Electronic Communications of the EASST
Volume 6 (2007)

Proceedings of the
Sixth International Workshop on
Graph Transformation and Visual Modeling Techniques
(GT-VMT 2007)

Triple Patterns: Compact Specifications for the Generation of Operational Triple Graph Grammar Rules

Juan de Lara, Esther Guerra, Paolo Bottoni

14 pages

Triple Patterns: Compact Specifications for the Generation of Operational Triple Graph Grammar Rules

Juan de Lara[1], **Esther Guerra**[2], **Paolo Bottoni**[3]

[1] jdelara@uam.es
Escuela Politécnica Superior
Universidad Autónoma de Madrid (Spain)
[2] eguerra@inf.uc3m.es
Dep. Ingeniería Informática
Universidad Carlos III de Madrid (Spain)
[3] bottoni@di.uniroma1.it
Dip. Informatica
Università di Roma La Sapienza (Italy)

Abstract: Triple Graph Grammars (TGGs) allow the specification of high-level rules modelling the synchronized creation of elements in two graphs related through a correspondence graph. Low-level *operational* rules are then derived to manipulate concrete graphs. However, TGG rules may become unnecessarily verbose when elements have to be replicated from one graph to the other, and their actual derivation cannot exploit the presence of reoccurring patterns. Moreover they do not take advantage from situations in which a normal creation grammar for one of the graphs exists, from which TGG operational rules can be derived to build the other graph.

We present an approach to generating TGG operational rules from normal ones, reducing the information needed to derive them, through the definition of *Triple Patterns*, a high-level, compact, declarative, and visual notation for the description of admissible structures in a triple graph. Patterns can be expressed with respect to classes defined in a meta-model, and instantiated with derived classes at the model level, thus exploiting the inheritance hierarchies. The application of the generated rules results into the (synchronized or batch) creation of the structures specified in the patterns. We illustrate these concepts by showing their application to the synchronized incremental construction of visual models and of their semantics.

Keywords: Graph Transformation, Triple Graph Grammars, Visual Languages.

1 Introduction

Model transformation is becoming increasingly popular with the advent of model-driven development technologies, such as MDA [MSUW04], where model-to-model transformation plays a central role. In such transformations, an input model M_A conforming to a meta-model MM_A is transformed into an output model M_B conforming to a (possibly different) meta-model MM_B. Several scenarios are of interest here. For example, in a syntax directed visual modelling tool with separate models for concrete syntax and for semantic interpretation (which contains the relevant semantic roles, see [BDD+04]), one would like to model the synchronized evolution of both models (although a batch update of the semantic model could also make sense). For tool integration applications, a (bi-)directional – batch or incremental – transformation is desirable [Sch94]. Finally, one could be interested in checking the consistency of two given models.

Triple Graph Grammars (TGGs) [Sch94] were proposed by Schürr as a means to model the transformation of two graphs (source and target) related through a correspondence graph (whose nodes have morphisms to elements in the other two graphs). The main idea is to model the synchronized evolution of the two graphs, as well as the correspondence graph relating both, by means of triple rules. From these *creation* triple rules, algorithms were given to produce *operational* rules to perform a translation in either direction (from source to target or vice versa), to create the correspondence graph given two already existing source and target graphs, or to check the validity of the correspondence graph.

In the aforementioned scenario of a syntax directed visual modelling tool, the use of TGGs may be too cumbersome. In these environments, one models by means of rules the possible user editing actions. This brings advantages in cases when one has to model complex editing actions, where many elements are created in the concrete syntax, but requires the designer to define complex TGG creation rules as well. It is however possible to identify patterns for such situations, whereby certain elements in the concrete syntax always play the same role in the semantic model. Therefore, we propose an approach in which the designer has to provide the creation grammar for the concrete syntax only, and some triple patterns specifying admissible relations between concrete syntax elements and semantic roles. We have defined a collection of algorithms which exploit these patterns to produce sets of TGG operational rules that either do a batch translation from concrete syntax to the semantic model, or produce the synchronized evolution of both. This reduces the amount of information that the designer has to input, since many triple patterns can be "applied" to a normal graph transformation rule.

The approach we present is suitable for integration in meta-modelling environments, as the algorithms explicitly take into account the inheritance hierarchies of the meta-models. Although the presented examples are taken from the Visual Languages area, these ideas are readily applicable to general model-to-model transformations.

Paper Organization. Section 2 introduces TGGs, and some of the extensions we have provided to the underlying graph model [GL07]. Section 3 presents *triple patterns* and the algorithms for generating operational triple rules. In Section 4, we take into account the inheritance hierarchy of the meta-model, presenting the concept of *abstract triple patterns*. Section 5 compares with related research, and Section 6 discusses conclusions and future work.

2 Triple Graph Grammars

Triple graphs are made of three graphs: source, target and correspondence ones. Correspondence graph nodes are used to relate elements in source and target graphs. Triple graphs are depicted as $p : p_{src} \xleftarrow{ps} p_{corr} \xrightarrow{pt} p_{tar}$, where ps and pt are morphisms from the nodes in graph p_{corr} to nodes in the source and target graphs. The structure of each graph p_X (for $X \in \{src, corr, tar\}$) is given by $p_X = (V_X, E_X, src_X : E_X \rightarrow V_X, tar_X : E_X \rightarrow V_X)$, where V_X is the set of vertices, E_X is the set of edges, and src_X and tar_X are functions defining the source and target nodes of every edge $e \in E_X$. Labels for nodes and edges can also be given.

