should create one target graph vertex for every source graph vertex, using the source vertices as archetypes.

This operation allows for replacing one CreateVertexClass operation call followed by arbitrary many CreateAttributes operation calls with one single instruction. For example, the sequence of operation calls

```
1 CreateVertexClass SimpleAttribute
2 <== #bedsl# V{SimpleAttribute};
3
4 CreateAttribute SimpleAttribute.value : String
5 <== #bedsl# from sa : V{SimpleAttribute}
6              reportMap sa -> sa.value end;
```

specified in the example in Sect. 3.2 could be replaced with one single operation call:

```
1 CopyVertexClass #bedsl# SimpleAttribute;
```

The operation expects the alias of the source graph, whose schema contains the vertex class to be copied. The qualified name of the vertex class itself is given as the second parameter.

Specifying the operation's semantics in Java Given the DSL syntax for the new operation, it can be implemented as a Java transformation class. We derive the new operation class from Transformation, and we set its type parameter to VertexClass in analogy to CreateVertexClass.

```
1 public class CopyVertexClass extends Transformation<VertexClass> {
2   private VertexClass sourceVC; private String alias;
3   public CopyVertexClass(Context c, VertexClass sourceVC, String alias){
4     super(c); this.sourceVC = sourceVC; this.alias = alias;
5   }
```

The constructor requires the mandatory context object, the source vertex class to be copied, and the alias of the source graph which contains the vertex instances to be copied.

The behavior of the new operation is specified in its transform() method. The important point here is that we implement it by composing already existing elementary operations.

```
 6  @Override protected VertexClass transform() {
 7    String qname = sourceVC.getQualifiedName();
 8    VertexClass targetVC = new CreateVertexClass(context, qname,
 9      "#" + alias + "# V{" + qname + "!}").execute();
10    for (Attribute sourceAttr : sourceVC.getOwnAttributeList()) {
11      Domain d = new CopyDomain(context,sourceAttr.getDomain()).execute();
12      new CreateAttribute(context,
13        new AttributeSpec(targetVC, sourceAttr.getName(), d),
14        "from v: keySet(img_" + qname + ")                    "
15      + "reportMap v -> v." + sourceAttr.getName() + " end").execute();
16    }
17    return targetVC;
18  }
```

First, CreateVertexClass creates a vertex class with the same qualified name in the target schema. The semantic expression expands into *V{QName!}* where *QName* is the qualified name of the source vertex class. This expression evaluates to the set of vertices which are direct instances (note the exclamation mark) of the vertex class *QName*.

Thereafter, one CreateAttribute operation is executed for each attribute defined for the copied vertex class. The semantic expression evaluates to a function that assign to every archetype vertex v its attribute value in the source graph. Thus, the attributes are simply copied over, and the same applies to the attribute values on the instance level.

Note that the implementation of transformations as Java classes allows to use the full power of Java to compute any relevant information. Every elementary transformation operation has a constructor that receives a semantic expression as GReQL string, which is evaluated by the transformation object when it is executed. Additionally, all GReTL operations have another constructor that directly receives an already calculated result. Thus, instead of using GReQL to specify the vertex archetypes and the value assignment function for attributes, it is possible to compute those algorithmically in Java and pass the calculated results to the transformation constructor.

Making the operation callable from the GReTL DSL Until now, the implementation of CopyVertexClass is complete with respect to the Java API. To make it also usable in the GReTL DSL, the operation has to provide an additional parseAndCreate() factory method that parses the operation arguments and returns a transformation instance. The method receives the current GReTL interpreter of type ExecuteTransformation as a parameter which provides a set of matching methods.

```
19  public static CopyVertexClass parseAndCreate(ExecuteTransformation et) {
20    String alias = Context.DEFAULT_SOURCE_GRAPH_ALIAS;
21    if (et.tryMatchGraphAlias())
22      alias = et.matchGraphAlias();
23    String qname = et.matchQualifiedName();
24    VertexClass sourceVC = et.context.getSourceGraph(alias).getSchema()
25        .getGraphClass().getVertexClass(qname);
26    return new CopyVertexClass(et.context, sourceVC, alias);
27  }
```

Because the CopyVertexClass operation is generally useful, the source graph alias is made optional here, so that it can be omitted if there is only one source graph. In that case, the default alias is used. The vertex class to be copied is retrieved from the source graph's schema, and finally a new CopyVertexClass instance is instantiated with the interpreter's context and returned.

