Electronic Communications of the EASST
Volume 29 (2010)

Proceedings of the Ninth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2010)

Preserving constraints in horizontal model transformations

Paolo Bottoni, Andrew Fish, Francesco Parisi Presicce

14 pages

Guest Editors: Jochen Küster, Emilio Tuosto
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Preserving constraints in horizontal model transformations

Paolo Bottoni[1], **Andrew Fish**[2], **Francesco Parisi Presicce**[1]

[1] Dipartimento di Informatica, "Sapienza" Università di Roma, Italy, [2]Computing, Mathematical and Information Sciences, University of Brighton, UK

Abstract: Graph rewriting is gaining credibility in the model transformation field, and tools are increasingly used to specify transformation activities. However, their use is often limited by special features of graph transformation approaches, which might not be familiar to experts in the modeling domain. On the other hand, transformations for specific domains may require special constraints to be enforced on transformation results. Preserving such constraints by manual definition of graph transformations can be a cumbersome and error-prone activity. We explore the problem of ensuring that possible violations of constraints following a transformation are repaired in a way coherent with the intended meaning of the transformation. In particular, we consider the use of transformation units within the DPO approach for intra-model transformations, where the modeling language is expressed via a type graph and graph conditions. We derive additional rules in a unit from a declarative rule expressing the principal objective of the transformation, so that the constraints set by the type graph and the graph conditions hold after the application of the unit. The approach is illustrated with reference to a diagrammatic reasoning system.

Keywords: DPO, automatic generation, model transformation.

1 Introduction

Graph rewriting-based tools are increasingly used in the field of model transformation. However, their use is often limited by the special features of the different graph transformation approaches, which might not be familiar to experts in the modeling domain. On the other hand, transformations for specific domains may require constraints to be enforced on the results of the transformation. In this paper we explore the problem of ensuring that possible violations of constraints are managed in a way coherent with the intended meaning of the transformation.

We consider horizontal (or in-place) model transformations which destructively update a model expressed in a given language, for the case where the modeling language is expressed via a type graph and a set of graph conditions. In particular, we study transformations in reasoning processes deriving inferences via logical steps creating or deleting model elements.

While modelers are generally clear on what they want to achieve by defining a transformation, the evaluation of all of its consequences may be complex, and the definition of the implied preserving or enforcing actions cumbersome and error-prone.

We propose an approach to the automatic construction of transformation units achieving the effect of an intended model transformation while ensuring that all conditions are satisfied at the end of the unit if they held at its start. We consider transformations consisting of the creation or deletion of elements of a specific type, expressed as *principal* declarative rules. As their appli-

cation may violate some conditions, they have to be applied in a proper (*condition preserving*) context, or (*condition enforcing*) repair actions have to be taken to restore the satisfaction of such conditions. Hence, additional rules are defined, derived from the principal one and the conditions to be enforced. The approach is illustrated with reference to a diagrammatic reasoning system.

Paper organisation. Section 2 discusses related work on constraint preservation in graph transformation, and Section 3 provides the relevant formal notions. Section 4 introduces Spider Graphs (SGs) as running example, before presenting the approach in Section 5 and applying it to SGs in Section 6. Finally, Section 7 draws conclusions and points to future developments.

2 Related work

Rensink and Kuperus have exploited the notion of nested graphs to deal with the amalgamated application of rules to all matches of a rule. In [RK09], they define a language to specify nested graph formulae. A match can be found from a nested graph rule to a graph satisfying a formula, according to a given morphism, and the application of a composite rule ensues. Their approach is focused on avoiding control expressions when all the matches of a rule have to be applied, while we focus here on preserving constraints with reference to a single match.

Bottoni *et al.* have defined methods to extend single declarative rules for model transformation so that they comply with specific patterns defining consistency of interpretation in triple graphs [BGL08]. They define completions of single rules with respect to several patterns, while we are interested here in constructing several rules, navigating along different sets of constraints.

Taentzer *et al.* have proposed the management of inconsistencies among different viewpoints of a model in distributed graph rewriting. For example, the *resolve* strategy requires the definition of the right-hand sides of rules to be applied when the left-hand side identifying the inconsistency is matched [GMT99]. The detection of inconsistencies between rules representing different model transformations has been attacked by static analysis methods in [HHT02]. Similarly, Münch *et al.* have added *repair actions* to rules in case some post-conditions are violated by rule application [MSW00]. In all these cases, actions were modeled through single rules.

Habel and Pennemann [HP09] unify theories about application conditions from [EEHP06] and nested graph conditions from [Ren04], lifting them to high-level transformations. They transform rules to make them preserve or enforce both universal and existential conditions. Their approach leads to the generation of a single rule incorporating several application conditions derived from different conditions with reference to the possible matches of the rule on host graphs. In his dissertation [Pen09], Pennemann expands on the topic, also introducing programs with interfaces, analogous to transformation units, but allowing passing of matches.

In [OEP08], Orejas *et al.* define a logic of graph constraints to allow the use of constraints for language specification, and to provide rules for proving satisfaction of clausal forms.

The idea of introducing basic rules derived from entities and associations defined in a meta-model is exploited in [BQV06] to define constraints on the interactive composition of complex rules, by allowing their presence in the rule left or right-hand sides only in accordance with their roles in the meta-model, where only the abstract syntax is taken as a source of constraints.

Ehrig *et al.* describe a procedure, exploiting layers, which derives a grammar to generate (rather than transform) instances of the language defined by a meta-model with multiplici-

ties [EKTW06]. Satisfaction of OCL constraints is checked a posteriori on a generated instance.

3 Background

For a graph $G = (V(G), E(G), s, t)$, $V(G)$ is the set of *nodes*, $E(G) \subset V(G) \times V(G)$ the set of *edges* and $s,t : E \to V$ the *source* and *target* functions. In a *type graph* $TG = (V_T, E_T, s^T, t^T)$, V_T and E_T are sets of node and edge types, while $s^T : E_T \to V_T$ and $t^T : E_T \to V_T$ define source and target node types for each edge type. G is typed on TG via a graph morphism $type\colon G \to TG$, where $type_V : V \to V_T$ and $type_E\colon E \to E_T$ preserve s^T and t^T, i.e. $type_V(s(e)) = s^T(type_E(e))$ and $type_V(t(e)) = t^T(type_E(e))$. $|\, V(G) \,|_t$ is the number of nodes of type $t \in V_T$ in G.

