Applications and Rewriting of Omnigraphs – Exemplified in the Domain of MDD

Oliver Denninger, Tom Gelhausen, and Rubino Geiß

Institute for Program Structures and Data Organization (IPD)
University of Karlsruhe (TH), Germany
http://www.ipd.uni-karlsruhe.de/

Abstract. Graph rewrite systems provide only elementary primitives – many applications require more complex structures though. We present a rewrite system for omnigraphs, a formal extension of hypergraphs with the ability to connect multiple nodes *and edges* with a single edge. We exemplify the adequacy of this approach in the domain of Model Driven Development (MDD): Using our system trivializes the representation and transformation of advanced UML structures that are awkward to handle with common approaches.

Key words: graph rewriting, hypergraph, omnigraph, supergraph

1 Introduction

Graph rewrite systems elegantly handle various tasks; they have sound and concise fundamentals and their computational power is Turing equivalent. But the operational primitives of current graph rewrite systems are quite elementary, quite assembler-language-like. Several application domains demand more powerful primitives. One example for such a domain is the representation and transformation of UML within MDD.

UML class diagrams allow n-ary associations which are de facto hyperedges [1] (cf. Fig. 2 for example). Furthermore, they allow relations between associations (cf. Fig. 3). In order to express these relations directly, we would need to additionally allow *edges* to be end points of edges – and that is precisely what omnigraphs[1] are about.

In 1998, Minas showed the advantage of hypergraphs over traditional graphs for representing various kinds of diagrams [9], but no available graph rewrite system has support for hypergraphs so far, not to mention omnigraphs. Therefore, we developed languages for model definition, graph definition and rewrite specifications for omnigraphs. Compilers [18] translate these languages into semantically equivalent definitions for a traditional graph rewrite system. In this paper, we present these languages (Section 2), their theoretical fundamentals (Section 3), and the functionality of the compilers (Section 4).

[1] In previous work [5, 7], we referred to 'omnigraphs' as 'supergraphs', but we changed the name in order to avoid further confusion with the antonym of 'subgraphs'.

2 Omnigraphs in Use – A Problem-oriented Introduction

Before giving a formal definition of omnigraphs in the next section, we will introduce omnigraphs by means of their application to a specific problem: representation and transformation of UML. We thereby demonstrate how the concepts of omnigraphs ease the handling of advanced UML structures.

2.1 UML Models as Omnigraphs

We show how to represent UML class diagrams using the syntax of our custom-made graph rewrite system Ogre (OmniGraph REwriting[2]). For the complete syntax of Ogre, please refer to [18]. We have taken all examples from the "UML Superstructure Specification" version 2.1.1 [12].

Fig. 1: Simple association, Fig. 7.19 from the UML Superstructure Specification

Defining a Model. Figure 1 shows a simple association between two classes A and B with multiplicities and the roles endA and endB. Listing 1 shows the definition of an accordant model for omnigraphs: Ogre provides nodes, omniedges, and roles as graph primitives. A definition starts with the type of the primitive, so line 1 defines a type for nodes named Class. Line 4 defines a type for omniedges named Association. Constraints for omniedges (in parentheses) specify the allowed types of roles. The constraint in line 4 states that the omniedges need a least one (+) end point of role type AssociationEnd. In line 5, this class of roles is defined with the constraint that it is only applicable on nodes of the type Class. An AssociationEnd has two attributes: name and multiplicity. Please note that already managing these two attributes is cumbersome in graph rewrite systems that do not support attributed endpoints on edges.

```
1 node Class {
2   name : string;
3 }
4 omniedge Association (AssociationEnd+);
5 role AssociationEnd (Class) {
6   name : string;
7   multiplicity : string;
8 }
```

Listing 1: Model for UML classes and associations

[2] Previous name: SUGR – SUperGraph Rewriting [5]

Instantiation. Having defined the model, we will now set up an instance of this model – an omnigraph representing the UML class diagram depicted in Figure 1. Let's begin with the one line statement depicted in Listing 2: The graph definition starts with an omniedge of the type Association. Inside the body of the omniedge we define two new nodes of type Class. Their corresponding roles (AssociationEnd in both cases) follow, separated by vertical bars. The two nodes with their roles form the end points of the surrounding omniedge. In general, nodes and omniedges are declared by an optional identifier (before) and a type (after the colon). The declaration of an omniedge additionally has a body (in square brackets) defining its end points. As we will not refer to any of the declared graph elements again in the code snippet in Listing 2, we omit the identifiers here. But what we effectively lack in this declaration are attributes!

```
1 :Association[:Class|AssociationEnd :Class|AssociationEnd]
```