In [GL07], we extended the underlying triple graph structure originally proposed in [Sch94] with attributes for nodes and edges, and a typing by a type triple graph (or meta-model triple) which may contain inheritance relations between nodes or edges. Moreover, we made the relation between the source and target graphs more flexible, by allowing *partial* morphisms from nodes in the correspondence graph to nodes and edges in the other two graphs.

Figure 1(a) shows an example meta-model triple taken from the area of visual modelling languages. The lower part (source graph) contains a simplified meta-model with the base classes for the concrete syntax of a visual language [BG04]. Briefly, in a diagrammatic language, significant *spatial relations* exist among *identifiable elements*. The latter are recognizable entities in the language, to which a semantic role can be associated, and which are univocally materialized by means of a complex graphic element. Each such element is composed in turn of simpler graphic elements, each possessing one or more attach zones, which define its availability to participating in different spatial relations, such as containment or touching.

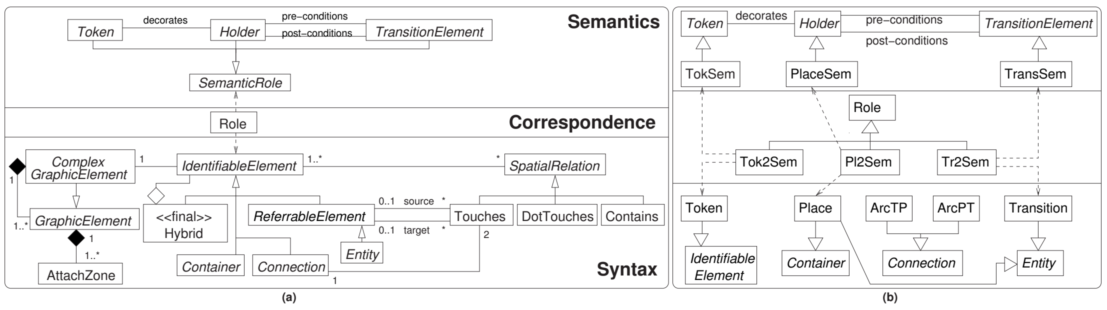


Figure 1: (a) Meta-Model Triple for the Syntax and Semantics of Visual Languages (b) Specialized Meta-Model for Petri Nets.

The upper part (target graph) contains a meta-model that describes the possible abstract roles for a transition-based (i.e. token-holder) semantics (i.e. semantics in the style of Petri nets, UML 2.0 activity diagrams and automata). The correspondence graph assigns semantic roles to syntactic elements. When a meta-model for a new visual language is defined, the newly defined concrete syntax concepts inherit from the classes in the syntax meta-model. If the language has a transition-based semantics, then the designer can create concrete roles by subclassing the classes in the semantics meta-model. Thus, predefined, customizable model transformation libraries implementing the operational semantics can be reused for the new language. Figure 1(b) shows the definition of the syntactic and semantic roles for Petri nets (we have omitted arc weights for simplicity of presentation). The significant spatial relations are refined (by means of a creation graph grammar, with some rules shown in Figures 2(a) and 3) to be the *Touches* relation between instances of *ArcPT* (*ArcTP*) and a source *Place* (*Transition*) or a target *Transition* (*Place*), and the *Contains* relation, between instances of *Place* and *Token*. Note that a *Place* can play both the role of an *Entity*, in relation to the arcs which refer to it, and that of a *Container*, in relation to the *Tokens* it holds.

TGG rules model the transformation of triple graphs. In [GL07] we adapted TGG rules to the Double Pushout approach (DPO) [EEPT06], in which rules are modelled using three components, L, K and R, where: L (the left hand side, LHS) contains the elements to be found in the host graph where the rule is applied; K contains the elements preserved by the rule application; and R (the right hand side, RHS) contains the elements that should replace the part identified by L in the host graph. The DPO approach has been lifted to work with any (weak) adhesive HLR category [EEPT06] (such as those for graphs, Petri nets, etc.). In [GL07] we showed that the category $\mathbf{TriAGraph}_{\mathbf{TriATG}}$ of attributed typed triple graphs (short triple graphs) and morphisms is an adhesive HLR category. Therefore, in our case, L, K and R are triple graphs.

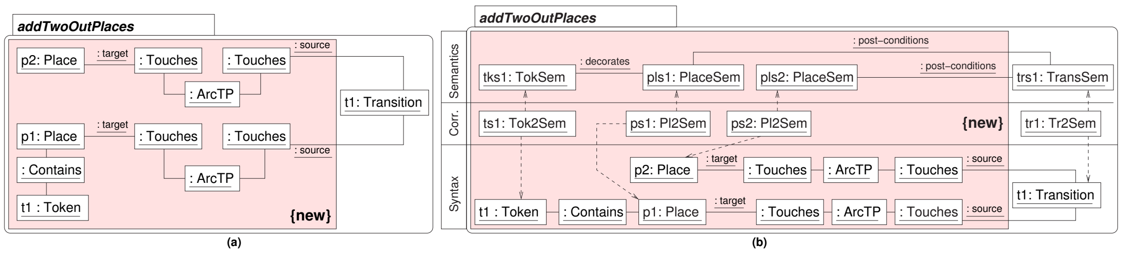


Figure 2: (a) Rule Modelling an Editing Action. (b) Triple Rule Modelling the Synchronized Creation of Semantic Roles.