A DPO rule [EEPT06] consists of three graphs: left- and right-hand side (L and R) and interface graph K. Two morphisms[1] $l\colon K \to L$ and $r\colon K \to R$ model the embedding of K (containing the elements preserved by the rule) in L and R. Figure 1 shows a DPO direct derivation diagram. Square (1) is a pushout (i.e. G is the union of L and D through their common elements in K), modeling the deletion of the elements of L not in K, while pushout (2) adds the new elements, i.e. those present in R but not in K. Figure 1 also illustrates the notion of *negative application condition* (NAC), as the association of a set of morphisms $n_i\colon L \to N_i$, also noted $NAC \xleftarrow{\vec{n}} L$, with a rule. A rule is applicable on G through a match $m\colon L \to G$ if there is no morphism $q_i\colon N_i \to G$, with N_i in NAC, commuting with m (i.e. $q_i \circ n_i = m$). We exploit the partial order $\leq$ induced, up to isomorphisms, by monomorphisms on the set of graphs, i.e. $g_1 \leq g_2 \Leftrightarrow \exists m : g_1 \hookrightarrow g_2$.

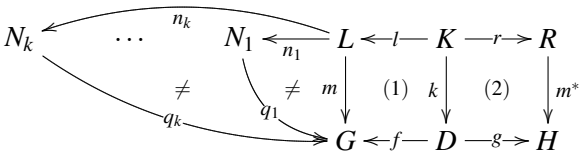

Figure 1: DPO Direct Derivation Diagram for rules with NAC.

Graph conditions allow the specification of models by forbidding the appearance of certain subgraphs, or by enforcing others to appear in given contexts. We use here a class of conditions $\mathcal{Q}$ similar to those in [HP09], where a condition over a graph A is either of the form `true` or of the form $\exists(a,q)$, with $a : A \to Q$ a morphism from A to some graph Q and q a condition over Q. Conditions are also obtained by using the Boolean connectives $\neg$ and $\vee$, and can be written in the form $\forall(a,q)$, equivalent to $\neg\exists(a,\neg q)$. We assume that all conditions in a set $\Theta \subset \mathcal{Q}$ differ for the a morphism, so that $(a_1,q_1),(a_2,q_2) \in \Theta \Rightarrow (A_1 \not\simeq A_2) \vee (Q_1 \not\simeq Q_2)$. We will also use the short forms $\exists(Q)$ for $\exists(a : \emptyset \to Q, true)$ and $\nexists(Q)$ for $\neg\exists(a : \emptyset \to Q, true)$. We restrict here to *positive* conditions of types $\exists(Q)$ or $\forall(a : \emptyset \to Q, q)$, noted $\forall(Q,q)$ with $q = \bigvee_{j\in J} q_j : Q \to W_j$ a disjunction of existential conditions. In this case, all the conditions of the form $\exists(Q_i) \in \Theta$ can be collapsed into a single condition $\exists\overline{Q}$, with $\overline{Q}$ the colimit of all Q_i on the diagram constructed with all pairwise maximal common subgraphs. Simple *negative* conditions have the form $\nexists(Q)$.

Definition 1 Given a graph G, we say:

[1] In this paper, when we speak of morphisms, we will always consider them injective.

- A morphism $m : X \to G$ satisfies a condition C, ($m \models C$), *iff* one of the following holds:
  1. $C = true$.
  2. $C = \exists(Y)$ and $Y \leq X$.
  3. $C = \forall(X, \bigvee_{j \in J} q_j : X \to W_j)$ and $\exists m_j : m(X) \to W_j$ s.t. $q_j = m_j \circ m$ for some q_j.
  4. $C = \nexists(Y)$ and $Y \not\leq X$.
  5. $C = C_1 \vee C_2$ and $m \models C_1$ or $m \models C_2$.
- A graph G *satisfies* C ($G \models C$), *iff* one of the following holds:
  1. $C = true$.
  2. $C = \exists(Y)$ and there exists $m : Y \to G$ s.t. $m \models C$.
  3. $C = \forall(X, q)$ and for each $m : X \to G$, $m \models C$.
  4. $C = \nexists(Y)$ and there is no morphism $m : Y \to G$.
  5. $C = C_1 \vee C_2$ and $G \models C_1$ or $G \models C_2$.

We say that a graph G typed on TG is a *model* for Θ, noted $G \models \Theta$, if for each $C_i \in \Theta$, $G \models C_i$. We assume Θ to be a consistent set of conditions, whose models are finite non-empty graphs; in particular, simple graphs, with no two instances of the same edge type between two nodes.

Transformation units control rule application through control words over rule names [KKS97]. Given: 1) $\mathscr{G}$ the class of typed graphs; 2) $\mathscr{R}$ the class of DPO rules with NACs on $\mathscr{G}$; 3) $\Longrightarrow$ the DPO derivation relation; 4) $\mathscr{E}$ a class of graph expressions (here defined by type graphs and graph conditions), where the *semantics* of an expression e is a subclass $sem(e) \subset \mathscr{G}$; 5) $\mathscr{W}$ a class of control words over identifiers of rules in $\mathscr{R}$ exploiting single rules, the sequential construct ';', the iteration construct w^*, with $w \in \mathscr{W}$, the alternative choice '|'; a *transformation unit* is a construct $TU = (e_1, e_2, P, imp, w)$, with $e_1, e_2 \in \mathscr{E}$ initial and terminal graph class expressions, $P \subset \mathscr{R}$ a set of DPO rules, imp a set of references to other, *imported*, units, whose rules can be used in the current one, and $w \in \mathscr{W}$ a control word enabling rules from P, and units from imp, to be applied. TUs have a transactional behaviour, i.e. a unit succeeds *iff* it can be executed according to the control condition; it fails otherwise. The semantics of a TU is the set $sem(TU) = \{(g_1, g_2) \mid g_1 \in sem(e_1), g_2 \in sem(e_2), g_1 \overset{TU\downarrow}{\Longrightarrow} g_2\}$, where $\downarrow$ indicates successful termination.

