Electronic Communications of the EASST
Volume 47 (2012)

Proceedings of the 11th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2012)

A Flattening Approach for Attributed Type Graphs with Inheritance in Algebraic Graph Transformation

Christine Natschläger and Klaus-Dieter Schewe

14 pages

Guest Editors: Andrew Fish, Leen Lambers
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A Flattening Approach for Attributed Type Graphs with Inheritance in Algebraic Graph Transformation

Christine Natschläger[1] **and Klaus-Dieter Schewe**[2]

[1] christine.natschlaeger@scch.at
Software Competence Center Hagenberg GmbH
Hagenberg, Austria

[2] kd.schewe@scch.at, kd.schewe@cdcc.faw.jku.at
Software Competence Center Hagenberg GmbH
Hagenberg, Austria and
Research Institute for Applied Knowledge Processing
Johannes Kepler University Linz, Linz, Austria

Abstract: The algebraic graph transformation approach was initiated in 1973 and supports the rule-based modification of graphs based on pushout constructions. The vertex and edge types used within the rules (or productions) as well as possible inheritance relationships defined between them are specified in the type graph. However, the termination proof can only be accomplished for graph transformation systems without inheritance relationships. Thus, all graph transformation systems with inheritance relationships in the type graph must be flattened. To this end, the algebraic graph transformation approach provides a formal description for how to flatten the type graph as well as a definition of abstract and concrete productions.

In this paper, we will extend the definitions to also consider vertices in negative application conditions with finer node types and positive application conditions. Furthermore, we will prove the semantic equivalence of the original and the flattened graph transformation system. The whole flattening algorithm is then implemented in a prototype which supports an abstract or concrete flattening of a given graph transformation system. The prototype is finally evaluated within a case study.

Keywords: Graph Transformation, Inheritance, Flattening

1 Introduction

The main idea of graph transformation is the rule-based modification of graphs from a source to a target graph. The application of a production p (or rule) consisting of a graph L (left-hand side (LHS)) and a graph R (right-hand side (RHS)) ($p = (L,R)$) means finding a match of L in the source graph and replacing L by R leading to the target graph [EEPT06, p. 6]. The algebraic graph transformation approach was initiated by Ehrig, Pfender and Schneider in 1973 (see [EPS73]) and is based on pushout constructions, which are used to model the gluing of graphs (detailed description in [EEPT06]).

In order to show that a graph transformation system (GTS) is confluent and globally deterministic, it is necessary to prove local confluence and termination. A graph transformation $G \overset{*}{\Rightarrow} H$ is

called terminating if no further production is applicable to H anymore, so that there is no infinite sequence of graph transformations [EEPT06, p. 59f]. Termination of a GTS can be concluded if the criteria described in the termination theorem in [EEPT06, p. 63f] are met. However, the termination theorem can only be used for GTSs without inheritance in the type graph. If a type graph includes inheritance relationships or abstract nodes, then the graph transformation system must be flattened.

The algebraic graph transformation approach describes the flattening of the type graph and provides a definition for abstract and concrete productions. In our research, we further consider vertices in negative application conditions with finer node types and positive application conditions. Moreover, we prove the semantic equivalence of the original and the flattened GTS.

The application of the algebraic graph transformation approach is supported by the tool AGG (see [AGG11]). AGG was introduced by Löwe and Beyer in 1993 (see [LB93]) and later redesigned and extended by Taentzer (see [Tae04]). In December 2010, AGG was updated to version 2.0 [AGG11]. AGG provides a graphical editor and can be used for specifying graph grammars with a start graph or for typed attributed graph transformations. Furthermore, AGG offers analysis techniques as for consistency checking, critical pair analysis and termination evaluation. Since AGG does not support the flattening of GTSs with inheritance, we further implement the extended flattening algorithm within a prototype.

2 Motivation

In [Nat11], we extended BPMN with deontic logic to explicitly highlight the modality of tasks (obligatory, permissible or alternative) and reduce the structural complexity of the process flow. We called this extension Deontic BPMN and further defined an algebraic graph transformation from BPMN to Deontic BPMN named DeonticBpmnGTS. DeonticBpmnGTS specifies an attributed type graph with inheritance as shown in Fig. 1.

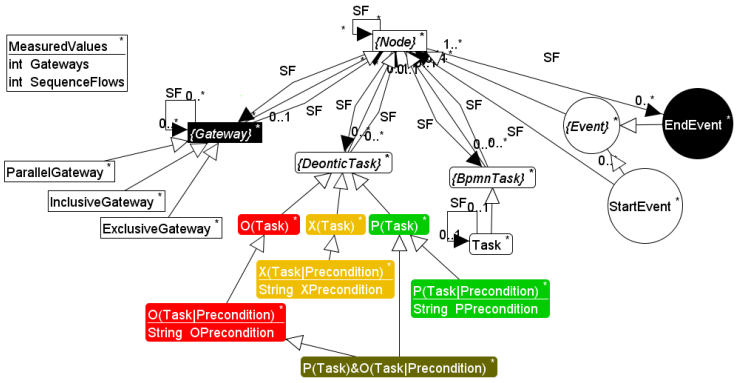
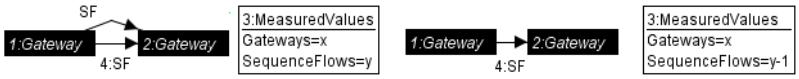


