

Ensuring Structural Constraints in Graph-Based Models with Type Inheritance

Gabriele Taentzer¹ and Arend Rensink²

¹ Computer Science Department, Technical University of Berlin,
Franklinstr. 28/29, 10587 Berlin, Germany

² Computer Science Department, University of Twente,
P.O. Box 217, 7500 AE 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 is a valuable principle for the description of model domains. In the object-oriented approach, *class diagrams* are the basic means to specify classification structures; e.g., in UML (Unified Modeling Language) [11] for software systems and MOF (Meta Object Facility) [11] 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. Inheritance allows much more compact representations by reducing redundancy. The principle of inheritance has been carried over and formalized for graph transformation in [2]; there we have shown that node inheritance in typed graph transformation leads to a denser form of a graph transformation system, by a similar reduction of redundancy.

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 are 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 [11]. On the other hand, typed graphs can be equipped with *graph constraints*, as proposed first in [9], 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. In this paper we show how to express multiplicities and *edge inheritance* by graph constraints over type graphs with inheritance. Furthermore, we give a translation of constrained type graphs with inheritance to constrained flat type graphs. From existing theory [6] it then follows that graph constraints can be translated into (necessary and sufficient) pre-conditions for typed graph transformation rules. Our result can be regarded as a necessary further 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 [4] (leading to a theory of behavioral verification) and *refactoring* as in [10] (leading to a formal underpinning). We use a running example from the former area.

The paper is organized as follows: In the next section, we recall type graphs with node type inheritance as introduced in [2]; this will be the basis for further development. In Section 3 defined graph constraints over type graphs with inheritance and presents a translation to constraints over simple type graphs. Then Section 4 shows that multiplicities and edge inheritance are expressible by graph constraints. Section 5 describes how graph constraints can be ensured by typed graph transformation systems, reusing and extending the results in [6]. All proofs are omitted due to lack of space.

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

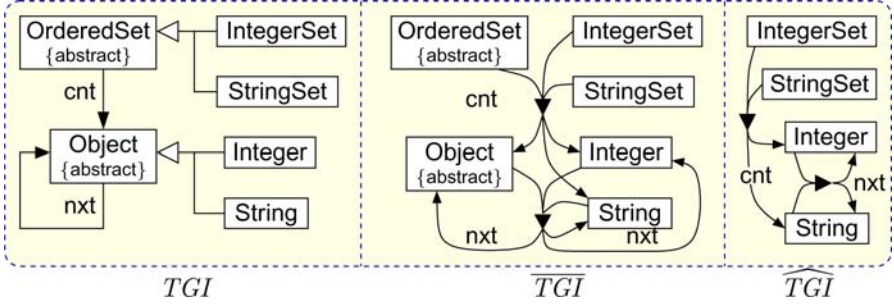


Fig. 1. A sample type graph with node type inheritance, and its abstract and concrete closure

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 $s, t : E \rightarrow N$), 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 reflexive-transitive closure of I .

Example 1. As sample type graph we use TGI in Fig. 1. This describes a special kind of sets, namely ordered sets, which contain a number of objects (indicated by cnt -edges from $OrderedSet$ -nodes to $Object$ -nodes) which can be put into some order (indicated by nxt -edges among the object). We consider two possible specializations of ordered sets, namely $StringSet$ and $IntegerSet$, which are intended to contain $Strings$ and $Integers$, respectively. Note that the type graph by itself does not yet enforce this constraint: that is, it does not rule out that $StringSets$ contain also $Integers$, and vice versa.

To benefit from the existing theory of graph transformation [5], 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 $\widehat{TGI} = \overline{TGI}|_{N-A}$.³

³ Given a graph $G = (N, E, s, t)$ and a set $X \subseteq N$, we denote by $G|_X$ the sub-graph $(X, E_X = \{e \in E \mid s(e), t(e) \in X\}, s|_{E_X}, t|_{E_X})$.