In only 27 lines of code, we have added a completely new transformation operation to the GReTL framework by composing it using the elementary operations CreateVertexClass and CreateAttributes. The new CopyVertexClass operation is convenient, whenever a transformation scenario requires the unmodified transfer of several vertex classes from source schemas into the target schema including the migration on the instance level.

Except that a custom transformation has to extend the Transformation class and override its abstract transform() method, GReTL does not place any restrictions on its behavior. For most cases, it suffices to compose the elementary operations in some way, but it may also introduce completely new semantics and even a new syntax for its arguments.

Although not the topic of this article, it should be mentioned that the GReQL query language is also extensible. To add a new function to GReQL, one has to extend an abstract Function class and implement an evaluate() method with arbitrary arguments, returning an arbitrary object as a result. So if some information needed in the semantic expressions given to the GReTL operations can better be calculated algorithmically or even requires communication with auxiliary services, there is nothing hindering users from doing so.

The complete power of Java with its huge ecosystem and thousands of libraries can always be exploited in both GReQL queries and GReTL transformations.

5 Related work

In this section, GReTL is compared to today's most relevant and widely used transformation languages. A coarse-grained distinction can be made between whether a transformation language is especially targeted to a certain use case (*special-purpose*) or whether it is general enough to preform arbitrary transformations (*general-purpose*).

First, we give a brief overview of the field of coupled evolution which is related to GReTL's idea of transformations

on both metamodels and models. We then outline the general concepts of graph rewriting systems pointing out the major differences compared to GReTL, and finally we compare GReTL to the most prominent model transformation languages.

Because GReTL is unidirectional and operational, we omit a discussion on logic-based transformation languages like QVT Relations [18]. Since triple graph grammar approaches like the TGG Interpreter [19] or MOFLON [20] are very similar to graph rewriting systems when applied as forward transformations, we only discuss them briefly at the end of the graph rewriting paragraph.

5.1 Comparison with coupled evolution approaches

Two special-purpose approaches tailored to *coupled evolution* of metamodels and models are Epsilon Flock [21] and COPE [22]. Their intent is to ease the creation of transformations that migrate models conforming to a metamodel version *a* to models conforming to a (usually newer) metamodel version *b*. Usually, the differences between successive metamodel versions are relatively small compared to the commonalities. With Epsilon Flock and COPE, transformation developers have to specify the transformation only for the elements whose metamodel type has been changed between versions. Thus, the size of the transformations scales with the amount of differences between the metamodel versions instead of the size of the metamodels.

Epsilon Flock implements a model migration approach named *conservative copy*. With that approach, all elements that conform to both the source and the target metamodel are implicitly copied to the target model. Rules need to be specified only for elements directly affected by the metamodel changes.

COPE is not a separate transformation language, but a complete modeling workbench integrated into the Eclipse Ecore editor. It provides a set of about 60 so-called *reusable coupled operations* that are used to evolve a given metamodel interactively. Examples for such operations are renaming a class or extracting an abstract superclass. The sequence of operations applied to the Ecore metamodel in the editor is recorded in a history from which model update transformations can be generated. Thus, when metamodeling using the COPE workbench, the update transformations from any metamodel version *a* to a later version *b* emerge as a by-product of modeling.

GReTL is not especially targeted at coupled evolution of metamodels and models, but its emphasis lies on the incremental and parallel construction of target schema and graph which can be seen as a generalization of coupled evolution. In Sect. 4.3, a generic copy operation for a vertex class and its instances has been implemented to demonstrate how to extend the language. Likewise, a similar operation could be implemented that copies an edge class into the target schema and its instances into the target graph. If we wanted to evolve a schema, we could use GReTL to do so. First, we apply the generic copy operations to the schema elements that should stay as is, and thereafter we can specify the changed parts of the schema using the elementary operations introduced in Sect. 2.6 and exemplified in Sect. 3.2. This is similar to the COPE approach in that the evolution is specified in terms of operations applied to the source metamodel version. Of course, such a GReTL transformation would be specific to exactly one source schema version and generate exactly one target schema version with a conforming graph, whereas COPE is capable of generating update transformations from any metamodel version to any later version.