The motivation for this work is the following. Given a normal graph grammar modelling the possible editing actions in a modelling environment (i.e. working in the concrete syntax only), how can we obtain triple rules that update or build synchronously the semantic model? As an example, Figure 2(a) shows a rule modelling a complex editing action by which, given an existing transition, two places are created, connected with arcs from the transition to the places, and a token is inserted into one of the places. Figure 2(b) illustrates the desired corresponding triple rule that synchronously creates the semantic elements together with the syntactic ones, designating the created places as post-conditions for the transition. We could build by hand a TGG rule for each single syntax editing rule. However, this task is repetitive, as elements in the concrete syntax are related in the same way to elements in the semantic model (as specified in the meta-model triple), i.e. a reoccurring pattern can be identified in the triple rules.

3 Triple Patterns

In this section we present the concept of *Triple Pattern*, together with an algorithm that, given a rule (like the one in Figure 2(a)) and a set of patterns, generates an operational TGG rule (like the one in Figure 2(b)). For simplicity of presentation, we assume the simple graph structure mentioned in the first paragraph of Section 2 (i.e. untyped graphs, with nodes in the correspondence graph having two morphisms: one to a node in the target and one to a node in the source graph, like in [Sch94]). The adaptation of the algorithm to more complex graph models is straightforward. Assuming that the input rule acts on the source graph only, the algorithm generates a TGG rule that synchronously creates the necessary elements in the target graph. Symmetrically, the input rule could act on the target graph, and the generated TGG rule would complete the source graph. Moreover, as in [Sch94], it is also easy to generate slightly different TGG rules: batch rules (i.e. rules assuming that the source elements are already created, and which then create the target graph elements), rules for creating the correspondence graph, assuming that the source and target graphs are created, and rules for checking the validity of the correspondence graph.

A triple pattern $p : p_{src} \xleftarrow{ps} p_{corr} \xrightarrow{pt} p_{tar}$ is a triple graph conformant to a meta-model triple. Formally, given a triple pattern p and a triple graph G, we say that G satisfies p (written $G \models p$) if an injective triple graph morphism $m : p \rightarrow G$ exists.

Example. We first start by giving an intuition of the algorithm through an example. In this paper, we use triple patterns in order to specify in a visual, high level, acausal notation the kind of configurations we want to find in our semantic models when certain syntactic configurations are met (or the other way round). Thus, our triple patterns are triple graphs conforming to the meta-model of Figure 1. Figure 3 shows a triple pattern depicting the needed structure in the syntactic model for a holder to have a token in the semantic model. In this case, a *Place* in the syntactic model has an associated *PlaceSem* role (a subclass of *Holder*) in the semantic model. Similarly, a *Token* in the syntactic model has a *TokSem* role in the semantic model (a subclass of class *Token* of the semantic meta-model). In the semantic model, a token *decorates* a holder, while at the syntactic level the place *contains* the token.

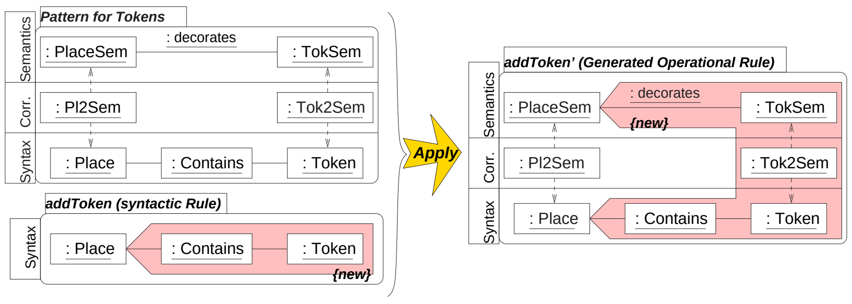


Figure 3: Applying a Pattern to a Rule.

Figure 3 also shows a syntactic editing rule ("addToken") modelling the creation of a token inside a place in the syntactic model. The objective of the algorithm ("Apply") is to obtain the triple rule shown in the figure, where information about the actions to be done at the semantic level has been incorporated, together with the mapping between the syntactic and semantic models. Roughly, we first try to find a match from the pattern to the rule's RHS. Then we glue the pattern and the RHS of the syntactic rule through the matching, to obtain the triple rule's RHS. Finally, we construct the triple rule's LHS by taking the elements in the correspondence and semantic graphs (of the RHS) which are related to elements which were already present in the syntactic rule's LHS.

The following algorithm describes the application of a set of patterns to a non-deleting normal rule, resulting in one triple rule. Later, we show how the algorithm can be easily modified for its application to deleting (and non-creating) rules.

```
Apply(P: SetOfTriplePatterns, rl:  Rule):  TripleRule
```

Let $P = \{p^i\}_{i \in I}$ be a set of triple patterns of the form $p^i : p^i_{src} \xleftarrow{ps^i} p^i_{corr} \xrightarrow{pt^i} p^i_{tar}$ and rl a non-deleting normal rule $rl : L \xleftarrow{l} K \xrightarrow{r} R$ with $L = K$, and which therefore can be written as $rl : L \xrightarrow{r} R$. The application of P to rule rl results in a triple rule rl' as follows:

1. Initialize the triple rule rl', copying rule rl in the source part of rl'. The resulting triple rule is written as $rl' : L_{rl'} \xrightarrow{r'} R_{rl'}$, where r' is a triple graph morphism (see Figure 4).

$$\begin{tikzcd}
 & L_{tar} = \emptyset \arrow[r, "r'_{tar}=\emptyset"] & R_{tar} = \emptyset \\
rl' = & L_{corr} = \emptyset \arrow[u, "lt=\emptyset"] \arrow[d, "ls=\emptyset"] \arrow[r, "r'_{corr}=\emptyset"] & R_{corr} = \emptyset \arrow[u, "rt=\emptyset"'] \arrow[d, "rs=\emptyset"'] \\
 & L_{src} = L \arrow[r, "r'_{src}=r"] & R_{src} = R
\end{tikzcd}$$

Figure 4: Initialization of Triple Rule rl'.

