An Automata-Theoretic Approach to Instance Generation *

Jan Steffen Becker

Universität Oldenburg
jan.steffen.becker@informatik.uni-oldenburg.de

Abstract. In this paper, we elaborate on a filter problem for graph grammars. The problem is to find, for some graph grammar and some constraint, all graphs generated by the grammar that satisfy the constraint. We show that there is, for every graph grammar and nested graph constraint, a derived graph grammar that solves the filter problem. However, the general solution is not well-suited for most practical applications. Because it is undecidable for arbitrary constraints whether there is some graph satisfying the constraint, there will be no efficient solution for the filter problem in the general case. We present an automata-based approach to solve a weaker form of the filter problem on labeled directed graphs that seems to be sufficient for practical applications. The filtering graph grammars derived this way are well-suited for instance generation since they ensure termination and can be implemented without the need of backtracking.

Keywords: finite automaton, graph grammar, instance generation, meta-modeling

1 Introduction

Generating instances from meta-models is an important task in meta-modeling. For this graph grammars [8] are well-suited, because they provide constructive means of manipulating graphs. In general, a meta-model consists of a type graph [6], usually specified in UML, and a set of constraints specified in an arbitrary constraint language (e.g. OCL) that are separated from the type graph. There are already approaches to translate the type graph into a graph grammar or similar construct for generating instances [17]. However, these approaches are very restricted since they do not take additional constraints beyond multiplicities into account. The challenge is to integrate the additional constraints into such a graph grammar derived with existing methods in order to derive a modified grammar that generates only those instances that satisfy the additional constraints.

* This work is partly supported by the German Research Foundation (DFG), Grants HA 2936/4-1 and TA 2941/3-1 (Meta modeling and graph grammars: integration of two paradigms for the definition of visual modeling languages).

In this work, we abstract from the meta-modeling case and lift the problem to the general case that we call the *(strong) filter problem*: Given a graph grammar GG and a constraint c, construct a graph grammar GG_1 such that[1] $\mathcal{L}(GG_1) = \mathcal{L}(GG) \cap [\![c]\!]$. We use the term *complete* to denote that a grammar solves the strong filter problem. For labeled directed graphs, this problem has a quite simple solution that we show for completeness sake in Section 3 in detail. But this solution requires a lot of backtracking and is therefore not considered efficient. So we weaken the problem to the *weak filter problem*: Given a graph grammar $GG = \langle N, S, \mathcal{R} \rangle$ and a constraint c, construct a graph grammar GG_2 such that $\{G \in \mathcal{L}(GG) \mid S \Rightarrow_{\mathcal{R}} G \models c\} \subseteq \mathcal{L}(GG_2) \subseteq \mathcal{L}(GG) \cap [\![c]\!]$ (i.e. GG_2 constructs at least those graphs satisfying c that are generated with one derivation step). There is also a simple solution for this problem that is efficient but only generates the empty language in many cases. In this paper, we enhance the solution for the weak filter problem to generate larger subsets of $\mathcal{L}(GG) \cap [\![c]\!]$ but remain efficient.

In this paper, we present our solution for labeled directed graphs whereas the core constructions may be generalized to similar structures, e.g. typed attributed graphs, as well. The paper is structured as follows. In Section 2, we formally introduce graph grammars and nested graph constraints over labeled directed graphs. In Section 3, we present our solution to the filter problem for grammars and constraints on labeled graphs. In Section 4, we show how the solution can be adapted for typed graphs to solve the Petri net use case presented in [14]. In Section 5, we compare to related work and conclude with a short summary and outlook in Section 6.

2 Preliminaries

In this paper, we consider labeled directed graphs in the sense of Ehrig et al. [6]. In the following we recall the formal definitions of nested graph conditions and graph grammars [8, 6].

Graph conditions are nested constructs which can be represented as trees of morphisms equipped with quantifiers and Boolean connectives.

Definition 1 (nested graph conditions). A *(nested) graph condition*, short *condition*, over a graph P is of the form *true* or $\exists(a, c)$ where $a\colon P \to C$ is a morphism and c is a condition over C. Boolean formulas over conditions over P yield conditions over P, that is, for conditions c, c_i $(i \in I)$ over P, $\neg c$ and $\bigwedge_{i \in I} c_i$ are conditions over P. In the context of rules, conditions are called *application conditions*. Conditions over the empty graph $\emptyset$ are called *constraints*.

Satisfaction of a condition by a morphism is inductively defined as follows: Every morphism satisfies *true*. A morphism $p\colon P \to G$ satisfies $\exists(a, c)$ over P with $a\colon P \to C$ if there exists an injective morphism $q\colon C \hookrightarrow G$ such that $q \circ a = p$ and q satisfies c (see Figure 1, left). A morphism $p\colon P \to G$ satisfies $\neg c$

[1] $\mathcal{L}(GG)$ denotes the set of all graphs generated by graph grammar GG; $[\![c]\!]$ denotes the set of all graphs satisfying constraint c (see Section 2)

over P if p does not satisfy c, and p satisfies $\bigwedge_{i\in I} c_i$ over P if p satisfies each c_i ($i \in I$). We write $p \models c$ if $p\colon P \to G$ satisfies the condition c (over P). A graph G satisfies a constraint c, short $G \models c$, if the morphism $p\colon \emptyset \hookrightarrow G$ satisfies c. The set of all graphs satisfying c is denoted by $[\![c]\!]$.

We restrict ourselves to finite conditions, i.e. all index sets I are finite. Conditions can be written in a more compact form by writing only the co-domain of morphisms, e.g. $\exists(C, \exists(D))$ instead of $\exists(\emptyset \hookrightarrow C, \exists(C \hookrightarrow D, \mathit{true}))$. We write $\forall(a, c)$ instead of $\neg\exists(a, \neg c)$ and merge $\neg\exists$ to $\nexists$.

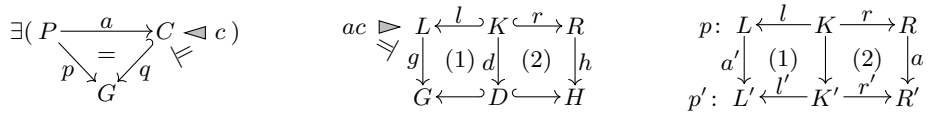


Fig. 1. Commuting Diagrams for Definitions 1, 2 and Construction 1.

Definition 2 (rules and transformations). A *rule* $\rho = \langle p, \mathrm{ac}\rangle$ consists of a *plain rule* $p = \langle L \leftarrow K \rightarrow R\rangle$ with injective morphisms $l\colon K \to L$ and $r\colon K \to R$ and an application condition ac over L. A *direct transformation* from a graph G to a graph H applying rule ρ at morphism g consists of two pushouts (1) and (2) as in Figure 1 (middle) where $g \models ac$. We write $G \Rightarrow_\rho H$ if there exists such a direct transformation. We say ρ is applicable on G if a graph H with $G \Rightarrow_\rho H$ exists. For words $w = \rho_1 \dots \rho_k \in \mathcal{R}^*$ over some set of rules $\mathcal{R}$ we write $G \Rightarrow_w H$ if there is a chain of derivations $G \Rightarrow_{\rho_1} \dots \Rightarrow_{\rho_k} H$. We write $G \Rightarrow_{\mathcal{R}} H$ ($G \Rightarrow^k_{\mathcal{R}} H$, $G \Rightarrow^*_{\mathcal{R}} H$) if there is some $\rho \in \mathcal{R}$ ($w \in \mathcal{R}^*$ of length k, of arbitrary length) s.t. $G \Rightarrow_\rho H$ ($G \Rightarrow_w H$). In this paper, ε denotes the empty (plain) rule $\langle \emptyset \leftarrow \emptyset \rightarrow \emptyset\rangle$.