4 A Running Example: Spider Diagrams and Spider Graphs

Spider Diagrams are a reasoning system based on Euler diagrams. Several variants exist, differing in syntax and semantics [HMT+01]. We adopt a simplified version, based on Venn, rather than Euler, diagrams and omitting shading and strands. We first provide an indication of the concrete syntax of the diagrams and an informal semantics. Then we propose a graph-based abstract model for them, called *Spider Graphs*, which differs from the usual algebraic abstract models and is in fact slightly closer to the concrete model, even modelling spider's feet.

Let $C = \{C_1, \ldots, C_n\}$ be a collection of simple closed curves in the plane with finitely many points of intersection between curves. A *zone* is a region of the form $X_1 \cap \cdots \cap X_n$, where $X_i \in$

$\{int(C_i), ext(C_i)\}$, the interior of C_i or the exterior of C_i, for $i \in \{1,\ldots,n\}$. If each of the 2^n possible zones of C are non-empty and connected then C is a *Venn diagram* (see [Rus97] for more details). Each zone z defines a unique partition of the set C, according to whether z is *inside* or *outside* a curve. Two zones are called *twins* if their inside and outside relations are switched for exactly one curve. In this paper, a *Spider Diagram* is a Venn diagram whose curves are labelled, together with extra syntax called *spiders*, which are trees whose vertices (called *feet*) are placed in unique zones. The set of zones containing a spider's feet is called its *habitat*. Special arcs, called *ties*, can be drawn between feet of different spiders in the same zone.

Intuitively, each curve represents a given set (indicated by the label) and each zone represents some set intersection. A spider indicates the existence of an element within the set determined by its habitat, whilst a tie between a pair of feet of different spiders within a zone indicates equality of elements, if both spiders represent an element in the set represented by the zone.

Figure 2 (left) shows an example of a Spider Diagram, with two curves $\{A,B\}$ and four zones described by $\{(\{A\},\{B\}), (\emptyset,\{A,B\}), (\{B\},\{A\}), (\{A,B\},\emptyset)\}$. Here, these zones are the four minimal region of the plane determined by the curves; for example, the zone described by $(\{A\},\{B\})$ is the region $int(A) \cap ext(B)$ which is inside A but outside B. The habitat of spider s is the set of zones $\{(\{A\},\{B\}),(\{A,B\},\emptyset)\}$, while that of t is the singleton $\{(\{A,B\},\emptyset)\}$. Informally, the diagram semantics is: there are two sets A and B, there exists an element named s in A and an element named t in $A \cap B$. Moreover, if s is in $A \cap B$ then $s = t$.

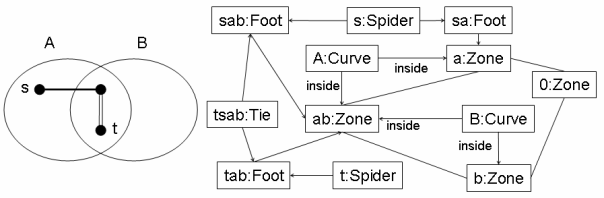

Figure 2: A Spider Diagram on the left, with the corresponding Spider Graph on the right.

We provide here an abstract graph-based model of a Spider Diagram, called a *Spider Graph*, not taking into account its concrete geometry. Since we are interested here only in syntactic aspects, we do not consider the labeling of the curves. We obtain the type graph of Figure 3 (left), where nodes represent the diagram elements *Curve*, *Foot*, *Spider* and *Zone*, and edges represent relations between them. A *twin* edge indicates that two zones are twins w.r.t. some curve and an *inside/outside* edge indicates whether a curve contains/excludes a zone, respectively.

In Figure 2 (right) the Spider Graph associated with the Spider diagram on the left is shown. The names of the nodes show the correspondence with the objects in the diagram. We have two curve nodes in each possible relation with four zones[2]. For ease of reading, the zone nodes are given names consisting of a list of the lower case letters corresponding to the upper case letters used as names of the curves the zones are inside, and we use O for the name of the node corresponding to the zone outside all curves in the diagram. Zone node pairs ab and b, and O and a are twinned due to curve A, whilst ab and a, and O and b are twinned due to curve B.

We now present the conditions completing the definition of the class of Spider Graphs. Fig-

[2] To keep the graph simple, we have omitted the *outside* edges, which are complementary to the *inside* ones.

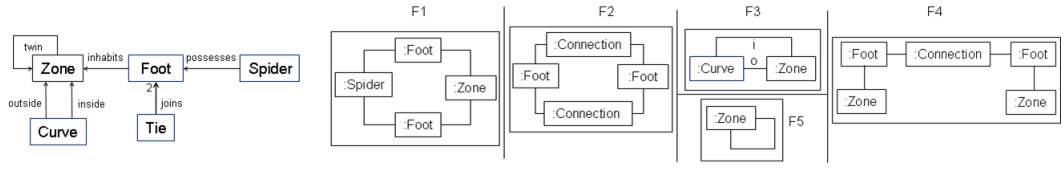

Figure 3: The type graph (left) and negative conditions (right) for Spider Graphs.

ure 3 (right) shows a set of conditions of the form $\nexists Q$, presented as forbidden graphs. They prevent duplication or inconsistency of information and state the uniqueness of relations between zones and curves. Moreover, we assume the existence of all negative conditions forcing the graphs to be simple. We omit the direction of edges and their labels, when understood from the type graph, and use the abbreviations `i` and `o` for the inside/outside case. The remaining conditions force the existence of a partition of the set of curves for all zones, and require suitable contexts for zones and feet. We present them adopting a visual syntax where a condition $\exists(a : A \to Q, q)$ is represented by a box, separated into two parts by a horizontal line, with the top part containing a depiction of the morphism a and the bottom part containing a box depicting the condition q on Q. An empty bottom box corresponds to `true`. Each condition box has an external tab containing either quantifier information or the boolean connective $\vee, \wedge$ or $\neg$. As we use conditions with $A = \emptyset$, we only present Q and we do not repeat Q in the depiction of q. Numbers indicate identification in the morphisms, while not numbered nodes indicate a hidden existential quantification, as usual. Edges between identified nodes are also assumed to be identified in the morphisms. The class of Spider Graphs is the intersection of the languages defined by the type graph and the negative conditions of Figure 3, and the positive conditions in Figures 4 to 6.