A coupled evolution approach more similar to Epsilon Flock could be developed by integrating a schema comparison approach into a new GReTL operation. This operation would receive a schema of a base version, a graph conforming to this base version, and a schema of another version. It would calculate the schema elements that did not change between the versions and copy the given source graph instances into the target graph conforming to the other schema version. Again, only operations for elements whose type has been modified in the schema evolution would need to be specified.

5.2 Comparison with graph rewriting systems

Today, graph transformation languages in the sense of *graph rewriting systems* are widely applied in the fields of model transformation [23,24] as well as simulation and verification [25–27], and program optimization [28,29].

The basic building blocks of graph transformations are rules consisting of a left-hand side and a right-hand side. The left-hand side is a graph pattern, and the right-hand side is a replacement graph. Both sides consist of symbols for nodes and edges connecting those nodes. When a rule is applied, one arbitrary occurrence of a subgraph matching the graph pattern is searched in the host graph and replaced by an instance of the replacement graph. The symbols occurring both in the graph pattern and in the replacement graph denote nodes and edges that are preserved by a rule application. Symbols occurring only in the graph pattern denote nodes and edges that are deleted, and symbols occurring only in the replacement graph denote nodes and edges that are created by a rule application. Most languages support negative application conditions (NAC) in the left-hand side, which forbid the rule's application if they can be matched in the host graph.

For specifying the control flow of a transformation, many graph rewriting systems provide separate rule application languages. For example, PROGRES [30] specifies the order

of rule applications using so-called transactions, which are in essence transformation procedures triggering rule applications or other transactions, by using control structures like conditional application, nondeterministic choice, and iterated application. Fujaba[7] and MDELab[8] use story diagrams [31], which is a visual language similar to UML activity diagrams. VIATRA2 [32] uses abstract state machines [33], and GrGen.NET [34] provides a custom graph rewrite sequence language based on logical and regular expressions. Henshin [35,36], the successor of the Tiger EMF Transformation Project [37], provides a visual rule application control language similar to graph rewrite sequences, where different control units can be nested inside each other. There are units for sequential application, application in arbitrary order, iterated application, conditional application, priority-based application, and so called amalgamation units providing a forall operator. AGG [38] supports priority-based layering of rule application, where all rules of the layer with highest priority are applied as long as possible, then all rules of the following layer, etc. The transformation terminates either when no rule of the lowest layer can be applied anymore, or optionally if no rule of no layer can be applied. Furthermore, users can interactively select a rule to be applied in a debugger-like stepwise execution mode.

Graph transformation languages focusing on enumerating all possible graphs that can be generated from a given start graph by applying a set of rules usually execute all applicable rules in parallel. These languages are used for verification purposes, i.e., the system is modeled as a graph, and the dynamic semantics of a system is modeled in terms of rules that transition the system from one state into another. If two rule application sequences result in isomorphic graphs, they transit to the same state. For each state, invariants may be checked, or properties like confluence can be proven. Languages used for such purposes, although not limited to this scenario, are for example GROOVE [39], AGG, and Henshin.

The kind of graphs used by the individual graph transformation tools varies from simple directed graphs with nodes and edges over labeled graphs, graphs with attributed nodes and/or edges to typed and attributed graphs with inheritance between node types [40]. The types available in a graph are defined by a type graph or schema. Often, visual metamodeling approaches similar to UML class diagrams are used to define schemas. In the TGraph technological space on which GReTL is based, graphs are ordered in addition to being typed and attributed. Furthermore, inheritance is not restricted to node types, but possible also for edge types.