Rules may be collected to graph grammars [2, 7].

Definition 3 (graph grammar). A graph grammar $GG = \langle N, \mathcal{R}, S\rangle$ over directed graphs that are labeled over some finite alphabet Lab, consists of a set $N \subseteq Lab$ of node and edge labels called *non-terminals*, a finite set $\mathcal{R}$ of rules and a start graph S. The labels in $Lab - N$ are called terminals. The *graph language* generated by GG consists of all terminal graphs derivable from S by $\mathcal{R}$: $\mathcal{L}(\mathcal{G}) = \{G \mid S \Rightarrow^*_{\mathcal{R}} G \wedge G \text{ is terminal}\}$, where a graph is called terminal, if it does not contain items with a label from N.

In the following, we consider weakest preconditions of rules according to nested graph constraints:

Definition 4 (guaranteeing rule, weakest precondition). For a rule ρ and a constraint c, the constraint $\mathrm{wp}(\rho, c)$ denotes the *weakest precondition* for ρ with respect to c, i.e. $G \models \mathrm{wp}(\rho, c)$ if there is some $H \models c$ with $G \Rightarrow_\rho H$ and for all H, $G \Rightarrow_\rho H$ implies $H \models c$. A rule ρ *guarantees* a constraint c, if c is a

postcondition to ρ, i.e. $H \models c$ for all $G \Rightarrow_\rho H$. A rule ρ' is the *c-guaranteeing rule* to some rule ρ, if $G \Rightarrow_{\rho'} H$ if $G \Rightarrow_\rho H$ and $H \models c$.

Guaranteeing rules and weakest preconditions can be effectively constructed.

Fact 1 (Gua **and** Wp **[8]).** There are a transformations Gua and Wp such that, for every rule ρ and every constraint c, $\mathrm{Gua}(\rho, c)$ is the c-guaranteeing rule for ρ and $\mathrm{Wp}(\rho, c)$ is a weakest precondition of ρ relative to c.

Construction 1 ([8, 7]). For some rule $\rho = \langle p, ac\rangle$ with plain rule $p = \langle L \xleftarrow{l} K \xrightarrow{r} R\rangle$ we have $\mathrm{Gua}(\rho, c) = \langle p, \mathrm{L}(p, \mathrm{A}(p, c)) \wedge ac\rangle$ and $\mathrm{Wp}(\rho, c) = \forall(L, (ac \wedge \mathrm{Appl}(\rho)) \Rightarrow \mathrm{L}(\rho, \mathrm{A}(\rho, c))) \wedge \exists(L, ac \wedge \mathrm{Appl}(\rho))$ and $\mathrm{A}(p, c) = \mathrm{Shift}(\emptyset \hookrightarrow R, c)$, $\mathrm{Appl}(\rho) = \bigwedge_{a\in\mathcal{A}} \nexists a$ where $\mathcal{A}$ ranges over all injective morphisms $a\colon L \to L'$ s.t. (a, l) has no pushout complement and there is no decomposition $a = a_2 \circ a_1$ s.t. a_2 is not an isomorphism and (a_1, l) has no pushout complement. Note that $\mathcal{A}$ is finite for labeled directed graphs.

The constructions Shift and L are inductively defined: $\mathrm{Shift}(b, \mathit{true}) = \mathit{true}$, $\mathrm{Shift}(b, \exists(a, c)) = \bigvee_{(a',b')\in\mathcal{F}} \exists(a', \mathrm{Shift}(b', c))$ with $\mathcal{F}$ the set of all jointly surjective pairs (a', b') of injective morphisms such that $a' \circ b = b' \circ a$. $\mathrm{Shift}(b, \neg c) = \neg\,\mathrm{Shift}(b, c)$, $\mathrm{Shift}(b, \bigwedge_{i\in I} c_i) = \bigwedge_{i\in I} \mathrm{Shift}(b, c_i)$.

$\mathrm{L}(p, \mathit{true}) = \mathit{true}$, $\mathrm{L}(p, \exists(a, c)) = \exists(a', \mathrm{L}(p', c))$ if pushouts (1) and (2) as in Figure 1 (right) exist, *false* otherwise. $\mathrm{L}(p, \neg c) = \neg\mathrm{L}(p, c)$, $\mathrm{L}(p, \bigwedge_{i\in I} c_i) = \bigwedge_{i\in I} \mathrm{L}(p, c_i)$.

Note, that $G \models \mathrm{Wp}(\rho, \mathit{true})$ if, and only if, ρ is applicable on G. Instead of $\mathrm{Wp}(\rho, \mathit{true})$ we write $\mathrm{Wp}(\rho)$ for short. For finite conditions, the constructions Wp and Gua yield finite (application) conditions.

3 Filter Theorems

In this section, we consider filter problems for graph grammars and nested graph constraints.

Theorem 1 (filtering). There are constructions `SFilter` and `WFilter` such that, for arbitrary graph grammars GG over labeled directed graphs and nested graph constraints c, $GG_1 = \texttt{SFilter}(GG, c)$ and $GG_2 = \texttt{WFilter}(GG, c)$ are graph grammars with $\mathcal{L}(GG_1) = \mathcal{L}(GG) \cap [\![c]\!]$ resp. $\mathcal{L}(GG_2) \subseteq \mathcal{L}(GG) \cap [\![c]\!]$.

The names `WFilter` and `SFilter` stand for *weak* and *strong filtering*, respectively. The constructions are based on the integration of condition into rules (Fact 1). While the first construction yields a graph grammar that generates a subset of the desired language, the second construction generates the desired language: In the last step, it checks, whether the derived graph satisfies the considered condition. We add a "state" node with non-terminal label, extend the rules of $\mathcal{R}$ by the node, and add a rule that deletes the state node whenever the graph satisfies the condition.

Construction 2. For a graph grammar $GG = \langle N, \mathcal{R}, S\rangle$, and a nested graph constraint c, let $\texttt{SFilter}(GG, c) = GG_1$, $\texttt{WFilter}(GG, c) = GG_2$ where $GG_i = \langle N \uplus \{q\}, \mathcal{R}_i \cup \{\mathrm{Gua}(\texttt{delete}, c)\}, S{+}\textcircled{q}\rangle$ $(i \in \{1, 2\})$ are the graph grammars with $\mathcal{R}_1$ consists of the rules of $\mathcal{R}$, extended by a q-labeled node and $\texttt{delete} = \langle \textcircled{q} \hookleftarrow \emptyset \hookrightarrow \emptyset\rangle$, $\mathcal{R}_2 = \{\mathrm{Gua}(\rho, c) \mid \rho \in \mathcal{R}\}$.

Proof. By Fact 1 and Definition 4, for each derivation $S \Rightarrow^*_{\mathcal{R}} G$ there is a derivation $S' = S{+}\textcircled{q} \Rightarrow^*_{\mathcal{R}_2} G{+}\textcircled{q}$ and $G{+}\textcircled{q} \Rightarrow_{\mathrm{Gua}(\texttt{delete}, c)} G$ iff $G \models c$. Therefor $\mathcal{L}(GG_1) = \mathcal{L}(GG) \cap [\![c]\!]$. For all rules $\rho \in \mathcal{R}$, $G \Rightarrow_{\mathrm{Gua}(\rho, c)} H$ implies $G \Rightarrow_\rho H$. Thus, $\mathcal{L}(GG_2) \subseteq \mathcal{L}(GG)$. Since all rules in $\mathcal{R}_c$ are c-guaranteeing, $\mathcal{L}(GG_2) \subseteq \mathcal{L}(GG) \cap [\![c]\!]$. □

In the following examples we assume some implicit label alphabet consisting of the used non-terminals and an empty label that is not drawn and always terminal.