Listing 2: Graph definition for a simple association

Now we define a new graph *with* attributes (cf. Listing 3). This time we also show how to declare identifiers and how to reference graph elements. In line 1 and 2, the required Class nodes are defined. They have identifiers (a and b) and attributes to hold the names "A" and "B". In line 4 and 5, a and b are referenced (indicated by the @ character). By this means, they are defining end points. The nodes take the role AssociationEnd as above, but this time we additionally define attributes for the end points: "endA" respectively "endB" as value for the name attribute, and "*" as value for the multiplicity attribute of the respective ends of the association. Now we have completely represented the association from Figure 1.

```
1 a:Class(%name="A")
2 b:Class(%name="B")
3 :Association[
4   @a|AssociationEnd(%{name="endA", multiplicity="*"})
5   @b|AssociationEnd(%{name="endB", multiplicity="*"})
6 ]
```

Listing 3: Graph definition for a simple association (with attributes)

Ternary Associations. UML enables the declaration of n-ary associations which are not directly expressible by simple binary associations. The ternary association in Figure 2 is an example.

Binary as well as ternary edges are only special cases of omniedges; we do not need to extend the previous model from Listing 1: It already accepts an arbitrary number of AssociationEnds for each Association. So we can immediately denote the example as an omnigraph definition as shown in Listing 4. We simply define a third end point inside the Association omniedge. The Class nodes are defined within the omniedge body.

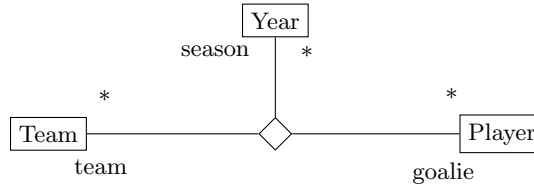

Fig. 2: Ternary association, Fig. 7.21 from the UML Superstructure Specification

```
1 :Association[
2   :Class(%name="Team")|AssociationEnd(%{name="team", multiplicity="*"})
3   :Class(%name="Year")|AssociationEnd(%{name="season", multiplicity="*"})
4   :Class(%name="Player")|AssociationEnd(%{name="goalie", multiplicity="*"})
5 ]
```

Listing 4: Graph definition for the ternary association

Higher Order Predicates. Figure 3 shows an example for a second order predicate in UML class diagrams: The {xor} constraint is a predicate over two associations which are predicates over (Class-) nodes themselves. Omnigraphs remove the restriction of hypergraphs by allowing higher order predicates. To express this UML constraint, our model needs an extension: a new type of omniedge named Constraint and a new role ConstraintEnd. We only want omniedges of the type Association to take this role, so we restrict the role to this type. The complete model definition is shown in Listing 5. The graph definition in Listing 6 consists of two associations (line 1 and 4). The constraint is an additional omniedge with the two associations as end points (line 7). The two associations share a common node, so this node is identified by a (line 2) and referenced (line 5).

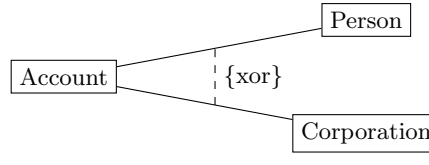

Fig. 3: {xor} constraint, Fig. 7.34 from the UML Superstructure Specification

2.2 Transforming UML with Ogre

After having shown how to represent a UML class diagram as omnigraph, we will show how to define an elementary transformation: We want to transform the ternary association form the preceding example (cf. Figure 2 and Listing 4) into adequate binary associations. This transformation is an inevitable step in every model driven process; we repeat it every time we decompose our model to obtain executable code. The rule according to this transformation demonstrates the pragmatical simplification of writing rules the omnigraph approach has been

```
node Class {
  name : string;
}
omniedge Association (AssociationEnd+);
role AssociationEnd (Class) {
  name : string;
  multiplicity : string;
}
omniedge Constraint (ConstraintEnd+) {
  type : string;
}
role ConstraintEnd (Association);
```

Listing 5: Model for UML classes, associations, and constraints

```
a1:Association[
  a:Class(%name="Account")|AssociationEnd
  :Class(%name="Person")|AssociationEnd]
a2:Association[
  @a|AssociationEnd
  :Class(%name="Corporation")|AssociationEnd]
:Constraint[@a1|ConstraintEnd @a2|ConstraintEnd](%type="xor")
```

Listing 6: Graph definition for the {xor} constraint

designed for: no need to think about any extra nodes and edges, their names, types, and directions – simply because we have omniedges with attributed end points.

Transforming Ternary Associations. The rewriting rule in Listing 7 decomposes ternary associations into adequate binary associations. The left-hand side, the `pattern` graph, matches an omniedge a of type Association with three end points c1, c2, and c3 of type Class and with roles ae1, ae2, and ae3 of type AssociationEnd. The syntax is similar to graph definitions: we use an identifier followed by a colon and a type. The role type is separated by a vertical bar; in contrast to graph definitions we can use identifiers for roles. On the right-hand side of the rule, the `modify` graph, we first delete the ternary association a (line 9) and create a new node c4 of type Class (line 10) – serving as new connection node. Then we create three new associations between the connection node and the former end points of the ternary association. Finally, we have to set the attributes for the new graph elements, which is done in the `eval` section. Line 15, for example, sets the name for the newly created class c4: It consist of the names of the classes c1, c2, c3 and the suffix "Triple".