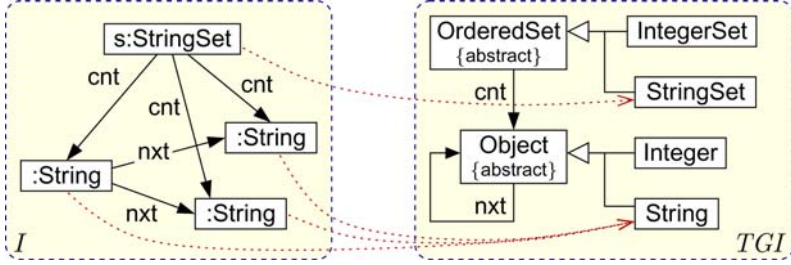


Fig. 2. Sample clan-typed graph

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 they give rise to different instances. We will define abstract graph transformation rules of which the left hand and right hand sides are typed over the abstract closure (see Sect. 5), whereas ordinary host graphs and concrete rules are typed over the concrete closure.

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

Note that, due to the canonical inclusion $inc_{TG}: \widehat{TGI} \hookrightarrow \widehat{TGI}$, all concrete instance graphs are abstract instance graphs. 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, from nodes to node types and from edges to edge types, respectively. 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 $ctp = (ctp_N: N_G \rightarrow N_{TG}, ctp_E: E_G \rightarrow E_{TG})$ 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. Fig. 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.

Proposition 1 (universal clan morphism, see [1]). *Let $TGI = (TG, I, A)$ be a type graph with inheritance. There is a universal clan morphism $u_{TG} : \overline{TGI} \rightarrow TG$ such that for each clan morphism $ctp : G \rightarrow TG$ there is a unique graph morphism $tp : G \rightarrow \overline{TGI}$ with $u_{TG} \circ tp = ctp$.*

We often write G for the clan-typed graph (G, ctp_G) . To formalize the relationship between abstract and concrete rules (see Sect. 5), we introduce the notion of *type refinement*. This imposes an order over the set of clan morphisms of a given instance graph: one clan morphism is said to be *finer* than another if it assigns more concrete node types to the instance graph nodes.

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

- $ctp_N(n) \in \text{clan}_I(ctp'_N(n))$ for all $n \in N_G$, and
- $ctp_E = ctp'_E$.

Given clan-typed graphs (G, ctp_G) and (H, ctp_H) over TGI , a morphism $g : G \rightarrow H$ is called type-refining if $g \circ ctp_H \leq ctp_G$, and type-preserving if $g \circ ctp_H = ctp_G$.

We write $(G, ctp_G) \leq (H, ctp_H)$ if $G = H$ and $ctp_G \leq ctp_H$. We write $g : G \rightarrow_c H$ to denote that G and H are both concrete and g is an injective type-preserving morphism, and $g : G \rightarrow_a H$ 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 $g : G \rightarrow H$ be type refining.*

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

3 Structural Properties over Type Graphs with Inheritance

The following definition extends the concept of graph constraints, originally introduced in [9] (where they are called consistency constraints). There are two points of change:

- We define constraints over concrete clan-typed graphs rather than ordinary 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.
- We allow constraints with multiple, disjunctively interpreted conclusions, rather than a single conclusion, as in [9, 6]. 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 $(n: L \rightarrow_c P, \text{Con})$ [$(n: L \rightarrow_a P, \text{Con})$], 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 $m: L \rightarrow_c G$ [$m: L \rightarrow_a G$], denoted $m \models^c A$ [$m \models^a A$], if for all injective type-preserving [type-refining] morphisms $p: P \rightarrow_c G$ [$p: P \rightarrow_a G$] such that $m = p \circ n$, there is a $(q: P \rightarrow C) \in \text{Con}$ and an injective type-preserving [type-refining] morphism $c: C \rightarrow_c G$ [$c: C \rightarrow_a G$] such that $p = c \circ q$. If $L = \emptyset$ (i.e., the empty graph) then we also write $G \models^c A$ [$G \models^a A$].
- A concrete [abstract] graph formula F over L is a boolean formula over concrete [abstract] graph atoms over L . The satisfaction relation \models^c [\models^a] is extended to graph formulae by defining the semantics of the boolean operators in the usual way. We call F a constraint if $L = \emptyset$, and an application condition otherwise.