Example 1. The graph grammar GG with the start graph $S = \bullet$ and the rules p_1: $\underset{1}{\bullet} \Rightarrow \underset{1}{\bullet}\ \ \underset{2}{\bullet}$, p_2: $\underset{1}{\bullet}\ \ \underset{2}{\bullet} \Rightarrow \underset{1}{\bullet}\!\!\rightarrow\!\!\underset{2}{\bullet}$ generates the set of all loop-free graphs with at least one node. For the constraint $c = \exists \bullet \rightarrow \bullet$, $L(GG) \cap [\![c]\!]$ consists of all loop-free graphs with at least two nodes and an edge with distinct source and target.

The graph grammar $GG_1 = \texttt{SFilter}(GG, c)$ with the start graph $\textcircled{q}\bullet$ and the rules p_1': $\underset{1}{\bullet}\textcircled{q} \Rightarrow \underset{1}{\bullet}\ \ \underset{2}{\bullet}\textcircled{q}$, p_2': $\underset{1}{\bullet}\ \ \underset{2}{\bullet}\textcircled{q} \Rightarrow \underset{1}{\bullet}\!\!\rightarrow\!\!\underset{2}{\bullet}\textcircled{q}$, and $\texttt{delete}'$: $\langle \textcircled{q} \Rightarrow \emptyset, \exists(\textcircled{q}\bullet\rightarrow\bullet)\rangle$ allows to generate all loop-free graphs with at least two nodes and an edge with distinct source and target, i.e. $\mathcal{L}(GG_1) = \mathcal{L}(GG) \cap [\![c]\!]$. The derivation is done as usual. At some point, we check whether $\textcircled{q}\bullet\rightarrow\bullet$ exists, and if so, we delete the q-labeled node.

The graph grammar $GG_2 = \texttt{WFilter}(GG, c)$ with the start graph $\textcircled{q}\bullet$ consists of the rules $p_1' = \langle p_1, \exists(\underset{1}{\bullet}\rightarrow\bullet) \vee \exists(\bullet\rightarrow\underset{1}{\bullet}) \vee \exists(\underset{1}{\bullet}\ \bullet\rightarrow\bullet)\rangle$, $p_2' = \langle p_2, \mathit{true}\rangle \equiv p_2$ and $\texttt{delete}'$: $\langle \textcircled{q} \Rightarrow \emptyset, \exists(\textcircled{q}\bullet\rightarrow\bullet)\rangle$. If the start graph of GG consists of a single node, no rule can be applied, i.e. $\mathcal{L}(GG_2) = \emptyset \neq \mathcal{L}(GG) \cap [\![c]\!]$.

Both grammars derived via `WFilter` and `SFilter` have disadvantages and we consider them not to behave well in practice. Usually, the output from `WFilter` can generate only a small subset of $\mathcal{L}(GG) \cap [\![c]\!]$ and make some important rules not applicable. On the other hand, the result of `SFilter` relies on generate-and-test too much to be considered efficient. We are interested in constructing grammars that are better suited for instance generation. We call them *goal oriented*.

Definition 5 (goal-oriented graph grammar). A graph grammar $GG = \langle N, \mathcal{R}, S\rangle$ is *goal-oriented* if there is a terminating[2] set $\mathcal{R}_t \subseteq \mathcal{R}$ of rules, s.t. for all graphs G with[3] $S \Rightarrow^+_{\mathcal{R}} G$ there exists some $\rho \in \mathcal{R}_t$ that is applicable on G, or $G \in \mathcal{L}(GG)$.

[2] A set of rules is terminating if there is no infinite derivation.

[3] $G \Rightarrow^+ H$ means a derivation with at least one step, i.e. $G \Rightarrow^+ H$ if $\exists G'.\ G \Rightarrow G' \Rightarrow^* H$.

Example 2 (goal-oriented graph grammar). The graph grammar $GG = \langle \{A, B\}, \mathcal{R}, \textcircled{A} \rangle$ with

$$\mathcal{R} = \{\textcircled{A} \Rightarrow \textcircled{A}\ \bullet,\ \textcircled{B}\ \bullet \Rightarrow \textcircled{B}\ \bullet\!\circlearrowright,\ \langle \textcircled{A} \leftarrow \emptyset \rightarrow \textcircled{B} \rangle,\ \langle \textcircled{B} \leftarrow \emptyset \rightarrow \emptyset \rangle\}$$

and $\mathcal{R}_t = \{\langle \textcircled{A} \leftarrow \emptyset \rightarrow \textcircled{B} \rangle,\ \langle \textcircled{B} \leftarrow \emptyset \rightarrow \emptyset \rangle\}$ is goal-oriented.

The benefit from using a goal-oriented graph grammar is that an implementation does not need to use backtracking. When using a goal-oriented graph grammar for instance generation, it is very easy to guarantee that the generation process terminates. A possible implementation could use the complete set $\mathcal{R}$ of rules until some abortion criterion is reached, e.g. a maximum number of derivation steps or graph size. Then the implementation switches to the reduced set $\mathcal{R}_t$ of rules. They guarantee that the process terminates with a valid graph.

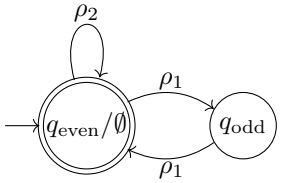
We will use some kind of finite automata as an intermediate structure. These so-called *rule-automata* can – in case of labeled directed graphs – be transformed to graph grammars.

Definition 6 (rule automaton). A *non-deterministic finite automaton (NFA)* [9] over an alphabet Σ is a tuple $NFA = \langle Q, \Sigma, \delta, q_0, F \rangle$ with a finite set Q of states, an initial state $q_0 \in Q$, a set $F \subseteq Q$ of final states, and transition relation $\delta \colon Q \times \Sigma \to 2^Q$. The language of the NFA is defined as $\mathcal{L}(NFA) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$ where $\hat{\delta} : Q \times \Sigma^* \to 2^Q$ is inductively defined as $\hat{\delta}(q, \varepsilon) = \{q\}$ and $\hat{\delta}(q, wa) = \bigcup_{q' \in \hat{\delta}(q,w)} \delta(q', a)$ for $w \in \Sigma^*$, $a \in \Sigma$.

A *rule automaton* over a set $\mathcal{R}$ of rules is a pair $A = \langle NFA, S \rangle$ where NFA is a NFA over $\mathcal{R}$ and S a graph, called *start graph*. The *graph language* of a rule automaton is defined as $\mathcal{L}(A) = \{G \mid \exists w \in \mathcal{L}(NFA).\ S \Rightarrow_w G\}$.

We draw rule automata in the usual notation for finite automata. As an extension we label the initial state q_0 by q_0/S where S is the automaton's start graph.

Example 3 (even number of nodes). The rule automaton on the right with rules $\rho_1 \colon \emptyset \Rightarrow \bullet$ and $\rho_2 \colon \bullet\ \bullet \Rightarrow \bullet\!\rightarrow\!\bullet$ creates all loop-free graphs with an even number of nodes.



Lemma 1 (from rule automata to graph grammars). There is a construction `Grammar` that creates, for every rule automaton A over labeled directed graphs, a graph grammar $\texttt{Grammar}(A)$ such that $\mathcal{L}(\texttt{Grammar}(A)) = \mathcal{L}(A)$.

In the following, we interpret the transition function δ of an automaton as relation $\delta \subseteq Q \times \mathcal{R} \times Q$, i.e. we say $\langle q_1, \rho, q_2 \rangle \in \delta$ if $q_2 \in \delta(q_1, \rho)$.

