Ensuring Structural Constraints in Graph-Based Models with Type Inheritance

Gabriele Taentzer[1] and Arend Rensink[2]

[1] Computer Science Department
Technical University of Berlin
Berlin, Germany
[2] Computer Science Department
University of Twente
Enschede, The Netherlands

Abstract Graphs are a common means to represent structures in models and meta-models of software systems. In this context, the description of model domains by classifying the domain entities and their relations using *class diagrams* or *type graphs* has emerged as a very valuable principle. The constraints that can be imposed by pure typing are, however, relatively weak; it is therefore common practice to enrich type information with *structural properties* (such as local invariants or multiplicity conditions) or *inheritance*.
In this paper, we show how to formulate structural properties using *graph constraints* in type graphs with inheritance, and we show how to translate constrained type graphs with inheritance to equivalent constrained simple type graphs. From existing theory it then follows that graph constraints can be translated into preconditions for productions of a typed graph transformation system which ensures those graph constraints. This result can be regarded as a further important step of integrating graph transformation with object-orientation concepts.

1 Introduction

Graphs and graphical representations play a central role in modeling and meta-modeling of software systems. Graphs are used to describe essential structures of entities and their relations. Their representation ranges from simply formatted, graph-like notations such as class diagrams, Petri nets, automata, etc. to more elaborated diagram kinds such as message sequence charts and to more application-specific notations for modeling, e.g., for industrial production processes.

In graph-based modeling and meta-modeling, graphs are used to define the static structure, such as class and object structures, data base schemes, as well as visual symbols and interrelations, i.e., visual alphabets and sentences. Graph manipulations describe the dynamic changes of these structures. Classifying the possible entities and interrelations in static system structures or visual language constructs has emerged as a valuable principle for the description of model domains. In the object-oriented approach, *class diagrams* are the basic means to specify classification structures, for instance in UML (Unified Modeling Language) [12] for software systems and MOF (Meta

Object Facility) [12] for visual language specification. When applying graph transformation for modeling or meta-modeling, *type graphs* are used to classify graph nodes and edges.

One of the main principles to handle complex classification structures comes from the object-orientation paradigm: class inheritance enhances the typing principle by adding more abstract types on top of the ones concretely used in the (meta)models. Thus, inheritance allows much more compact representations by reducing redundancy. The principle of inheritance has been carried over and formalized for graph transformation in [4]; there we have shown that node inheritance in typed graph transformation leads to a denser form of a graph transformation system, since similar transformation rules can be abstracted to one.

The power of pure typing to describe and constrain the static structure is, however, relatively weak (and is not enhanced by inheritance). It is therefore common practice to enrich type information with *structural properties* which further constrain the correct instances. A typical class of such structural properties are *multiplicity conditions*, which restrict correctly typed structures to those where the numbers of entities and interrelations must be within given ranges. Further constraints can be *local invariants* which require, e.g., the existence or non-existence of certain substructures. In class diagrams, some of these constraint kinds are built-in, like multiplicities, while others have to be stated by separate constraints using, e.g., OCL [12]. On the other hand, typed graphs can be equipped with *graph constraints*, as proposed first in [10], which can be used to describe a variety of local invariants. Note, however, that graph constraints have so far been studied for *flat* graphs only (i.e., without node type inheritance).

The object-oriented and graph transformation approaches can be integrated by identifying classes with node types, and associations with edge types. In this way, class inheritance naturally corresponds to node type inheritance. At this stage, therefore, we have already the possibility to manipulate object structures by rule applications, which is the *constructive* element in system modeling and meta-modeling. *Declarative* elements come in through constraints, formulated on top of type graphs with inheritance. For instance, we show how multiplicities and *edge inheritance* can be expressed by graph constraints. To have a precise definition at hand, we give a translation of constrained type graphs with inheritance to constrained flat type graphs. From existing theory [8] it then follows that graph constraints can be translated into pre-conditions for the rules of a typed graph transformation system which ensure those graph constraints. Our result can be regarded as a further important step of integrating graph transformation with object-orientation concepts. Application areas for the resulting theory are for instance: *operational semantics* for object-oriented systems as in [6] (leading to a theory of behavioral verification) and *refactoring* as in [11] (leading to a formal underpinning). We use an example from the former area as a running example in the paper.

The paper is organized as follows: In the next section, we review the basis of integrating graph transformation with object-orientation concepts by first recalling type graphs with node type inheritance introduced in [4]. In Section 3, graph constraints over type graphs with inheritance are defined and a translation to constraints over simple type graphs is presented. Then in Section 4, multiplicities and edge inheritance are shown to be expressible by graph constraints. Section 5 presents the basic transformation con-

cepts for typed graphs using type inheritance, and describes how graph constraints can be ensured by typed graph transformation systems, reusing and extending the results in [8].

2 Type Graphs with Node Type Inheritance

The basic idea for specifying node type hierarchies is to introduce a special kind of (directed) edges, called *inheritance edges*, into type graphs. The source node of an inheritance edge is said to be a sub-type of the target node, which is called the former one's super-type. Moreover, nodes are marked either as *concrete* or *abstract*; we will see that only concrete type nodes can have direct instances. In host graphs only nodes of concrete types shall occur, while graphs in rules may contain nodes of both types.

Definition 1 (type graph with inheritance) *A type graph with inheritance is a triple* $TGI = (TG, I, A)$ *consisting of a type graph* $TG = (N, E, s, t)$ *(with a set* N *of nodes, a set* E *of edges, source and target functions* $s, t : E \to N$*), an acyclic inheritance relation* $I \subseteq N \times N$*, and a set* $A \subseteq N$*, called abstract nodes. For each* $x \in N$*, the* inheritance clan *is defined by* $clan_I(x) = \{y \in N \mid (y, x) \in I^*\}$*, where* I^* *is the transitive closure of* I*.*

Example 1. As sample type graph we use a graph description of a special kind of sets, namely ordered sets, which contain a number of objects which can be put into some order indicated by edges. We consider two possible specializations of ordered sets, namely StringSet and IntegerSet, which are intended to contain Strings and Integers, respectively. Considering the corresponding type graph TGI in Figure 1, we use edge types cnt and nxt to describe the containment and order relations on objects in ordered sets.

The type graph does not yet demand a strict separation of strings and integers in their kinds of sets. E.g. this type graph does not rule out that StringSets contain also Integers, and vice versa.