Figure 1: Type Graph of *DeonticBpmnGTS*

The type graph of DeonticBpmnGTS comprises five abstract node types (*Node*, *Gateway*, *DeonticTask*, *BpmnTask*, and *Event*), several concrete node types (e.g., *O(Task)* (obligatory task), *P(Task)* (permissible task), *X(Task)* (alternative task), ...), and inheritance relationships defined

between them. The node type *MeasuredValues* is used to store meta-information like the number of gateways and sequence flows in order to study the reduction of structural complexity. In addition, DeonticBpmnGTS provides one edge label called SF for sequence flows representing several edge types.

The goal of DeonticBpmnGTS is to transform BPMN diagrams to Deontic BPMN diagrams thereby replacing all *BpmnTasks* with *DeonticTasks*. For this purpose, DeonticBpmnGTS defines 18 productions with 27 application conditions (21 negative application conditions (NACs) and 6 positive application conditions (PACs)). The productions cover sequences, gateways (parallel, exclusive and inclusive) and iterations and are distributed across four layers. For example, the production *SeveralPhiReductionRule* addresses duplicate empty (or *Phi*) paths between two gateways by removing one of them (see Fig. 2). Since both sequence flows are unconditional, they are semantically equivalent and nothing (Φ) has to be done if a token traverses a path. The element *MeasuredValues* shows that the transformation leads to a reduction of one sequence flow.



(a) LHS (b) RHS

Figure 2: SeveralPhiReductionRule

Afterwards, a typed graph can be created for a concrete BPMN model and is then transformed to Deontic BPMN. This is demonstrated by an example whose BPMN model is shown in Fig. 3 in the image view. According to the *MeasuredValues* element, the BPMN model consists of eight gateways and twenty-nine sequence flows.

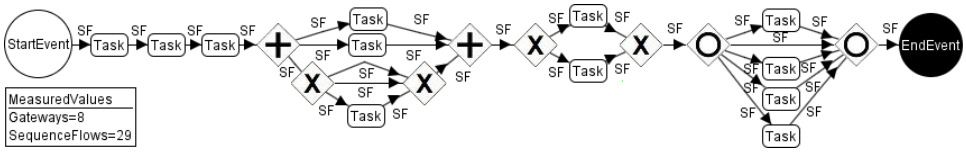


Figure 3: Example: BPMN Model

The resulting Deontic BPMN model is shown in Fig. 4. This model consists of six gateways and twenty-four sequence flows. Thus, the transformation leads to a reduction of two gateways and five sequence flows. In addition, the obligatory (O), alternative (X) and permissible (P) tasks can be distinguished on first sight based on their prefix and color.

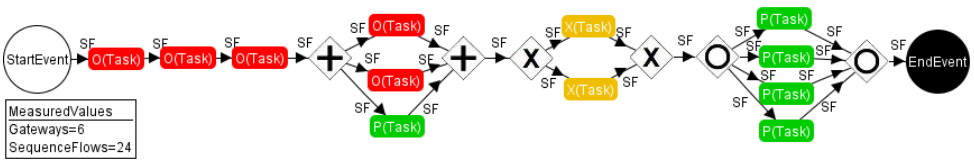


Figure 4: Example: Deontic BPMN Model

Our next goal was then to prove that DeonticBpmnGTS is a trusted model transformation. According to [VVE+06], the most important correctness properties of a trusted model transformation are termination, uniqueness (confluence), and behavior preservation. In a first step, we proved that the transformation from BPMN to Deontic BPMN is unique and behavior preserving. However, since DeonticBpmnGTS specifies an attributed type graph with inheritance, it is necessary to flatten the GTS in order to prove termination.

3 Related Work

The fundamentals of the algebraic graph transformation approach are presented in [EEPT06]. According to the authors, supporting node type inheritance leads to a denser form of graph transformation systems, since several similar productions can be abstracted into one. However, in order to benefit from the well-founded theory of typed attributed graph transformation, it is in some cases recommended to flatten graph transformations with an attributed type graph with inheritance (ATGI) to equivalent typed attributed graph transformations without inheritance (see [EEPT06, p. 260ff] and [LBE+07]). For example, the termination theorem provided in [EEPT06, p. 63f] can only be used for GTSs without inheritance in the type graph. Since a termination analysis has to consider all possible replacements of an abstract node to correctly calculate the creation and deletion layers, a kind of flattening (either explicitly or implicitly) is necessary. An implicit flattening implies an adaptation of the termination theorem to also consider derived nodes. For example, Golas et al. adapted the critical pair analysis and introduced abstract critical pairs to also cope with abstract productions (see [GLEO12]). In contrast, an explicit flattening requires the flattening of the entire graph transformation system. In the following, explicit flattening is used for the termination analysis, since the flattening approach can be based on already existing definitions and the flattening results can be validated. Three definitions that are relevant for the flattening algorithm are presented in subsequent paragraphs.