Construction 3 (from automata to grammars). For some rule automaton A and start graph S let $\texttt{Grammar}(A) = \langle Q, \mathcal{R}', S + \textcircled{q_0} \rangle$ with

$$\mathcal{R}' = \{\langle p', ac' \rangle \mid \langle q_1, \langle p, ac \rangle, q_2 \rangle \in \delta\} \cup \{\langle \textcircled{q_f} \leftarrow \emptyset \rightarrow \emptyset \rangle \mid q_f \in F\}$$

where $p' = \langle L{+}\textcircled{q_1} \leftarrow I \rightarrow R{+}\textcircled{q_2}\rangle$ for $p = \langle L \leftarrow I \rightarrow R\rangle$ and $ac' = \text{Shift}(L \hookrightarrow L{+}\textcircled{q}, ac)$.

Proof. We show by induction over $k \in \mathbb{N}$ that, for all graphs G, H and states $q_1, q_2 \in Q$, there is some word $w \in \mathcal{R}^k$ with $G \Rightarrow_w H$ and $q_2 \in \hat{\delta}(q_1, w)$ iff $G{+}\textcircled{q_1} \Rightarrow^k_{\mathcal{R}'} H{+}\textcircled{q_2}$. This is trivial for $k = 0$. Assume we have shown this for some arbitrary but fixed k.

$$\begin{aligned}
& G{+}\textcircled{q_1} \Rightarrow^{k+1}_{\mathcal{R}'} H{+}\textcircled{q_2} \\
\iff & \exists q \in Q, K.G{+}\textcircled{q_1} \Rightarrow_{\mathcal{R}'} K{+}\textcircled{q} \Rightarrow^k_{\mathcal{R}'} H{+}\textcircled{q_2} \\
\iff & \exists w \in \mathcal{R}^k, \rho \in \mathcal{R}'.G{+}\textcircled{q_1} \Rightarrow_\rho K{+}\textcircled{q} \Rightarrow_w H{+}\textcircled{q_2} \wedge q_2 \in \hat{\delta}(q, w) \\
\iff & \exists \rho \in \mathcal{R}'.\ G{+}\textcircled{q_1} \Rightarrow_{\rho w} H{+}\textcircled{q_2} \wedge q_2 \in \hat{\delta}(q_1, \rho w)
\end{aligned}$$

Since $H{+}\textcircled{q_2} \Rightarrow_{\mathcal{R}'} H$ iff $q_2 \in F$ follows $H \in \mathcal{L}(\texttt{Grammar}(A))$ iff $H \in \mathcal{L}(A)$. □

Our approach to solve the filter problem is an enhancement of `WFilter`. Instead of requiring that the constraint c is satisfied immediately after every derivation step, we allow the intermediate graphs to be up to k derivation steps "away" from a graph that satisfies c. As a consequence, the resulting graph grammar generates at least all graphs satisfying c that can be derived from the start graph within k steps.

Theorem 2 (weak integration). There are constructions $\texttt{Integrate}^k$, $k \in \mathbb{N}$ such that, for every graph grammar $GG = \langle N, \mathcal{R}, S\rangle$ over labeled directed graphs, constraint c and natural number $k \in \mathbb{N}$, $GG_c = \texttt{Integrate}^k(GG, c)$ is a a goal-oriented graph grammar with

$$\{G \in \mathcal{L}(GG) \cap [\![c]\!] \mid S \Rightarrow^\ell_{\mathcal{R}} G, \ell \leq k\} \subseteq \mathcal{L}(GG_c) \subseteq \mathcal{L}(GG) \cap [\![c]\!]$$

We construct these grammars via Construction 3 (`Grammar`) from rule automata. In order to cover also graph grammars with non-terminals we have to encode that a graph is terminal.

Lemma 2. For every set N of node and edge labels exists a nested constraint $\texttt{terminal}_N$ such that every labeled graph $G \models \texttt{terminal}_N$ if, and only if, G is terminal wrt. N.

Proof. Define

$$\texttt{terminal}_N = \bigwedge\nolimits_{x \in N_V} \nexists(\textcircled{x}) \wedge \bigwedge\nolimits_{y \in N_E} (\nexists(\bullet \xrightarrow{y} \bullet) \wedge \nexists(\bullet \circlearrowright y))$$

with $N = N_V \cup N_E$, where N_V and N_E are the alphabets of non-terminal node and edge labels. Then $G \models \texttt{terminal}$ iff G has no node with label in N_V and edge (or loop) with label in N_E. □

Construction 4 ($\texttt{Integrate}^k$). Let $GG = \langle N, \mathcal{R}, S\rangle$ be a graph grammar and c a constraint. For $k = 0$ let $\texttt{Integrate}^0 = \langle N \uplus \{q\}, \{\langle \textcircled{q} \Rightarrow \emptyset, c\rangle\}, S{+}\textcircled{q}\rangle$. For $k > 0$, we can construct the graph grammar $\texttt{Integrate}^k(GG, c)$ as $\texttt{Grammar}(A_c)$ from the automaton A_c constructed via one of the Algorithms 1 or 2 with maxRounds $= k$.

Input : Grammar $GG = \langle N, \mathcal{R}, S\rangle$, constraint c, maxRounds $\in \mathbb{N} \cup \{\infty\}$
Output : Automaton $A_c = \langle Q, \mathcal{R}', \delta, q_0, F, S\rangle$, terminating transition function $\delta_t \subseteq \delta$
Requires : maxRounds $= \infty \Rightarrow$ ICS of c and $\mathcal{R}$ is finite
$Q, Q' \leftarrow \{c \wedge \texttt{terminal}_N\}$; $\mathcal{R}' \leftarrow \emptyset$; $\delta, \delta_t \leftarrow \emptyset$; $j \leftarrow 0$;
Repeat
  $Q' \leftarrow Q$;
  For Each $c' \in Q'$, $\rho \in \mathcal{R}$ **Do**
    Add $\mathrm{Gua}(\rho, c')$ to $\mathcal{R}'$;
    Let $\mathcal{C}$ be a minimal set of constraints such that $\bigvee_{c'' \in \mathcal{C}} c'' \equiv \mathrm{Wp}(\mathrm{Gua}(\rho, c'))$;
    For Each $c'' \in \mathcal{C}$ **Do**
      If $\exists c''' \in Q.\, c''' \equiv c''$ **Then**
        Add $\langle c''', \mathrm{Gua}(\rho, c'), c'\rangle$ to δ;
        If $c''' \notin Q'$ **Then** Add $\langle c''', \mathrm{Gua}(\rho, c'), c'\rangle$ to δ_t;
      Else
        Add c'' to Q;
        Add $\langle c'', \mathrm{Gua}(\rho, c'), c'\rangle$ to δ and δ_t;
      EndIf
    EndFor
  EndFor
  $j \leftarrow j + 1$;
Until $Q = Q'$ **or** $j \geq$ maxRounds;
Add $\langle 0, \langle \varepsilon, c'\rangle, c'\rangle$ to δ for all $c' \in Q$;
Return $A_c := \langle Q_j \cup \{0\}, \mathcal{R}', \delta, 0, \{c\}, S\rangle$;

Algorithm 1: Creating the automaton for `Integrate1`.

For constructing the set $\mathcal{C}$ in Algorithm 1 we, bring the result of $\mathrm{Wp}(\mathrm{Gua}(\rho, c))$ into a disjunctive normal form and remove all clauses that are implied by another clause in the disjunction.