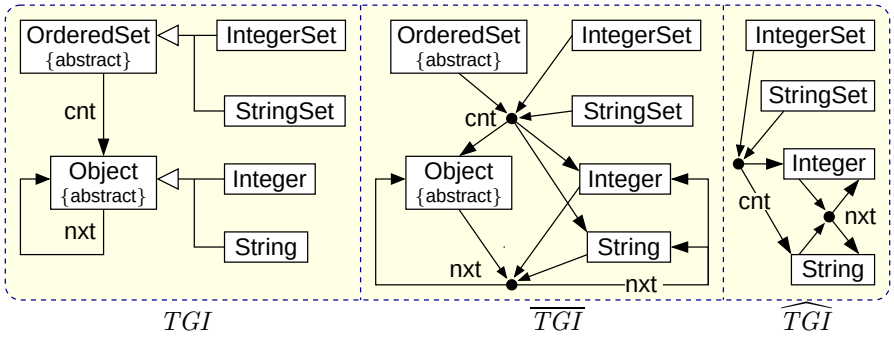

Figure1. A sample type graph with node type inheritance, and its abstract and concrete closure

To benefit from the existing theory of graph transformation [7], which does not recognize the notion of inheritance, we define the *flattening* or *closure* of type graphs with inheritance to ordinary ones.

Definition 2 (Closure of type graph with inheritance) *Let $TGI = (TG, I, A)$ be a type graph with inheritance, and let $TG = (N, E, src, tar)$. The* abstract closure *of TGI is the graph $\overline{TGI} = (N, \overline{E}, \overline{src}, \overline{tar})$ with*
- $\overline{E} = \{(n_1, e, n_2) \mid e \in E, n_1 \in clan_I(src(e)), n_2 \in clan_I(tar(e))\}$;
- $\overline{src}((n_1, e, n_2)) = n_1$;
- $\overline{tar}((n_1, e, n_2)) = n_2$.

The concrete closure *of TGI is the graph $\widehat{TGI} = \overline{TGI}|_{N-A}$.*[1]

Example 2. Fig. 1 also shows the abstract and concrete closure of the type graph with inheritance TGI. Please note that for better readability of the closures, the edge types are bundled using auxiliary nodes. Note that the inheritance edges are no longer present in the closure, and the abstract node types and adjacent edge types are absent from the concrete closure. Instead, for all combinations of corresponding sub-types a new edge type is inserted — including those which do not follow our intuition, like edge type nxt between String and Integer. We will use structural graph properties in addition to rule out those unwanted structures.

The distinction between the abstract and the concrete closure of a type graph is necessary, since we use instance graphs with respect to either one. For instance, we will define abstract graph transformation rules of which the left hand and right hand sides are typed over the abstract closure (see Section 5), whereas ordinary host graphs and concrete rules are typed over the concrete closure. Note that, due to the canonical inclusion $inc_{TG}\colon \widehat{TGI} \hookrightarrow \overline{TGI}$, all graphs typed over $\widehat{TGI}$ are also typed over $\overline{TGI}$.

Definition 3 (instance graph) *An abstract instance graph (G, tp_A) of a type graph with inheritance TGI is an instance graph of $\overline{TGI}$; i.e., $tp_A\colon G \to \overline{TGI}$. Analogously, a concrete instance graph (G, tp_C) of TGI is a graph typed over $\widehat{TGI}$.*

The construction of the closure in Def. 2 gives rise to a characterization of instance graphs directly on type graphs with inheritance. Namely, instance graphs can be typed over the type graph with inheritance by a pair of functions, one assigning a node type to each node and the other one assigning an edge type to each edge. This pair of functions does not constitute a graph morphism, but will be called *clan morphism*; it uniquely characterizes the type morphism into the flattened type graph.

Definition 4 (clan morphism) *Let $TGI = (TG, I, A)$ be a type graph with inheritance. A* clan-morphism *from G to TGI is a pair $ctp = (ctp_N\colon N_G \to N_{TG}, ctp_E\colon E_G \to E_{TG})$ such that for all $e \in E_G$ the following holds:*
- *$ctp_N \circ s_G(e) \in clan_I(s_{TG} \circ ctp_E(e))$ and*
- *$ctp_N \circ t_G(e) \in clan_I(t_{TG} \circ ctp_E(e))$.*

(G, ctp) is called a clan-typed graph. *ctp is called* concrete *if $ctp_N^{-1}(A) = \emptyset$.*

Example 3. Figure 2 shows a sample instance graph typed over TGI of Fig. 1. The edge typing is not shown explicitly, but follows uniquely from the node typing. The typing is done by a clan morphism which maps each node to its node type and each edge to an edge type between potentially more abstract node types holding the source and target types of the instance edge in their clans.

[1] Given a graph $G = (N, E, s, t)$ and a set $X \subseteq N$, we denote by $G|_X$ the sub-graph $(X, E_X = \{e \in E \mid s(e), t(e) \in X\}, s|_{E_X}, t|_{E_X})$.

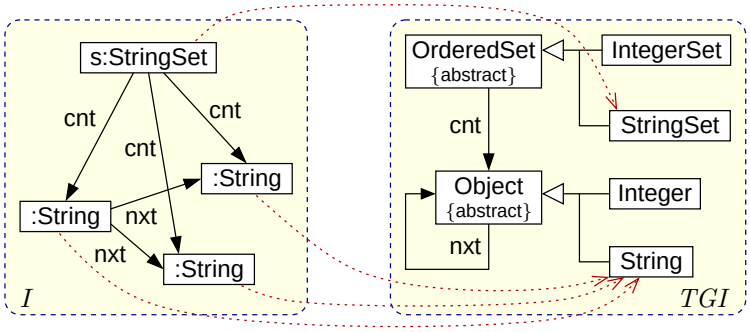

Figure2. Sample clan-typed graph

Proposition 1 (universal clan morphism). *Let $TGI = (TG, I, A)$ be a type graph with inheritance. There is a universal clan morphism $u_{TG}\colon \overline{TGI} \to TG$ such that for each clan morphism $ctp\colon G \to TG$ there is a unique graph morphism $tp\colon G \to \overline{TGI}$ with $u_{TG} \circ tp = ctp$. Moreover, there is a bijective correspondence between clan morphisms $ctp\colon G \to TG$ and graph morphisms $tp\colon G \to \overline{TGI}$ such that $u_{TG} \circ tp = ctp$.*

For the proof see [3]. By abuse of notation we often use G to stand for the clan-typed graph (G, ctp_G).

To formalize the relationship between abstract and concrete rules (see Section 5), we introduce the notion of *type refinement*. Thus imposes an order over possible typing morphisms for a given instance graph. A typing morphism is said to be *finer* than another one if it assigns more concrete node types to the nodes of the instance graph.

Definition 5 (type refinement and typed graph morphism) *Let $TGI = (TG, I, A)$ be a type graph with inheritance, and let $ctp, ctp'\colon G \to TG$ be clan typings. ctp is a* refinement *of ctp', denoted $ctp \leq ctp'$, if*

- *$ctp_N(n) \in clan_I(ctp'_N(n))$ for all $n \in N_G$, and*
- *$ctp_E = ctp'_E$.*

Given two clan-typed graphs $(G, ctp_G\colon G \to TG)$ and $(H, ctp_H\colon H \to TG)$, a graph morphism $g\colon G \to H$ is called type-refining *if $g \circ ctp_H \leq ctp_G$, and* type-preserving *if $g \circ ctp_H = ctp_G$.*

We also write $(G, ctp_G) \leq (H, ctp_H)$ if $G = H$ and $ctp_G \leq ctp_H$. We write $g\colon G \to_{\mathsf{c}} H$ to denote that G and H are both concrete and g is an injective type-preserving morphism, and $g\colon G \to_{\mathsf{a}} H$ to denote that g is an injective type-refining morphism.

