Electronic Notes in Theoretical Computer Science 44 No. 4 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume44.html 11 pages

How to Specify a Graph Transformation Approach: A Meta Model for Fujaba ⋆

Reiko Heckel [1] Albert Zündorf [2]

Dept. of Mathematics and Computer Science
University of Paderborn
D-33095 Paderborn, Germany

Abstract

Application-oriented approaches to graph transformation provide structural features beyond vertices and edges, like composition in hierarchical graphs, inheritance in object-oriented graphs, multiplicity constraints, etc. Often, these features have a specific dynamic interpretation which requires complex embedding mechanisms and context conditions. For example, the deletion of a compound node usually implies the deletion of its components.

In this paper, we propose the use of a *meta graph grammar* for the definition of such a complex graph transformation approach. A meta graph grammar is a typed graph grammar whose type graph provides a static description of the structure of graphs, rules, and transformations of the approach. This static *meta model*, which is comparable to the meta model in the Uml specification, is extended by a specification of the rule application operator by means of *graphical embedding rules*, i.e., the productions of the meta graph grammar. These embedding rules allow a concise visual description of the admissible context embeddings of a rule and of the side effects of the rule application on the context.

As a case-study, a meta graph grammar for selected features of the object-oriented graph transformation approach Fujaba is given.

Key words: graph transformation and graph grammars, meta modeling, fujaba

⋆ Research partially supported by the ESPRIT Working Group APPLIGRAPH and the TMR network GETGRATS.
[1] Email: reiko@upb.de
[2] Email: zuendorf@upb.de

Fig. 1. A sample class diagram

Fig. 2. A sample object diagram

1 Introduction

A graph transformation approach is formally given by its notions of *graph, rule*, and *transformation*. In addition, some approaches provide concepts like *graph schemata* or *control structures*. Usually, formal definitions of these notions are given in mathematical (e.g., set-theoretic, logic, algebraic, or categorical) terms (cf. [15] for a collection of such definitions). While mathematical definitions are an indispensable tool for the development of the theory, they are often not very useful for explaining the concepts and constructions to potential users, in particular, if the mathematical language has a very different nature than the approach to be defined.

A solution to this problem, which is popular for visual modeling languages like the Uml [13], is the approach of *meta modeling* [12]. Here, a modeling language is defined in a kind of boot-strapping process using as a meta language a simple subset of the language to be defined. (In order to avoid cyclic definitions, this subset has to be given an independent formalization.) For example, in the Uml specification [13], class diagrams and constraints are used in order to specify the abstract syntax and static semantics of the Uml. The dynamic semantics is just described in informal text.

In this paper we propose meta modeling as a technique for the specification of both the static and dynamic aspects of application-oriented graph transformation approaches. Based on a Uml-like meta model defining statically the classes of graphs, rules, and transformations of the approach, a *meta graph grammar* is used for generating, from each rule, a class of transformations. As a case study, this technique is applied to (a subset of) the object-oriented graph transformation approach Fujaba which includes features like *composition, multiplicity constraints*, and *inheritance*.

2 The Fujaba Approach

The Fujaba project [6] (see also `www.fujaba.de`) aims at promoting the use of graph transformation in object-oriented software development. To attract object-oriented software developers, Fujaba employs Uml collaboration diagrams as a notation for graph rewrite rules. In addition, the Fujaba environment provides a code generator that generates a Java implementation of class diagrams and graph rewrite rules. We generate plain Java code that uses very simple implementation concepts. Graph rewrite rules are applied to usual main memory objects and employ standard Java Beans access methods for look-up and manipulation of object structures.