Example 4. Fig. 3 shows three atoms over the type graph with inheritance TGI in Fig. 1. In this and later pictures we depict graph atoms $(L \rightarrow P, \{P \rightarrow C_i\}_i)$ more compactly as $L \rightarrow P \rightarrow \{C_i\}_i$.

- A_1 is satisfied by a morphism that selects an element without an outgoing `next`-edge;
- A_2 is satisfied by a graph if every `OrderedSet` is empty, i.e., contains no elements;
- A_3 is satisfied by a graph if for every `OrderedSet` and every pair of distinct elements contained in it, (at least) one element as an outgoing `next`-edge. Note that the graphs in the set on the right hand side are to be interpreted disjunctively.

A_2 and A_3 range over the same graph L , viz. the empty graph: in fact, they are *constraints* and can be combined into the formula $A_2 \wedge A_3$. A_1 , on the other hand, cannot be combined with A_2 or A_3 into one formula, since they are atoms over different graphs.

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 $A = \langle n: L \rightarrow_a Q, \text{Con} \rangle$, the K -flattening of A is defined by:

$$\begin{aligned} \text{flat}_K(A) &= \bigwedge \{ (n: K \rightarrow_c P, \text{flat}_P(\text{Con})) \mid P \leq Q \} \\ \text{flat}_P(\text{Con}) &= \{ q: P \rightarrow_c C \mid (q: Q \rightarrow_a D) \in \text{Con}, C \leq D \} . \end{aligned}$$

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

In the next section we give some examples of flattening. 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 $m: L \rightarrow_a G$ 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 $m: K \rightarrow_c G$ is type-preserving (which uniquely exists due to Prop. 2.1). This allows us to re-use existing theory on concrete graph formulae.

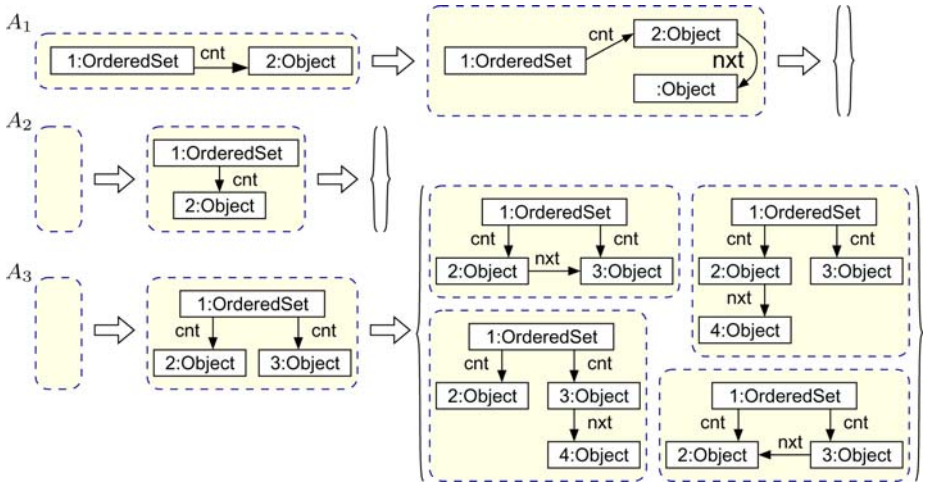


Fig. 3. Three example graph atoms

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

$$\begin{aligned} (m: L \rightarrow_a G) \models^a A &\text{ iff } (m: K \rightarrow_c G) \models^c \text{flat}_K(A) \\ (m: L \rightarrow_a G) \models^a F &\text{ iff } (m: K \rightarrow_c G) \models^c \text{flat}_K(F) . \end{aligned}$$

The proof relies on the fact that the flattening defined in Def. 7 “predicts” all concrete ways in which the abstract atom and formula could be satisfied, by taking conjunctions resp. disjunction over all concrete instance graphs that are \leq -predecessors of the abstract premises and conclusions.