The following proposition states some facts regarding type-refining and type-preserving morphisms.

Proposition 2. *Let G, H be clan-typed graphs, and let $g\colon G \to H$ be type refining.*

1. *There is a unique clan-typed graph $K \leq G$ such that $g\colon K \to H$ is type-preserving;*
2. *For any clan-typed graph $K \geq G$, $g\colon K \to H$ is type-refining.*
3. *For any clan-typed graph $K \leq H$, $g\colon G \to K$ is type-refining.*

Proof. 1. Let us write out G in full as (G, ctp_G). Let $(K, ctp_K) = (G, ctp_H \circ g)$; then clearly $(K, ctp_K) \leq (G, ctp_G)$ and $g\colon K \to H$ is a type-preserving graph morphism. On the other hand, if $(L, ctp_L) \leq (G, ctp_G)$ and $g\colon L \to H$ is a type-preserving morphism, then $L = G = K$ and $ctp_L = ctp_H \circ g = ctp_K$, hence $(L, ctp_L) = (K, ctp_K)$.
2. g is clearly a graph morphism, and $ctp_K \geq ctp_G \geq ctp_H \circ g$; since $\geq$ is transitive, $g\colon K \to H$ is type-refining.
3. g is clearly a graph morphism, and $ctp_G \geq ctp_H \circ g \geq ctp_K \circ g$ due to $ctp_H \geq ctp_K$ and the fact that composition of morphisms is monotonic in $\geq$; hence $g\colon K \to H$ is type-refining.

3 Structural Properties over Type Graphs with Inheritance

The following definition extends the concept of graph constraints, originally introduced in [10] (where they are called consistency constraints). In fact there are two points of change:

- Rather than using ordinary typed graphs we are defining constraints over concrete clan-typed graphs. However, this is not a real extension since (due to Prop. 1), there is a one-to-one correspondence between concrete clan morphisms and type morphisms to the concrete closure of the type graph.
- Rather than having constraints consisting of a single premise and conclusion, as in [10,8], we generalize to multiple conclusions. This is a real extension, as it properly enlarges the set of properties expressible through graph constraints.

Whenever we mention "clan-typed graphs" in the following, we mean graphs with a clan morphism to some implicit, globally given type graph with inheritance TGI.

Definition 6 (graph atoms and formulae) *Let L, G be clan-typed graphs, such that G is concrete.*

- *A concrete [abstract] graph atom A over L is a tuple $(n\colon L \to_{\mathsf{c}} P, Con)$ [$(n\colon L \to_{\mathsf{a}} P, Con)$], where n is an injective type-preserving [type-refining] morphism, and Con is a set of injective type-preserving [type-refining] morphisms starting in P. If $L = \emptyset$ we also write (P, Con) for A.*
- *A is said to be satisfied by an injective type-preserving [type-refining] morphism $m\colon L \to_{\mathsf{c}} G$ [$m\colon L \to_{\mathsf{a}} G$], denoted $m \models^{\mathsf{c}} A$ [$m \models^{\mathsf{a}} A$], if for all injective type-preserving [type-refining] morphisms $p\colon P \to_{\mathsf{c}} G$ [$p\colon P \to_{\mathsf{a}} G$] such that $m = p \circ n$, there is a $(q\colon P \to C) \in Con$ and an injective type-preserving [type-refining] morphism $c\colon C \to_{\mathsf{c}} G$ [$c\colon C \to_{\mathsf{a}} G$] such that $p = c \circ q$. If $L = \emptyset$ (i.e., the empty graph) then we also write $G \models^{\mathsf{c}} A$ [$G \models^{\mathsf{a}} A$].*
- *A concrete [abstract] graph formula F over L is a boolean formula over concrete [abstract] graph atoms over L. The satisfaction relation $\models^{\mathsf{c}}$ [$\models^{\mathsf{a}}$] is extended to graph formulae by defining the semantics of the boolean operators in the usual way. We sometimes call F a* constraint *if $L = \emptyset$, and an* application condition *otherwise.*

We can now define the flattening of an abstract atom and an abstract formula.

Definition 7 (flattening) *Let K, L be clan-typed graphs, such that $K \leq L$ and K is concrete.*

- *For any abstract graph atom $A = \langle n\colon L \to_{\mathsf{a}} Q, Con\rangle$, we define the K-*flattening *of A as follows:*

$$\begin{aligned} \mathit{flat}_K(A) &= \bigwedge \{(n\colon K \to_{\mathsf{c}} P, \mathit{flat}_P(Con)) \mid P \leq Q\} \\ \mathit{flat}_P(Con) &= \{q\colon P \to_{\mathsf{c}} C \mid (q\colon Q \to_{\mathsf{a}} D) \in Con, C \leq D\} \ . \end{aligned}$$

- *For any abstract graph formula F over L, we define the K-flattening $\mathit{flat}_K(F)$ by replacing each abstract graph atom A occurring in F by the corresponding K-flattening $\mathit{flat}_K(A)$.*

The following theorem is the main contribution of this paper. It states that satisfaction of an abstract atom or formula over an abstract clan-typed graph L by a type-refining morphism $m\colon L \to_{\mathsf{a}} G$ is equivalent to satisfaction of the flattening of that atom or formula with respect to the concrete clan-typed graph $K \leq L$ for which $m\colon K \to_{\mathsf{c}} G$ is type-preserving (which uniquely exists due to Prop. 2.1). This allows us to re-use existing theory on concrete graph formulae.

Theorem 1 (flattening of abstract graph formulae). *Let K, L, G be clan-typed graphs such that $K \leq L$, and let $m\colon K \to_{\mathsf{c}} G$. For any abstract graph atom A and graph formula F over L the following holds:*

$$\begin{aligned} (m\colon L \to_{\mathsf{a}} G) &\models^{\mathsf{a}} A \text{ iff } (m\colon K \to_{\mathsf{c}} G) \models^{\mathsf{c}} \mathit{flat}_K(A) \\ (m\colon L \to_{\mathsf{a}} G) &\models^{\mathsf{a}} F \text{ iff } (m\colon K \to_{\mathsf{c}} G) \models^{\mathsf{c}} \mathit{flat}_K(F) \ . \end{aligned}$$

Proof. We prove the case for atoms; the case for formulae follows by a straightforward induction over the structure of boolean formulae. Let $A = (n\colon L \to_{\mathsf{a}} Q, Con)$.