Class and object diagrams. Figure 1 shows a fragment of a class diagram that models the control software of an automatic material flow system in an industrial assembly hall, developed within the ISILEIT project [7]. In FUJABA, the classes of a class diagram are translated into usual JAVA classes. Class attributes become private data members of the corresponding JAVA classes with appropriate `set-` and `get`-methods. Class diagram associations represent bi-directional relationships between classes. Therefore, associations are translated into pairs of pointers within the corresponding JAVA classes. Associations in a class diagram may be adored with multiplicity constraints. For example, each robot may hold an arbitrary number of goods (multiplicity "∗"), while every good may belong to at most one robot. For a *to-many* end of an associations (multiplicity "∗", the FUJABA code generator creates a private data member of type `HashSet`, i.e., a standard container class provided by the JAVA runtime library. In addition, a number of access methods that is created allowing, e.g., to add and to remove links from/to an association and to enumerate all its links. Thus, for the *to-many* end of association holds in Fig. 1 (attached to class Good) a private data member `HashSet holds;` in class `Robot` is generated with methods `addToHolds`, `removeFromHolds` and `iteratorOfHolds`.

To-one associations are most naturally implemented by pointers. Thus, we generate a private data member of the neighbors type and `set-` and `get`-methods. The *to-one* end of association holds at class Robot creates a private data member `Robot robot` in class `Good` and methods `setRobot` and `getRobot`.

The implementation of *to-one* associations via usual pointers saves memory-space and reduces the read-access to a minimum. However, this implies that our implementation is not able to store more than one pointer for a given association at a certain object, not even temporarily. If, for example, a good `g` already belongs to a certain robot `r1` and if we assign a new robot `r2` to hold `g` by calling `g.setRobot(r2)`, then we create a conflict in ownership between robots `r1` and `r2`. One has to deal with this conflict by either rejecting the new owner or by overriding the old owner or by raising a runtime exception. In the FUJABA approach, the default behavior is to override the old owner with the new owner.

In order to guarantee the consistency of the pairs of pointers that represent a link between two objects, the write methods for the corresponding data members call each other, mutually. For example, method `r.addToGoods(g)` calls method `g.setRobot(r)` and vice versa. Similarly, the call `r.removeFromGoods(g)` results in call `g.setRobot(null)` and vice versa. Since the data members are private and may only be changed via their access methods, we are able to guarantee that each time a link is added to or removed from the object structure, the corresponding pair of pointers is established or removed properly. This guarantee turns usual JAVA object structures into a proper implementation of *object-oriented graphs*.

Fig. 3. A story pattern creating a *to-one* association

Fig. 4. A story pattern dealing with compositions

The consequently bi-directional implementation of associations guarantees that each object knows all its neighbors. Therefore, we are able to generate a method `removeYou` that removes all links between an object and its neighbors. Such an isolated object becomes easily garbage collected. Thus, our method `removeYou` allows to implement a behavior where nodes may be removed without explicit knowledge of their context while avoiding the severe problem of dangling references and guaranteeing referential integrity.

The `removeYou`-method is also used to implement a simple concept of *composition*. In Uml class diagrams, an association marked with a black diamond represents a composition relationship indicating so-called *co-incident life time*. We implement this co-incident life time as a weak existence dependency: If a component object looses its compound, it is removed, too. We achieve this by forwarding recursively `removeYou` calls along composition relations. Thus, in the object diagram in Fig. 2 which instantiates the class diagram in Fig. 1, the call `g1.removeYou()` would recursively isolate (and thus garbage collect) all Good-objects.

Rules and transformations. Figure 3 and 4 show two Fujaba *story patterns*, i.e., graph rewrite rules in Uml collaboration diagram notation. A story pattern is represented as a group of connected objects optionally with additional attribute constraints. Objects and links may be marked with constraints {new} and {destroyed} and the attribute compartment of objects may contain attribute assignments. One may derive the left-hand side of a usual graph rewrite rule from such a collaboration diagram by collecting all unmarked elements and attribute constraints and all elements marked with {destroyed}. The right-hand side of the corresponding graph rewrite rule consists of all unmarked elements, the attribute assignments, and the elements marked with {new}. Common elements of the left- and right-hand sides are mapped identically.