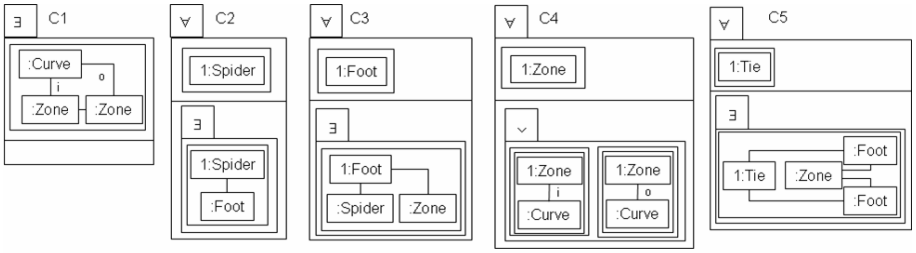

Figure 4: Conditions on single elements.

Reasoning rules are derived on top of the algebraic abstract models for Spider diagrams. These are syntactic transformations whose application corresponds to logical deduction, according to the semantics. They are usually specified by complex algorithmic procedures, during which the intermediate diagrams may not be logical consequences of the premise diagram, with pre and post conditions taking into account the stated semantics of the diagram. For instance a rule to add a new curve must split every zone into two zones, one inside and one outside each existing zone, as well as duplicating spider's feet in zones. Whereas the first effect derives from the syntactical conditions, the second is a semantic aspect.

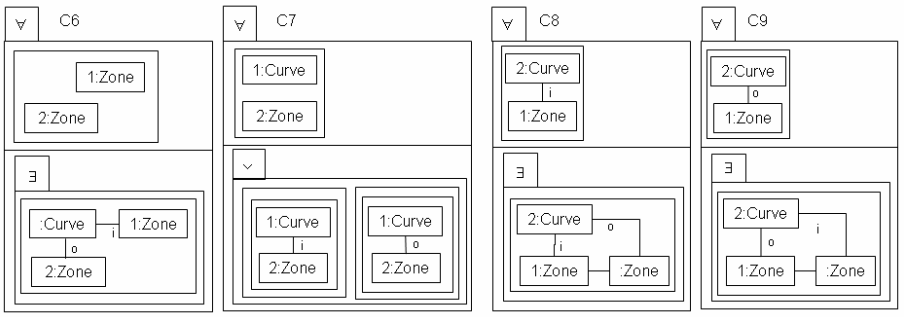

Figure 5: Conditions on pairs of elements.

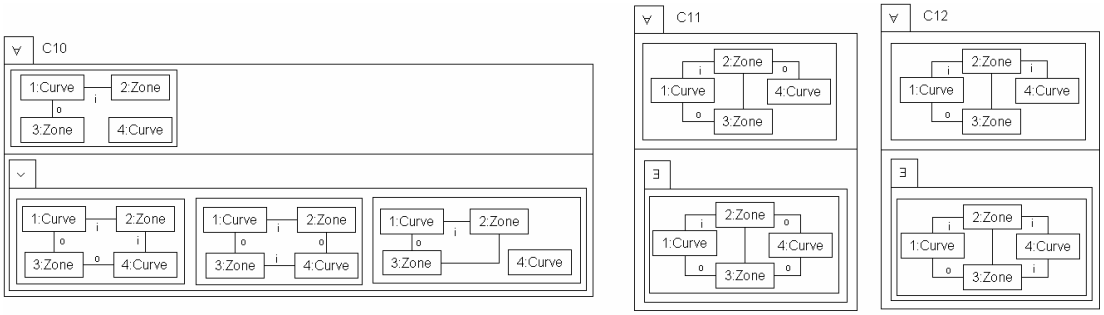

Figure 6: Conditions on existence and uniqueness of twins.

5 Condition preserving rules

We discuss the derivation of a condition-preserving transformation unit TU_g^t for the generation of an element of type t. The initial and terminal expressions e_1 and e_2 for TU_g^t define the class of graphs typed on TG and satisfying Θ. TU_g^t is associated with the execution of[3] $r : \emptyset \leftarrow \emptyset \rightarrow \boxed{t}$ and is constructed so that given a graph $G \in sem(e_1)$, for $G \overset{TU_g^t\downarrow}{\Longrightarrow} H$, $(G,H) \in sem(TU_g^t)$, and $G \leq G + \boxed{t} \leq H$, where $+$ indicates the pushout along the empty subgraph.

Note that in general $G + \boxed{t} \not\models \Theta$, but $G + \boxed{t} \models \Theta'$ for some $\Theta' \subset \Theta$. Hence, we admit that some conditions may not be satisfied at intermediate steps of the unit application, and define an *operational* class in which to perform transformations. Graphs in this class satisfy a subset of the graph conditions and may be typed on some TG' with additional types and edges w.r.t. TG. In particular, we use here the subset Θ' containing $\exists(\overline{Q})$ and all the conditions $\nexists(Q_i)$ in Θ.

Before presenting the algorithm, we give its rationale. We only have to consider universal and negative existential conditions, as positive existential conditions cannot be violated by adding an element. However, adding $\boxed{t}$ produces a graph $G + \boxed{t}$ which may not satisfy Θ in two ways: either it contains a forbidden subgraph, or it provides a new match for the premise of a universal condition, but it fails to satisfy the conclusion.

[3] Here and in the rest of the paper, $\boxed{t}$ denotes the graph consisting of a single node of type t.

To solve the first problem, given[4] a rule $r : L \to R$ in TU_g^t (including $r : \emptyset \to \boxed{t}$), for each condition $\nexists(X) \in \Theta$, the function $genNAC(r,X)$ adds to r the set of NACs formed according to the construction in Figure 7 (left). Here M_j is a maximal common subgraph of R and X and M'_j is a maximal common subgraph to M_j and L, s.t. all the squares are pushouts. Hence, $L \to X'_j \leftarrow X_j$ is the pushout for $L \leftarrow M'_j \to X_j$, with the second morphism given by arrow composition. The set of NACs contains all the morphisms $n'_j : L \to X'_j$ preserving the image of L in X_j. This prevents the application of r on a match which could create the forbidden subgraph X (see [HHT96]).