If. Assume $(m\colon K \to_{\mathsf{c}} G) \models^{\mathsf{c}} \mathit{flat}_K(A)$, and let $p\colon Q \to_{\mathsf{a}} G$ be arbitrary such that $m = p \circ n$; that is, $ctp_G \circ p \leq ctp_Q$. Since G is concrete, it follows that $ctp' = ctp_G \circ p$ is a concrete clan morphism for P. We denote $P = (Q, ctp')$; it follows that $P \leq Q$ and $p\colon P \to_{\mathsf{c}} G$. Since $m\colon K \to_{\mathsf{c}} G$ is also type-preserving, we may conclude that so is $n\colon K \to_{\mathsf{c}} P$.
Due to $(m\colon K \to_{\mathsf{c}} G) \models^{\mathsf{c}} \mathit{flat}_K(A) = (n\colon K \to_{\mathsf{c}} P, \mathit{flat}_P(Con))$, it follows that there is a $q\colon P \to_{\mathsf{c}} C \in \mathit{flat}_P(Con)$ and a $c\colon C \to_{\mathsf{c}} G$ such that $p = c \circ q$. By construction of $\mathit{flat}_P(Con)$, there is a $(q\colon Q \to_{\mathsf{a}} D) \in Con$ such that $C \leq D$; hence (due to Prop. 2.2), $c\colon D \to_{\mathsf{a}} G$ is type-refining. We may conclude $m \models^{\mathsf{a}} A$.

Only if. Assume $(m\colon L \to_{\mathsf{a}} G) \models^{\mathsf{a}} A$. Let $P \leq Q$ and $n\colon K \to_{\mathsf{c}} P$ be arbitrary. We prove $(m\colon K \to_{\mathsf{c}} G) \models^{\mathsf{c}} (n\colon K \to_{\mathsf{c}} P, \mathit{flat}_P(Con))$; this then implies $m \models^{\mathsf{c}} \mathit{flat}_K(A)$.
Let $p\colon P \to_{\mathsf{c}} G$ be arbitrary such that $m = p \circ n$; that is, $ctp_G \circ p = ctp_P$. Due to $(m\colon L \to_{\mathsf{a}} G) \models^{\mathsf{a}} A$, there is a $(q\colon Q \to_{\mathsf{a}} D) \in Con$ and $c\colon D \to_{\mathsf{a}} G$ such that $p = c \circ q$. Due to Prop. 2.1 there is a (unique) concrete $C \leq D$ such that $c\colon C \to G$ is type-preserving. Hence $ctp_C = ctp_G \circ c$, implying $ctp_C \circ q = ctp_G \circ p = ctp_P$. It follows that $q\colon P \to_{\mathsf{c}} C$, and hence $(q\colon P \to_{\mathsf{c}} C) \in \mathit{flat}_P(Con)$. This establishes $(m\colon K \to_{\mathsf{c}} G) \models^{\mathsf{c}} \mathit{flat}_K(A)$.

4 Multiplicities and Edge Inheritance as Graph Formulae

In this section we show that two existing classes of constraints on type graphs with inheritance can be translated to abstract graph formulae. This serves to give some intuition about graph formulae, and to demonstrate that they are expressive enough to cover practically useful examples.

Multiplicities By enriching a type graph with multiplicities we can restrict the class of instance graphs to those which are not only correctly typed but also satisfy additional constraints concerning the number of nodes and edges for each type. These constraints are expressed using so-called *multiplicities*.

Definition 8 (multiplicities) *A multiplicity is a pair $[i, j] \in \mathcal{N} \times (\mathcal{N} \cup \{*\})$ with $i \leq j$ or $j = *$. The set of multiplicities is denoted $Mult$. The special value $*$ indicates that the maximum number of nodes or edges is not constrained. For an arbitrary finite set X and $[i, j] \in Mult$, we write $|X| \in [i, j]$ if $i \leq |X|$ and either $j = *$ or $|X| \leq j$.*

As usual, we use multiplicities to decorate the nodes and edges of type graphs. For the nodes, the multiplicity indicates the total number of instances; for the edges, we use multiplicities expressing the number of incoming, respectively outgoing edges *for each target, respectively source instance*.

Definition 9 (Type graph with multiplicities) *A type graph with multiplicities is a tuple $TGM = (TGI, m_N, m_{src}, m_{tar})$ consisting of a type graph with inheritance TGI and additional functions $m_N : N_{TGI} \to Mult$, called* node multiplicity function*, and $m_{src}, m_{tar} : E_{TGI} \to Mult$, called* edge multiplicity functions*.*

The satisfaction of multiplicity constraints is expressed by counting inverse images with respect to the clan typing.

Definition 10 (Semantics of type graphs with multiplicities) *A clan-typed graph G over $TGI = (TG, I, A)$ is said to* satisfy *a type graph with multiplicities $(TGI, m_N, m_{src}, m_{tar})$ if the following conditions hold:*

- *for all $n \in N_{TG}$, $|ctp_G^{-1}(clan_I(n))| \in m_N(n)$;*
- *for all $e \in E_{TG}$ and $p \in ctp_G^{-1}(clan_I(src(e)))$, $|ctp_G^{-1}(e) \cap src_G^{-1}(p)| \in m_{tar}(e)$;*
- *for all $e \in E_{TG}$ and $p \in ctp_G^{-1}(clan_I(tar(e)))$, $|ctp_G^{-1}(e) \cap tar_G^{-1}(p)| \in m_{src}(e)$.*

We now show how a type graph with multiplicities TGM can be translated to an abstract graph formula that is satisfied by precisely those clan-typed graphs that also satisfy TGM. In order to do that, we introduce two special types of graphs: for all $i \in \mathcal{N}$,

- For all $n \in N$, G_i^n is the graph consisting of i distinct n-typed nodes.
- For all $e \in E$, $\mathcal{G}_i^{e,src}$ is the *set* of graphs with i distinct e-typed edges and all source nodes glued together; dually, $\mathcal{G}_i^{e,tar}$ is the set of graphs with i distinct e-typed edges and all target nodes glued together.