One major difference between graph rewriting systems and GReTL is that the former always work in-place, i.e., they modify their source graph directly, and source and target graphs are the same. This also means that there cannot be many input graphs to a transformation, as it was the scenario in the example of Sect. 3. Of course, it is possible to copy the elements of many graphs into one, but then the identities of the individual graphs are lost, so that additional constructs are needed to restrict pattern matching to a subgraph that has been a single input graph before. VIATRA2 has a special keyword `below`, which can be used to restrict an element to be contained directly or indirectly in some other element for purposes like that.

GReTL transformations usually work out-place. They construct a completely new target graph (including a new target schema) out of arbitrary many source graphs, where arbitrary many also includes zero. GReTL transformations may also use the given source graph as target graph, but then they are restricted to the instance level, i.e., they cannot modify the schema.

The in-place nature of graph rewriting systems has the consequence that they are, in principle, restricted to endogenous transformations, i.e., transformations whose source and target graphs conform to the same schema. However, in practice this restriction is circumvented by the use of a combined schema encompassing at least source and target schema plus optionally node and edge types used only during the execution of a transformation. This makes graph rewriting systems also applicable for typical model transformation tasks, where the schemas of source and target model differ.

Graph rewriting systems use pattern matching to locate one single occurrence of a rule's left-hand side in the host graph which is then replaced by the right-hand side. Instead of pattern matching, GReTL uses graph querying on the source graphs to compute sets of arbitrary archetypes, and for each archetype a new target graph element is created. For many use cases, this novel concept enabled us to define very concise and elegant transformations like in [11], but it is not suited out-of-the-box for most typical graph rewriting domains like optimization and verification. However, because GReTL is extensible, it can be adapted to such domains as well. For solving the TTC 2011 compiler optimization case [12], some custom in-place operations were developed to make GReTL fit this purpose [13].

It should be noted that although we do not compare GReTL with TGGs in detail here, most differences between GReTL and graph rewriting systems apply in a similar vein for TGGs. A TGG forward (i.e., uni-directional) transformation is similar to a graph rewriting transformation, except that it does not work in-place. However, the mentioned main differences apply: TGGs use pattern matching, whereas GReTL is based on querying, and GReTL is able to construct a new metamodel and instantiate a conforming graph, whereas TGGs assume a fixed, pre-existing target metamodel.

[7] http://www.fujaba.de.

[8] http://www.mdelab.com.

5.3 Detailed comparison with model transformation languages

There are only a few widely used *general-purpose* transformation languages, the most well known being ATL [41], the Epsilon Transformation Language (ETL, [42,43]), and the QVT languages Operational Mappings (QVTo) and Relations [18].

GReTL claims to be competitive to ATL, ETL, and QVTo. Therefore, we compare GReTL to these languages in more detail along the main language properties.

GReTL is meant to be a *graph-based* transformation language. It shares with the competitors the property that transformations are usually executed out-place, i.e., a new target model is created from a given input model. The property of transformations also constructing the target metamodel is a novel concept of GReTL. All other model transformation languages require a pre-existing target metamodel.

ATL, ETL, and QVTo as well as GReTL support transformations using *more than one* input model simultaneously. Because of its flexible archetype concept which only requires that the semantic expressions given to the GReTL operations evaluate to sets of arbitrary objects, GReTL also allows the creation of a new target graph with no source graph at all. ATL, ETL, and QVTo allow for transformations that create more than one target model, too, whereas GReTL transformations are restricted to one single output graph at the current point in time.

While all cited transformation languages are *rule-based* with several differences in rule application semantics, e.g., imperative (QVTo, ATL called rules) versus declarative (relations, ATL matched rules, ETL rules), the concept of specifying the target graph of a transformation by defining the extensions of the target schema constituents is a novel property of GReTL.

ATL and ETL rules have a source pattern declaring identifiers with source metamodel types and a target pattern that specifies elements to be created. In the case of ETL, rules can only declare one input element in their source patterns. The source pattern may also contain constraints, and for all matches of the source pattern in the source model and fulfilling the constraints, the elements in the target pattern are created in the target model.