Figure 4 shows the ternary association after transforming it into an extra class and appropriate binary associations. The rule also changed all multiplicities as necessary for a correct transformation.

The Rewriting Semantics of Ogre. In Ogre, rules consist of a `pattern` graph and a `replace` or a `modify` graph. Each element of these graphs has a name, either user defined or internally defined. Consider a graph element defined in the `pattern` part: If its name is used in the `replace` graph, the denoted

```
pattern {
  a:Association[
    c1:Class|ae1:AssociationEnd
    c2:Class|ae2:AssociationEnd
    c3:Class|ae3:AssociationEnd
  ];
}
modify {
  delete(a);
  c4:Class;
  :Association[c4|ae11:AssociationEnd c1|ae21:AssociationEnd];
  :Association[c4|ae12:AssociationEnd c2|ae22:AssociationEnd];
  :Association[c4|ae13:AssociationEnd c3|ae23:AssociationEnd];
  eval {
    c4.name = c1.name+c2.name+c3.name+"Triple";
    ae11.multiplicity = ae1.multiplicity;
    ae21.multiplicity = "1";  ae21.name = ae1.name;
    ae12.multiplicity = ae2.multiplicity;
    ae22.multiplicity = "1";  ae22.name = ae2.name;
    ae13.multiplicity = ae3.multiplicity;
    ae23.multiplicity = "1";  ae23.name = ae3.name;
  }
}
```

Listing 7: Rule for processing ternary associations

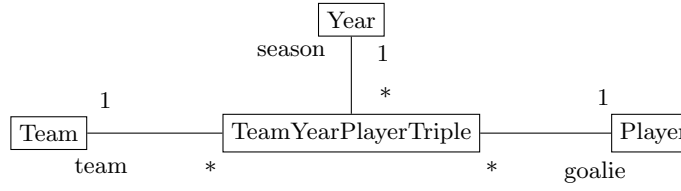

Fig. 4: Ternary Association after the transformation by the rule

graph element will be kept during the execution of the rule. Otherwise the graph element will be deleted from the host graph. A graph element is created in the host graph by defining a name in the `replace` graph. Anonymous graph elements in a `replace` graph always create new elements in the host graph. Using a name multiple times has the same effect as a single occurrence.

The `modify` variant is syntactic sugar for copying the `pattern` graph to the `replace` graph – in this case deletions from this replacement graph are triggered by the `delete` keyword; additions work the usual way. In case of a conflict between deletion and preservation, deletion is prioritized. It is convenient to use the `modify` variant for modifying only small parts of a large pattern graph.

For a proper graph rewriting system, we need a sound approach on how incident objects are treated when other objects are deleted. Traditional graph rewrite systems can be classified according to SPO or DPO, but both approaches are obviously not applicable for omnigraphs. Our definition of omnigraphs (cf. Section 3.2) allows edges to have an arbitrary number of end points, including zero. Deletion of an incident node of an omniedge just reduces the number of

end points of that omniedge. Deletion of an omniedge always requires an explicit statement, and removing it does not bother the objects it connected any further. Thus deletion can never lead to a data structure that is not an omnigraph.

2.3 Advanced UML Structures in Practice

The UML structures we are referring to in this publication are surprisingly uncommon and many software engineers are unfamiliar with these features of UML. Nevertheless, one may have a hard time trying to encode their semantics *without* these structures. In our opinion, this already justifies their existence and their use – leading to the necessity for their support in modelling tools. Some more structures of UML that lend themselves to be realized via omniedges are attributed associations, qualified associations, fork-, join, merge- and decision nodes, or duration constraints.

3 Formal Definition of Omnigraphs

Before we present the formalism we discuss some issues regarding our approach of generalizing "direction". This discussion should explain our perspective on hypergraphs and demonstrate that the given definition is adequate.

3.1 Roles

In a traditional (directed) graph, each edge has a direction, a point of origin and an aiming point. But how can we specify something comparable for omnigraphs with arbitrary numbers of end points? The following paragraphs present an approach that renders the ordinary directed edge a special case of a more general concept.

Every formalism – every way of representing information – provides certain primitives to store pieces of information and other primitives to relate these fragments. The available primitives determine the semantics that can be encoded directly with this formalism. Graphs, too, are just a special way of representing information. Their primitives are usually nodes and edges with labels. We use the labels to store pieces of information. Contiguity relates the information stored in a graph.

On closer inspection, one can see that there is a third primitive in graphs that allows information storage: the direction of edges. Direction enables us to store several extra bits per edge – one extra bit if only unidirectional edges are permitted, two extra bits if multidirectional edges are permitted, and no extra bits if only undirected edges are permitted in the graph. Initially, these bits encode the direction of an edge. Additional information, for example "who loves whom" or "which code block precedes another", is an interpretation that has been agreed on. This agreement constitutes which bit-value represents which role in the relation. Thus, we are effectively interested in the *roles* an edge assigns and *not* in its direction.