The algorithms work as follows: Both start with constraint c as a final state for the automaton and continue by iteratively creating new states from existing ones. For every state, which is also a constraint c, created in the round before and every rule $\rho \in \mathcal{R}$ we calculate the guaranteeing rule $\mathrm{Gua}(\rho, c)$ and its weakest precondition $\mathrm{Wp}(\mathrm{Gua}(\rho, c))$. Algorithm 1 decomposes the preconditions into sets of constraints, whereas Algorithm 2 composes all preconditions into one. We add these new conditions, provided no equivalent one already exists, as states and connect them to the existing ones via transitions that are labeled with the guaranteeing rules. If there was no equivalent condition before, the transition is part of δ_t (so δ_t is a spanning tree). We repeat this until no new states are found anymore or the maximum number of rounds is reached. Finally, Algorithm 1 adds a separate initial state, while Algorithm 2 merges the two last states into one with a loop and uses it as initial state.

Example 4. Take the grammar $\langle \emptyset, \{\rho_1 \colon \emptyset \Rightarrow \bullet, \rho_2 \colon \bullet \Rightarrow \bullet \circlearrowright\}, \emptyset\rangle$ and constraint $c = \forall(\bullet, \exists(\bullet \circlearrowright))$. Constructing an automaton with maxRounds $= 2$ leads to the

Input : Grammar $GG = \langle N, \mathcal{R}, S\rangle$, constraint c, maxRounds $\in \mathbb{N} \cup \{\infty\}$
Output : Automaton $A_c = \langle Q, \mathcal{R}', \delta, q_0, F, S\rangle$, terminating transition function $\delta_t \subseteq \delta$
Requires : maxRounds $= \infty \Rightarrow$ ICS of c and $\mathcal{R}$ is WQO
Initialize $Q \leftarrow \{q_0\}$, $\mathcal{R}' \leftarrow \emptyset$; $c_0 \leftarrow c \wedge \texttt{terminate}_N$; $j \leftarrow 0$;
Repeat
  $j \leftarrow j + 1$; $\delta_{last} \leftarrow \emptyset$;
  $c_j \leftarrow c_{j-1} \vee \bigvee_{\rho\in\mathcal{R}} \mathrm{Wp}(\mathrm{Gua}(\rho, c_{j-1}))$; add c_j to Q ;
  For Each $\rho \in \mathcal{R}$ **Do**
    Let $\rho' = \mathrm{Gua}(\rho, c_{j-1})$; add ρ' to $\mathcal{R}'$;
    Add $\langle c_j, \rho', c_{j-1}\rangle$ to δ, δ_t, and δ_{last};
  EndFor
  Add $\langle q_j, \langle \varepsilon, c_{j-1}\rangle, q_{j-1}\rangle$ to δ;
Until $c_j \equiv c_{j-1}$ **or** $j \geq$ *maxRounds*;
For Each $\langle c, \rho, c'\rangle \in \delta_{last}$ **Do** *// add loops to* c_{j-1}
  Add $\langle c', \rho, c'\rangle$ to δ;
EndFor
If $j \geq 2$ **Then**
  Remove c_j with all outgoing transitions;
  Let $s \leftarrow c_{j-1}$
Else $s \leftarrow c_j$;
Return $A_c = \langle Q, \mathcal{R}', \delta, s, \{c_0\}, S\rangle$, δ_t;

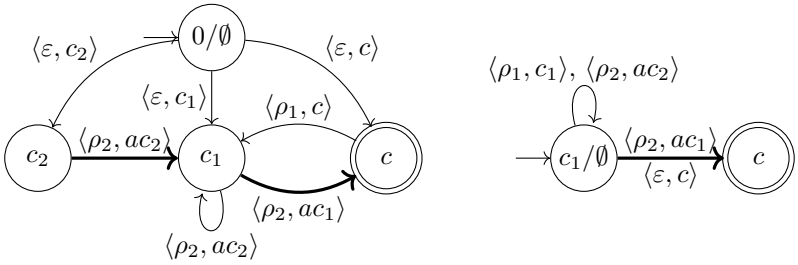
Algorithm 2: Creating the automaton for `Integrate2`.

constraints/conditions

$$c_1 = \exists(\bullet_1, \forall(\bullet\bullet_{1\,2}, \exists(\bullet\bullet_{1\,2}\circlearrowright))) \vee c \qquad c_2 = \exists(\bullet\bullet_{1\,2}, \forall(\bullet\bullet\bullet_{1\,2\,3}, \exists(\bullet\bullet\bullet_{1\,2\,3}\circlearrowright))) \vee c_1$$

$$ac_1 = \forall(\bullet_1 \hookrightarrow \bullet\bullet_{1\,2}, \exists(\bullet\bullet_{1\,2}\circlearrowright)) \qquad ac_2 = \exists(\bullet_1 \hookrightarrow \bullet\bullet_{1\,2}, \forall(\bullet\bullet\bullet_{1\,2\,3}, \exists(\bullet\bullet\bullet_{1\,2\,3}\circlearrowright))) \vee ac_1$$

and the automata for Algorithms 1 (left) and 2 (right) with start graph $\emptyset$



Proof of Theorem 2 $\{G \models c \mid S \Rightarrow^{\ell}_{\mathcal{R}} G, \ell \leq k\} \subseteq \mathcal{L}(\texttt{Integrate}^k(GG, c))$: By induction over state(s)/constraint(s) c_j introduced in round j. Let $\mathcal{R}'^j$ be $\mathcal{R}'$ in round j. For every derivation $G \Rightarrow^{j}_{\mathcal{R}} H \models c$, there is $w \in \mathcal{R}'^j$, c_j, st. $G \Rightarrow_w H$, $G \models c_j$, and the final state $c \in \hat{\delta}(c_j, w)$. This is trivial for $j = 0$. For $j + 1$ we split the derivation into $G \Rightarrow_\rho G' \Rightarrow^{j}_{\mathcal{R}} H$. By induction hypothesis we have

$w \in \mathcal{R}'^j$, c_j, st. $G' \Rightarrow_w H$, $G' \models c_j$, $c \in \hat{\delta}(c_j, w)$. By properties of Wp and Gua, $G \Rightarrow_{\mathrm{Gua}(\rho,c_j)} G' \Rightarrow_w H$ and $G \models \mathrm{Wp}(\mathrm{Gua}(\rho, c_j))$. As a consequence exists c_{j+1} st. $G \models c_{j+1}$, $c \in \hat{\delta}(c_{j+1}, \mathrm{Gua}(\rho, c_j)w)$. If $S \models c_j$ we have a word w of ε-rules st. $S \Rightarrow_w S$ and $q_j \in \hat{\delta}(s, w)$ with s the initial state. So $G \in \mathcal{L}(A_c)$ if $S \Rightarrow^{\ell}_{\mathcal{R}} G$, $\ell \leq k$.

$\mathcal{L}(\texttt{Integrate}^k(GG, c)) \subseteq \mathcal{L}(GG) \cap [\![c]\!]$: By construction every non-empty derivation $S \Rightarrow^+_{\mathcal{R}'} H$ ends with a c-guaranteeing rule, so $G \models c$ for all $G \in \mathcal{L}(\texttt{Integrate}^k(GG)) \subseteq \mathcal{L}(GG)$.

$\texttt{Integrate}^k(GG, c)$ is goal oriented: Let $\mathcal{R}'$ be the rules from $\texttt{Integrate}^k(GG, c)$ and $\mathcal{R}_t$ be the rules originating from transitions δ_t. The transitions δ_t build a spanning tree with root c on the states in A_c. Every transition to some state c_j is labeled with a c_j-guaranteeing rule. By construction, c_j is the disjunction of weakest preconditions for rules on outgoing transitions in δ_t. So at least one rule from $\mathcal{R}_t$ is applicable and we finally end in the final state. □

For systems that are closed and well quasi-ordered in the sense of [1] we can solve the strong filter problem effectively with a goal-oriented graph grammar.

Definition 7 (transition and constraint system [1]). A *transition system* $\mathcal{T} = \langle \Sigma, \rightarrow \rangle$ consists of a set Σ of states and a transition relation $\rightarrow$ on Σ. For $S \subseteq \Sigma$ let $pre(\rightarrow, S) := \{\sigma \in \Sigma \mid \exists s \in S.\ \sigma \rightarrow s\}$.