Figure 3 represents a story pattern that looks up a good g1 with attribute prodName equal to " bobby car" and a shuttle s. This object structure is "rewritten" by the same objects plus a holds-link connecting g1 and s. The application of this story pattern to the object structure shown in Fig. 2 would match objects g1 and s and it would create a holds-link between them. So far, this creates the situation that good g1 is attached to two holds-links connecting it to assembly cell a and shuttle s, respectively. This would violate the multiplicity constraint of association holds that allows at most one robot as holder of a certain good. In addition, our pointer based implementation is not able to hold more than one robot pointer within one good. As already discussed, the Fujaba approach resolves this conflict by removing existing holds-links attached to goods as soon as new holds-links are attached. Thus, in

our example, the holds-link connecting assembly cell c and good g1 is removed as a side-effect of adding the holds-link between g1 and s.

Figure 4 outlines how our approach deals with compositions. The shown story pattern looks up a good g1 with attribute prodName equal to " bobby car" and a good g3 with prodName equal to " seat" and a parts-link between g1 and g3. This object structure is rewritten by destroying good g1 and the parts-link to good g3 and by creating a new good g9 and a new parts-link to g3. Although this operation looks as it does not change anything but the object id of good g1, it has dramatic side effects. As Fig. 1 shows, the parts-association is a composition relationship. This means that the deletion of a parent good removes all child goods that become orphans, too. Thus, in Fig. 2, the deletion of good g1 would cause the deletion of all its child goods and, recursively, of all descendants. However, the story pattern of Fig. 4 explicitly removes the parts-link from g1 to g3. Thereby the child object g3 is separated from its parent g1 and, thus, it is not affected by the removal of good g1. However, the other children of g1 are not rescued by some other object in the story diagram and they are removed together with their parent, i.e., objects g2 and g4 are removed and their children g5, g6, g7, and g8 become orphans and are removed, too.

At first glance, the survival of good g3 may look strange, since intermediately it became an orphan. However, in our project this semantics has proven to be very useful since it allows to deconstruct composition hierarchies and to unmount parts from their parents in order to store them separately for later reuse. In our example, the unmounted seat is directly reused for the new good g9.

To summarize, the application of graph transformation to object-oriented data models creates challenges for the semantics definition of graph rewrite rules. In particular, multiplicity constraints for associations and composition relations imply certain side effects of the creation and deletion of objects and links that need to be carefully specified. Semantic decisions like the overriding of links of *to-one* associations are guided by the implementation in Java and a in contrast with most other graph transformation approaches. In the following two sections, we present a formalization for the static and dynamic aspects of these non-standard features.

3 Abstract Syntax: Typed Graphs

As in the Uml specification [13], we adopt a meta modeling approach [12] in order to specify the abstract syntax of object-oriented graphs and graph schemata, story patterns, and transformations which form the core of the Fujaba language. The relation between the (static) meta model—the language definition—and the individual models—the elements of the language—is captured by the concept of *typed graphs* [2].

By *graphs* we mean directed unlabeled graphs $G = \langle G_V, G_E, src^G, tar^G \rangle$

Fig. 5. Fujaba static meta model

with set of vertices G_V, set of edges G_E, and functions $src^G : G_E \to G_V$ and $tar^G : G_E \to G_V$ associating to each edge its source and target vertex, respectively. A graph homomorphism $f : G \to H$ is a pair of functions $\langle f_V : G_V \to H_V, f_E : G_E \to H_E \rangle$ compatible with source and target, i.e., for all edges e in G_E, $f_V(src^G(e)) = src^H(f_E(e))$ and $f_V(tar^G(e)) = tar^H(f_E(e))$.

Given a graph TG, called *type graph*, a *TG-typed (instance) graph* consists of a graph G together with a typing homomorphism $g : G \to TG$ associating to each vertex and edge x of G its type $g(x) = t$ in TG. In this case we also write $x : t \in G$. The collection of all instance graphs typed over TG is denoted by $Graph_{TG}$.

Throughout this paper, the type graph TG shall be given by the Fujaba meta model depicted in Fig. 5. Following the Uml meta model [13], we distinguish between two levels of very similar structure: the *schema level* consisting of the meta types Class, Assoc, AssocEnd, Attribute, Type and Schema and the *instance level* given by meta types Object, Link, LinkEnd, AttribLink, Value and Graph. Each instance-level graph element is associated to a schema element by an instOf-edge.