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. (It should be noted, however, that there are many graph constraints that do not fall into either of these classes: for instance, A_3 in Fig. 3 cannot be expressed through multiplicities or edge inheritance.

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 $[i, j] \in \text{Mult}$, 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 $m_N : N_{TGI} \rightarrow \text{Mult}$, called node multiplicity function, and $m_{src}, m_{tar} : E_{TGI} \rightarrow \text{Mult}$, 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 $n \in N_{TG}$, $|ctp_G^{-1}(\text{clan}_I(n))| \in m_N(n)$;
- for all $e \in E_{TG}$ and $p \in ctp_G^{-1}(\text{clan}_I(\text{src}(e)))$, $|ctp_G^{-1}(e) \cap \text{src}_G^{-1}(p)| \in m_{tar}(e)$;
- for all $e \in E_{TG}$ and $p \in ctp_G^{-1}(\text{clan}_I(\text{tar}(e)))$, $|ctp_G^{-1}(e) \cap \text{tar}_G^{-1}(p)| \in m_{src}(e)$.

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

$$A_{n \geq i} = (\emptyset, \{\emptyset \rightarrow G_i^n\})$$

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

- 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

$$A_{e \geq i}^{src} = (G_1^{tar(e)}, \{q^{tar} : G_1^{tar(e)} \rightarrow H \mid H \in \mathcal{G}_i^{e,tar}\})$$

$$F_{e \leq j}^{src} = \bigwedge \{(H, \emptyset) \mid H \in \mathcal{G}_{j+1}^{e,tar}\}$$

with q^{tar} mapping the sole node of $G_1^{tar(e)}$ to the unique target node of H .

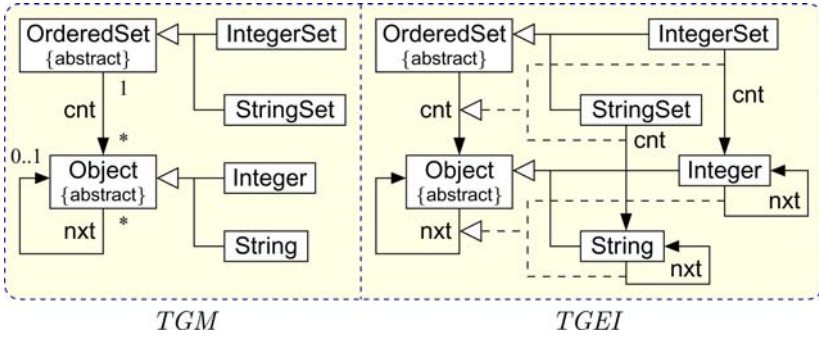


Fig. 4. Type graph with multiplicities, respectively edge inheritance

- 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).

The following theorem states that this formula indeed expresses the multiplicity semantics according to Def. 10. The proof is omitted here.

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 $G \models^a F_{TGM}$.*

Example 5 (multiplicity constraints). In Figure 4 (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 5. (Note that we have omitted the empty initial graph.) 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.

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. 6. The first of these is the complete flattening of the first constraint in Fig. 5; the second and third show two of the four atomic constraints that constitute the flattening of the second constraint in Fig. 5.

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.