In contrast, QVTo and GReTL are based on querying. QVTo uses the Object Constraint Language (OCL, [44]) whereas GReTL uses the GReQL language. We believe that GReQL provides more expressive constructs than most other querying languages and comparable components of other transformation languages. Especially when very complex, non-local connections of elements with arbitrary distance in terms of edges in between have to be described, and GReQL's *regular path expressions* provide a powerful means.

When not considering the optional creation of the target schema, GReTL's operational *execution semantics* are quite similar to those of QVTo. For the latter, starting with a top-level mapping operation, sets of source model elements are selected using OCL, and for each of them another mapping is applied, creating a new target model element and possibly applying further mapping operations. Usually, this call hierarchy of mapping operations is aligned to the containment hierarchy of the source model. With GReTL, there is no hierarchy of operation calls. The semantic expressions define the archetype sets for each operation invocation, and similar to QVTo, for any member in these sets, a new target graph element is created.

The strict *separation of concerns* in GReTL's concept of compositional semantics with respect to the constituents of a schema results in a very slim set of only four transformation operations which suffice to perform arbitrary transformations. These operations have a very simple and clear semantics, and can easily be combined to more expressive and convenient operations, as the example in Sect. 4.3 demonstrates. In contrast, all cited languages mix several concerns. Rules create output elements for some input elements and additionally set the new elements' attribute values and assign references or create edges between elements. This may lead to duplicate code in rules creating instances of subclasses of some common superclass. Therefore, ATL, ETL, and QVTo provide advanced concepts like *rule inheritance* that allow for aligning a transformation towards the type hierarchy of the target metamodel. This way, inherited attributes and references are set by some parent rule, and specialized rules only have to deal with the direct properties of their output elements. However, the implementation of rule inheritance differs across languages [45]. For example, while ETL supports multiple inheritance between rules, ATL and QVTo support only single inheritance.

The usual *extension* and *reuse* mechanism supported by most transformation languages are *libraries*, which consist of *helper functions* and sometimes even rules. These helpers (and rules) can then be used in transformations. Beyond that is the *black-box implementation* concept defined by the QVT standard. It allows for implementing a QVTo mapping operation with an arbitrary MOF operation [46] with the same signature and semantics. However, GReTL allows for even more extensibility, because an extension operation developer is mostly free both in the choice of the syntax in the GReTL DSL and the semantics of the new operation. We acknowledge that there are scenarios, where the conceptual purity of GReTL's elementary transformation operations is less convenient. But GReTL was designed with *extensibility* in mind. An extended convenience operation like CopyVertexClass discussed in Sect. 4.3 is usually implemented in only a few lines of Java code.

With respect to *traceability*, the image and archetype functions that are created automatically during the execution of a GReTL transformation form a very fine-grained traceability model. They provide the possibility for navigating from any target graph element (image) to its archetype, and vice versa. These functions can be persisted as an XML file after the transformation has succeeded and loaded back later in order to navigate between target and source graphs. ATL, ETL, and the implementations of the QVT languages have similar traceability concepts, in that for each rule the mappings from input to output elements are retained and can also be persisted. The main difference is that in GReTL, the archetypes can be chosen freely and are not restricted to source graph elements.

6 Conclusion and future work

This article introduced the graph-based GReTL transformation language with a focus on its concrete syntax. A non-trivial example (merging the metamodels of two domain-specific languages including the migration of their instances graphs) was explained in detail to demonstrate the power of the approach. (A simpler introductory example is discussed in [5]).

Also, a brief insight into its design as a Java API was given, and, using that, an extended copy operation was exemplarily implemented to show GReTL's extensibility.

Based of the concept of compositional semantics of TGraph schemas, GReTL introduces a novel concept to model transformations. GReTL transformations follow the conception of constructing the target schema, while simultaneously creating the target graph as an extension.

GReTL's API supplies a basis for the definition of expressive transformations that support frequently occurring transformation tasks appropriately. The encapsulation of transformations in plain Java transformation objects supports this goal, facilitating reuse and composition in terms of inheritance and nesting.