All vertex types have an attribute name: string which is not shown in the diagram. Unless stated otherwise, the multiplicity of edges is $*$ (any). Properties like *multiplicity* and *composition* which belong to the ends of an association (rather than to the association as a whole) are modeled as attributes of the vertex type AssocEnd. The super-links model inheritance between Fujaba classes pointing from the sub- to the superclass.

We follow the approach of [10] (also used in the Uml meta model [13]) of regarding attribute instances as links from objects to attribute values. Correspondingly, at the type level, each attribute is associated to its class and its type. The latter provides a meta attribute sort which refers to a sort of an algebraic signature $\Sigma = \langle S, OP \rangle$ while Value-vertices are attributed with the elements of (the corresponding carrier of) a Σ-algebra A. We assume that the signature Σ and the algebra A are fixed so as to reflect the built-in data types of the language.

The association of schema elements to schemata and of graph elements to graphs is represented by el-edges as modeled in the lower part of Fig. 5. It also defines a transformation Trafo as a pair of two graphs (not necessarily disjoint).

Various *integrity constraints* have to be imposed on the instances of this meta model in order to represent well-formed Fujaba models. Many of them are obvious, like the commutativity of instOf-edges with the el-edges inside object-oriented graphs and graph schemata. For example, the Object connected to a LinkEnd should be an instance of the Class connected to the corresponding AssocEnd. Also, we require that the elements associated with a Graph-vertex through el-edges form indeed an object-oriented graph being an instance of the

Fig. 6. Abstract syntax of sample graph schema and story pattern

corresponding schema. The formalization of such constraints using a formal constraint language is beyond the scope of this paper.

The representation of the story pattern of Fig. 3 according to this meta model is given in Fig. 6. The upper part represents the graph schema of Fig. 1 where we have omitted some of the obvious instOf- and el-edges for readability. The lower part describes the story pattern as a pair of graphs where, in this case, the pre-graph is a subgraph of the post-graph. Notice, that the meta model does not distinguish between *story patterns* (i.e., rewrite rules) and *transformations*. In fact, in the following section, a story pattern is considered as a minimal transformation from which all transformations using this pattern can be generated by means of context embedding rules.

4 Operational Semantics: Embedding Rules

In this section, a *meta graph grammar* is used to specify, for each story pattern, the set of transformations resulting from applying the pattern to legal object-oriented graphs. The productions of this meta graph grammar generate the contexts in which the pattern can be placed. Each production describes, at the same time, the matching conditions and the effect of the transformation in the given context. The productions are presented as *embedding rules* like below on the left

$$\frac{P}{C} \quad T \qquad \rightsquigarrow \qquad P \cup T \hookleftarrow P \cap C \cup T \hookrightarrow C \cup T$$

where P is called the *premise*, C the *conclusion*, and T the *typing condition* of the rule, such that the union $P \cup C \cup T$ is well-defined. Usually, P and C represent transformations (or story patterns) while T is a fragment of the graph schema.

Formally, our meta graph grammar is based on the *algebraic double-pushout (DPO) approach* to graph transformation [4,3] using *typed graphs* [2] and *negative application conditions* [8]. The premise P and the condition T jointly form the left-hand side of a graph grammar production as shown above on the right. The right-hand side is given by the conclusion C and the typing condition C, and the interface graph in the center is the intersection of the two. The application of such a production to a TG-typed graph representing a transformation yields another TG-typed graph which represents a transformation with additional objects, links, or attribute instantiations. The set of all transformations using a given story pattern G_0 is given by all well-formed TG-typed graphs G derivable from G_0 by means of the graph grammar productions. This use of embedding rules is inspired by the contextualization rules in SOS [14], like the rule for *restriction* in CCS [11] stating under which conditions an action can be performed in the context of the restriction operator.

Fig. 7. DPO-like embedding rules: adding disconnected nodes, adding edges between preserved nodes, and merging of preserved nodes

Fig. 8. Abstract syntax of embedding rules: interface vertices (upper left), dangling edges (right), and generalization (lower left).