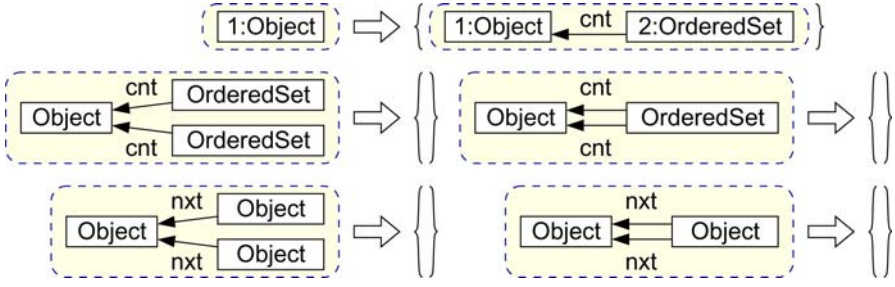


Fig. 5. Multiplicity constraints as abstract graph atoms

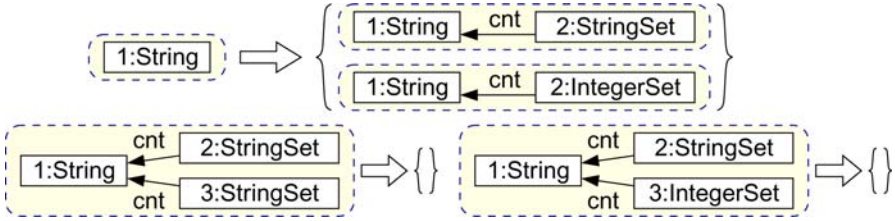


Fig. 6. Flattened multiplicity constraints

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 $src(e) \in \text{clan}_I(src(f))$ and $tar(e) \in \text{clan}_I(tar(f))$.

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 $(e, \text{ctp}_G(x)) \in I$, $\text{ctp}_G(src_G(x)) \notin \text{clan}_I(src_{TG}(e))$ and $\text{ctp}_G(tar_G(x)) \notin \text{clan}_I(tar_{TG}(e))$.

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 $TGEI = (TG, I, A)$, define $F_{TGEI} = \bigwedge_{(e,f) \in I_E} A_{e,f}^{src} \wedge A_{e,f}^{tar}$ where

$$A_{e,f}^{src} = (G^{src(e),f,tar(f)}, \{q_{e,f} : G^{src(e),f,tar(f)} \rightarrow G^{src(e),e,tar(e)}\})$$

$$A_{e,f}^{tar} = (G^{src(f),f,tar(e)}, \{q_{e,f} : G^{src(f),f,tar(e)} \rightarrow G^{src(e),e,tar(e)}\}) .$$

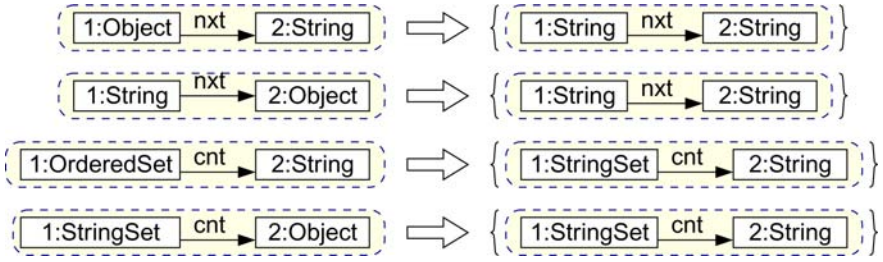


Fig. 7. Edge inheritance as graph constraints

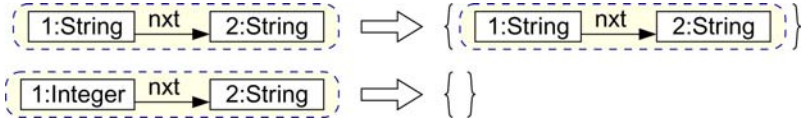


Fig. 8. Flattened edge inheritance constraints

with G^{n_1, e, n_2} for $n_1 \in \text{clan}_I(\text{src}(e))$ and $n_2 \in \text{clan}_I(\text{tar}(e))$ 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.

The following theorem states that this formula indeed expresses the satisfaction of the edge inheritance relation, according to Def. 13. The proof is omitted here.

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 $G \models^a F_{TGEI}$.*

Example 6 (edge inheritance constraints). In Figure 4 (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).

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. 8 are the complete flattening of the first constraint in Fig. 7. Note that the first flattened constraint is always true, and the second describes a handle not allowed by the edge inheritances.

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 [6].

We first define abstract and concrete rules with application conditions, and their matching. The following definition extends that in [2].

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 \hat{p} .