2. $\forall p^i : p^i_{src} \xleftarrow{ps^i} p^i_{corr} \xrightarrow{pt^i} p^i_{tar} \in P$:

   (a) $\forall p^i_{src} \xrightarrow{m^i_j} R_{src}$, with m^i_j an injective match from the source part of p^i (i.e. p^i_{src}) to the source part of $R_{rl'}$ (i.e. R_{src}):

      i. **if** $\exists p^i_{src} \xrightarrow{m'^i_j} L_{src}$ with $m^i_j = r'_{src} \circ m'^i_j$ **then** do nothing (as no elements in the source part of the rule have been newly created for this match) **else**

      ii. $\forall O : (O_{src} = p^i_{src}) \xleftarrow{os} O_{corr} \xrightarrow{ot} O_{tar}$ such that the diagram of Figure 5(a) commutes, and that $\nexists O'_x | O_x \subset O'_x \subseteq p^i_x,\ x \in \{corr, tar\}$:

$$\begin{tikzcd}
 & O_{tar} \arrow[r, hook] \arrow[dl, "mt"'] & p^i_{tar} \\
R_{tar} & & \\
 & O_{corr} \arrow[r, hook] \arrow[uu, "ot"] \arrow[dl, "mc"'] \arrow[dd, "os"] & p^i_{corr} \arrow[uu, "pt^i"'] \arrow[dd, "ps^i"'] \\
R_{corr} \arrow[uu, "rt"] \arrow[dd, "rs"] & & \\
 & p^i_{src} \arrow[r, "id"] \arrow[dl, "m^i_j"'] & p^i_{src} \\
R_{src} & &
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
 & O_{tar} \arrow[rr, hook] \arrow[dl, "mt"'] & & p^i_{tar} \arrow[dl, "tp"'] \\
R_{tar} \arrow[rr, "tr"] & & R'_{tar} & \\
 & O_{corr} \arrow[rr, hook] \arrow[uu, "ot"] \arrow[dl, "mc"'] \arrow[dd, "os"] & & p^i_{corr} \arrow[uu, "pt^i"'] \arrow[dd, "ps^i"'] \arrow[dl, "cp"'] \\
R_{corr} \arrow[uu, "rt"] \arrow[dd, "rs"] \arrow[rr, "cr"] & & R'_{corr} \arrow[uu, "rt'"] \arrow[dd, "rs'"] & \\
 & p^i_{src} \arrow[rr, "id"] \arrow[dl, "m^i_j"'] & & p^i_{src} \arrow[dl, "tp"'] \\
R_{src} \arrow[rr, "tr"] & & R'_{src} &
\end{tikzcd}$$

(b)

Figure 5: (a) Glueing p^i with the Right Hand Side of rl'. (b) Building the Pushout.

A. replace $R_{rl'} : R_{src} \xleftarrow{rs} R_{corr} \xrightarrow{rt} R_{tar}$ with the pushout object of the previous diagram, i.e. $R_{rl'} = PushOut(R_{rl'}, O, p^i)$ [1]. The pushout is shown in Figure 5(b).

B. **Add appropriate nodes to L**

```
[ 1] $\forall n \in V_{R_{src}}$ // Check if we have to copy the correspondence node to $L$
[ 2]   $New_{tar} = New_{corr} = New_{unconn} = \emptyset$ // Sets with newly added nodes to $L$
[ 3]   if $(\exists n' \in V_{L_{src}}$ s.t. $r'_{src}(n') = n)$ then // $n$ is also in $L$
[ 4]     // seek correspondence nodes which are not in $L$
[ 5]     $\forall c \in V_{R_{corr}}$ s.t. $rs(c) = n \wedge \nexists c'' \in V_{L_{corr}}$ s.t. $r'_{corr}(c'') = c$
[ 6]       // add correspondence node to L
[ 7]       $V_{L_{corr}} = V_{L_{corr}} \uplus \{c'\}$ and set $ls(c') = n'$, $r'_{corr}(c') = c$
[ 8]       $New_{corr} = New_{corr} \uplus \{c'\}$ // add it to the set
[ 9]       // check for nodes in the target graph of $R$ which are not in $L$
[10]       if $(\exists n'' \in V_{R_{tar}} | rt(c) = n'' \wedge \nexists o \in V_{L_{tar}} | r'_{tar}(o) = n'')$ then
[11]         // add node to target graph of $L$ and to $New_{tar}$ set
[12]         $V_{L_{tar}} = V_{L_{tar}} \uplus \{n'''\}$ and set $lt(c') = n'''$, $r'_{tar}(n''') = n''$
[13]         $New_{tar} = New_{tar} \uplus \{n'''\}$
[14] $\forall n \in New_{tar}$// Add nodes to target graph for which no correspondence exists
[15]   $\forall m \in V_{R_{tar}}$ s.t. $(\nexists m' \in V_{L_{tar}}$ s.t. $r'_{tar}(m') = m \wedge \exists path_U(r'_{tar}(n), m))$[2]
[16]     $V_{L_{tar}} = V_{L_{tar}} \uplus \{m'\}$, set $r'_{tar}(m') = m$, $New_{unconn} = New_{unconn} \uplus \{m'\}$
[17] $New_{tar} = New_{tar} \uplus New_{unconn}$
```