Fig. 9. SPO-like embedding rules: deletion of dangling edges, merging destroyed nodes, and conflict resolution

Fig. 10. Composition rules: implicit deletion of dependent objects, unless there exists another dependency

The context embedding rules are formally defined on the level of abstract syntax. However, as it is obvious from the abstract syntax graph depicted in Fig. 6 of the story pattern in Fig. 3, this presentation is quite complex even in simple examples. Moreover, it is not adequate for explaining the application of story patterns to users of the language (e.g., software developers or domain experts). Therefore, we present the context embedding rules in the style of graphical deduction rules [1] based on the concrete syntax of the language.

The rules of Fig. 7 specify the embedding policy of the DPO approach. The first rule states that, given any transformation, for every vertex type C we can add a vertex o:C to obtain another transformation. The vertex is added to the interface, i.e., it occurs both in the pre and the post graph, because there is no qualification with {destroyed} or {new} (cf. Sect. 2). The second rule states that a transformation can be extended by introducing edges between vertices in the interface, provided this is permitted by the schema graph. The third rule specifies the possibility of gluing two vertices in the interface if they have the same type. A similar rule could be defined for edges. Thus, given a DPO production (represented as an instance of the meta model in Fig. 5), the rules in Fig. 7 allow us to derive all legal DPO transformations, i.e., they specify the notion of production application in the DPO approach. For the first and second rule, the formal presentation based on the abstract syntax as specified by the meta model is shown in Fig. 8 in the upper left and in the right, respectively.

In Fig. 9, three additional rules are shown that are needed in order to specify the more general SPO approach. The three rules correspond to the three situations ruled out by the DPO gluing condition [3]: the deletion of dangling edges, the merging of vertices that are both deleted, and the conflict between deletion and preservation, which is resolved in favor of deletion. Notice, how the difference between DPO and SPO (which is quite difficult to tell looking at the original definitions) boils down to a few additional embedding rules.

As far as normal associations (without composition or multiplicities other than "∗") are concerned, Fujaba behaves like the SPO approach. The be-

Fig. 11. Multiplicity rules: adding links to constrained associations, and overwriting of links

Fig. 12. Inheritance and embedding of attribute links

havior w.r.t. composition is shown in Fig. 10. The idea is that a component object depends on the existence of at least one composite, i.e., it is deleted when the last composite is removed. As specified by the first rule, a transformation which destroys an object o can be extended by attaching a composition relation with another object p so that both the object p and the composition relation are also destroyed. If there exists a second composition relation of p, the object is preserved in the transformation, as specified by the rule on the right. In order to be complete, we would have to add a third rule, analogous to the second rule in Fig. 7, which adds a composition between two objects that are both preserved.

Figure 11 specifies the possibly most surprising feature of Fujaba, the overriding of links of multiplicity 0..1. As discussed in Sect. 2, the idea is to consider such a link as a pointer to an object which is assigned a new value once a new link of this type is created. We essentially distinguish two situations in dealing with links of multiplicity 0..1. On its left, Fig. 11 shows the simple case of introducing such a link to the interface if none is existing so far. This is specified by the *negative application condition* in the premise of this rule depicted by the crossed-out object p:D. Similar rules are needed in order to introduce the A-edge to the pre- or the post-graph of the transformation. On the right, it is shown that a transformation in which a new A-link is created can happen in the presence of a second A-link. As a side effect, the latter is destroyed.

The rule in the left of Fig. 12 specifies the semantics of inheritance: An object o: C in a story pattern may be matched by an object o: D in a transformation if D is a subclass of C. The abstract syntax of this rule is shown in the lower left of Fig. 8. In the right of Fig. 12, the instantiation of attribute links is explained. The rule in the center specifies that, given a class C with an attribute attr:type, an object o:C may be linked to an attribute value val of the same type. The rule in the right of Fig. 12 describes how the deletion of an object leads to the deletion of its attribute links. Despite the different concrete syntax for links and attribute links, these rules are very similar to those for embedding and implicit deletion of links.