Example 7 (abstract rules). Fig. 9 shows two abstract rules, modelling the insertion of a new string into an ordered set. `InsertFirstString` inserts a string into an empty set (the emptiness is ensured by the application condition), whereas `InsertNextString` handles the case of a non-empty set: an existing object will become the predecessor of the newly inserted `String`. Note that, if this existing object already has a successor, application of the rule will violate the multiplicity constraint in Fig. 5. We will show below (Ex. 8) that this condition is obtained automatically by translation from the multiplicity constraints. (Note that Fig. 9 only shows the left and right hand sides; the interface graph can be deduced from the node identities.)

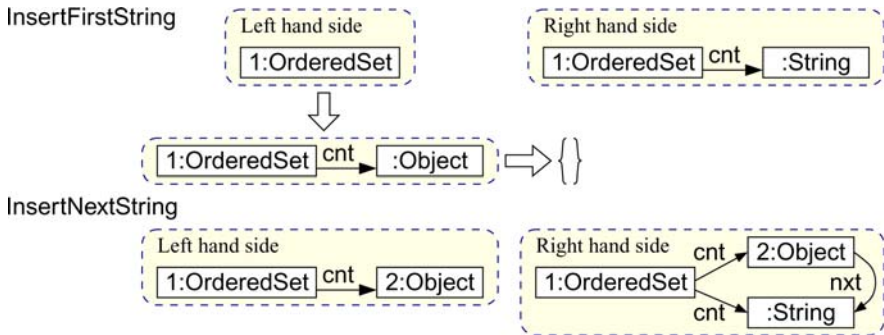


Fig. 9. Abstract rules for inserting a String into an OrderedSet

Definition 16 (rule matching and application). Let $p = (L \xleftarrow{l} K \xrightarrow{r} R, F_L, F_R)$ be a derivation rule, G and H concrete clan-typed graphs, and $m: L \rightarrow G$ a type-preserving graph morphism.

- If p is a concrete rule, then m is a match of p in G if

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

Given a match m , a concrete direct derivation $G \xrightarrow{p,m} H$ exists if there is a span of type-preserving morphisms $G \leftarrow D \rightarrow H$ and a co-match $m^* : R \rightarrow H$ of p in H that give rise to a derivation in the classical theory of (untyped) graph transformations [5]. The derivation is valid if $m^* \models^c F_R$.

– If p is an abstract rule, then m is a match of p in G if

- m is a match of the untyped rule $\langle L \xleftarrow{l} K \xrightarrow{r} R \rangle$ 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 $r(x_1) = r(x_2)$;
- $m \models^a F_L$.

Given a match m , an abstract direct derivation $G \xrightarrow{p,m} H$ exists if there is a span of type-preserving morphisms $G \leftarrow D \rightarrow H$ and a co-match $m^* : R \rightarrow H$ of p in H that give rise to a derivation in the sense of [2]. The derivation is valid if $m^* \models^a F_R$.

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

Theorem 4 (equivalence of abstract and concrete derivations). *Given an abstract rule $p_a = (L \leftarrow K \rightarrow R, F_L, F_R)$, concrete clamped graph G, H and a structural match morphism $m : L \rightarrow G$ (i.e. a match with respect to the untyped rule $\langle L \leftarrow K \rightarrow R \rangle$), the following statements are equivalent:*

1. m is a match of p_a in G , yielding a valid abstract direct derivation: $G \xrightarrow{p_a, m} H$.
2. m is a match of the concrete rule $p_c = (L_c \leftarrow K_c \rightarrow R_c, F_L^c, F_R^c)$ in G with $p_c \in \widehat{p}_a$ and $m : L_c \rightarrow_c G$ type-preserving, yielding a valid concrete direct derivation: $G \xrightarrow{p_c, m} H$.