C. **Add appropriate edges to L_X (for $X \in \{corr, tar\}$):**

```
[1] $\forall n \in New_X$ // visit all new nodes...
[2]   // check if some edge stems from $r'_X(n)$ and ends in a node $\in L$
[3]   if $(\exists e \in E_{R_X}, m' \in V_{L_X}$ s.t. $source_{R_X}(e) = r'_X(n) \wedge target_{R_X}(e) = r'_X(m') \wedge$
[4]     $\nexists e' \in E_{L_X}$ s.t. $r'_X(e') = e)$ then
[5]     // Add the edge to $L$
[6]     $E_{L_X} = E_{L_X} \uplus \{e'\}$, $r'_X(e') = e$, $source_{L_X}(e') = n$, $target_{L_X}(e') = m'$
```

Note that in step ii, we look for a total match from p^i_{src} to R_{src}, and partial matches from p^i_{corr} and p^i_{tar}. Thus, the triple graph O models the domain of such partial matches. With the pushout, we add the part of p^i which was not matched to the rule. In step $ii.B$ we copy the necessary correspondence nodes to the LHS, if they were added to the RHS by the pushout and the RHS node they refer to also belongs to the LHS. More than one correspondence node can be connected to a source or target graph node (line [5]). We also allow nodes in the target graph which are not connected with any correspondence graph node (added to the RHS by the pushout, and appropriately copied to LHS by lines [14-17]). These are useful if we want to model a source graph node related with many elements in the target graph, or "composite" connections in the target graph.

The application of a set of patterns to a rule acting on the target graph simply requires substi-

[1] The pushout of triple graphs [GL07, Sch94] is built component-wise, where in addition all the faces of the two cubes commute. Examples are shown in Figures 5(b) and 6

[2] Predicate $path_U(a,b)$ is true if a path from a to b exists (without taking into account the edge direction) where no node in the path receives a morphism from correspondence graph nodes, except a. Moreover, b should not be connected (directly or indirectly) with a newly created node, i.e. a node p s.t. $\nexists p' \in V_{L_{tar}}$ with $r'_{tar}(p') = p$.

tuting *src* by *tar* in the previous algorithm. It is also easy to apply patterns to deleting rules (i.e. rules which delete elements but do not create anything), by substituting *L* by *R* in the algorithm.

Example (continued). Figure 6 shows some details about the execution of steps *2.a.ii.B* and *2.a.ii.C* of the algorithm in the case of the rule and the patterns shown in Figure 3. The upper part shows how the pushout is performed, and the lower part also shows how the new elements *c'* and *n"'* are added to *L* in the generated TGG rule. Note that the *PlaceSem* object associated with the *Place* belongs to *L*, as the *Place* object also belongs to *L* (step B in the algorithm).

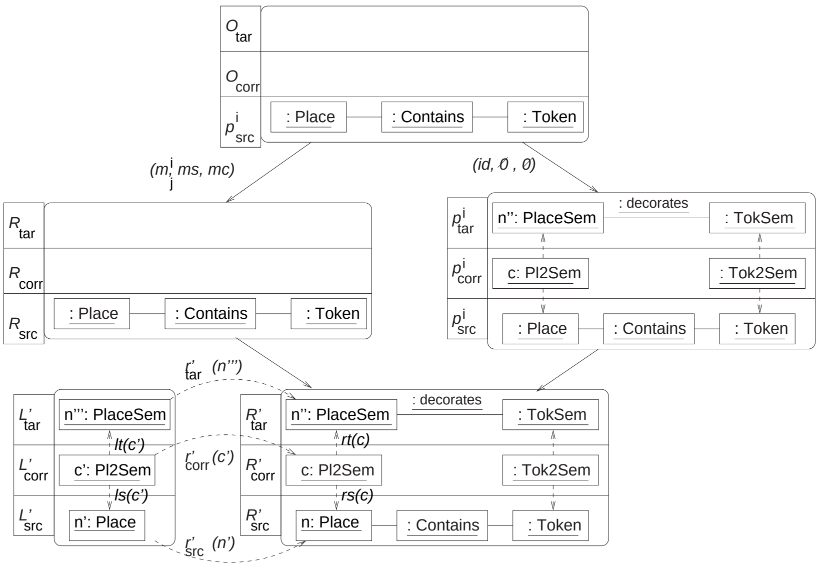


Figure 6: Steps in the Application of the Pattern in Figure 3.

Figure 7 shows additional patterns for the Petri nets example. According to the left pattern, output places of a transition in the syntactic graph are post-condition *PlaceSem* objects for the *TransSem* object associated with the transition. The pattern to the right models the correspondence for input places. By applying the three patterns to the rule in Figure 2(a) (twice the pattern for post-conditions, and once that for tokens), we obtain the operational triple rule in Figure 2(b).