Taking into account that we are interested in roles rather than directions, we could as well provide roles immediately in our way of representing information. Instead of storing one direction per edge, each end of an edge is assigned a role. The advantage of this approach is that it scales a lot better: Now, it is irrelevant how many ends (these "ends" are called "tentacles" in hypergraphs) an edge has, including the special cases "one" and "zero".

Another conclusion we can draw from the above consideration is, that the roles we are effectively interested in are seldom "source" and "target". We would rather allow arbitrary roles. Accordingly, the number of available roles does not need to be limited to two.

For these reasons, our omniedges do not have a direction and no inherent order or numerical limitations of their tentacles. Instead, each tentacle is assigned a role out of an arbitrary, finite set of roles. Initially, the combinations of roles within one omniedge are unrestricted. One might want to impose constraints about the legal role sets per omniedge, though.

A classic approach to assign meaning to the tentacles of a hyperedge (or the components of a tuple) is position: Any term-based syntax for the declaration of hyperedges imposes a sequence of tentacle declarations, and the position within this sequence assigns its meaning to a tentacle. The downside of this approach is that the sequence always needs to be specified completely and in order, and that no meaning can be assigned to additional tentacles. In programming languages like Eiffel or Visual Basic, some of these drawbacks can be resolved by *named* function arguments. The role-based approach is a generalization thereof.

For illustration, we show that an ordinary (directed, two-ended) edge can be expressed immediately in terms of this approach: It is an omniedge with two tentacles of which one has the role "source" and the other has the role "target". A traditional graph is thus completely representable as omnigraph. As also omnigraphs are representable via traditional graphs (cf. Section 4) both formalisms are *theoretically* equally expressive. Yet the *practical* expressiveness of omnigraphs is more suitable for certain applications as we show in this publication.

3.2 Definition

Definition 1 (Omnigraph). *Let N, O, T, R be arbitrary finite pairwise disjoint*[3] *sets, $C := N \cup O$, and src, tgt, and rol total but not necessarily injective or surjective functions with*

$$\begin{aligned} src &: T \longrightarrow O \\ tgt &: T \longrightarrow C \\ rol &: T \longrightarrow R \end{aligned}$$

then the 7-tuple $G = (N, O, T, src, tgt, R, rol)$ is an omnigraph.

[3] For clearness we require N, O, T, R to be pairwise disjoint. Formally it is only necessary that $N \cap O = \emptyset$ holds.

Explanation and Implications. We call N the nodes, O the omniedges, T the tentacles, and R the roles of an omnigraph. $C = N \cup O$ is the set of all connectable objects. The tentacles link the elements of C to their connections $o \in O$; these links are denoted by the src and tgt functions assigning the obvious direction[4] to the tentacle. It is specific to omnigraphs that the tentacles can only start at omniedges but end at omniedges and nodes. So by definition, no node can have an outgoing tentacle whereas incoming tentacles are allowed. Each tentacle linking a connectable object $c \in C$ to an omniedge $o \in O$ also specifies a role $r \in R$ that c takes in o. This is denoted by the rol function.

As omniedges may connect other omniedges, the tentacles are directed to make clear which omniedge establishes the connection between the other ones. It is only usual to utilize the concept of 'direction' here – but it is not necessary: It is sufficient to *somehow* distinguish which tentacles belong to which omniedge. Accordingly, it is irrelevant which direction the tentacles exactly have, as long as it is consistent for all omniedges.

Multigraphs are defined by their allowance for multiple edges between two distinct nodes. For omnigraphs this property is obtained by our function-based definition instead of the commonly used tuple-based definition. Thus an arbitrary number of omniedges can occur between every set $C' \subseteq C$. Furthermore, this property allows multiple tentacles of the same or different roles between an omniedge and one connected object $c \in C$. Moreover, if not every node or omniedge has an incident tentacle, src and tgt are not surjective. We require src and tgt to be *total* such that no dangling tentacles can occur.

Omnigraphs as defined here have two properties that may appear strange, but are harmless consequences of the generality of the concept: (a) Omniedges may have one or zero tentacles and (b) omniedges may connect to themselves.

Discussion. Our definition is rather close to the function-based definition of ordinary hypergraphs. One *could* picture the set T as "edges" and C as "vertices". But this picture is only half true, because the set C has an internal structure: The "edges" can only start at elements $o \in O \subseteq C$ and end at *any* element $c \in C = O \cup N$. This way, the property of being representable as bipartite graphs (like ordinary hypergraphs) is lost. Clearly, every omnigraph G can be turned into an omnigraph G' without nodes by turning every node into an omniedge not having any outgoing tentacles ("virtual nodes"). Yet in this case, certain runtime checks and validation procedures on the graph and its model must be put into place if we want to distinguish omniedges and (virtual) nodes in a typesave way. We chose the intuitive and computationally cheaper alternative, namely to enforce this distinction by the formalism and in the Ogre language.