Definition 11 (Multiplicities as abstract graph formulae) *Given a type graph with multiplicities $TGM = (TGI, m_N, m_{src}, m_{tar})$, we define*

$$F_{TGM} = \bigwedge_{n \in N_{TGI}} F_n \wedge \bigwedge_{e \in E_{TGI}} (F_e^{src} \wedge F_e^{tar})$$

where F_n, F_e^{src} and F_e^{tar} are abstract graph formulae defined as follows:

– *F_n regulates the node multiplicity of n. Let $m_N(n) = [i,j]$; then $F_n = A_{n\geq i} \wedge A_{n\leq j}$ if $j \neq *$ and $F_n = A_{n\geq i}$ otherwise, where*

$$A_{n\geq i} = (\emptyset, \{\emptyset \to G_i^n\})$$
$$A_{n\leq j} = (G_{j+1}^n, \emptyset) \ .$$

– *F_e^{src} regulates the edge source multiplicity of e. Let $m_{src}(e) = [i,j]$; then $F_e^{src} = A_{e\geq i}^{src} \wedge F_{e\leq j}^{src}$ if $j \neq *$ and $F_e^{src} = A_{e\geq i}^{src}$ otherwise, where*

$$A_{e\geq i}^{src} = (G_1^{tar(e)}, \{q^{tar} : G_1^{tar(e)} \to H \mid H \in \mathcal{G}_i^{e,tar}\})$$
$$F_{e\leq j}^{src} = \bigwedge \{(H, \emptyset) \mid H \in \mathcal{G}_{j+1}^{e,tar}\}$$

with q^{tar} mapping the sole node of $G_1^{tar(e)}$ to the unique target node of H.
– *F_e^{tar} regulates the edge target multiplicity of e, and is the exact dual of F_e^{src} (obtained by switching src and tar everywhere in the above definition).*

Theorem 2 (semantics of multiplicities). *For all type graphs with multiplicity TGM and all graphs G clan-typed over TGI, G satisfies TGM (in the sense of Def. 10) if and only if $G \models^{\mathsf{a}} F_{TGM}$.*

Proof. The essence of the construction of F_{TGM} is that for each condition arising out of the satisfaction of the multiplicities, there is a graph (sub)-formula expressing exactly that condition. We show this for the first two conditions in Def. 10; the third is analogous to the second.

– For all $n \in N_{TG}$, $V = ctp_G^{-1}(clan_I(n))$ is the set of all (distinct) nodes typed by a node in the clan of n. It follows that there is an injective type-refining morphism $a\colon G_i^n \to G$ to that set for all $i \leq |V|$ but not for $i > |V|$; hence $G \models^{\mathsf{a}} A_{n\geq i}$ for all $i \leq |V|$ and $G \not\models^{\mathsf{a}} A_{n\leq j}$ for all $j > |V|$. This implies that $G \models^{\mathsf{a}} A_{n\geq i} \wedge A_{n\leq j}$ if and only if $|V| \in [i,j]$ and $G \models^{\mathsf{a}} A_{n\geq i}$ if and only if $|V| \in [i,*]$.
– For all $e \in E_{TG}$ and $p \in ctp_G^{-1}(clan_I(src(e)))$, $E_p = ctp_G^{-1}(e) \cap src_G^{-1}(p)$ is the set of all e-typed edges in G starting at p. It follows that for all $i \leq |E_p|$ there is an injective type-refining morphism $a\colon H \to G$ for *some* $H \in \mathcal{G}_i^{e,src}$ such that a maps the unique source node of H to p; but no such morphism exists if $i > |E_p|$. This implies that $A_{e\geq i}^{src} \wedge F_{e\leq j}^{src} \models^{\mathsf{a}} G$ if and only if $|E_p| \in [i,j]$ and $A_{e\geq i}^{src} \models^{\mathsf{a}} G$ if and only if $|E_p| \in [i,*]$ □

Example 4 (multiplicity constraints). In Figure 3 (left hand side), the type graph TGI of Fig. 1 has been extended with multiplicities at edge types. For the notation of multiplicities we follow UML. Each object has always to belong to precisely one ordered set. This statement contains two constraints: a lower and an upper bound, which in this case are both equal to 1. Vice versa, ordered sets are allowed to contain arbitrarily many objects, which is indicated by an asterisk. The nxt relation on objects is constrained to a partial order where at most one object is nxt, but each object may have arbitrarily many predecessors. This results in the five graph constraints depicted in Figure 4, where the premises are depicted on the left and the conclusions on the right of the block arrow, and

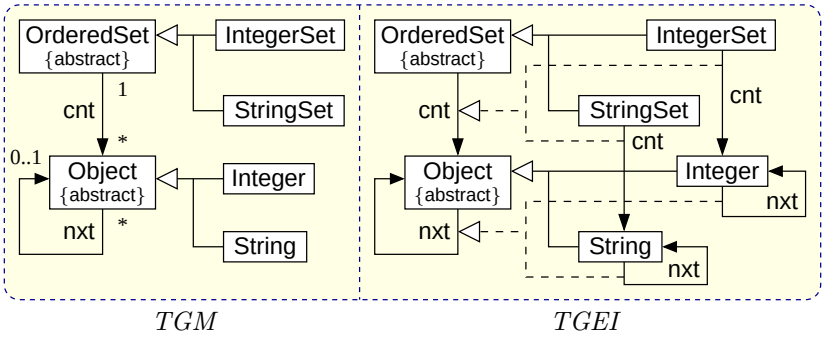

Figure3. Type graph with multiplicities, respectively edge inheritance

graph atoms of the form $(P, \{P \to C_i\})$ are depicted more compactly as $(P \to \{C_i\})$. The first constraint states that every object is contained in a set (which is a *positive* constraint), the next two that an object is not allowed to have two outgoing containment edges, neither to different nor to the same OrderedSet node (which are *negative* constraints), and the last two constraints (also negative) express that an object does not have two successor objects.

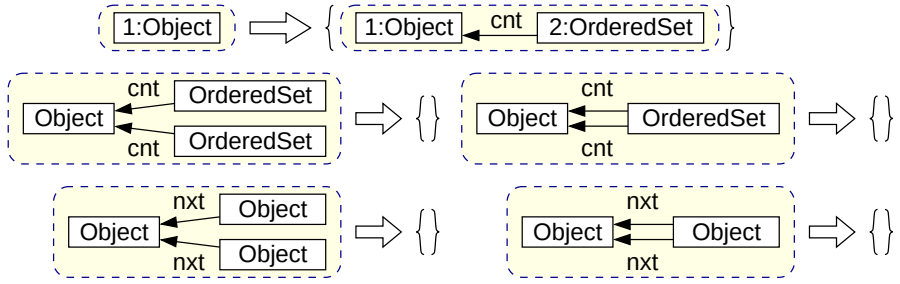

Figure4. Multiplicity constraints as abstract graph atoms

The next step is to flatten these graph constraints; i.e., we formulate graph constraints w.r.t. the concrete closure $\widehat{TGI}$ also given in Fig. 1. Some representatives of the flattened constraints are shown in Fig. 5. The first of these is the complete flattening of the first constraint in Fig. 4; the second and third show two of the four atomic constraints that constitute the flattening of the second constraint in Fig. 4.