This concludes our presentation of the context embedding rules for Fujaba story patterns. We have considered structural features like multiplicity constraints 0..1, composition, attributes, and inheritance. For lack of space, several other features have been omitted, like more sophisticated multiplicity constraints, ordered or qualified associations, and multi-objects. Some of these require a more powerful notion of graph grammar which supports, e.g., the distinction between terminal and non-terminal types. Whether, besides the

basic concepts, also the results of graph grammar theory can be of benefit for the specification graph transformation approaches is a question of future work.

5 Conclusion

In this paper, we have provided a meta model for both the static and dynamic aspect of a subset of the Fujaba language—an object-oriented graph transformation approach based on the Uml and Java. The static aspect is described by the type graph of a meta graph grammar whose productions are used to specify the generation of transformations from story patterns, i.e., Fujaba graph rewrite rules.

The technique presented in this paper is not limited to the specification of an individual graph transformation approach. In fact, our use of graphical deduction rules is closely related to the *Graphical Operational Semantics (GOS) approach* [1] where similar rules are used to specify the operational semantics of UML diagram languages [5].

The correspondence between graph transformation rules and full Uml collaboration diagrams which describe, in addition to structural changes, the interaction between objects, is further extended in [9]. In this context very similar problems can be identified. For example, the distinction in the Uml between collaboration diagrams on the specification level and on the instance level corresponds to the one between graph transformation rules and transformations. Embedding rules can be a general means to specify the relation between these two levels.

Acknowledgement

Thanks to Stefan Sauer for many helpful comments on a preliminary version of this paper.

References

[1] Corradini, A., R. Heckel and U. Montanari, *Graphical operational semantics*, in: A. Corradini and R. Heckel, editors, *Proc. ICALP2000 Workshop on Graph Transformation and Visual Modelling Techniques* (2000), `http://www.uni-paderborn.de/cs/ag-engels/Papers/2000/CorradiniGTVMT00.pdf`.

[2] Corradini, A., U. Montanari and F. Rossi, *Graph processes*, Fundamenta Informaticae **26** (1996), pp. 241–266.

[3] Corradini, A., U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Löwe, *Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach*, in: Rozenberg [15] pp. 163–245.

[4] Ehrig, H., M. Pfender and H. Schneider, *Graph grammars: an algebraic approach*, in: *14th Annual IEEE Symposium on Switching and Automata Theory* (1973), pp. 167–180.

[5] Engels, G., J. Hausmann, R. Heckel and S. Sauer, *Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML*, in: A. Evans, S. Kent and B. Selic, editors, *Proc. UML 2000, York, UK*, LNCS **1939** (2000), pp. 323–337.

[6] Fischer, T., J. Niere, L. Torunski and A. Zündorf, *Story diagrams: A new graph transformation language based on UML and Java*, in: H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, editors, *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn, November 1998*, LNCS **1764** (2000).

[7] Gausemeier, J., U. Glässer, W. Schäfer and (project managers), *Integrative specification of distributed control systems for the flexible automated manufacturing*, `www.upb.de/cs/isileit`, funded by the German Research Foundation (DFG).

[8] Habel, A., R. Heckel and G. Taentzer, *Graph grammars with negative application conditions*, Fundamenta Informaticae **26** (1996), pp. 287 – 313.

[9] Heckel, R. and S. Sauer, *Strengthening UML collaboration diagrams by state transformations*, in: H. Hußmann, editor, *Proc. Fundamental Approaches to Software Engineering (FASE'2001), Genova, Italy*, LNCS (2001).

[10] Löwe, M., M. Korff and A. Wagner, *An algebraic framework for the transformation of attributed graphs*, in: M. Sleep, M. Plasmeijer and M. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, John Wiley & Sons Ltd, 1993 pp. 185–199.

[11] Milner, R., "Communication and Concurrency," Prentice-Hall, 1989.

[12] Object Management Group, *Meta object facility (MOF) specification* (1999), `http://www.omg.org`.

[13] Object Management Group, *UML specification version 1.3* (1999), `http://www.omg.org`.

[14] Plotkin, G., *A structural approach to operational semantics*, Technical Report DAIMI FN-19, Aarhus University, Computer Science Department (1981).

[15] Rozenberg, G., editor, "Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations," World Scientific, 1997.