Specifying Graph Languages with Type Graphs

Andrea Corradini[1], Barbara König[2], and Dennis Nolte[2]

[1] Università di Pisa, Italy
andrea@di.unipi.it
[2] Universität Duisburg-Essen, Germany
{barbara_koenig,dennis.nolte}@uni-due.de

Abstract. We investigate three formalisms to specify graph languages, i.e. sets of graphs, based on type graphs. First, we are interested in (pure) type graphs, where the corresponding language consists of all graphs that can be mapped homomorphically to a given type graph. In this context, we also study languages specified by restriction graphs and their relation to type graphs. Second, we extend this basic approach to a type graph logic and, third, to type graphs with annotations. We present decidability results and closure properties for each of the formalisms.

1 Introduction

Formal languages in general and regular languages in particular play an important role in computer science. They can be used for pattern matching, parsing, verification and in many other domains. For instance, verification approaches such as reachability checking, counterexample-guided abstraction refinement [5] and non-termination analysis [11] could be directly adapted to graph transformation systems if one had a graph specification formalism with suitable closure properties, computable pre- and postconditions and inclusion checks. Inclusion checks are also important for checking when a fixpoint iteration sequence stabilizes.

While regular languages for words and trees are well-understood and can be used efficiently and successfully in applications, the situation is less satisfactory when it comes to graphs. Although the work of Courcelle [9] presents an accepted notion of recognizable graph languages, equivalent to regular languages, this is often not useful in practice, due to the sheer size of the resulting graph automata. Other formalisms, such as application conditions [20, 13] and first-order or second-order logics, feature more compact descriptions, but there are problems with expressiveness, undecidability issues or unsatisfactory closure properties.[3]

Hence, we believe that it is important to study and compare specification formalisms (i.e., automata, grammars and logics) that allow to specify potentially infinite sets of graphs. In our opinion there is no one-fits-all solution, but we believe that specification mechanisms should be studied and compared more extensively.

In this paper we study specification formalisms based on type graphs, where a type graph T represents all graphs that can be mapped homomorphically to T,

[3] A more detailed overview over related formalisms is given in the conclusion (Section 6).

This final publication is avaliable at Springer via https://doi.org/10.1007/978-3-319-61470-0_5

potentially taking into account some extra constraints. Type graphs are common in graph rewriting [7, 21]. Usually, one assumes that all items, i.e., rules and graphs to be rewritten, are typed, introducing constraints on the applicability of rules. Hence, type graphs are in a way seen as a form of labelling. This is different from our point of view, where graphs (and rules) are – a priori – untyped (but labeled) and type graphs are simply a means to represent sets of graphs.

There are various reasons for studying type graphs: first, they are reasonably simple with many positive decidability results and they have not yet been extensively studied from the perspective of specification formalisms. Second, other specification mechanisms – especially those used in connection with verification and abstract graph transformation [19, 23, 24] – are based on type graphs: abstract graphs are basically type graphs with extra annotations. Third, while not being as expressive as recognizable graph languages, they retain a nice intuition from regular languages: given a finite state automaton M one can think of the language of M as the set of all string graphs that can be mapped homomorphically to M (respecting initial and final states).

We in fact study three different formalisms based on type graphs: first, pure type graphs T, where the language consists simply of all graphs that can be mapped to T. We also discuss the connection between type graph and restriction graph languages. Then, in order to obtain a language with better boolean closure properties, we study type graph logic, which consists of type graphs enriched with boolean connectives (negation, conjunction, disjunction). Finally, we consider annotated type graphs, where the annotations constrain the number of items mapped to a specific node or edge, somewhat similar to the proposals from abstract graph rewriting mentioned above.

In all three cases we are interested in closure properties and in decidability issues (such decidability of the membership, emptiness and inclusion problems) and in expressiveness. Proofs for all the results and an extended example for annotated type graphs can be found in [6].

2 Preliminaries

We first introduce graphs and graph morphisms. In the context of this paper we use edge-labeled, directed graphs.

Definition 1 (Graph). *Let Λ be a fixed set of edge labels. A* Λ-labeled graph *is a tuple $G = \langle V, E, src, tgt, lab \rangle$, where V is a finite set of nodes, E is a finite set of edges, $src, tgt \colon E \to V$ assign to each edge a source and a target node, and $lab \colon E \to \Lambda$ is a labeling function.*

We will denote, for a given graph G, its components by V_G, E_G, src_G, tgt_G and lab_G, unless otherwise indicated.