A *constraint system* $\mathcal{C}$ over Σ is a set of constraints where each $\phi \in \mathcal{C}$ characterizes a set $[\![\phi]\!] \subseteq \Sigma$ of states that satisfy ϕ. For $\phi_1, \phi_2 \in \mathcal{C}$ we define $\phi_1 \sqsubseteq \phi_2$ if $[\![\phi_2]\!] \subseteq [\![\phi_1]\!]$. A constraint system is *closed* with respect to a transition system $\mathcal{T} = \langle \Sigma, \rightarrow \rangle$, if for each $\phi \in \mathcal{C}$, one can compute $\Phi \subseteq \mathcal{C}$ such that $pre(\rightarrow, [\![\phi]\!]) = \bigcup_{\phi' \in \Phi} [\![\phi']\!]$. A constraint system is *well quasi ordered (WQO)* if $\sqsubseteq$ is a well quasi-ordering, i.e. each infinite sequence $\phi_1, \phi_2, \cdots \in \mathcal{C}$ has indices $i < j$ such that $\phi_i \sqsubseteq \phi_j$.

Note that all sets in Definition 7 are possibly infinite sets.

Definition 8 (induced transformation/constraint system). Let $\mathcal{G}$ be the set of all graphs. For a graph transformation system $\mathcal{R}$ and graph constraint c the transition system $\mathcal{T} = \langle \mathcal{G}, \Rightarrow_{\mathcal{R}} \rangle$ is the *induced transition system (ITS)* and the set $\mathcal{C}$ of constraints with $c \in \mathcal{C}$ and $\mathrm{Wp}(\mathrm{Gua}(\rho, c')) \in \mathcal{C}$ if $c' \in \mathcal{C}$ and $\rho \in \mathcal{R}$ the *induced constraint system (ICS)*.

The constraints constructed in Algorithms 1 and 2 are from the ICS. Note, that for constraints $c_1, c_2 \in \mathcal{C}$, $c_1 \sqsubseteq c_2$ if $c_2 \Rightarrow c_1$. The following fact is a direct consequence of Definitions 7 and 8.

Fact 2. For graph transformation systems $\mathcal{R}$, and constraints c, the ICS $\mathcal{C}$ is closed wrt. the ITS $\mathcal{T}$.

Theorem 3 (strong integration). There is a construction $\texttt{Integrate}^*$ s.t. for every graph grammar $GG = \langle N, \mathcal{R}, S \rangle$ on labeled directed graphs and every constraint c where the induced constraint system of $\mathcal{R}$ and c is well quasi ordered, $\texttt{Integrate}^*(GG, c)$ is a goal-oriented graph grammar with $\mathcal{L}(\texttt{Integrate}^*(GG, c)) = \mathcal{L}(GG) \cap [\![c]\!]$.

Construction 5 (`Integrate*`). We construct $\texttt{Integrate}^*(GG, c)$ as $\texttt{Grammar}(A_c)$ from the automaton A_c constructed via Algorithm 1 or – if the induced constraint system for c and $\mathcal{R}$ is finite – Algorithm 2 with maxRounds $= \infty$.

Proof. By Definition 7 every infinite chain c_1, c_2 of constraints calculated in Algorithm 1 or 2 has indices $i < j$ such that $c_j \Rightarrow c_i$. So we will reach a fixpoint Q_k, resp. c_k, of states and, by Theorem 2 $\texttt{Integrate}^\ell(GG, c) = \texttt{Integrate}^k(GG, c)$ for $\ell \geq k$. The rest follows from Theorem 2 with $\ell \to \infty$. $\square$

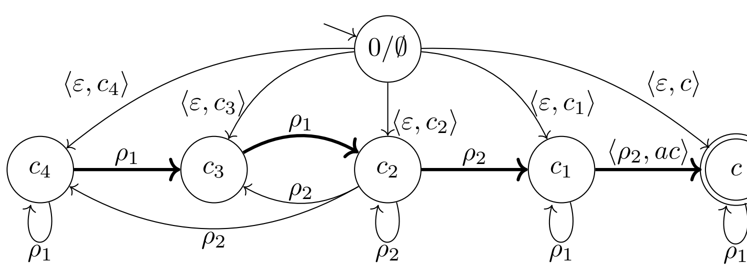
The algorithms do not reach the fixpoint $c_j \equiv c_{j-1}$ resp. $Q = Q'$ in the repeat-loop in all cases (except the induced constraint system is well quasi ordered, resp. finite). In fact it is undecidable whether the fixpoint can be reached. Also the implication problem for nested graph constraints is undecidable [8]. That why we allow to stop the generation process in an earlier state, after a maximum of maxRounds rounds. However, Theorem 2 does not depend on deciding these problems.

In the following we write `Integrate` if we do not refer to a specific version of `Integrate`. If a specific version is meant, we use `Integrate1` resp. `Integrate2` to indicate use of Algorithm 1 resp. 2. We present two examples where we apply both algorithms to simple grammars and constraints. In Example 5 we reach a fixpoint, whereas there is no fixpoint in Example 4 given earlier in this section, but the generated language is still complete.

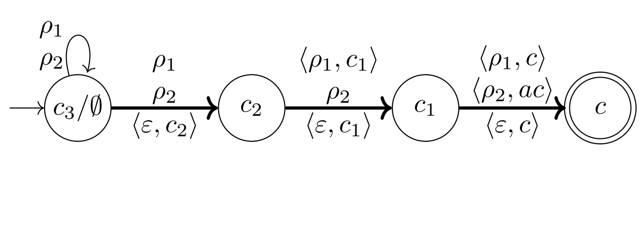
Example 5 (terminating example). Take the graph grammar with rules $\rho_1\colon \emptyset \Rightarrow \bullet$, $\rho_2\colon \underset{1}{\bullet}\ \underset{2}{\bullet} \Rightarrow \underset{1}{\bullet}\!\rightarrow\!\underset{2}{\bullet}$ and start graph $\emptyset$, and the constraint $c = \exists(\bullet\rightleftarrows\bullet)$. Using Algorithm 1 leads to the helper constraints $c_1 = \exists(\bullet\!\rightarrow\!\bullet)$, $c_2 = \exists(\bullet\ \bullet)$, $c_3 = \exists(\bullet)$, $c_4 = \mathit{true}$, the application condition

$$\begin{aligned} ac = \mathrm{L}(\rho_2, \mathrm{A}(\rho_2, c)) = {} & \exists(\underset{1}{\bullet}\ \underset{2}{\bullet} \hookrightarrow \underset{1}{\bullet}\!\rightarrow\!\underset{2}{\bullet}) \vee \exists(\underset{1}{\bullet}\ \underset{2}{\bullet} \hookrightarrow \underset{2}{\bullet}\!\rightleftarrows\!\underset{1}{\bullet}) \\ & \vee \exists(\underset{1}{\bullet}\ \underset{2}{\bullet} \hookrightarrow \underset{1}{\bullet}\ \underset{2}{\bullet}\!\rightleftarrows\!\bullet) \vee \exists(\underset{1}{\bullet}\ \underset{2}{\bullet} \hookrightarrow \underset{1}{\bullet}\ \bullet\!\rightleftarrows\!\underset{2}{\bullet}) \\ & \vee \exists(\underset{1}{\bullet}\ \underset{2}{\bullet} \hookrightarrow \underset{1}{\bullet}\!\rightleftarrows\!\bullet\ \underset{2}{\bullet}) \vee \exists(\underset{1}{\bullet}\ \underset{2}{\bullet} \hookrightarrow \bullet\!\rightleftarrows\!\underset{1}{\bullet}\ \underset{2}{\bullet}) \end{aligned}$$

and the automaton (transitions in δ_t are drawn thick)