The first definition defines an attributed type graph with inheritance (ATGI) (see [EEPT06, p. 260f, Definition 13.1]). An ATGI ($ATGI = (TG, Z, I, A)$) consists of an attributed type graph ($ATG = (TG, Z)$), an inheritance graph ($I = (I_V, I_E, s, t)$) and a set of abstract nodes ($A \subseteq I_V$). Furthermore, the inheritance clan of a node ($clan_I(n)$) represents all its subnodes.

The second definition defines the closure (or flattening) of ATGIs (see [EEPT06, p. 262f, Definition 13.4]). In a first step, all inheritance relationships are removed from the type graph and the edges and attributes of a parent node are copied for every child node. Thus, additional graph-, node attribute-, and edge attribute edges may be inserted in the type graph. The result is called the abstract closure (or abstract flattening) of the type graph. In a second step, all abstract nodes together with adjacent edges are removed from the type graph. This is called the concrete closure (or concrete flattening) of the type graph.

The third definition defines abstract and concrete productions (see [EEPT06, p. 272f, Definition 13.16]). An abstract production typed over ATGI is given by $p = (L \xleftarrow{l} K \xrightarrow{r} R, type, NAC)$, where $type$ is a triple of typing morphisms, e.g. $type_L : L \to ATGI$, and NAC is a set of triples ($nac = (N, n, type_N)$) with an attributed graph N, a morphism $n : L \to N$ and a typing ATGI-clan morphism $type_N : N \to ATGI$, such that the following conditions hold: (I) types in K are equal with the morphism image types in L and R, (II) all nodes that only exist in R must not be of an

abstract type, and (III) all nodes in a *NAC* with a morphism image in L have the same or a finer type as the morphism image. In a concrete production ($p_t = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R, t, \overline{NAC})$) again (I) the types in K are equal with the morphism image types in L and R, but (II) the types can be equal or finer than in the abstract production. Furthermore, (III) all nodes that only exist in R must remain of the same type and (IV) all nodes in a *NAC* with a morphism image in L are flattened to the same type whereas nodes without a morphism image are flattened to all subtypes. Considering the last aspect, we suggest that a node in a *NAC* should only be set to the same morphism image type, if the original node types in L and *NAC* were equal (node type in a *NAC* might be finer).

The flattening approach was extended in [LBE+07]. The main results of this publication show the equivalence of the abstract and the corresponding concrete transformation as well as the equivalence of attributed graph grammars with and without inheritance. Open issues of the flattening approach are changed attributes, dependency relationships with multiplicity and edge inheritance (cf. [LBE+07]). The last two aspects were defined for graph transformations without attributes in [TR05].

However, we identified open issues in the definition of abstract and concrete productions. Thus, we will extend the definition to also consider vertices in negative application conditions with finer types as well as positive application conditions. The extensions are generic and can be used to flatten any graph transformation system with inheritance. Furthermore, we will prove the semantic equivalence of the original and the flattened graph transformation system.

4 Flattening Algorithm

First of all, we provide some suggestions and extensions concerning the definition of abstract and concrete productions (see [EEPT06, p. 272f, Definition 13.16]). The fourth condition of concrete productions is adapted from:

- "for each $(N, n, type_N) \in NAC$, we have all $(N, n, t_N) \in \overline{NAC}$ for concrete ATGI-clan morphisms t_N satisfying $t_N \circ n = t_L$ and $t_N \leq type_N$."

to:

- for each $(N, n, type_N) \in NAC$, we have all $(N, n, t_N) \in \overline{NAC}$ for concrete ATGI-clan morphisms t_N satisfying $t_N \leq type_N$ and $\forall x \in L_{V_G}$:
  - if $(type_N \circ n)(x) = type_L(x)$ or $(type_N \circ n)(x) > t_L(x)$ then $(t_N \circ n)(x) = t_L(x)$
  - else if $(type_N \circ n)(x) < type_L(x)$ and $(type_N \circ n)(x) \leq t_L(x)$ then $t_N(x) = type_N(x)$
  - else $(N, n, t_N) \notin \overline{NAC}$

This definition also supports a stepwise flattening as required for the prototype implementation. The stepwise flattening flattens one production after the other before removing the inheritance relationships from the type graph. In the first case, $type_N$ and $type_L$ are equal or the flattened t_L is finer than $type_N$, thus the node in N is flattened to the same type as the node in L. This case is also addressed by the original definition. In the second case, $type_N$ is finer than $type_L$ and also finer or equal than the flattened t_L, so $type_N$ is not flattened. This case need not be considered

in the original definition, because all inheritance relationships are removed from the type graph during the flattening and, thus, a NAC with a finer node type is not applicable. However, this case is necessary for a stepwise flattening, since a NAC with a finer node type of, e.g., a concrete production is relevant for the further flattening of the production. In the third case, however, the NAC is removed from the set of $\overline{NAC}$ of this production. For example, if $type_L$ defines a *Node*, $type_N$ a *Gateway* and the node in L is flattened to an *Event* (t_L), then the NAC can be removed, because it is neither applicable nor relevant for a further flattening, since an *Event* cannot be replaced by a *Gateway*.