Formally speaking, our definition is a direct extension of hypergraphs. But we define names and interpretations of the sets C, O, N, and T to suit our need for a vocabulary of concepts on a higher level of abstraction.

[4] Please note that only the tentacles but not the omniedges themselves are directed.

3.3 Examples

We will present two of the examples from Section 2 as formal omnigraph definitions. The first example is the formal definition of the hyperedge depicted in Figure 2. The second example is the formal definition of the constraint edge between two association edges depicted in Figure 3.

The ternary association from Figure 2 has three nodes: Team, Year, and Player. They respectively take the roles team, season, and goalie in the omniedge Association.

$$
\begin{aligned}
&N = \{\mathsf{Team}, \mathsf{Year}, \mathsf{Player}\} \quad O = \{\mathsf{Association}\} \\
&T = \{t_0, t_1, t_2\} \quad R = \{\mathsf{team}, \mathsf{season}, \mathsf{goalie}\} \\
&\mathit{src}(t_0) = \mathsf{Association} \quad \mathit{tgt}(t_0) = \mathsf{Team} \quad \mathit{rol}(t_0) = \mathsf{team} \\
&\mathit{src}(t_1) = \mathsf{Association} \quad \mathit{tgt}(t_1) = \mathsf{Year} \quad \mathit{rol}(t_1) = \mathsf{season} \\
&\mathit{src}(t_2) = \mathsf{Association} \quad \mathit{tgt}(t_2) = \mathsf{Player} \quad \mathit{rol}(t_2) = \mathsf{goalie}
\end{aligned}
$$

The formal definition of the {xor} constraint has three nodes: Account, Person, and Corporation, and three omniedges: $\mathsf{Association}_0$, $\mathsf{Association}_1$, and Constraint. Each omniedge has two tentacles, so that we have in total six tentacles, but only two different roles associationEnd and constraintEnd. An illustration of the formal definition is shown in Figure 5.

$$
\begin{aligned}
&N = \{\mathsf{Account}, \mathsf{Person}, \mathsf{Corporation}\} \\
&O = \{\mathsf{Association}_0, \mathsf{Association}_1, \mathsf{Constraint}\} \\
&T = \{t_{a0}, t_{a1}, t_{a2}, t_{a3}, t_{c0}, t_{c1}\} \quad R = \{\mathsf{associationEnd}, \mathsf{constraintEnd}\} \\
&\mathit{src}(t_{a0}) = \mathsf{Association}_0 \quad \mathit{tgt}(t_{a0}) = \mathsf{Account} \quad \mathit{rol}(t_{a0}) = \mathsf{associationEnd} \\
&\mathit{src}(t_{a1}) = \mathsf{Association}_0 \quad \mathit{tgt}(t_{a1}) = \mathsf{Person} \quad \mathit{rol}(t_{a1}) = \mathsf{associationEnd} \\
&\mathit{src}(t_{a2}) = \mathsf{Association}_1 \quad \mathit{tgt}(t_{a2}) = \mathsf{Account} \quad \mathit{rol}(t_{a2}) = \mathsf{associationEnd} \\
&\mathit{src}(t_{a3}) = \mathsf{Association}_1 \quad \mathit{tgt}(t_{a3}) = \mathsf{Corporation} \quad \mathit{rol}(t_{a3}) = \mathsf{associationEnd} \\
&\mathit{src}(t_{c0}) = \mathsf{Constraint} \quad \mathit{tgt}(t_{c0}) = \mathsf{Association}_0 \quad \mathit{rol}(t_{c0}) = \mathsf{constraintEnd} \\
&\mathit{src}(t_{c1}) = \mathsf{Constraint} \quad \mathit{tgt}(t_{c1}) = \mathsf{Association}_1 \quad \mathit{rol}(t_{c1}) = \mathsf{constraintEnd}
\end{aligned}
$$

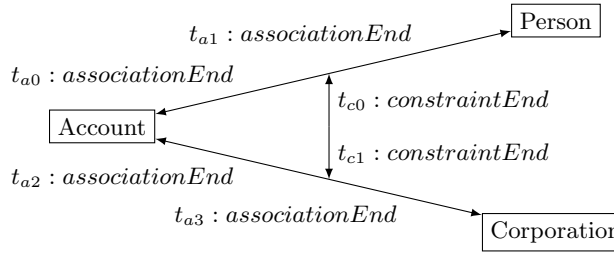

Fig. 5: Formal graph for the {xor} constraint (names of omniedges omitted).

3.4 Extensions

For an efficient use of a graph rewriting *tool* we need (a) labels on nodes, edges, and roles, and want (b) these labels to obey certain constraints, i. e. typing. Furthermore, (c) inheritance relations among these types are needed for an easier declaration of rules. Accordant extensions to the formal basis of omnigraphs can be defined the usual way without difficulty.