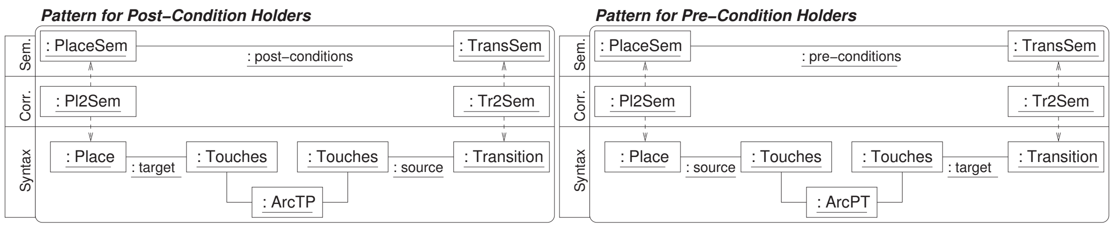


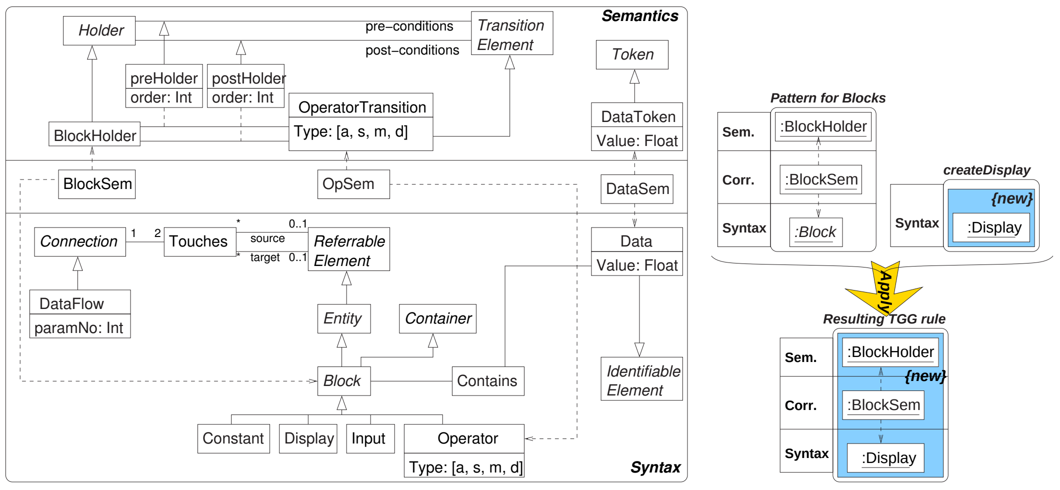
Figure 7: Additional Patterns for the Example.

The advantage of these patterns is that they are specified once, and can then be applied to complex syntactic rules. The visual language designer does not have to modify by hand each syntactic rule to add the semantic information, but only has to specify the patterns once. Moreover, the patterns do not have to take into account which elements are created and which are already existing, as this is specified in the normal rules to which patterns are applied. Thus, the pattern may be used in several ways (i.e. in parts of the rule which are newly created or in existing ones).

4 Abstract Triple Patterns

We consider now patterns with "abstract objects" and their application to rules also containing "abstract objects" (i.e. abstract rules). When looking for a match from an abstract pattern to a rule, abstract objects in the pattern can be matched with objects of more concrete classes in the rules. An abstract rule is equivalent to the set of concrete rules resulting from the valid substitutions of the abstract objects by instances of the subclasses of the abstract object class [BELT04].

In order to illustrate these concepts, we introduce a new example, which models the concrete syntax and the semantic roles for a visual language of arithmetic expressions. The meta-model triple is shown in Figure 8(a). The language is made of blocks (abstract *Block* class), which can be interconnected through data flows (*DataFlow* class). Blocks contain data values (*Data* class), which are propagated through the data flows and processed by the blocks. There are four types of blocks: constants (i.e. blocks which store non-modifiable data values), displays (blocks that output a value), inputs (blocks which capture a value from outside the system), and operators. The latter manipulate data values, and can be adders, subtractors, multipliers and dividers. Data flows contain the argument position, which is needed for non-commutative operations.



(a) (b)

Figure 8: (a) Meta-Model Triple for an Arithmetic Expressions Language. (b) Application of an Abstract Triple Pattern.

In the semantic level, blocks are considered holders, data is considered a token (with value), and operators are transitions. Differently from the Petri net case, operators have both the roles of transitions and holders (for the value resulting from the operation).

Before presenting the algorithm, we show the intuition using simple examples. Figure 8(b) shows an abstract pattern to the left. The pattern shows the desired relation between blocks (any kind of block, as *Block* is an abstract class) at the syntactic level and *BlockHolder* objects in the semantic graph. The syntactic rule to the right models the creation of a display object. The application of the pattern to the rule results in a triple rule where the block object in the triple pattern has been matched to a display object, as display has a more concrete type.

Consider now the situation depicted in Figure 9. The left part shows two patterns. The first one describes that a *BlockHolder* is a post-condition for an *OperatorTransition* object at the semantic level when an *Operator* object is connected through a *DataFlow* to a *Block* abstract object. This pattern is abstract, and would be equivalent to four patterns, resulting from the substitutions of the *Block* object by objects of each one of its subclasses. The second pattern associates an *OperatorTransition* object with an *Operator* object.

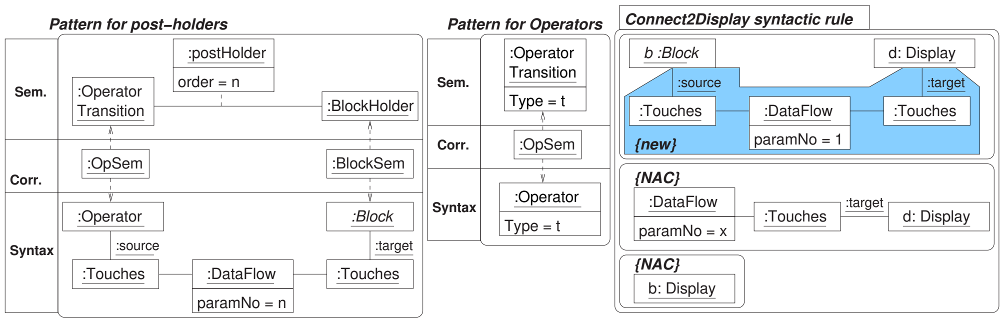


Figure 9: Abstract Triple Patterns and Abstract Rule.