Furthermore, we also consider PACs. The flattening of PACs is similar to that of NACs described in [EEPT06, p. 272f, Definition 13.16] and extended above. However, if a PAC comprises a refined node and this node is still finer than the flattened t_L (case 2) or the morphism image in L is flattened to a sibling (case 3), then the PAC must remain to ensure that the production is never applied (so case 2 is mandatory for PACs). Alternatively, the whole production can be deleted.

In addition, there is a difference between flattening of NACs and PACs concerning nodes without morphism image in L. If a NAC has an abstract node without morphism image, then this node is flattened to all concrete nodes resulting in several flattened NACs for one production. All NACs must be fulfilled in order to apply the production. However, if such a node is part of a PAC, then the semantics of the original definition is that only one instance with a concrete node must be fulfilled. However, if the PAC is flattened, then all flattened PACs must be satisfied in order to apply the production. For example, if the original PAC defines that a node X must address an abstract node Y, and Y has no morphism image in L, then flattening this PAC means that node X must address all subnodes of Y, whereas the original definition states that only one subnode must be addressed. Thus, it is necessary to convert all flattened PACs to general application conditions (GACs) and to specify a formula. The formula defines which GACs must be fulfilled under which circumstances. For example, all flattened GACs of one abstract PAC are defined to be disjunctive (concatenated with $\vee$) and, thus, only one of the GACs must be fulfilled.

All other PACs which only comprise abstract nodes that have a morphism image in L are flattened together with the production. These PACs can either remain as positive application condition (PAC) or can be transformed to a GAC. GACs that originate from different former PACs are defined to be conjunctive (concatenated with $\wedge$). The two approaches are semantically equivalent. In the following, we will transform these PACs to GACs to be consistent with PACs that have abstract nodes without morphism image. Then we can generally define:

- $\overline{PAC}(x) \rightarrow \overline{GAC}(x)$;
- $PAC(x) \equiv \overline{GAC_1}(x) \vee \ldots \vee \overline{GAC_n}(x)$;
- $PAC(x) \wedge PAC(y) \equiv \overline{GAC}(x) \wedge \overline{GAC}(y)$

Summing up, we provided two extensions concerning the definition of concrete productions. So we now have to prove that, despite of the extensions, the original and the flattened graph transformation system are still semantically equivalent. As already mentioned in the related work, the equivalence of abstract and corresponding concrete transformations as well as the equivalence of attributed graph grammars with and without inheritance is shown in [LBE+07]. The theorem for equivalence of attributed graph grammars as well as the corresponding proof is

not affected by the extensions (see Theorem 3 in [LBE+07]). However, Theorem 2 and Lemma 3 of [LBE+07] must be adapted to also cover the proposed extensions.

Theorem 1 *Equivalence of Transformations (based on [LBE+07] Theorem 2, adapted): Given an abstract production* $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R, type, AC)$ *over an attributed type graph ATGI with inheritance, a concrete typed attributed graph* $(G, type_G)$ *and a match morphism* $m : L \rightarrow G$ *(which satisfies the gluing condition w.r.t. the untyped production* $L \leftarrow K \rightarrow R$*). Then the following statements are equivalent, where* $(H, type_H)$ *is the same concrete typed graph in both cases:*

1. $m : L \rightarrow G$ *is a consistent match w.r.t. the abstract production p yielding an abstract direct transformation* $(G, type_G) \overset{p,m}{\Rightarrow} (H, type_H)$.
2. $m : L \rightarrow G$ *is a consistent match w.r.t. the concrete production* $p_t = (L \leftarrow K \rightarrow R, t, \overline{AC})$ *with* $p_t \in \widehat{p}$ *(set of concrete productions) and* $t_L = type_G \circ m$ *(where* t_K*,* t_R *and* $\overline{AC}$ *are uniquely defined by Lemma 1(1)) yielding a concrete direct transformation* $(G, type_G) \overset{p_t,m}{\Rightarrow} (H, type_H)$.