Algorithm 2 instead leads to the automaton



The difference between the two algorithms is that we calculate in Algorithm 1 a preceding state for every rule separately, whereas in Algorithm 2 only one state is created per round. The state c_j calculated in Algorithm 2 can be seen as a collection of all the states created until round j in Algorithm 1. Note, that the resulting grammars do not create identical languages. In general $\mathcal{L}(\texttt{Integrate1}^k(GG, c)) \subseteq \mathcal{L}(\texttt{Integrate2}^k(GG, c))$. Experiments showed that the automaton created in Algorithm 1 is a good visualization how graphs can be constructed that satisfy the desired constraint, since almost every path through the automaton gives rise to an executable sequence of rules. This also leads to a high flexibility in instance generation. E.g. one can find a solution where selected rules are applied a fixed number of times by searching an appropriate path through the automaton. This is not possible with Algorithm 2. On the other hand the automaton may become very large in states and requires much computation power during construction due to the implications to check. The automaton derived via Algorithm 2 is much smaller in states and gives less information about the transition system. But experiments showed that it can be created much faster. For example, our prototype creates the automata in Example 5 in 231ms using Algorithm 1 and 104ms using Algorithm 2. In opposite, the Petri-Net example (without arcs) introduced in Section 4 is calculated in about 4min using Algorithm 1 and in 2.5s using Algorithm 2. When setting the timeout for the used implication solver from 1s to 200ms, the computation times are shortened to 1min 49s for Algorithm 1 and 960ms for Algorithm 2. Surprisingly, setting the timeout to 100ms increases the computation time for Algorithm 1 slightly to 1min 59s, because less equivalent states are found which leads to bigger search spaces in the last round of the algorithm. The experiments have been performed on a Laptop PC with Intel Core i5-3210M CPU with 2.50GHz and 8GB RAM. The prototype is implemented on top of the ENFORCe framework [4] using the implication prover for graph conditions ProCon [13].

4 Application to instance generation

In this section we revisit an excerpt of the usecase from [14]. In [14] a graph grammar generating petri nets and a set of example constraints is presented, that had been translated from OCL. The graph grammar does not guarantee the constraints, so also invalid instances can be generated. In [14] an approach similar to `WFilter` is used to integrate the constraints into the rules leading to unapplicable rules such that no non-empty instance can be generated at all. We show that via $\texttt{Integrate}^k$ the constraints can be successfully integrated into the grammar such that the resulting grammar generates all valid instances.

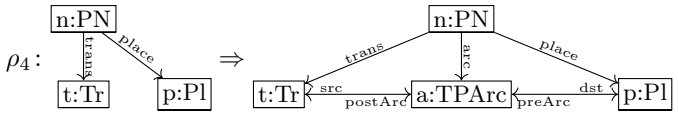
We adopt the approach from the last section to typed attributed graphs as used in [14]. Here types correspond to labels. For simplicity we ignore the attributes here, because they are not involved in the crucial parts of the example. We have the following rules.

1. Creating a new Petri net: `createPetriNet` $\rho_1 \colon \emptyset \Rightarrow \boxed{:\text{PN}}$

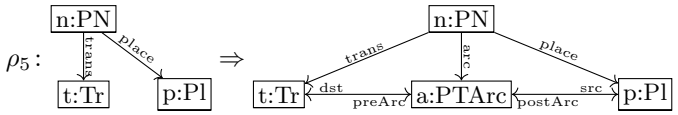
2. Inserting a new place (`insertPlace`,ρ_2)/new transition (`insertPlace`,ρ_3) in a given Petri net:

$$\rho_2 : \boxed{\text{p:PN}} \Rightarrow \boxed{\text{p:PN}} \xrightarrow{\text{place}} \boxed{\text{:Pl}} \qquad \rho_3 : \boxed{\text{p:PN}} \Rightarrow \boxed{\text{p:PN}} \xrightarrow{\text{trans}} \boxed{\text{:Tr}}$$

3. Connecting a transition to a place: `insertTPArc`



4. Connecting a place to a transition: `insertPTArc`



5. Inserting a token in a given place: `insertToken`

$$\rho_6 : \boxed{\text{p:Pl}} \Rightarrow \boxed{\text{p:Pl}} \xrightarrow{\text{token}} \boxed{\text{:Token}}$$

Here we only look at two of the constraints of depth 2. All other constraints are negative constraints.

– Every Petri net has at least one token.

$$c_A = \forall(\boxed{\text{1:PN}}, \exists(\boxed{\text{1:PN}} \xrightarrow{\text{place}} \boxed{\text{2:Pl}} \xrightarrow{\text{token}} \boxed{\text{3:Tk}}))$$

– A Petri net has at least one place and at least one transition.

$$c_B = \forall(\boxed{\text{1:PN}}, \exists(\boxed{\text{1:PN}} \xrightarrow{\text{place}} \boxed{\text{:Pl}})) \qquad c_C = \forall(\boxed{\text{1:PN}}, \exists(\boxed{\text{1:PN}} \xrightarrow{\text{trans}} \boxed{\text{:Tr}}))$$

Using Algorithm 2 we get (among others) the helper constraints

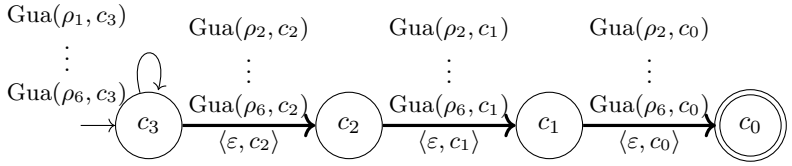
$$hc_1 = \exists(\boxed{\text{1:Pl}}, \forall(\boxed{\text{1:Pl}}\ \boxed{\text{2:PN}}, \exists(\boxed{\text{2:PN}} \xrightarrow{\text{place}} \boxed{\text{1:Pl}}) \vee \exists(\boxed{\text{1:Pl}}\ \boxed{\text{2:PN}} \xrightarrow{\text{place}} \boxed{\text{:Pl}} \xrightarrow{\text{token}} \boxed{\text{:Tk}})))$$
$$hc_2 = \exists(\boxed{\text{1:PN}}, \forall(\boxed{\text{1:PN}}\ \boxed{\text{2:PN}}, \exists(\boxed{\text{1:PN}}\ \boxed{\text{2:PN}} \xrightarrow{\text{trans}} \boxed{\text{:Tr}})))$$
$$hc_3 = \exists(\boxed{\text{1:PN}}, \forall(\boxed{\text{1:PN}}\ \boxed{\text{2:PN}}, \exists(\boxed{\text{1:PN}}\ \boxed{\text{2:PN}} \xrightarrow{\text{place}} \boxed{\text{:Pl}} \xrightarrow{\text{token}} \boxed{\text{:Tk}})))$$

and the constraints[4]

$$c_0 = c_A \wedge c_B \wedge c_C \qquad c_1 = c_0 \vee (hc_1 \wedge c_C) \vee (hc_2 \wedge c_A) \vee \ldots$$
$$c_2 = c_1 \vee (hc_3 \wedge c_C) \vee (hc_1 \wedge hc_2) \vee \ldots \qquad c_3 = c_2 \vee (hc_3 \wedge hc_2) \vee \ldots$$

The resulting automaton is

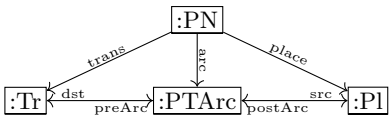
[4] The ellipses (...) indicate fewer subconditions that occur in the full calculations but are ignored here for brevity. Since they are part of a disjunction we can skip them without getting invalid results.