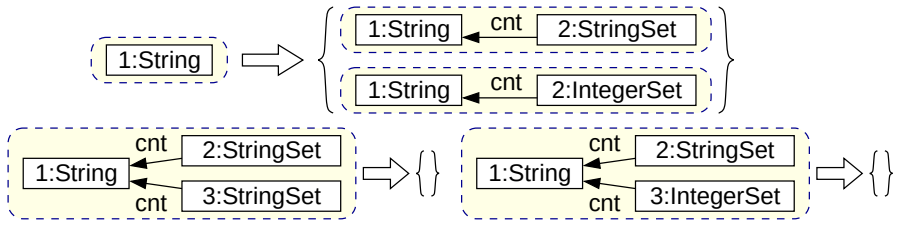

Figure5. Flattened multiplicity constraints

Edge inheritance As we have seen, node inheritance is used to formulate a compact type graph in the sense that edge types between super types stand for all combinations of edge types between their sub-types (including themselves). This might lead to a type graph with too loose type information concerning edges. In the following, we introduce *edge type inheritance*, which aims at restricting the combinations of sub-types allowed.

Definition 12 (type graph with edge inheritance) *A type graph with edge inheritance is a tuple (TG, I, A) where $I \subseteq (N \times N) \cup (E \times E)$ is an acyclic relation such that $TGI = (TG, I|_N, A)$ is a type graph with (node) inheritance, and moreover, $(e, f) \in I \cap (E \times E)$ implies $src(e) \in clan_I(src(f))$ and $tar(e) \in clan_I(tar(f))$.*

The idea is that if a type edge e inherits from another type edge f, then f can occur as an edge type only for concrete graph edges whose source and target node types are not in the clan of the source type, resp. target of e. The semantics of edge inheritance can either be expressed by redefining the closure, or directly as a constraint on the clan morphism. In other words, if the source or target node of an edge would allow e as an edge type, then no proper super-type of e may be used.

Definition 13 (semantics of type graphs with edge inheritance) *A clan-typed graph G over TGI is said to* satisfy *a type graph with edge inheritance (TG, I, A) for which $TGI = (TG, I|_N, A)$ if for all $x \in E_G$ and $(e, ctp_G(x)) \in I$, $ctp_G(src_G(x)) \notin clan_I(src_{TG}(e))$ and $ctp_G(tar_G(x)) \notin clan_I(tar_{TG}(e))$.*

We now construct an abstract graph formula which expresses the same constraint.

Definition 14 (edge inheritance as an abstract formula) *Given a type graph with edge inheritance $TGEI = (TG, I, A)$, we define $F_{TGEI} = \bigwedge_{(e,f)\in I} A^{src}_{e,f} \wedge A^{tar}_{e,f}$ where*

$$A^{src}_{e,f} = (G^{src(e),f,tar(f)}, \{q_{e,f} : G^{src(e),f,tar(f)} \to G^{src(e),e,tar(e)}\})$$
$$A^{tar}_{e,f} = (G^{src(f),f,tar(e)}, \{q_{e,f} : G^{src(f),f,tar(e)} \to G^{src(e),e,tar(e)}\}) \ .$$

with G^{n_1,e,n_2} for $n_1 \in clan_I(src(e))$ and $n_2 \in clan_I(tar(e))$ being the graph consisting of two nodes typed over n_1 and n_2, and one edge typed over e. $q_{e,f}$ is the unique type-refining morphism between the source and target graph.

Theorem 3 (semantics of edge inheritance). *For all type graphs with edge inheritance $TGEI = (TG, I, A)$ and all graphs G clan-typed over $(TG, I|_N, A)$, G satisfies $TGEI$ (in the sense of Def. 13) if and only if $G \models^{\mathsf{a}} F_{TGEI}$.*

Proof. Let $(e, f) \in I \cap (E \times E)$. We show that $A^{src}_{e,f}$ precisely captures the condition on the source typing G-edges mapped to f; the proof for the target typing is analogous. Note that, below, we extend the notion of inheritance clan to E by defining $clan_I(x) = \{y \in E \mid (y, x) \in I^*\}$ for all $x \in E$.

If. Assume $G \models^{\mathsf{a}} A^{src}_{e,f}$ and let $x \in E_G$ such that $f = ctp_G(x)$. If $ctp_G(src_G(x)) \in clan_I(src_{TG}(e))$ then there is an injective type-refining morphism mapping the unique edge of $G^{src(e),f,tar(f)}$ to x, but which cannot be refined further to $G^{src(e),e,tar(e)}$ because (due to the acyclicity of I) $ctp_G(x) \notin clan_I(e)$. This contradicts $G \models^{\mathsf{a}} A^{src}_{e,f}$.

Only if. Assume G satisfies $TGEI$, and let $a\colon G^{src(e),f,tar(f)} \to G$ be an injective type-refining morphism. It follows that $ctp_G(x) \in clan_I(f)$ for the sole edge $x \in E_G$ in the image of a; however, $ctp_G(x) = f$ contradicts the assumption that G satisfies $TGEI$. We conclude $ctp_G(x) \in clan_I(e)$, and hence $a\colon G^{src(e),e,tar(e)} \to G$ is also a type-refining morphism. This implies $G \models^{\mathsf{a}} A^{src}_{e,f}$. □

Example 5 (edge inheritance constraints). In Figure 3 (right hand side) we extended the type graph of Fig. 1 with edge type inheritance, depicted by (dashed) inheritance arrows between edges. Hence this type graph expresses (among other things) that an instance may not contain a nxt-edge from a String-typed node to anything but another String-typed node — in particular not to an Integer-typed node — or to a node typed by a subtype of String(of which there are none in this example).

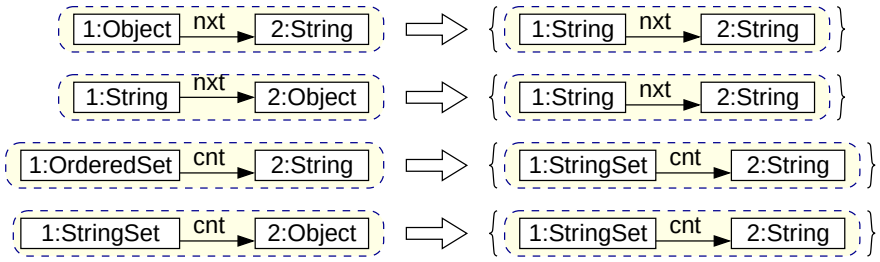

Figure6. Edge inheritance as graph constraints

Similarly to the example above, we flatten these graph constraints, i.e., we formulate graph constraints w.r.t. the concrete closure $\widehat{TGI}$ given in Fig. 1. The constraints shown in Fig. 7 are the complete flattening of the first constraint in Fig. 6. Note that the first flattened constraint is always true, and the second describes a handle not allowed by the edge inheritances.