We see several advantages for using the GReTL transformation language:

- GReTL is operational. Thus, it can be included in arbitrary (Java) application code.
- GReTL is formal, since it is built on TGraphs, grUML, and GReQL, which are formally defined.
- GReTL builds on the powerful GReQL querying language. Thus, GReTL transformations can use arbitrary information extracted from the source graph. Because GReQL has proven to be highly efficient for complex querying of large graphs, the same can be said for GReTL for the transformation of large graphs.
- GReTL builds on a powerful conception. Since grUML has compositional semantics, the full power expected from a transformation language is granted by a kernel of four elementary operations.
- GReTL DSL has an easy syntax. Thus, a quick notation of simply structured transformations is possible.
- GReTL is embedded in Java. Thus, arbitrary computations can be executed to steer the transformation operations.
- GReTL transformations can be executed from the command line using a convenient interface which enables batch processing of transformations.
- GReTL is extensible. Higher-level operations that provide more expressiveness or are especially suited for a particular domain can be built on top of the provided elementary operations.
- GReTL supports full traceability, since the img/arch-maps are available and can be persisted. Thus, traceability information can be used inside the transformation code and later on.

First promising experiences with GReTL have been achieved when using it in a reengineering project, where Java software was parsed into TGraphs representing the abstract syntax of the source code. Here, GReTL transformations have been used for the extraction of state machines that had been implemented in plain Java using a set of conventions. The input graph to this transformation had about 2.5 million vertices and edges and it could still be executed on a usual laptop in a few seconds.

Furthermore, the Transformation Tool Contests 2010 and 2011 were won using GReTL and other TGraph technologies like GReQL.

In future work, further issues have to be tackled, e.g., better applicability for endogenous in-place transformations, and the inclusion of ordering information. Also, instead of having transformations that construct new schemas including conforming graphs, the concepts should be extended to support modification of existing schemas including the parallel migration of existing graphs.

Although GReTL's concepts were derived from the formal definition of TGraphs and are aligned to the constituents that make up a grUML schema (vertex classes, edge classes, and attributes), it seems feasible to apply them on other technological spaces like EMF [47] as well. A grUML vertex class is equivalent to Ecore's `EClass`, and grUML attributes correspond to Ecore `EAttributes`. The main difference is that EMF does not consider edges as first class objects, but relations between elements are expressed as (pairs of) `EReferences` instead. However, it is still possible to imagine an operation that creates a reference (and an opposite reference) in an Ecore metamodel, and assigns that reference for elements in an instance model with a provided set

of tuples specifying the archetypes of the elements that have to be connected. Using EMF technology, the Object Constraint Language [44] or EMF Model Query [48,49] could be used for specifying semantic expressions.

Because the GReTL archetype concept decouples the retrieval of source model information from the creation of target model elements, it seems feasible to allow for transformations crossing the borders of technological spaces, e.g., archetypes calculated on a source TGraph using GReQL could be passed to transformation operations creating elements in an EMF model and vice versa.