The syntactic rule shown to the right models the connection of a block to a display and forbids the connection of two displays. In principle, the first pattern cannot be applied to the rule because, although the *Block* object in the pattern can get instantiated to the *Display* object in the rule, class *Operator* is more concrete than class *Block*. However, there are cases when a *Block* is an operator. Therefore what we have to do is to consider all concrete rules equivalent to the abstract one, and then apply the patterns. Here, we want to distinguish the case when an object is both a *Block* and an *Operator*, and the case where the object is a *Block* and not an *Operator*.

In order to define the construction of concrete rules from abstract patterns, we rely on the partial order $\preceq$ induced by the inheritance relationship on the set of classes in the meta-model, so that $A \preceq B$ if A inherits, even indirectly, from B, or if A is equal to B.

The following algorithm describes the previous processes.

```
AbstractApply(P: SetOfTriplePatterns, rl:  Rule):  SetOfTripleRules
```

Let $P = \{p^i\}_{i \in I}$ be a set of triple patterns of the form $p^i : p^i_{src} \xleftarrow{ps^i} p^i_{corr} \xrightarrow{pt^i} p^i_{tar}$ and $rl : L \xleftarrow{l} K \xrightarrow{r} R$ a non-deleting rule with $L = K$, and therefore $rl : L \xrightarrow{r} R$. The application of P to rl results in a set of triple rules $R'_{rl} = \{rl'_j\}$ as follows:

1. Set $R'_{rl} = \emptyset$.

2. Let $R_{rl} = \{rl_k\} \cup \{rl\}$ be the set of concrete rules equivalent to rl (see [BELT04]) and rl itself.

3. $\forall rl_k \in R_{rl}$, $R'_{rl} = R'_{rl} \cup Apply(P, rl_k)$. That is, we apply the patterns to each rule. Note that we allow a match from a pattern to a rule if a structural match is found, and if all types in the rule are more concrete or equal to the corresponding types in the abstract pattern.

4. $\forall rl' \in R'_{rl}$: if $\exists rl'' \in R'_{rl}$ s.t. rl' is more concrete than rl'' ($rl' \preceq rl''$) then $R'_{rl} = R'_{rl} \setminus \{rl'\}$. That is, we eliminate rules "subsumed" by others (same structure, equal or more concrete types).

5. $\forall rl' \in R'_{rl}$: if $\exists rl'' \in R'_{rl}$ s.t. $rl''_{src} \preceq rl'_{src}$ then add a NAC to rl' with all the nodes in rl'' that are refinements of nodes of rl', where rl'_{src} is the normal rule resulting by taking the source graphs of triple rule rl'.

Figure 10 shows the result of applying the patterns in Figures 8(b) and 9 to the abstract rule in Figure 9. The first rule considers the case when *Block* objects are not *Operators*. The second considers the case when objects are *Operators*. Note how, due to the NACs, the application of these rules is mutually exclusive

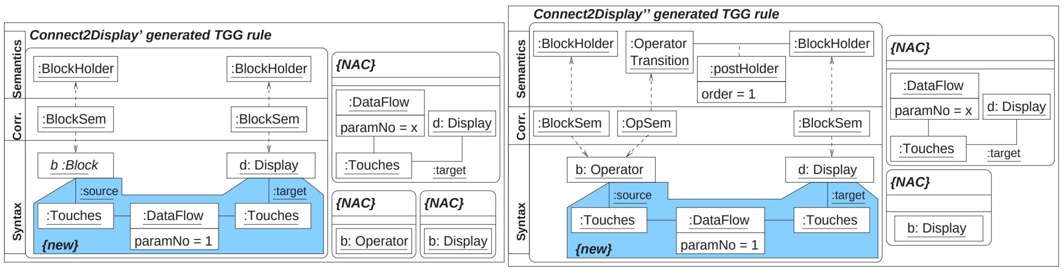


Figure 10: Generated Operational TGG rules.

5 Related Work

Our approach is inspired by the seminal work in [Sch94] and aims at providing an efficient way to obtain TGG operational rules, whenever a grammar for one of the graphs already exists. In this scenario, patterns do not need to specify which elements should be created and which should already exist in one of the graphs, as this is expressed in the normal rule to which the pattern is applied. When specifying a declarative TGG rule, one has still to indicate which elements should be present, and which ones are new. Thus, patterns may be used in several ways, which makes them more flexible and declarative than normal TGG rules as defined in [Sch94].

In addition, we have taken advantage of meta-models with the concept of abstract patterns. These are more compact than normal patterns, as they are equivalent to a number of concrete patterns resulting from the substitution of each object by instances of subclasses of the former object class. The concept of inheritance in TGG rules is of course not new, as existing TGG approaches based on meta-models such as [KS06] and [BGN$^+$04] already consider inheritance.

Our contribution in this aspect is the observation that abstract patterns can be applied to rules with more abstract typing, and that a set of operational triple rules is generated which discriminates the right types by using negative application conditions. In addition, the concepts of triple rules with inheritance is fully formalized in the DPO approach in [GL07].

Due to lack of space, we have only presented the algorithms for translating in both directions; however generating operational rules for the scenarios described in [KS06] is also possible. In particular, it is possible to produce rules for creating the correspondence graph (assuming the source and target graphs already exist), to check the validity of the correspondence graph and for incremental updates. Further developments of TGGs can also be taken into account. For example, in [GW06], an efficient algorithm for incremental transformation was suggested, by relating the created nodes in the correspondence graph. Our patterns can also be used to create such relations.

A precedent to this work can be found in [G79̈], where Göttler describes a programming language as a triple with the syntax, the semantics and a function ϕ describing how the semantic model is built from the syntactic one. In addition, he proposes meta-rules that modify either syntactic or semantic standard rules. In our case, we use triple patterns instead of meta-rules. Our triple patterns generate triple rules that are used to build the semantic model. Thus, they play the role of the ϕ function in Göttler's approach.