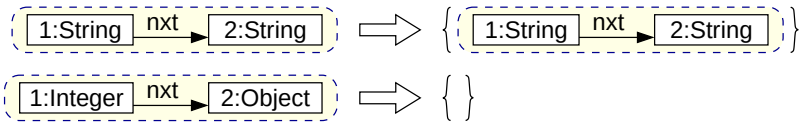

Figure7. Flattened edge inheritance constraints

5 Ensuring Abstract Graph Formulae

Having defined the concept of abstract graph formulae and shown their utility in formalizing node multiplicities and edge inheritance, we now turn to the issue of *ensuring* graph constraints (not arbitrary formulae) in a given graph transformation system. A graph transformation system is said to ensure a graph constraint if all the graphs that can be derived satisfy the constraint; in other words, if the constraint is an invariant on the derivable graphs. The method for enforcing a constraint is by including appropriate *preconditions* (which are themselves graph formulae) in the rules, using a technique worked out recently for sub-classes of concrete constraints in [8].

We first define abstract and concrete rules with application conditions, and their matching. The following definition extends that in [4].

Definition 15 (abstract and concrete rules) *An* abstract *rule typed over a type graph $TGI = (TG, I, A)$ with inheritance is given by $p = (L \xleftarrow{l} K \xrightarrow{r} R, F_L, F_R)$, where L, K, R are abstract clan-typed graphs, l and r are type-preserving graph morphisms, F_L and F_R are abstract graph formulae, and $ctp_R^{-1}(A) \subseteq r(N_K)$.*

p is called concrete *if L, K, R are concrete clan-typed graphs and F_L, F_R are concrete graph formulae.*

Concrete rule p' refines *abstract rule p, if $L' \leq L$, $K' \leq K$, $R' \leq R$ and $ctp'_R|_{N'_R} = ctp_R|_{N'_R}$, and moreover, $F'_L = flat_{L'}(F_L)$ and $F'_R = flat_{R'}(F_R)$. The set of all concrete refinements of an abstract rule p is denoted by $\widehat{p}$.*

Definition 16 (matching and application of concrete rules) *Let $p = (L \xleftarrow{l} K \xrightarrow{r} R, F_L, F_R)$ be a concrete rule, G a concrete clan-typed graph, and $m\colon L \to G$ a type-preserving graph morphism. m is a* match *of p in G if*

- *m is a match of the untyped rule $L \xleftarrow{l} K \xrightarrow{r} R$ in the untyped graph G,*
- *$m \models^{\mathsf{c}} F_L$.*

A direct derivation step is denoted by $G \overset{p,m,m^*}{\Longrightarrow} H$, where H is a concrete clan-typed graph and there is a span of type-preserving morphisms $G \longleftarrow D \longrightarrow H$ riving rise to a derivation in the classical theory of (untyped) graph transformations [7], with $m^*\colon R \to H$ as the co-match of p in H. This derivation step is only performed if $m^* \models^{\mathsf{c}} F_R$.

Definition 17 (matching and application of abstract rules) *Let $p = (L \xleftarrow{l} K \xrightarrow{r} R, F_L, F_R)$ be an abstract rule typed over TGI, G a concrete clan-typed graph, and $m\colon L \to G$ a type-refining graph morphism. Then m is a match of p in G if*

- *m is a match of the untyped rule $L \xleftarrow{l} K \xrightarrow{r} R$ in the untyped graph G;*
- *$t_K(x_1) = t_K(x_2)$ for $t_K = ctp_G \circ m \circ l$ and $x_1, x_2 \in N_K$ with $r(x_1) = r(x_2)$;*
- *$m \models^{\mathsf{a}} F_L$.*

Given a match m, the abstract rule can be applied to G yielding an abstract direct derivation $G \overset{p,m}{\Longrightarrow} H$, where H is a concrete clan-typed graph and there is a span of type-preserving morphisms $G \longleftarrow D \longrightarrow H$ riving rise to a derivation in the sense of [4], with $m^*\colon R \to H$ as the co-match of p in H. This derivation step is only performed if $m^* \models^{\mathsf{a}} F_R$.

The following is the main theorem of [3], extended to the more general application conditions used in the paper and proved using Theorem 1.

Theorem 4 (equivalence of abstract and concrete derivations). *Given an abstract rule $p_{\mathsf{a}} = (L \longleftarrow K \longrightarrow R, F_L, F_R)$, concrete clan-typed graph G, H and a structural match morphism $m\colon L \to G$ (i.e. a match with respect to the untyped rule $L \longleftarrow K \longrightarrow R$), the following statements are equivalent:*

1. *m is a match of p_{a} in G, yielding an abstract direct derivation: $G \overset{p_{\mathsf{a}},m}{\Longrightarrow} H$.*

2. *m is a match of the concrete rule $p_{\mathsf{c}} = L_{\mathsf{c}} \longleftarrow K_{\mathsf{c}} \longrightarrow R_{\mathsf{c}}, F^{\mathsf{c}}_L, F^{\mathsf{c}}_R)$ in G with $p_{\mathsf{c}} \in \widehat{p_{\mathsf{a}}}$ and $m\colon L_{\mathsf{c}} \to_{\mathsf{c}} G$ type-preserving, yielding a concrete direct derivation: $G \overset{p_{\mathsf{c}},m}{\Longrightarrow} H$.*

In the following, we want to use the translation of graph constraints to application conditions of graph rules as described in [8]. Therefore, we have to restrict the class of graph formulae we use to the ones defined in [8]. If we restrict our concrete graph constraints $GC = (P, Con)$ to those with $|Con| \leq 1$, they become equivalent to the positive and negative graph constraints of [8]: the case of $|Con| = 1$ corresponds to positive graph constraints, while the case of $|Con| = 0$ correspond to negative graph constraints.[2] Another difference is that, in [8], the morphisms in Con are allowed to be arbitrary, but that does not add expressiveness (although it does add compactness) to those we have defined here, which have injective morphisms only. The following is the relevant result from [8].

Theorem 5 (from concrete constraints to left application conditions). *Given a concrete constraint GC and a concrete rule $p = \langle L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R\rangle$, there is a left application condition acc_L such that for all direct derivations $G \overset{p,m}{\Longrightarrow} H$ we have: $m \models^{\mathsf{c}} \mathrm{acc}_L \Leftrightarrow H \models^{\mathsf{c}} GC$.*

By combining this with Theorems 1 and 4, we can prove the following:

Theorem 6 (from abstract constraints to left application conditions). *Given an abstract constraint GC_{a} and an abstract rule p_{a} with left hand side L_{a}, there is a set S of concrete application conditions such that for all direct derivations $G \overset{p_{\mathsf{a}},m}{\Longrightarrow} H$ we have: $(\exists F \in S : m \models^{\mathsf{c}} F) \Leftrightarrow H \models^{\mathsf{a}} GC_{\mathsf{a}}$.*

Proof. We fix $GC = \mathit{flat}_{\emptyset}(GC_{\mathsf{a}})$ for the duration of this proof. For an arbitrary concrete rule $p = \langle L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R\rangle \in \widehat{p_{\mathsf{a}}}$, let us write acc_p for the left application condition (which is guaranteed to exist by Theorem 5) such that for all direct derivations $G \overset{p_{\mathsf{c}},m}{\Longrightarrow} H$ we have: $m \models^{\mathsf{c}} \mathrm{acc}_p \Leftrightarrow H \models^{\mathsf{c}} GC$.