Lemma 1 *Construction of Concrete and Abstract Transformations (based on [LBE+07] Lemma 3, extended): Given an abstract production* $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R, type, AC)$ *with* $AC = (NAC \cup PAC)$ $(NAC = \{(N_i, n_i, type_{N_i}) | i \in I\}$ *and* $PAC = \{(P_i, p_i, type_{P_i}) | i \in I\})$*, a concrete typed attributed graph* $(G, type_G : G \rightarrow ATGI)$ *and a consistent match morphism* $m : L \rightarrow G$ *w.r.t. p and* $(G, type_G)$*, we have [...]:*

1. *There is a unique concrete production* $p_t \in \widehat{p}$ *with* $p_t = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R, t, \overline{AC})$ *and* $t_L = type_G \circ m$*. In this case,* t_K*,* t_R *and* $\overline{AC}$ *are defined by:*
   - $t_K = t_L \circ l$;
   - $t_{R,V_G}(x) =$ *if* $x = r_{V_G}(x')$ *then* $t_{K,V_G}(x')$ *else* $type_{R,V_G}(x)$ *for* $x \in R_{V_G}$;
   - $t_{R,X} = type_{R,X}$ *for* $X \in \{V_D, E_G, E_{NA}, E_{EA}, D\}$;
   - $\overline{AC} = (\overline{NAC} \cup \overline{GAC})$;
   - $\overline{NAC} = \bigcup_{i \in I} \{(N_i, n_i, t_{N_i}) | t_{N_i}$ *is a concrete ATGI-clan morphism with* $t_{N_i} \leq type_{N_i}$ *and* $\forall x \in L_{V_G}$:
     - *if* $(type_{N_i} \circ n_i)(x) = type_L(x)$ *or* $(type_{N_i} \circ n_i)(x) > t_L(x)$ *then* $(t_{N_i} \circ n_i)(x) = t_L(x)$
     - *else if* $(type_{N_i} \circ n_i)(x) < type_L(x)$ *and* $(type_{N_i} \circ n_i)(x) \leq t_L(x)$ *then* $t_{N_i}(x) = type_{N_i}(x)$
     - *else* $(N_i, n_i, t_{N_i}) \notin \overline{NAC}\}$;
   - $\overline{PAC} = \bigcup_{i \in I} \{(P_i, p_i, t_{P_i}) | t_{P_i}$ *is a concrete ATGI-clan morphism with* $t_{P_i} \leq type_{P_i}$ *and* $\forall x \in L_{V_G}$:
     - *if* $(type_{P_i} \circ p_i)(x) = type_L(x)$ *or* $(type_{P_i} \circ p_i)(x) > t_L(x)$ *then* $(t_{P_i} \circ p_i)(x) = t_L(x)$
     - *else if* $(type_{P_i} \circ p_i)(x) < type_L(x)$ *then* $t_{P_i}(x) = type_{P_i}(x)\}$;
   - $\overline{PAC}(x) \rightarrow \overline{GAC}(x)$;

In addition, all $\overline{GAC}$ are concatenated by a formula as follows:

- $PAC(x) \equiv \overline{GAC_1}(x) \vee ... \vee \overline{GAC_n}(x)$;
- $PAC(x) \wedge PAC(y) \equiv \overline{GAC}(x) \wedge \overline{GAC}(y)$

[...]

The proof for Theorem 2 is provided in [LBE+07] and can be extended from NAC to AC. Thus, it can be concluded that the original and the flattened graph transformation system are semantically equivalent and every concrete graph in a confluent graph transformation system is transformed to the same resulting graph independent of whether the GTS is flattened or not.

5 Implementation

Based on the flattening algorithm a concrete prototype was developed using the programming language C#. The prototype flattens the inheritance of a GTS in order to execute the termination analysis. It thereby considers abstract nodes, inheritance relationships, multiple inheritance, attributes and dependency relationships. An overview of the *Flattening Prototype* is given below:

```
ReadXmlFile();
ReadNodeTypes();
FindAndOrderInheritancesBetweenNodes();
foreach (Inheritance in Inheritances)
    foreach (Rule in Rules)
        foreach (AC in Rule.ACs)
            foreach (Node in AC.Nodes)
                if (Node.Type == ParentType)
                    if (!HasMorphismNodeInLHS())
                        CloneAC();
                        ReplaceParentWithChildNodeInAC();
                        ChangeIds();
                        InsertACInDict();
    FindCombinationsOfNodeReplacementsInAC();
RemoveDuplicateACs();
InsertACs();
foreach (Inheritance in Inheritances)
    foreach (Rule in Rules)
        foreach (Node in Rule.LHS.Nodes)
            if (Node.Type == ParentType)
                CloneRule();
                ReplaceParentWithChildNodeInLHS();
                if (HasMorphismNodeInRHS())
                    rhsNode = GetMorphismNodeInRHS();
                    ReplaceParentWithChildNodeInRHS(rhsNode);
```

```
                foreach (AC in Rule.ACs)
                    acNode = GetMorphismNodeInAC();
                    ReplaceParentWithChildNodeInAC(acNode);
                ChangeIds();
                InsertRuleInDict();
    FindCombinationsOfNodeReplacementsInRule();
    RemoveInheritanceBetweenNodes();
RemoveDuplicateRules();
InsertRules();
RemoveAbstractNodes();
SaveXmlFile();
```

First of all, the user provides the AGG-file (XML) and a boolean parameter, which defines whether an abstract or concrete flattening should be executed. The *Flattening Prototype* starts with reading in the XML-file and the node types together with their hierarchy. Then, all inheritance relationships are identified and ordered starting with those relationships where the parent node is not derived from any further node. The ordering is necessary, since a node Z may be derived from a node Y which is in turn derived from a node X that defines an attribute *name*. If the inheritance relationship between Z and Y is considered first, then no attribute is taken over from Y to Z. Thus, it is necessary to first flatten the inheritance relationship between Y and X thereby taking the attribute *name* over to Y, followed by the flattening of Z and Y.