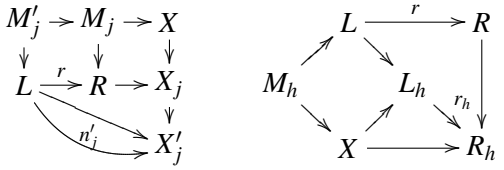

Figure 7: Constructing NAC (left) and incorporating available context (right).

To solve the second problem, given a (universal) condition $C = \forall(Q, \bigvee_{j\in J} q_j : Q \to W_j)$, s.t. $\boxed{t} \leq Q$, the function $genUniRules(C)$ produces the set of rules $R(C)$ where each rule has the form $NAC(C) \overset{\overrightarrow{n}}{\leftarrow} Q \overset{r_{C,j}}{\to} W_j$. TU_g^t will contain an alternative choice among these rules, produced by the function $alt(R(C))$. In order to prevent these rules from being applied indefinitely in case of iteration on the choice, $NAC(C)$ contains a copy of each W_j so the same match is not reused twice. Intuitively, these rules adjust the relations of the newly added element w.r.t. the contexts defined in their premises. However, several aspects have to be taken into account. For example, consider conditions $C2$ in Figure 4 and suppose we want to add a *Spider*. Then, the derived rule will have to create a *Foot* (condition $C2$), but this will require a *Zone* (condition $C3$), which will require a *Curve* (condition $C4$), hence other additional *Zones* (conditions $C8$ and $C9$), with several relations to other curves and zones (conditions $C10-C12$). On the other hand, a *Zone* for a *Foot* is already guaranteed to be present by $C1$, so that one can reuse existing context to satisfy this. To deal with such situations, given a rule $r : L \to R$ and a context X to be reused (more on this later), the function $reuseContext(r,X)$ produces a collection of rules of the form $r_h : L_h \to R_h$ according to the construction in Figure 7 (right). Here, $L \to L_h \leftarrow X$ is the pushout along a maximal common subgraph M_h of L and X and $X \to R_h \leftarrow R$ is the pushout of $X \leftarrow M_h \to R$.

In general, one wants to obtain a TU_g^t which, after applying $r : \emptyset \to \boxed{t}$ to G, proceeds through the following abstract steps, so that context is progressively constructed for the next step.

1. define all edges between the added node and existing nodes of G as required by conditions;
2. generate new nodes as required by the conditions;
3. generate all edges for the new nodes, as required by the conditions.

For example, when adding a `Curve`, one has to: 1) define relations between the new curve and existing zones; 2) create new zones, while defining relations with the new curve; 3a) establish relations between new zones and existing curves; 3b) establish relations between zones.

[4] Where not needed, we will omit K.

Two things have to be considered. In general, satisfaction of $\forall(Q,q)$ requires iterating through all possible matches for Q. However, when Q consists of just one node, no iteration is necessary, and if Q is the graph $\boxed{t}$, the derived rule has to be applied only to the newly added node, as it is already satisfied for the nodes of type t which were in G originally. Hence, we extend TG to admit a special type of loop edge: the first rule is changed to $r : \emptyset \to \boxed{t}^\dagger$, where $\boxed{t}^\dagger$ designates a node with a marker loop. For a rule[5] $r : L \to R$, the function $mark(r)$ produces a set $P_r^\dagger = \{r_h^\dagger : L_h^\dagger \to R_h^\dagger \mid h : \boxed{t} \to L\}$ where $L_h^\dagger$ and $R_h^\dagger$ are obtained by adding the loop to the images $h(L)$ and $r \circ h(L)$, the immersions $m_h : L \hookrightarrow L_h^\dagger$ and $m'_h : R \hookrightarrow R_h^\dagger$ preserve such images, and $r_h^\dagger$ is the unique morphism s.t. $L_h^\dagger \overset{r_h^\dagger}{\to} R_h^\dagger \overset{m'_h}{\leftarrow} R$ is the pushout of $L_h^\dagger \overset{m_h}{\leftarrow} L \overset{r}{\to} R$. TU_g^t will apply r or rules from $P_r^\dagger$ in different situations. The rule $delLoop : \boxed{t}^\dagger \to \boxed{t}$ will conclude TU_g^t deleting the loop.

Moreover, as in the examples above, some rules create new nodes if they cannot be provided by the context, and so conditions relative to the new nodes have to be satisfied. This potentially creates a situation in which an infinite recursion might start. To avoid this, we study the relations between types for which conditions are mutually recursive. In our example one such pair consists of `Curve` and `Zone`. Indeed, generating a curve implies the generation of a collection of zones, whilst the generation of a zone can imply the generation of a single curve and of the collection of zones related to the new curve: we need to distinguish between situations in which context, enriched with the new node which has started the process, has to be reused, and those in which a new node is needed to provide the correct context. Definition 2 provides the needed notation.

Definition 2 Let $t \in V_T$ be a type and $Q(t) \subset \Theta$ the set of conditions of the form $Op(a : A \to Q, q)$, for $Op \in \{\exists, \forall, \nexists\}$ s.t. $\boxed{t} \leq Q$ (i.e. a node of type t appears in Q). $\{Q_\exists(t), Q_\forall(t), Q_\nexists(t)\}$ is a partition of $Q(t)$ into *existential*[6], *universal and negative existential* conditions for t, respectively. $V_T^\exists = \{t \mid \boxed{t} \leq \overline{Q}\}$ is the set of *existentially quantified types*. A partial order $\leq_C$ is induced on $Q_\forall(t)$ by $(C_1 <_C C_2) \Leftrightarrow ((A_1 < A_2) \vee ((A_1 \simeq A_2) \wedge (Q_1 < Q_2)))$. $DAG(t) = (Q(t), \prec, s, t)$ is the directed acyclic graph induced on $Q(t)$, where $(q_1, q_2) \in \prec \Leftrightarrow q_1 <_C q_2 \wedge \nexists q_x$ s.t. $q_1 <_C q_x$, $q_x <_C q_2$. We call $Min(t)$ the set of minimal models for $\Theta \cup \{\exists(\overline{Q} + \boxed{t})\}$ for $t \in V_T \subset V_T^\exists$ and $MIN(S)$ the set of minimal models for $\Theta \cup \bigcup_{t \in S \subset V_T} \{\exists(\overline{Q} + \boxed{t})\}$.