Each such acc_p is a concrete graph formula, meaning that there is a concrete clan-typed graph K such that $A = (n\colon K \to_{\mathsf{c}} P_A, Con_A)$ for all (concrete) graph atoms occurring in acc_p; in fact we have $K = L$. Moreover, by the construction of $\widehat{p_{\mathsf{a}}}$, L uniquely identifies p.

We now define the set of concrete application conditions required in the theorem:

$$S = \{\, \mathrm{acc}_{p_{\mathsf{c}}} \mid p_{\mathsf{c}} \in \widehat{p_{\mathsf{a}}} \,\} \ .$$

Let $G \overset{p_{\mathsf{a}},m}{\Longrightarrow} H$ be a direct derivation. According to Theorem 4 it follows that for some $p_{\mathsf{c}} \in \widehat{p_{\mathsf{a}}}$, with left hand side L_{c} such that $m\colon L_{\mathsf{c}} \to_{\mathsf{c}} G$ is type-preserving, $G \overset{p_{\mathsf{c}},m}{\Longrightarrow} H$. It is important to note that, once more, p_{c} is uniquely determined by L_{c}, due to the fact that $m\colon L_{\mathsf{c}} \to_{\mathsf{c}} G$ is type-preserving.

[2] The result of [8] has since been extended in [9] to and beyond our graph formulae, namely to arbitrarily nested formulae as in [13], which means that the results below also hold for arbitrary formulae.

$\Rightarrow$ Assume $m \models^{\mathsf{c}} F$ for some $F \in S$, and let $p \in \widehat{p_{\mathsf{a}}}$ be such that $F = \mathrm{acc}_p$. This implies that $m\colon L \rightarrow_{\mathsf{c}} G$ (where L is the left hand side of p) is type-preserving. But then it follows that actually L and L_{c} are the same concrete clan-typed graphs, and hence $p = p_{\mathsf{c}}$, implying (by construction of acc_p) $H \models^{\mathsf{c}} GC$. Due to Theorem 1 it follows that $H \models^{\mathsf{a}} GC_{\mathsf{a}}$.

$\Leftarrow$ Assume $H \models^{\mathsf{a}} GC_{\mathsf{a}}$; hence (due to Theorem 1) $H \models^{\mathsf{c}} GC$, implying (by Theorem 5) $m \models^{\mathsf{c}} \mathrm{acc}_{p_{\mathsf{c}}}$. □

To conclude: we can start with some abstract graph formula F_{a} typed over type graph TGI with inheritance, flatten it to a concrete graph formula F_{c} as described in Section 3. F_{c} can be considered as simply typed over concrete closure $\widehat{TGI}$ and translated to a left application condition acc_L. Please note that acc_L is also typed over $\widehat{TGI}$. If there is an abstract rule p_a which has rule p as some concrete rule, then acc_L is also an application condition of p_a, since it is also typed over TGI due to Prop. 1.

6 Conclusions

In the literature, a variety of formal integrations of object-orientation and formal specification techniques exist. They are considered in the context of precise semantics for UML as well as for precise meta-modeling. It is the declared aim of the precise UML group [2,1] to come up with a precise standard semantics of the whole language UML, and then to use it for verification purposes. There are various approaches being developed, each formalizing certain aspects of UML with the intention of using the resulting precision for formal reasoning. In [5], the authors are especially concerned with the formalization of classes and their relations, inheritance and constraints on the basis of *description logics*. This work is dedicated entirely to the static part and does not regard the dynamic behavior of objects. Precise meta-modeling is considered in [14], where MOF and graph transformation concepts are integrated. While the aim and the basic ideas are similar to ours, the formalization chosen in [14] is different and not as comprehensive; in particular, it does not deal with constraints.

In addition to *formulating* a precise semantics, one has also to consider the process by which constraints are *enforced*. In this paper we have shown one way in which this can be done (by translation to application conditions). We are not aware of other approaches in the literature.

Summarizing, in this paper we have obtained a further, important step of integrating graph transformation with object-orientation concepts: now, type inheritance, constraints, and graph transformation concepts are integrated in one comprehensive formal framework. This offers the possibility to check properties for object-oriented software models. On the meta-model level, the results in our paper can be used to check constraints for model transformation. Further work is needed to carry over other analysis techniques to typed graph transformation with inheritance, to come up with a comprehensive visual and precise framework for object-oriented modeling and meta-modeling.

References

1. *Journal of Software and Systems Modeling* http:// www.sosym.org/ , 2004.

2. *The precise UML group* http://www.puml.org/ , 2004.
3. R. Bardohl, H. Ehrig, J. de Lara, O. Runge, G. Taentzer, and I. Weinhold. Node Type Inheritance Concepts for Typed Graph Transformation. Technical Report 2003–19, Technical University Berlin, Dept. of Computer Science, November 2003.
4. R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating Meta Modelling with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In M. Wermelinger and T. Margaria-Steffens, editors, *Proc. Fundamental Aspects of Software Engineering 2004*, volume 2984. Springer LNCS, 2004.
5. A. Cal, D. Calvanese, G. De Giacomo, and M. Lenzerini. A formal framework for reasoning on UML class diagrams. In *Proc. of the 13th Int. Sym. on Methodologies for Intelligent Systems (ISMIS 2002)*, volume 2366 of *Lecture Notes in Computer Science*, pages 503–513. Springer, 2002.
6. A. Corradini, F. L. Dotti, L. Foss, and L. Ribeiro. Translating Java into graph transformation systems. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Second International Conference on Graph Transformation*, volume 3256 of *Lecture Notes in Computer Science*, pages 383–389. Springer-Verlag, 2004.
7. A. Corradini, U. Montanari, and F. Rossi. Graph Processes. *Special Issue of Fundamenta Informaticae*, 26(3,4):241–266, 1996.
8. H. Ehrig, K. Ehrig, A. Habel, and K.-H. Pennemann. Constraints and application conditions: From graphs to high-level structures. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04)*, volume 3256. Springer LNCS, 2004.
9. A. Habel. Private communication, 2004.
10. R. Heckel and A. Wagner. Ensuring Consistency of Conditional Graph Grammars – A constructive Approach. *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, 2, 1995. http://www.elsevier.nl/locate/entcs/volume2.html.
11. T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *First International Conference on Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2002.
12. OMG. *MDA, MOF, UML and OCL specifications*. OMG, 2004. at the OMG web page: http://www.omg.org/.
13. A. Rensink. Representing first-order logic using graphs. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *International Conference on Graph Transformations (ICGT)*, volume 3256 of *Lecture Notes in Computer Science*, pages 319–335. Springer-Verlag, 2004.
14. D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modelling*, (1):1–24, 2003.