Then, all inheritance relationships are iterated and the ACs of all productions are taken into account. For every node that has the same type as the parent node type and no morphism image in L, the AC is duplicated and the parent node type is replaced by the child node type. If the child node type has an additional attribute, then this attribute remains unspecified due to the missing value. Attributes do not affect the termination analysis, but default values may be considered within further work. Then the IDs within the new AC are changed and the AC is inserted in a dictionary. Afterwards, possible combinations of replacements are identified. This is necessary in case an AC comprises two or more abstract nodes. For every combination a new AC is created, the nodes are replaced, IDs changed and the AC is inserted in the dictionary. Since the calculation of combinations may also lead to duplicate ACs, all generated ACs are iterated and the duplicate ACs are removed. Afterwards the remaining ACs are inserted in the XML-structure.

Similarly, all inheritance relationships are iterated again to consider the productions. Whenever a node in L corresponds with the parent node type, the production is duplicated and the node is replaced by the child node type. Morphism nodes in R or within ACs (except finer types) are replaced by the same child node type. Further nodes in R that are equivalent with the parent node type but not a morphism image are not replaced, since exactly this node type should be created. Then the IDs within the production are changed and the production is inserted in a dictionary. Again all combinations of replacements are identified and further productions generated.

Afterwards, all attributes as well as the dependency and inheritance relationships of the parent node type are copied for the child node type and the inheritance relationship between the two of them is deleted from the type graph (abstract closure). Then the duplicate productions within the dictionary are removed and the remaining productions are inserted in the XML-structure.

Depending on the user's choice, the abstract node types together with adjacent edges as well as all productions and ACs with abstract nodes are deleted afterwards (concrete closure). Finally, the XML-file with the flattened hierarchy is saved and can be opened with AGG.

6 Evaluation

The flattening prototype is evaluated within a case study in which the graph transformation system DeonticBpmnGTS is flattened. DeonticBpmnGTS defines a type graph (see Fig. 1) as well as 18 productions with 27 application conditions (6 PACs and 21 NACs). The abstract flattening of DeonticBpmnGTS with the flattening prototype results in 1.471 productions and 24.942 ACs (414 GACs and 24.528 NACs). All inheritance relationships have been removed from the type graph and the dependency relationships and attributes have been copied as shown in Fig. 5, e.g., *P(Task)&O(Task|Precondition)* now provides the attribute *OPrecondition* by itself. Furthermore, the concrete flattening of DeonticBpmnGTS leads to 822 productions and 10.170 ACs (234 GACs and 9.936 NACs). In this case, also the abstract nodes have been removed from the type graph as shown in Fig. 6 and all productions and ACs with abstract nodes have been deleted.

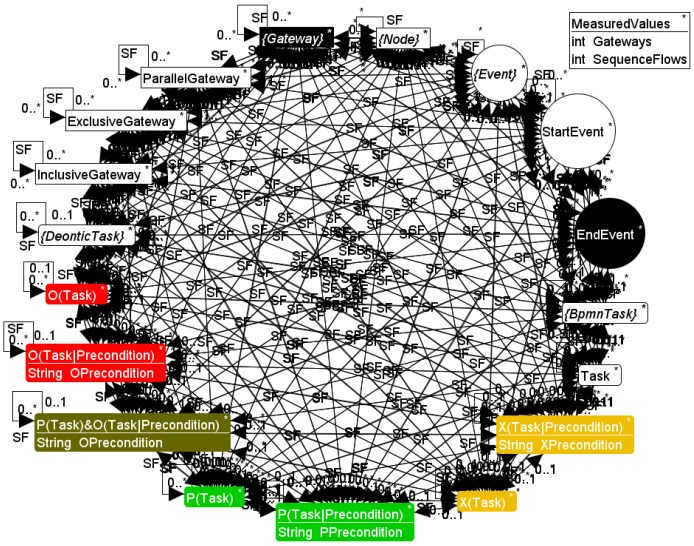


Figure 5: Abstract Closure of Type Graph

The flattening of DeonticBpmnGTS resulted in a large number of productions and ACs. So we also calculated the number of flattened productions and ACs manually as shown in Tab. 1. For example, the production *ParallelWithPhiDualRule* comprises two abstract *Nodes*. Every *Node* can be replaced by 18 node types of which 13 are concrete. Thus, $18 \times 18 = 324$ abstract and $13 \times 13 = 169$ concrete productions are generated. Furthermore, every node in an AC that has no morphism image in L leads to additional ACs that must then be multiplied with the number of productions. For example, the production *ParallelWithPhiDualRule* includes the NAC *NoFurtherNode* which comprises one abstract *Node* without morphism image in L. Thus, the flattening of the NAC leads to $18 \times 324 = 5.832$ abstract and $13 \times 169 = 2.197$ concrete NACs.