References

1. Bézivin, J.: Model driven engineering: an emerging technical space. In: Lämmel, R., Saraiva, J.A., Visser, J. (eds.) Generative and Transformational Techniques in Software Engineering. Lecture Notes in Computer Science, chap. 2., vol. 4143, pp. 36–64. Springer, Berlin (2006)
2. Object Management Group: MDA Guide Version 1.0.1. (2003)
3. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley/Longman, Boston (2003)
4. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Not. **35**, 26–36 (2000)
5. Horn, T., Ebert, J.: The GReTL transformation language. In: Cabot and Visser [50], pp. 183–197
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, Boston (1995)
7. Ebert, J., Riediger, V., Winter, A.: Graph technology in reverse engineering. The TGraph approach. In: 10th Workshop Software Reengineering (WSR 2008). GI Lecture Notes in Informatics, vol. 126. GI (2008)
8. Kurtev, I., Bézivin, J., Aksit, M.: Technological spaces: an initial appraisal. In: CoopIS, DOA'2002 Federated Conferences, Industrial track (2002)
9. Ebert, J., Bildhauer, D.: Reverse engineering using graph queries. In: Graph Transformations and Model Driven Engineering. LNCS, vol. 5765, Springer, Berlin (2010)
10. Horn, T.: Program understanding: a reengineering case for the transformation tool contest. In: Van Gorp et al. [51]
11. Horn, T.: Solving the TTC 2011 reengineering case with GReTL. In: Van Gorp et al. [51]
12. Buchwald, S., Jakumeit, E.: Compiler optimization: a case for the transformation tool contest. In: Van Gorp et al. [51]
13. Horn, T.: Solving the TTC 2011 compiler optimization case with GReTL. In: Van Gorp et al. [51]
14. Horn, T.: SHARE demo related to the paper Solving the TTC 2011 Reengineering Case with GReTL. http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu_10.04_TTC11_gretl-cases.vdi
15. Horn, T.: SHARE demo related to the paper Solving the TTC 2011 Compiler Optimization Case with GReTL. http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu_10.04_TTC11_gretl-cases.vdi
16. Walter, T., Ebert, J.: Foundations of Graph-Based Modeling Languages. Tech. Rep., University of Koblenz-Landau, Institute for Software Technology (in press). http://www.uni-koblenz.de/~horn/foundations-WE11-draft.pdf
17. Miksa, K., Kasztelnik, M., Sabine, P.: Case study design. Project Deliverable ICT216691/CMR/WP5-D2/D/RE/b1, MOST Project (2009)
18. Object Management Group: Meta Object Facility (MOF) 2.0: Query/View/Transformation Specification v1.0. (2008)
19. Greenyer, J., Kindler, E.: Comparing relational model transformation technologies: implementing Query/View/Transformation with Triple Graph Grammars. Softw. Syst. Model. **9**, 21–46 (2010). doi:10.1007/s10270-009-0121-8
20. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: a standard-compliant metamodeling framework with graph transformations. In: Rensink, A., Warmer, J. (eds.) Model Driven Architecture—Foundations and Applications: Second European Conference. Lecture Notes in Computer Science (LNCS), vol. 4066, pp. 361–375. Springer, Heidelberg (2006)
21. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Model Migration with Epsilon Flock. In: ICMT. Lecture Notes in Computer Science, vol. 6142, pp. 184–198. Springer, Berlin (2010)
22. Herrmannsdoerfer, M.: COPE—a workbench for the coupled evolution of metamodels and models. In: Malloy, B.A., Staab, S., van den Brand, M. (eds.) SLE. Lecture Notes in Computer Science, vol. 6563, pp. 286–295. Springer, Berlin (2010)
23. Taentzer, G., Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Varró, D., Varró-Gyapay, S.: Model transformation by graph transformation: a comparative study. In: ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems. Montego Bay, Jamaica (2005)
24. Ehrig, H., Ehrig, K.: Overview of formal concepts for model transformations based on typed attributed graph transformation. Electron. Notes Theor. Comput. Sci. **152**, 3–22 (2006)
25. Aksit, M., Rensink, A., Staijen, T.: A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In: Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development, AOSD '09, pp. 39–50, ACM, New York (2009)
26. Gönczy, L., Kovács, M., Varró, D.: Modeling and verification of reliable messaging by graph transformation systems. Electron. Notes Theor. Comput. Sci. **175**, 37–50 (2007)
27. Ráth, I., Vago, D., Varró, D.: Design-time simulation of domain-specific models by incremental pattern matching. In: VL/HCC, pp. 219–222. IEEE (2008)
28. Assmann, U.: Graph rewrite systems for program optimization. ACM Trans. Program. Lang. Syst. **22**, 583–637 (2000)
29. Schösser, A., Geiß, R.: Graph rewriting for hardware dependent program optimizations. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) Applications of Graph Transformations with Industrial Relevance, pp. 233–248. Springer, Berlin (2008)
30. Schürr, A., Winter, A., Zündorf, A.: The PROGRES approach: language and environment. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools, vol. 3, pp. 487–550. World Scientific, Singapore (1999)
31. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: a new graph rewrite language based on the unified modeling language and Java. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT. Lecture Notes in Computer Science, vol. 1764, pp. 296–309. Springer, Berlin (1998)
32. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Program. **68**(3), 214–234 (2007, special issue on Model Transformation)
33. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, Berlin (2003)
34. Jakumeit, E., Buchwald, S., Kroll, M.: GrGen. NET. Int. J. Softw. Tools Technol. Transf. (STTT) **12**, 263–271 (2010)
35. Ermel, C., Biermann, E., Schmidt, J., Warning, A.: Visual modeling of controlled EMF model transformation using HENSHIN. ECEASST, **32** (2010)
36. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF

model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MoDELS (1). Lecture Notes in Computer Science, vol. 6394, pp. 121–135. Springer, UK (2010)
37. Biermann, E., Ehrig, K., Ermel, C., Köhler, C., Taentzer, G.: The EMF model transformation framework. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE. Lecture Notes in Computer Science, vol. 5088, pp. 566–567. Springer, Berlin (2007)
38. Taentzer, G.: AGG: a graph transformation environment for modeling and validation of software. In: Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B. (eds.) Applications of Graph Transformations with Industrial Relevance. Lecture Notes in Computer Science, chap. 35, vol. 3062, pp. 446–453. Springer, Berlin (2004)
39. Kastenberg, H., Rensink, A.: Model checking dynamic states in GROOVE. In: Valmari, A. (ed.) SPIN. Lecture Notes in Computer Science, vol. 3925, pp. 299–305. Springer, Berlin (2006)
40. de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Attributed graph transformation with node type inheritance. Theor. Comput. Sci. **376**, 139–163 (2007)
41. ATLAS Group: ATL: User Guide. http://wiki.eclipse.org/ATL/User_Guide (2011)
42. Kolovos, D., Rose, L., Paige, R.: The Epsilon Book. http://dev.eclipse.org/svnroot/modeling/org.eclipse.gmt.epsilon/trunk/doc/org.eclipse.epsilon.book/EpsilonBook.pdf. Accessed June 2011
43. Kolovos, D.S., Paige, R.F., Polack, F.A.: The epsilon transformation language. In: Proceedings of the 1st International Conference on Theory and Practice of Model Transformations, ICMT '08, pp. 46–60. Springer, Berlin (2008)
44. Object Management Group: Object Constraint Language, Version 2.2. (2010)
45. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Kolovos, D.S., Paige, R.F., Lauder, M., Schürr, A., Wagelaar, D.: A comparison of rule inheritance in model-to-model transformation languages. In: Cabot and Visser [50], pp. 31–46
46. Object Management Group: Meta Object Facility (MOF) Core Specification, Version 2.4.1. (2011)
47. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley/Longman, Amsterdam (2009)
48. The Eclipse Project: EMF Model Query. http://www.eclipse.org/modeling/emf/?project=query
49. The Eclipse Project: EMF Model Query2. http://www.eclipse.org/modeling/emf/?project=query2
50. Cabot, J., Visser, E. (eds): Proceedings of the 4th International Conference on Theory and Practice of Model Transformations, ICMT 2011, Zurich, Switzerland, June 27–28, 2011. Lecture Notes in Computer Science, vol. 6707. Springer, Berlin (2011)
51. Van Gorp, P., Mazanek, S., Rose, L. (eds): Post-Proceedings of the TTC 2011: Fifth Transformation Tool Contest, Zürich, Switzerland, June 29–30 2011. EPTCS (2011)

Author Biographies

Jürgen Ebert holds the Chair of Software Engineering at the University of Koblenz-Landau since 1982. His research areas include software engineering, focusing on modeling, software architecture, and construction of generic tools, especially using graph-based methods. In the last decade, he published primarily in the area of graph technology, metamodeling, metaCASE, and software reengineering. He was an organizer or program committee chair for several workshops and conferences in the area of software engineering environments and reengineering.

Tassilo Horn is a researcher at the Institute for Software Technology at the University of Koblenz-Landau. He is currently working on his Ph.D. in which he explores the usage of the functional programming paradigm for model querying and transformation. His further interests include graph technology in general, software reengineering, metamodeling, relational programming, metaprogramming, API design, and free software, where he is a contributor to a diversity of projects.

Copyright of Software & Systems Modeling is the property of Springer Science & Business Media B.V. and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.