For each condition $C \in Q_\forall(t)$ the rules in $genUniRules(C)$ will be applied in an order established by a function $visit(DAG(t))$ which starts from initial nodes and proceeds from a join node only after all its incoming paths have been visited. In this way, progressively increasing contexts will have been produced, possibly providing new matches for the subsequent rules.

In order to follow the abstract steps discussed above, for a type t we organize the rules derived from $Q_\forall(t)$ into layers: $LAYER_1(t)$ contains rules which only add edges touching nodes of type t, $LAYER_2(t)$ contains rules which add at least one node (of any type) in a non-empty context (and possibly edges of any type), whilst $LAYER_3(t)$ contains rules which do not create nodes but add edges of any type, but with at least one edge between instances of some type other than t.

The sets $Min(t)$ provide context which is certainly present if a unit for the addition of an element of type t has already been applied, while $\overline{Q}$ is guaranteed to be always present. Hence, *reuseContext* will be invoked with parameter X equal to $\overline{Q}$ or $Min(t)$, depending on the situation.

[5] For each function operating on rules or types we overload the symbol to accept as argument sets.
[6] Note that $Q_\exists(t) = \{\exists(\overline{Q})\}$ if $\boxed{t} \leq \overline{Q}$, and $Q_\exists(t) = \emptyset$ otherwise.

Moreover, if an element of type t' is created as a consequence of the generation of $\boxed{t}$, rules derived from visiting $DAG(t')$ have also to be applied, in the context provided by the already applied rules. Hence, we introduce a notion of *domination* and a predicate $dominates(t,t') \equiv DAG(t') \leq DAG(t)$. Figure 8 shows the *DAG*s for the example introduced in Section 4. When adding a new zone, as we have $dominates(Zone, Curve)$, the construction of TU_g^{curve} should recursively be invoked. But then, rules from $Q_\forall(curve)$ would create new zones, thus requiring the invocation of rules from $Q_\forall(zone)$, etc. Hence, in the context of the construction of TU_g^t if $DAG(t') \leq DAG(t)$, then the rules from the conditions in $Q_\forall(t')$ are generated used via *reuseContext*, with $X = MIN(t')$, to take into account that the minimal context for t' can already exist. Also, a function $create(r)$ returns the set of types produced by r, i.e. in $V_T(R) \setminus V(L)$.

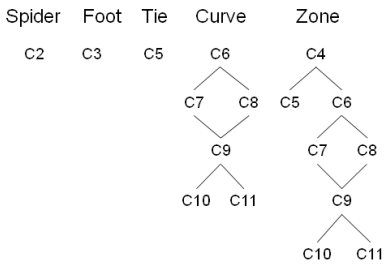

Figure 8: The DAGs for Spider Graphs.

The resulting algorithm *CreateGenUnit(t)* populates TU_G^t with rules derived from $Q_\forall(t)$, with added NACs to preserve conditions in $Q_\nexists(t)$, and organizes them according to ordering and layering: rules are applied only when context for their application is ready.

`Algorithm` $CreateGenUnit(t$ `:type)` `:TU`

initialize $UNIT$ with $r_t^\dagger : \emptyset \rightarrow \boxed{t}^\dagger$;
foreach condition $C = \forall(Q,q) \in \Theta$ **do** { $R(C) = genUniRules(C)$; }
return $RecursiveGen(t, \emptyset, false)$;

`Algorithm` $RecursiveGen(t$ `:type,` S`:setOfTypes,`$inner$`:boolean):TU`

$path = visit(DAG(t))$; $X = \emptyset$; $aux = \emptyset$;
if $isEmpty(S)$ **then** { **if** $t \in V_T \setminus V_T^\exists$ **then** { $X = \overline{Q}$; } } **else** { $X = MIN(S)$; }
foreach condition $C = \forall(Q, \bigvee_{j\in J} q_j : Q \rightarrow W_j) \in path$ **do** {
  foreach $k \in \{1,\ldots,3\}$ **do** {
    foreach $t' \in S$ { **if** $dominates(t,t')$ { $aux = aux \cup \{t'\}$ } };
    if $!isEmpty(aux)$ **then** { $X = MIN(aux)$ }; $single = \emptyset$; $nosingle = \emptyset$;
    foreach rule $r_{C.h} = NAC \xleftarrow{n} L \rightarrow R \in R(C) \cap LAYER_k(t)$ **do** {
      if $|\, V(L) \,| = 1$ **then** { $single = single \cup \{r_{C.h}\}$ } **else** { $nosingle = nosingle \cup \{r_{C.h}\}$; };
      if($inner$) **then** { $UNIT = concat(UNIT, alt(reuseContext(single, X)))$;
        $UNIT = concat(UNIT, (alt(reuseContext(nosingle, X)))^*)$;
      } **else** { $UNIT = concat(UNIT, alt(mark(reuseContext(single, X))))$;
        $UNIT = concat(UNIT, (alt(mark(reuseContext(nosingle, X)))))^*)$; }
      if $(k == 2)$ **then** { **foreach** $t' \in create(r_{C.h})$ **do** {

$UNIT = concat(UNIT, RecursiveGen(t', S \cup \{t\}, true))$; } } } } } ;
foreach rule $r : L \to R$ in $UNIT$ **do** {
foreach condition $C = \nexists(X) \in \Theta$ **do** { **replace** r with $genNAC(r, X)$; } };
$UNIT = concat(UNIT, delLoop)$;
return $UNIT$

Theorem 1 *A call CreateGenUnit$(t, \emptyset)$: 1) terminates, and 2) produces a correct unit TU_g^t s.t. given a graph G typed on TG s.t. $G \models \Theta$, $\forall H$ s.t. $G \overset{TU_g^t\downarrow}{\Longrightarrow} H$, we have $H \models \Theta \cup \{\exists(G + \boxed{t})\}$.*

Proof. (**Sketch**) 1) The first nested loop performs a finite number of iterations on conditions, layers and rules. The recursion on *recursiveGen* terminates since the set S increases in size on each call. The final iteration to add NACs occurs on a finite number of conditions and rules.