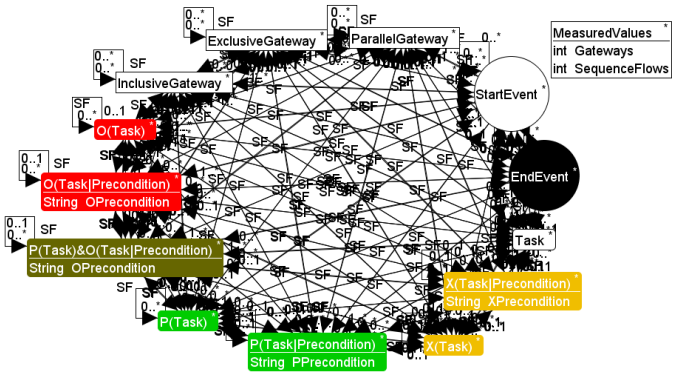


Figure 6: Concrete Closure of Type Graph

The correspondence of the flattening results and the calculated numbers allows us to assume a correct flattening. Duplicate productions and ACs can be ruled out due to the algorithm (see methods *RemoveDuplicateRules* and *RemoveDuplicateACs*) and a manual check.

Considering, for example, the rule *SeveralPhiReductionRule*, the flattening leads to nine concrete rules in which the two abstract *Gateways* are replaced in any combination. The left-hand side of the nine generated rules without the *MeasuredValues* element is shown in Fig. 7.

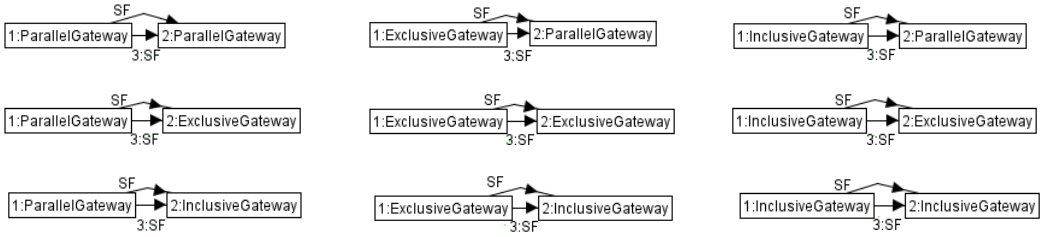


Figure 7: Flattened SeveralPhiReductionRule (LHS)

However, the flattening of DeonticBpmnGTS reveals a minor open issue concerning restricted dependency relationships of subtypes that has neither been considered in the original definition nor in our extension. If the flattening of the type graph results in two dependency relationships of the same type being specified between a source and target node, then the restricted dependency relationship of the subtype is taken. However, during the flattening of productions, every node is replaced by all subnodes without considering restricted multiplicities of dependency relationships. Thus, the flattening may provide invalid productions and ACs.

For example, the abstract vertex *Node* may have several incoming and outgoing sequence flows which is necessary for *Gateways*, but some subtypes are restricted to at most one incoming or one outgoing sequence flow. Thus, a *Node* in a production with two incoming sequence flows should not be replaced by all subtypes. The invalid productions and ACs are highlighted by the tool AGG and can easily be disabled or removed. An adaptation of the definitions and the implementation of a validator within the flattening prototype is complex and left for further work.

In general, it can be said that invalid ACs do not affect the termination analysis, since invalid

Table 1: Flattening of DeonticBpmnGTS: Calculation

Rule	Abstract	Concrete	AC	Abstract	Concrete
1	16	9			
2	1	1	NAC	1	1
			NAC	18	13
			PAC	324	169
3	18	13			
4	36	26			
5	36	26	NAC	648	338
6	324	169	NAC	5.832	2.197
7	1	1	PAC	18	13
8	9	9	NAC	9	9
9	324	169	NAC	5.832	2.197
10	1	1	PAC	18	13
11	9	9	NAC	9	9
12	1	1	NAC	1	1
			NAC	324	169
			NAC	324	169
			PAC	18	13
13	324	169	NAC	5.832	2.197
14	1	1	PAC	18	13
15	9	9	NAC	9	9
16	1	1	NAC	1	1
			NAC	324	169
			NAC	324	169
			PAC	18	13
17	72	26	NAC	288	78
			NAC	576	182
			NAC	144	26
18	288	182	NAC	1.152	546
			NAC	2.304	1.274
			NAC	576	182
	1.471	822		24.942	10.170