Ogre implements these features. It seamlessly incorporates them from the underlying graph rewrite system (cf. Section 4). In contrast to some formalizations of hypergraphs, we assign the type of an omniedge directly to it and do not derive it from the number or types of tentacles. In particular, omniedges of a certain type may have arbitrary numbers of tentacles. However, Ogre supports constraints about the legal role types for each omniedge type.

4 Implementation

To avoid developing a graph rewrite system from scratch, we chose to decompose omnigraph model-, rule-, and graph-definitions to model-, rule-, and graph-definitions for the traditional graph rewrite system GrGen.NET [6]. Ogre provides three compilers for this task. As the space in this paper is limited, we only give a rough outline of these transformations. Details can be found in [5], the compilers including source code are available from [18].

4.1 Mapping Ogre Definitions to GrGen.NET Definitions

As GrGen.NET has no support for omniedges and roles, we need a mapping to translate omniedges and roles into nodes and edges, the primitives provided by GrGen.NET. We map omniedges by introducing an additional interconnection node. As a consequence, each tentacle becomes an independent edge between the interconnection node and the node connected by the tentacle; the role of the tentacle is mapped to the type of the according edge. This approach is quite obvious and well-known from treating hyperedges as bipartite graphs. But in the context of omniedges we have to pay special attention to the direction of decomposed edges: their tentacles are directed, as discussed in Section 3.2. Correspondingly, we realize tentacles with directed edges. Figure 6 shows the mapping for the {xor} constraint example from Figure 3.

Mapping Models. Listing 8 shows the result of mapping our model from Listing 1 to GrGen.NET syntax. We can see that the omniedge type Association has become a node type (line 9) and the role type AssociationEnd has become an edge type (line 4). Line 5 shows a GrGen.NET constraint defining the allowed source and target types for this edge type.

Figure 7 shows the visualization of the mapped model for the ternary association. We can clearly see the interconnection node with the omniedge type and the edges with the role types.

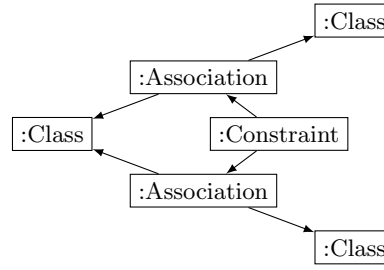

Fig. 6: Mapping of the {xor} constraint

```
1 node class Class extends NODE {
2   name : string;
3 }
4 edge class AssociationEnd extends ROLE
5     connect Association[1:*] ->  Class[*] {
6     name: string;
7     multiplicity: string;
8 }
9 node class Association extends OMNIEDGE;
```

Listing 8: Model from Listing 1 after translation *in GrGen.NET syntax*

Mapping Graphs. Listing 9 shows the ternary association from Listing 4 after the mapping to GrGen.NET syntax. The omniedge Association is decomposed into an interconnection node (line 1) and edges representing the tentacles (line 3, 5, and 7). In GrGen.NET edges are denoted by an arrow from the source to the target node. Identifier, type and attributes of the edge are stated between beginning and end of that arrow. The dollar sign is a build-in attribute of GrGen.NET keeping the identifier of the graph primitive for debugging purposes. The Class nodes and their declared attributes are preserved during the mapping.

This listing shows how the concept of omnigraphs disburdens the user from the necessity to fragment his or her thoughts for the input into a traditional graph rewrite system. Besides this semantic advantage, the Ogre syntax eliminates the need to constantly repeat identifiers for miscellaneous nodes and edges. This is not so much a quantitative but a qualitative alleviation, as repeating numerous – only technically induced – identifiers is an error prone work. However, an IDE or

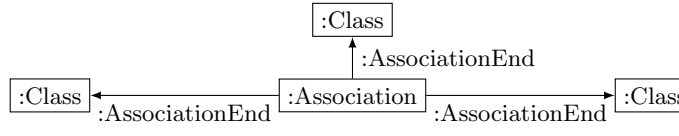

Fig. 7: Mapping of the **model** for the ternary association

```
new s_3:Association($="s_3")
new n_0:Class($="n_0", name="Team")
new s_3 -:AssociationEnd(name="team", multiplicity="*")-> n_0
new n_1:Class($="n_1", name="Year")
new s_3 -:AssociationEnd(name="season", multiplicity="*")-> n_1
new n_2:Class($="n_2", name="Player")
new s_3 -:AssociationEnd(name="goalie", multiplicity="*")-> n_2
```

Listing 9: Graph from Listing 4 after translation *in GrGen.NET syntax*

a graphical notation could alleviate this work, while the model and its instances will still be polluted with artificial entities.