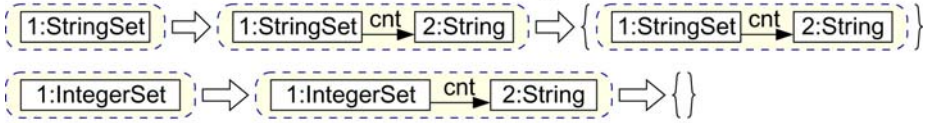
In the following, we want to use the translation of graph constraints to application conditions of graph rules as described in [6]. Therefore, we have to restrict the class of graph formulae we use to the ones defined in [6]. If we restrict our concrete graph constraints $GC = (P, Con)$ to those with $|Con| \leq 1$, they become equivalent to the positive and negative graph constraints of [6]: the case of $|Con| = 1$ corresponds to positive graph constraints, while the case of $|Con| = 0$ correspond to negative graph constraints.⁴ Another difference is that, in [6], 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 [6].

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

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

⁴ The result of [6] has since been extended in [8] 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.

Concrete application conditions for InsertFirstString



Concrete application condition for InsertNextString

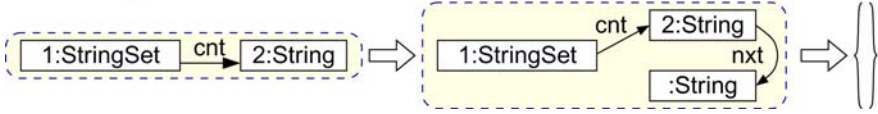


Fig. 10. Derived application conditions

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 $G \xrightarrow{p_a, m} H$ we have: $(\exists F \in S : m \models^c F) \Leftrightarrow H \models^a GC_a$.*

Thus, given some abstract graph constraint formula F_a typed over type graph TGI with inheritance, we can 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 concrete left application condition acc_L that guarantees F_a . Note that acc_L is also typed over \widehat{TGI} . Unfortunately, there is no straightforward way to translate acc_L to an abstract application condition.

Example 8 (additional application constraints for abstract rules). Consider the constraints in Figs. 5 and 7, respectively and the abstract rules in Fig. 9. Fig. 10 shows some of the elements of S derived for this case according to Th. 6.

For rule `InsertFirstString`, the multiplicity constraints (Fig. 5) do not lead to interesting application conditions, since the left-hand side does not contain an `Object`; but the edge inheritance constraints (Fig. 7) induce the two application conditions shown in the figure. These essentially express that the `OrderedSet` involved has to be a `StringSet`. For rule `InsertNextString`, the multiplicity constraint on `nxt`-edges leads to the third application condition of Fig. 10 (among others). This expresses that the node with identity 2 in the left hand side of the rule (which has the abstract type `Object` in the rule but concrete type `String` in the condition) may not have an outgoing `nxt`-edge; see also Fig. 3.

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 [12] 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 [3], 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. 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.
2. 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, *Fundamental Aspects of Software Engineering (FASE)*, volume 2984 of *LNCS*. Springer, 2004.
3. A. CalŒ, D. Calvanese, G. De Giacomo, and M. Lenzerini. A formal framework for reasoning on UML class diagrams. In *Int. Symp. on Methodologies for Intelligent Systems (ISMIS 2002)*, volume 2366 of *LNCS*, pages 503–513. Springer, 2002.
4. A. Corradini, F. L. Dotti, L. Foss, and L. Ribeiro. Translating Java into graph transformation systems. In Ehrig et al. [7], pages 383–389.
5. A. Corradini, U. Montanari, and F. Rossi. Graph Processes. *Special Issue of Fundamenta Informaticae*, 26(3,4):241–266, 1996.
6. H. Ehrig, K. Ehrig, A. Habel, and K.-H. Pennemann. Constraints and application conditions: From graphs to high-level structures. In Ehrig et al. [7].
7. H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors. *Second International Conference on Graph Transformation (ICGT)*, volume 3256 of *LNCS*. Springer, 2004.
8. A. Habel. Private communication, 2004.
9. 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>.

10. 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 (ICGT)*, volume 2505 of *LNCS*, pages 286–301. Springer, 2002.
11. OMG. *MDA, MOF, UML and OCL specifications*. OMG, 2004. at the OMG web page: <http://www.omg.org/>.
12. pUML. *The precise UML group* <http://www.puml.org/>, 2004.
13. A. Rensink. Representing first-order logic using graphs. In Ehrig et al. [7], pages 319–335.
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.