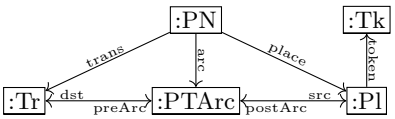


We cannot present the full grammar here that results from applying Construction 3 afterwards, but the generation from the automaton is straight forward. E.g. the resulting rule for the transition $\langle c_1, \text{Gua}(\rho_6, c_0), c_0 \rangle$ in the automaton is

$$\Big\langle \textcircled{c_1}\ \boxed{\text{p:Pl}} \hookleftarrow \boxed{\text{p:Pl}} \hookrightarrow \textcircled{c_0}\ \boxed{\text{p:Pl}} \xrightarrow{\text{token}} \boxed{\text{:Token}}, \quad \forall(\textcircled{c_1}\ \boxed{\text{p:Pl}}\ \boxed{\text{2:PN}},$$
$$(\exists(\textcircled{c_1}\ \boxed{\text{2:PN}} \xrightarrow{\text{place}} \boxed{\text{p:Pl}}) \vee \exists(\textcircled{c_1}\ \boxed{\text{p:Pl}}\ \boxed{\text{2:PN}} \xrightarrow{\text{place}} \boxed{\text{:Pl}} \xrightarrow{\text{token}} \boxed{\text{:Tk}}))$$
$$\wedge\ \exists(\textcircled{c_1}\ \boxed{\text{p:Pl}}\ \boxed{\text{2:PN}} \xrightarrow{\text{trans}} \boxed{\text{:Tr}})) \Big\rangle$$

Now we can use the automaton (or the grammar) to generate an instance. E.g. we can loop in state c_3 and generate a petri net with an arc applying $\text{Gua}(\rho_1, c_3); \text{Gua}(\rho_2, c_3); \text{Gua}(\rho_3, c_3); \text{Gua}(\rho_5, c_3);$ This results in the left graph below. Then we leave c_3. Constraints c_2 and c_1 are already satisfied so we can select the ε-rules switching to state c_1. We find that we can apply $\text{Gua}(\rho_6, c_0)$ and enter the final state c_0. So we finally get the right graph below that satisfies $c_A \wedge c_B \wedge c_C$.





5 Related concepts

Using some kind of control structure on top of graph transformation rules is not a new idea – rule automata use finite automata as a Control Condition in the sense of [2]. Alternatively to directly using constraints as states, as we do in Algorithms 1 and 2, rule automata could be defined with an (optional) labeling function assigning constraints to states. So the automata used in these constructions are closely related to Kripke transition systems [12]. Kripke transition systems consist of states and transitions, where states are labeled with sets of atomic constraints and transitions are labeled with actions.

In [15, 10] graph transformation systems are interpreted as transition systems where states are graphs and transitions are transformation rules. In [10] these so-called graph transition systems are mapped to Kripke structures where the atomic properties that hold in a state are names of rules that are applicable on the graph. However, state space reduction does not go beyond eliminating isomorphic graphs. In this paper we abstract from explicit graphs to classes of graphs sharing one property, i.e. they satisfy the same constraint. The weakest precondition calculus from [8] ensures that the abstraction is correct with respect to the rules.

Graph transformation systems can also be interpreted as transition systems and sets of graph constraints as constraint systems in the sense of [1], as shown in Section 3. The argumentation in the proof for Theorem 3 is identical to the termination part of the proof of Theorem 5.5 in [1]. Well structured systems have also been used in [11] but with other well quasi-orderings.

The instance generation problem (given a meta-model and a set of constraints, find instances of the meta-model that satisfy the constraints) has been addressed in several works. While in [14] graph transition systems are used, other approaches use SAT solving to find instances. Popular examples are [5, 16].

6 Conclusion

In this paper we have presented some algorithms to find grammars that solve the filter problem for graph grammars GG and constraints c, i.e. we construct graph grammars GG_c such that $\mathcal{L}(GG_c) \subseteq \mathcal{L}(GG) \cap [\![c]\!]$. The main result are two algorithms `Integrate1` and `Integrate2` for generating graph grammars GG_c. We give also lower bounds for $\mathcal{L}(GG_c)$ and give a characterization in which cases `Integrate` yields a complete grammar, i.e. $\mathcal{L}(GG_c) = \mathcal{L}(GG) \cap [\![c]\!]$. The constructions for `Integrate` use finite automata over rules, called rule automata, as an intermediate structure that we introduce for this purpose. Rule automata can be translated to graph grammars in a straight forward way. We also characterize special properties of the grammars generated by `Integrate`: The resulting grammars have no need of backtracking in their implementation (we call this goal-oriented), and therefore we assume our approach to be well-suited for generating large sets of graphs.

There is ongoing work on a prototype that implements `Integrate` to generate grammars and instances. The current prototype is based on ENFORCe [4]. We choose ENFORCe because the construction for weakest preconditions and an implication prover for graph constraints [13] are already present. A later implementation will probably be based on EMF Henshin [3] to better integrate with common tools for meta-modeling. Also a quantitative evaluation of the algorithms and a comparison to existing tools for instance generation (e.g. UML2CSP [5], Alloy [16]) are future work.

Acknowledgment. The author would like to thank Annegret Habel for lots of useful advices and support.

References

1. Abdulla, P.A., Jonsson, B.: Ensuring completeness of symbolic verification methods for infinite-state systems. Theor. Comput. Sci. 256(1-2), 145–167 (2001)
2. Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.J., Kuske, S., Plump, D., Schrr, A., Taentzer, G.: Graph transformation for specification and programming. Science of Computer Programming 34(1), 1–54 (1999)

3. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place emf model transformations. In: Model Driven Engineering Languages and Systems, LNCS, vol. 6394, pp. 121–135. Springer (2010)
4. Azab, K., Habel, A., Pennemann, K.H., Zuckschwerdt, C.: ENFORCe: A system for ensuring formal correctness of high-level programs. ECEASST 1 (2006)
5. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007). pp. 547–548 (2007)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science, Springer (2006)
7. Ehrig, H., Golas, U., Habel, A., Lambers, L., Orejas, F.: $\mathcal{M}$-adhesive transformation systems with nested application conditions. part 1: parallelism, concurrency and amalgamation. Mathematical Structures in Computer Science 24(4) (2014)
8. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science 19, 245–296 (4 2009)
9. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, 3rd edn. (2007)
10. Kastenberg, H., Rensink, A.: Model checking dynamic states in GROOVE. In: Model Checking Software, 13th International SPIN Workshop. LNCS, vol. 3925, pp. 299–305. Springer (2006)
11. König, B., Stückrath, J.: A general framework for well-structured graph transformation systems. In: CONCUR 2014. Proceedings. LNCS, vol. 8704, pp. 467–481. Springer (2014)
12. Müller-Olm, M., Schmidt, D.A., Steffen, B.: Model-checking: A tutorial introduction. In: Static Analysis (SAS '99). LNCS, vol. 1694, pp. 330–354 (1999)
13. Pennemann, K.H.: Development of Correct Graph Transformation Systems. Dissertation, Universität Oldenburg (2009)
14. Radke, H., Arendt, T., Becker, J.S., Habel, A., Taentzer, G.: Translating essential OCL invariants to nested graph constraints focusing on set operations. In: ICGT 2015, Proceedings. LNCS, vol. 9151, pp. 155–170. Springer (2015)
15. Rensink, A.: Explicit state model checking for graph grammars. In: Degano, P., Nicola, R.D., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 114–132. Springer (2008)
16. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to Alloy and back again. In: MoDELS Workshops. LNCS, vol. 6002, pp. 158–171 (2010)
17. Taentzer, G.: Instance generation from type graphs with arbitrary multiplicities. ECEASST 47 (2012)