Mapping Rules. Basically, rules are processed by treating the left-hand and right-hand side patterns individually as graph definitions and mapping them separately. As the syntax for patterns is quite similar to graph definitions we reuse the mappings for graph definitions with some minor changes. Listing 10 shows the rule from Listing 7 after mapping to GrGen.NET syntax. The content of the `eval` section can just be copied as it is, because it does not need to be changed by the mapping.

```
pattern {
  a:Association;
    a -ae1:AssociationEnd-> c1:Class;
    a -ae2:AssociationEnd-> c2:Class;
    a -ae3:AssociationEnd-> c3:Class;
}
modify {
  delete(a);
  c4:Class;
  x2:Association;
    x2 -ae11:AssociationEnd-> c4;
    x2 -ae21:AssociationEnd-> c1;
  x5:Association;
    x5 -ae12:AssociationEnd-> c4;
    x5 -ae22:AssociationEnd-> c2;
  x8:Association;
    x8 -ae13:AssociationEnd-> c4;
    x8 -ae23:AssociationEnd-> c3;
  eval {
      c4.name = c1.name+c2.name+c3.name+"Triple";
      ae11.multiplicity = ae1.multiplicity;
      ae21.multiplicity = "1";  ae21.name = ae1.name;
      ae12.multiplicity = ae2.multiplicity;
      ae22.multiplicity = "1";  ae22.name = ae2.name;
      ae13.multiplicity = ae3.multiplicity;
      ae23.multiplicity = "1";  ae23.name = ae3.name;
  }
}
```

Listing 10: Rule from Listing 7 after translation *in GrGen.NET syntax*

Obviously, the semantics of our rules are derived directly from the semantics of GrGen.NET rules [2]. GrGen.NET implements closely the SPO semantics, so deleting an incident node of an edge will also delete that edge. This behaviour is well suited for our needs: If we want to delete an omniedge, we can just delete

the interconnection node. The SPO semantics will lead to the deletion of edges representing its tentacles. As we allow omniedges to have an arbitrary number of tentacles (including zero), deleting an incident connectable object (and thus 'losing' a tentacle) cannot lead to an invalid omnigraph (cf. Section 2.2).

5 Related Work

Graph rewrite systems have been under research for several decades. Research in Model Driven Development has lead to a strong demand for model transformation technology. This has brought graph rewriting – as one possible solution for model transformation – to industrial relevance.

In this paper, we discuss omnigraph rewriting. To the best of our knowledge, no system with this capability has been published, yet. In Ogre, omniedges – and thus also hyperedges – are first-class citizens among the graph primitives: They are quasi materialized in the graph and can seamlessly be used in search and replacement patterns. Thus Ogre is also a hypergraph rewriting system. But even for hypergraph rewriting[5], there is no comparable system available. A property – besides allowing edges to be endpoints of edges – that distinguishes our notion of omnigraph rewriting from the usual *theoretical* notion of hypergraph rewriting is that omniedges have no fixed number of tentacles defining their type. Furthermore, the tentacles of omniedges have no inherent order. Both properties are by design, as we discuss in Section 3.

The Graph eXchange Language (GXL) supports omnigraphs using `rel`- and `relend`-elements. But as already suggested by its name, GXL is not a rewrite system but an exchange format and only serves to store graphs – it can neither store rules nor rewrite sequences. For solely exchanging omnigraphs, GXL would suit very well, but we could not find a tool to exchange omnigraphs graphs with: Holt [8] gives an overview of GXL capable tools, namely GRAS [15], DiaGen [10], Fujaba [13], GenSet [14] and PROGRES [19]. Except DiaGen[6], all these systems ignore `rel` tags, they have no support for hyper- or omniedges. The same holds for GROOVE [17].

AGG-graphs [3], a variant of ALR-graphs and the formal basis of AGG, explicitly enable edges between edges. But these edges are only binary, hyperedges are not first-class citizens in AGG-graphs. Instead, AGG-graphs come with a direct support for 'abstractions' which in turn are not first-class citizens in omnigraphs. So AGG-graphs and omnigraphs are skew to each other. However, AGG (the tool) has no support for edges between edges.

GReAT [16] and VIATRA [20] partially support edges between edges: GReAT can define edges between edges in models, but has no possibility to use them in

[5] Please note that 'hypergraph rewriting' and 'hyperedge replacement' [11] are different things: In hyperedge replacement, hyperedges are only special left-hand side patterns of replacement rules. Our concept of hypergraph rewriting is much broader.

[6] DiaGen is a tool for generating diagram editors based on hypergraphs. We aim to develop a general purpose graph rewrite system.