We believe this work is also relevant for the QVT community [QVT], as some efforts have been made to formalize QVT using TGGs. The most immediate similarities are found in the QVT relations, however attempts to formalize also the QVT Core have also been made[Gre06].

With respect to the application area of visual languages, Baar has proposed the use of TGGs to connect concrete and abstract syntax, so as to make it possible the static verification of the compliance between both [Baa06]. His proposal is however related to the structure of the visual sentence, and not to its operational interpretation. Moreover, it does not exploit inheritance, and requires the presence of display managers, relating the abstract and the concrete syntaxes.

6 Conclusions and Future Work

In this paper we have presented *Triple Patterns* as a compact means to obtaining operational TGG rules starting from normal graph grammar rules. We have shown that the approach is suitable for its combination with meta-modelling by considering the inheritance hierarchy in the meta-models. We have applied these ideas to the synchronized evolution of syntax and semantic models, improving previous work in [BDD$^+$04].

There are several open issues. The first one is to study to which degree patterns can be automatically derived from the meta-model triple. In general this process cannot be fully automated. However, for our particular application case, it could be possible, as we just model three kinds of structures at the semantic level: *Tokens* decorating *Holders*, and *Holders* being pre- and post-conditions for *TransitionElements*. We could take the information of which classes in the semantic meta-model inherit from the base classes, and to which classes they are related in the syntax graph. However, for general applications, only an approximation can be derived.

Although not explicitly mentioned, our abstract patterns can only have abstract objects in the source graph, as, typically, concrete elements in the target graph are created. One can however extend the notion of abstract rule [BELT04] to allow abstract nodes also in *R* and in the target graph. For our application, this would allow the designer to include predefined patterns in the semantic level (using only predefined classes *Token*, *Holder* and *TransitionElement*), thus helping towards the automatic generation of the triple patterns from the meta-model triple.

Up to now we have not considered problems related to the manipulation of attributes, which is up to future work. In addition we have only considered positive patterns, but we are currently working to extending this approach with patterns containing also negative conditions. The algorithms we have presented assume that the syntactic rules are "bigger" than the patterns. The study of the opposite case is still an open problem. In principle, by considering partial matches from patterns to rules, one could devise ways to generate additional triple rules with extended context. It is also worth studying whether we can extend the application of a pattern to general rules (i.e. not only deleting or non-deleting). A first line of attack might consider the splitting of a general rule into a sequence of one deleting and one non-deleting rule.

Finally, we are considering the application of the notion of triple graph grammars and meta-rules to the generation of operational semantics, for example the token game in the case of Petri nets.

Acknowledgements: This work has been partially sponsored by the Spanish Ministry of Education and Science with projects MOSAIC (TSI2005-08225-C07-06) and MODUWEB (TIN 2006-09678), and the EC's Human Potential Programme under contract HPRN-CT-2002-00275, SegraVis. The authors gratefully thank the referees for their useful suggestions.

References

[Baa06] T. Baar. Correctly defined concrete syntax for visual models. In *Proc. MoDELS/UML 2006*. LNCS 4199, pp. 111–125. Springer, 2006.

[BDD$^+$04] P. Bottoni, M. De Marsico, P. Di Tommaso, S. Levialdi, D. Ventriglia. Definition of visual processes in a language for expressing transitions. *J. Vis. Lang. Comput.* 15(3-4):211–242, 2004.

[BELT04] R. Bardohl, H. Ehrig, J. de Lara, G. Taentzer. Integrating Meta-modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In *Proc. FASE*. LNCS, pp. 214–228. Springer, 2004.

[BG04] P. Bottoni, A. Grau. A Suite of Metamodels as a Basis for a Classification of Visual Languages. In *Proc. VL/HCC*. Pp. 83–90. IEEE Computer Society Press, 2004.

[BGN$^+$04] S. Burmester, H. Giese, J. Niere, M. Tichy, J. Wadsack, R. Wagner, L. Wendehals, A. Zündorf. Tool integration at the meta-model level: the Fujaba approach. *J. Softw. Tools Technol. Transfer* 6:203–218, 2004.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[G7̈9] H. Göttler. Semantical Description by Two-Level Graph-Grammars for Quasihierarchical Graphs. In *Proc. Workshop Graph Theoretic Concepts in Computer Science – Graphs, Data Structures and Algorithms*. Applied Computer Science 13. Carl Hansen Verlag, 1979.

[GL07] E. Guerra, J. de Lara. Event-Driven Grammars: Relating Abstract and Concrete Levels of Visual Languages. *To appear in Software and Systems Modeling (SoSyM)*, 2007.

[Gre06] J. Greenyer. A Study of Model Transformation Technologies: Reconciling TGGs with QVT. Master/diploma thesis, University of Paderborn, July 2006.

[GW06] H. Giese, R. Wagner. Incremental Model Synchronization with Triple Graph Grammars. In *Proc. MoDELS/UML 2006*. LNCS 4199, pp. 543–557. Springer, 2006.

[KS06] A. Konigs, A. Schürr. Tool Integration with Triple Graph Grammars - A Survey. *Electronic Notes in Theoretical Computer Science* 148:113–150, 2006.

[MSUW04] S. J. Mellor, K. Scott, A. Uhl, D. Weise. *MDA Distilled*. Addison-Wesley, 2004.

[QVT] QVT specification, at http://www.omg.org/docs/ptc/05-11-01.pdf.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Proc. WG'94*. LNCS, pp. 151–163. Springer, 1994.