Rule Numbers:

[1] SeveralPhiReductionRule
[2] IterationRepeatUntilRule
[3] SequenceRuleBase
[4] SequenceRuleExtended
[5] SequenceRuleFinish
[6] ParallelWithPhiDualRule
[7] ParallelRule
[8] ParallelRuleFinish
[9] ExclusiveWithPhiDualRule
[10] ExclusiveWithPhiRule
[11] ExclusiveWithPhiRuleFinish
[12] ExclusiveWithoutPhiRule
[13] InclusiveWithPhiDualRule
[14] InclusiveWithPhiRule
[15] InclusiveWithPhiRuleFinish
[16] InclusiveWithoutPhiRule
[17] SequenceBpmnRulePragmatic
[18] SequenceDeonticRulePragmatic

NACs are redundant and PACs are not used for the termination proof. Concerning the flattening of productions, three cases are distinguished. If (I) a node is created and, thus, only part of the RHS, then this node is not flattened. Hence, no invalid productions may emerge in this case. If (II) the node is defined on the LHS and has a morphism image on the RHS, then the node is neither created nor deleted. An invalid production may emerge but does not affect the termination analysis. If (III), a node is defined on the LHS but not on the RHS, then this node is deleted. In this case, invalid productions that affect the termination analysis may occur.

In DeonticBpmnGTS, some invalid NACs are produced and highlighted by the tool AGG. However, NACs specify graphs that must not be part of the match in graph G. When the replacement has also been forbidden by restricted dependency relationships, then this definition is redundant. Furthermore, NACs are only necessary for the termination analysis of nondeletion layers. Since all layers in DeonticBpmnGTS are classified as deletion layers, the termination analysis of DeonticBpmnGTS is not affected by invalid NACs.

After flattening DeonticBpmnGTS, the resulting file is loaded with the tool AGG and the termination analysis (TA) is executed. Since every production of DeonticBpmnGTS deletes at least one node or edge, all layers are classified as deletion layer. All in all, AGG classifies DeonticBpmnGTS as terminating and calculates reasonable creation and deletion layers for every node and edge type. Thus, DeonticBpmnGTS can be called a trusted model transformation.

7 Conclusion

In this paper, we presented a flattening approach for attributed type graphs with inheritance. We, therefore, extended the current definitions and proved the semantic equivalence of the original and the flattened GTS. Afterwards, we developed a flattening prototype and evaluated it within a case study.

Further goals are to reduce the size of the flattened graph transformation system by deleting NACs with a finer node type (case 2) and productions with a contradictory PAC. In addition, restricted dependency relationships of subtypes must be considered.

Acknowledgements: The project *Vertical Model Integration* is supported within the program "Regionale Wettbewerbsfähigkeit OÖ 2007-2013" by the European Fund for Regional Development as well as the State of Upper Austria. In addition, we want to thank the anonymous reviewers for their constructive comments and suggestions.

Bibliography

[AGG11] AGG. AGG Homepage. http://user.cs.tu-berlin.de/~gragra/agg/, 2011. Last visited: Oct 2011.

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

[EPS73] H. Ehrig, M. Pfender, H. J. Schneider. Graph-Grammars: An Algebraic Approach. In *Proceedings of FOCS 1973*. Pp. 167–180. IEEE, 1973.

[GLEO12] U. Golas, L. Lambers, H. Ehrig, F. Orejas. Attributed graph transformation with inheritance: Efficient conflict detection and local confluence analysis using abstract critical pairs. *Theoretical Computer Science* 424:46 – 68, 2012.

[LB93] M. Löwe, M. Beyer. AGG – An Implementation of Algebraic Graph Rewriting. In *Rewriting Techniques and Applications*. Pp. 451–456. 1993.

[LBE+07] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. Attributed graph transformation with node type inheritance. *Theor. Comput. Sci.* 376:139–163, May 2007.

[Nat11] C. Natschläger. Deontic BPMN. In Hameurlain et al. (eds.), *Database and Expert Systems Applications*. Lecture Notes in Computer Science 6861, pp. 264–278. Springer Berlin / Heidelberg, 2011.

[Tae04] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In Pfaltz et al. (eds.), *In: Applications of Graph Transformations with Industrial Relevance (AGTIVE)*. Lecture Notes in Computer Science 3062, pp. 446–453. Springer Berlin / Heidelberg, 2004.

[TR05] G. Taentzer, A. Rensink. Ensuring Structural Constraints in Graph-Based Models with Type Inheritance. In Cerioli (ed.), *Fundamental Approaches to Software Engineering*. Lecture Notes in Computer Science 3442, pp. 64–79. Springer Berlin / Heidelberg, 2005.

[VVE+06] D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, G. Taentzer. Termination Analysis of Model Transformations by Petri Nets. In Corradini et al. (eds.), *Graph Transformations*. Lecture Notes in Computer Science 4178, pp. 260–274. Springer Berlin / Heidelberg, 2006.