2) If the first rule is applicable, then the application of $TU_G(t)$ terminates on each finite graph G s.t. $G \models \Theta$. Indeed, the NACs prevent repeated applications of a rule on identical matches, and even if new matches can be created, the layering prevents infinite repetition of the execution of a rule. Moreover, the application of *reuseContext* avoids arbitrary generation of new elements. If a graph H is obtained, then $H \models \exists(G + \boxed{t})$, as only increasing rules have been applied. Suppose now that $H \not\models \Theta$. Then either: 1) $H \not\models C_i$ for some C_i in some $Q_\nexists(t)$, but this is impossible as this is prevented by the use of *genNAC*; or 2) $H \not\models \exists\overline{Q}$, but this is impossible as $\overline{Q} \leq G \leq G + \boxed{t} \leq H$; or 3) $H \not\models C_i$ for some C_i in some $Q_\forall(t)$, but this is impossible as all the rules are derived from some $Q_\forall(t)$ and all matches for their premises have been considered. □

6 Application to Spider Diagrams

Contrasted to algorithmic definitions of inference figures for Spider Diagrams, the proposed approach allows the modeling both of syntactically correct Spider Diagram and of an operational system, admitting intermediary-type diagrams with some syntactic constraints relaxed.

We now apply the constructions in Section 5. Firstly, considering the addition of a `Curve`, we have conditions $Q_\exists(Curve) = \{C1\}$, $Q_\forall(Curve) = \{(C7), (C8, C9), (C10), (C11, C12)\}$, where we have abused notation for universal conditions to indicate their ordering according to $\leq_Q$; e.g. the premise of $C7$ is included in the premise of both $C8$ and $C9$. The layers associated to the type `Curve` are as follows. $LAYER_1(Curve)$ contains rules generated from $C7$, $C11$, $C12$ and from the first two graphs in the bottom box of $C10$ since these only add edges incident with nodes of type `Curve`. $LAYER_2(Curve)$ contains rules generated from $C8, C9$ which add `Zone` nodes, whilst $LAYER_3(Curve)$ consists of the rule generated from the last graph in the bottom box in $C10$ which adds an edge between nodes of type `Zone`.

Note that depending on which iteration of the rules derived from $C8$ or $C9$ is applied first, the other iteration will be performed vacuously. The same thing happens for $C11$ and $C12$.

Figure 9 shows a version of the rules derived from condition $C10$, with one choice of marking. Each possible conclusion of the rules from $C10$ give rise to a *NAC*, preventing re-application of the rule to the same match, and the set of three NACs (these define the n_j morphisms in the construction provided earlier) is presented together at the bottom left of the figure.

Using the same rule naming scheme as in Figure 9, and the initial rule $r_{curve} : \emptyset \to Curve$, the

algorithm produces a transformation unit of the form:

$$\texttt{TU(addCurve)} = \mathrm{r}^{\dagger}_{curve}\,;\ (\mathrm{r7.1}^{\dagger} \,|\, \mathrm{r7.2}^{\dagger})^{*}\,;\ (\mathrm{r10.1}^{\dagger} \,|\, \mathrm{r10.2}^{\dagger})^{*}\,;\ (r11^{\dagger})^{*}\,;\ (r12^{\dagger})^{*}\,;\ (\mathrm{r8}^{\dagger})\,;\ (\mathrm{r9}^{\dagger})^{*}\,;$$
$$(\mathrm{r4.1} \,|\, \mathrm{r4.2})^{*}\,;\ \mathrm{r6}^{*}\,;\ (\mathrm{r7.1} \,|\, \mathrm{r7.2})^{*}\,;\ (\mathrm{r10.1} \,|\, \mathrm{r10.2} \,|\, \mathrm{r10.3})^{*}\,;\ \mathrm{r11}^{*}\,;\ \mathrm{r12}^{*}\,;\ \mathrm{r10.3}^{*}$$

The iterations on rules from r4.i to r12 in the second row derive from the fact that some *Zone* is created in the previous rules, so that the second top-level loop must be started, reusing context to prevent the creation of new curves.

The analogous construction for the type *Zone* is based on the following specifications $Q_{\exists}(Zone) = \{C1\}$, $Q_{\forall}(Zone) = \{(C4),(C6,C7),(C8,C9),(C10),(C11,C12)\}$. $LAYER_1(Zone)$ contains the rules generated from $C7$, $C10$, $C11$, while $LAYER_2(Zone)$ contains the rules generated from $C8$ and $C9$. In this case, the unit will first define the relations of the new zone with the existing curves, according to the rule from $C4$, then create a required new curve (as the context does not provide one to satisfy $C6$). After the iteration of the rule for $C7$, the context again will not be sufficient for the application of the rules from $C8$ and $C9$. Finally, the rules from $C10$, $C11$ and $C12$ will adjust the relations with the newly created curves and among all zones.

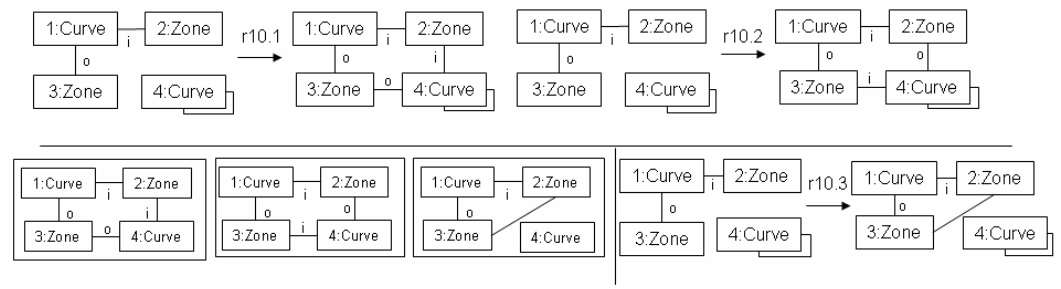

Figure 9: A marked version of the 3 rules derived from condition C10 and the non-marked NAC.