Table 1: Overview Graph Rewriting Tools

	Fujaba	**GReAT**	**VIATRA**	**GrGen.NET**	**Ogre**
Typed Domain	yes	yes	yes	yes	yes
Type Inheritance	single	multi	multi	multi	multi
Node Attributes	Java types	simple types, enumerations	simple and complex types	simple types, enumerations	simple types
Edge Attributes	same as nodes	no	same as nodes	same as nodes	same as nodes
Role Attributes	no	no	no	no	yes
NACs	yes	yes	yes	yes	yes
Hyperedges	no	no	no	no	yes
Omniedges	no	see text	see text	no	yes
Rule Definition	programmed	programmed	declarative	declarative	declarative
Rule Notation	graphical	graphical	textual	textual	textual
Parameterization	all types	no	all types	graph entities	no
Rule Scheduling	story diagrams	sequence diagrams	state machine	similar regular expressions	similar regular expressions
Rule Iteration	loop	loop, recursion	loop, recursion, fix point iteration	loop, fix point iteration, transaction	loop, fix point iteration, transaction

graphs or rules. VIATRA can use edges between edges in graphs and rules, but cannot define them in models. In both systems, edges are always binary.

Apart from the support for omniedges, we regard Ogre as ordinary general purpose graph rewrite system. Table 1 compares the features of Ogre with those of Fujaba, GReAT, VIATRA and GrGen.NET [6], some of the most popular general purpose graph rewriting tools today. The criteria are adopted from Czarnecki and Helsen [4].

6 Conclusion

We presented omnigraphs together with an appropriate rewrite system called Ogre. Omnigraphs are an extension of the well known hypergraphs, enabling the attachment of multiple nodes *and edges* to edges. In this paper, UML structures like n-ary associations and constraints between associations served as examples for the usefulness of omnigraphs. Besides model transformations, we use omnigraphs for the representation of natural language [7]. In this domain, hyper- and omniedges are essential as natural language includes complex sentence structures with higher order relations. Bond angles are an example from the domain of chemistry where one would like to declare edges between edges.

The rewrite system Ogre provides custom-made languages for the definition (i. e. typing) and instantiation of omnigraphs as well as the declaration of rules for their transformation. As the implementation of Ogre is based on a traditional graph rewrite system, we provide compilers [18] and can thus incorporate many features from our underlying system.

The reduction to normal graph rewrite systems is straight forward, but the provided abstraction eases the task of specifying transformations and graphs.

The automatic transformation unburdens the user from consistently and continuously regarding auxiliary nodes and edges. Instead, the user can directly express his intention. This makes omnigraphs and Ogre *practically* more expressive than common approaches – and thus well suited to simplify model transformation for advanced UML structures.

References

1. Berge, C.: Graphs and Hypergraphs. Elsevier Science Ltd. (1985)
2. Blomer, J., Geiß, R.: The GrGen.NET User Manual. University of Karlsruhe, Technical report, ISSN 1432-7864 (2007)
3. Conrad, M., Gajewsky, M., Holl-Biniasz, R., Rudolf, M., Demuth, J., Weber, S., Heckel, R., Müller, J., Taentzer, G., Wagner, A.: Graphische Spezifikation ausgewählter Teile von AGG – einem algebraischen Graphgrammatiksystem, Technical report, no. 95-07, TU Berlin, (1995)
4. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal, vol. 45, no. 3 (2006)
5. Denninger, O.: Erweiterung des Kantenkonzepts deklarativer Graphersetzungssysteme von Einfachkanten über Hyperkanten zu „Superkanten". Diplomarbeit, Universität Karlsruhe (2007)
6. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.M.: GrGen: A Fast SPO-Based Graph Rewriting Tool. In: Corradini et al. (eds.) ICGT 2006. LNCS, pp. 383–397, Springer, Heidelberg (2006)
7. Gelhausen, T., Tichy, W.F.: Thematic Role based Generation of UML Models from Real World Requirements. In: First IEEE International Conference on Semantic Computing (ICSC), pp. 282–289 (2007)
8. Holt, R., Schürr, A., Elliott, S., Winter, A.: GXL: A graph-based standard exchange format for reengineering, Science of Computer Programming (2005)
9. Minas, M.: Hypergraphs as a Uniform Diagram Representation Model, TAGT, pp. 281–295 (1998)
10. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. Science of Computer Programming, vol. 44, pp. 157–180 (2002)
11. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation. vol. 1: Foundations. World Scientific, Singapore (1997)
12. OMG: Unified Modeling Language: Superstructure, version 2.1.1 (2007)
13. Fujaba Tool Suite. University of Paderborn Software Engineering Group
14. GenSet: Design Information Fusion. University of Oregon
15. GRAS – A graph oriented database system for (software) engineering environments. Lehrstuhl für Informatik 3, University of Technology Aachen (RWTH)
16. GReAT – Graph Rewrite and Transform System. Institute for Software Integrated Systems, Vanderbilt University, Nashville
17. GROOVE – GRaphs for Object-Oriented VErification. University of Twente
18. OGRE – OmniGraphREwriting System. Institute for Program Structures and Data Organization (IPD), University of Karlsruhe (2007) `http://sf.net/projects/ogre-system/`
19. PROGRES – A Graph Grammar Programming Environment. Lehrstuhl für Informatik 3, University of Technology Aachen (RWTH)
20. VIATRA – Visual Automated model Transformations. Dept. of Measurement and Information Systems, Budapest University of Technology and Economics