For a Spider, we have $Q_{\exists}(Spider) = \emptyset$, $Q_{\forall}(Spider) = \{(C2)\}$, generating a rule in layer 2. While the creation of a `Spider` requires the creation of a `Foot`, the `Zone` will be taken from the context, due to its presence in $\overline{Q}$, so that it has been incorporated by the application of *reuseContext*. Insertion of a `Foot` will instead require the creation of a new `Spider`, if none exists, or its reuse if one had already been created. However, such a creation will fail if the spider has already a foot in each existing zone.

In a similar way, a unit for the deletion of a curve would first remove the twin edges between zones attached to the marked curve, then all edges from all other curves to these zones, then all zones attached with an *inside* or *outside* edge to the curve to be removed, then all remaining connections from the marked `Curve` node to be deleted, and finally the marked node itself. Removal of a spider would be preceded by removal of all its feet and their attachments to zones. The construction of such units is beyond the scope of this paper.

7 Conclusions

We have provided a methodology for the automatic derivation of transformation units from a principal rule via an algorithm that iteratively adds restorative rules to a unit for increasing rules.

As a result, membership in the model language is ensured before and after the application of the unit, but not necessarily throughout the unit. The methodology exploits a rule layering approach, and rules are generated from graph conditions taking into account the rule application context.

The automatic production of the rules needed to reassemble a syntactically correct diagram simplifies the specification of diagrammatic inference rules and supports therefore the development and comparison of syntactic and semantic variations of the systems. Future work will define a similar algorithm for deleting rules, adding preparatory rules for performing a final deletion.

Of course, semantic considerations play a greater role than simple syntactic constraints. However, the constructed rules may provide a basis to be extended with additional context and consequences. For example, the specification of a transformation via pre- and post-conditions can be used to integrate syntactic rules with specific side effects. In this sense, this construction provides more flexibility to modelers, who can define the language through conditions, the main goal of a transformation and the desired side effects in an independent manner. This removes the need to consider complex interplays between rules and constraints, as in approaches which derive amalgamated rules which have to achieve a global effect with a single specification.

We notice that most transformations involve redirection of associations from one element to another, or changing the context for an element. The construction presented in the paper can be adapted to define *accumulators* and *distributors* of associations, which would collect all edges to be redirected, while deleting or constructing elements. Hence, such redirections might be taken as primitive constructs. The approach has been presented only for typed graphs. Extensions to graphs with inheritance and with attributes have to be explored, in particular for the case where identifiers are used to describe the associations of an element with others.

This would be useful also in other domains. For instance, model refactoring often involves the elimination of elements, or the creation of suitable contexts for their insertion. One example is the elimination of a composite state in a Statechart which requires the elimination of all of its internal states. Then, given a set of conditions stating that each state must be contained within a composite state, the construction in Section 5 could be applied to generate transformation units to be recursively invoked to visit the nesting tree. Another refactoring example is that of moving a method. This requires placing it in a different class and redirecting all its invocations, as well as the messages which may originate from its invocation, to its new location. Our construction can thus be used to manage the identification of the arcs related to such a method.

Acknowledgements: Partially funded by UK EPSRC grant EP/E011160: Visualisation with Euler Diagrams.

Bibliography

[BGL08] P. Bottoni, E. Guerra, J. de Lara. Enforced generative patterns for the specification of the syntax and semantics of visual languages. *JVLC* 19(4):429–455, 2008.

[BQV06] P. Bottoni, P. Quattrocchi, D. Ventriglia. Constraining Concrete Syntax via Metamodel Information. In *Proc. IEEE VL/HCC 2006*. Pp. 85–88. IEEE CS Press, 2006.

[EEHP06] H. Ehrig, K. Ehrig, A. Habel, K.-H. Pennemann. Theory of Constraints and Application Conditions: From Graphs to High-Level Structures. *Fundam. Inform.* 74(1):135–166, 2006.

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

[EKTW06] K. Ehrig, J. M. Küster, G. Taentzer, J. Winkelmann. Generating Instance Models from Meta Models. In *Proc. FMOODS 2006*. LNCS 4037, pp. 156–170. Springer, 2006.

[GMT99] M. Goedicke, T. Meyer, G. Taentzer. ViewPoint-oriented software development by distributed graph transformation: towards a basis for living with inconsistencies. In *Proc. IEEE Requirements Engineering, 1999*. Pp. 92–99. 1999.

[HHT96] A. Habel, R. Heckel, G. Taentzer. Graph Grammars with Negative Application Conditions. *Fundam. Inform.* 26(3/4):287–313, 1996.

[HHT02] J. H. Hausmann, R. Heckel, G. Taentzer. Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In *Proc. ICSE '02*. Pp. 105–115. ACM Press, 2002.

[HMT+01] J. Howse, F. Molina, J. Taylor, S. Kent, J. Gil. Spider Diagrams: A Diagrammatic Reasoning System. *JVLC* 12(3):299–324, 2001.

[HP09] A. Habel, K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Math. Struc. in Comp. Sc.* 19(2):245–296, 2009.

[KKS97] H.-J. Kreowski, S. Kuske, A. Schürr. Nested graph transformation units. *Int. J. on SEKE* 7(4):479–502, 1997.

[MSW00] M. Münch, A. Schürr, A. J. Winter. Integrity Constraints in the Multi-paradigm Language PROGRES. In *Selected Papers from TAGT'98*. LNCS 1764, pp. 338–351. Springer, 2000.

[OEP08] F. Orejas, H. Ehrig, U. Prange. A Logic of Graph Constraints. In *Proc. FASE 2008*. LNCS 4961, pp. 179–198. Springer, 2008.

[Pen09] K.-H. Pennemann. *Development of Correct Graph Transformation Systems*. Ph.d thesis, Carl von Ossietzky Universität - Oldenburg, 2009.

[Ren04] A. Rensink. Representing First-Order Logic Using Graphs. In *Proc. ICGT*. LNCS 3256, pp. 319–335. Springer, 2004.

[RK09] A. Rensink, J.-H. Kuperus. Repotting the Geraniums: On Nested Graph Transformation Rules. *ECEASST - Proc. GT-VMT 2009* 18, 2009.

[Rus97] F. Ruskey. A Survey of Venn Diagrams. *Electronic Journal of Combinatorics*, 1997. www.combinatorics.org/Surveys/ds5/VennEJC.html.