Definition 2 (Graph morphism). *Let G, G' be two Λ-labeled graphs. A* graph morphism *$\varphi \colon G \to G'$ consists of two functions $\varphi_V : V_G \to V_{G'}$ and $\varphi_E \colon E_G \to E_{G'}$, such that for each edge $e \in E_G$ it holds that $src_{G'}(\varphi_E(e)) = \varphi_V(src_G(e))$,*

$tgt_{G'}(\varphi_E(e)) = \varphi_V(tgt_G(e))$ *and* $lab_{G'}(\varphi_E(e)) = lab_G(e)$*. If* φ *is both injective and surjective it is called an isomorphism.*

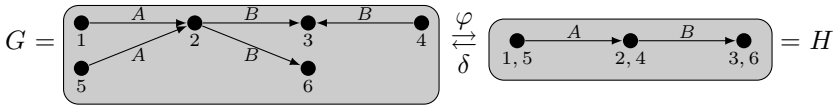
We will often drop the subscripts V, E and write φ instead of φ_V, φ_E. We will consider the category **Graph** having Λ-labeled graphs as objects and graph morphisms as arrows. The set of its objects will be denoted by $\mathbf{Gr}_{\mathbf{\Lambda}}$. The categorical structure induces an obvious preorder on graphs, defined as follows.

Definition 3 (Homomorphism preorder). *Given graphs* G *and* H*, we write* $G \to H$ *if there is a graph morphism from* G *to* H *in* **Graph**. *The relation* $\to$ *is obviously a preorder (i.e. it is reflexive and transitive) and we call it the* homomorphism preorder *on graphs. We write* $G \not\to H$ *if* $G \to H$ *does not hold. Graphs* G *and* H *are* homomorphically equivalent*, written* $G \sim H$*, if both* $G \to H$ *and* $H \to G$ *hold.*

We will revisit the concept of *retracts* and *cores* from [15]. *Cores* are a convenient way to minimize type graphs, as, according to [15], all graphs G, H with $G \sim H$ have isomorphic cores.

Definition 4 (Retract and core). *A graph* H *is called a* retract *of a graph* G *if* H *is a subgraph of* G *and in addition there exists a morphism* $\varphi\colon G \to H$*. A graph* H *is called a* core *of* G*, written* $H = core(G)$*, if it is a retract of* G *and has itself no proper retracts.*

Example 5. The graph H is a retract of G, where the morphism φ is indicated by the node numbering:



Since the graph H does not have a proper retract itself it is also the core of G.

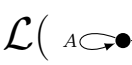
3 Languages Specified by Type or Restriction Graphs

In this section we introduce two classes of graph languages that are characterized by two somewhat dual properties. A *type graph language* contains all graphs that can be mapped homomorphically to a given *type graph*, while a *restriction graph language* includes all graphs that *do not contain* an homomorphic image of a given *restriction graph*. Next, we discuss for these two classes of languages some properties such as closure under set operators, decidability of emptiness and inclusion, and decidability of closure under rewriting via double-pushout rules. Finally we discuss the relationship between these two classes of graph languages.

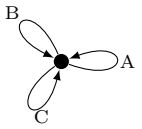
Definition 6 (Type graph language). *A* type graph T *is just a* Λ*-labeled graph. The language* $\mathcal{L}(T)$ *is defined as:*

$$\mathcal{L}(T) = \{G \mid G \to T\}.$$

Example 7. The following type graph T over the edge label set $\Lambda = \{A, B\}$ specifies a type graph language $\mathcal{L}(T)$ consisting of infinitely many graphs:

$\mathcal{L}\Big($  $\Big) = \Big\{$ $\varnothing$, $\Big\}$

The category **Graph** has a final object, that we denote $T^{\Lambda}_{\ast}$, consisting of one node (called *flower node* $\ast$) and one loop for each label in Λ. Therefore $\mathcal{L}(T^{\Lambda}_{\ast}) = \mathbf{Gr}_{\mathbf{\Lambda}}$. The graph $T^{\Lambda}_{\ast}$ for $\Lambda = \{A, B, C\}$ is depicted to the right.

$T_{\ast} =$ 

Specifying graph languages using type graphs gives us the possibility to forbid certain graph structures by not including them into the type graph. For example, no graph in the language of Example 7 can contain a B-loop or an A-edge incident to the target of a B-edge. However, it is not possible to force some structures to exist in all graphs of the language, since the morphism to the type graph need not be surjective. This point will be addressed with the notion of *annotated type graph* in Section 5.

Another way (possibly more explicit) to specify languages of graphs not including certain structures, is the following one.

Definition 8 (Restriction graph language). *A* restriction graph R *is just a* Λ*-labeled graph. The language* $\mathcal{L}_R(R)$ *is defined as:*

$$\mathcal{L}_R(R) = \{G \mid R \nrightarrow G\}.$$

We will consider the relationship between the class of languages introduced in Definitions 6 and 8 in Section 3.3.

3.1 Closure and Decidability Properties

The type graph and restriction graph languages enjoy the following complementary closure properties with respect to set operators.

Proposition 9. *Type graph languages are closed under intersection (by taking the product of type graphs) but not under union or complementation, while restriction graph languages are closed under union (by taking the coproduct of restriction graphs) but not under intersection or complementation.*

Instead the two classes of languages enjoy similar decidability properties.

Proposition 10. *For a graph language* $\mathcal{L}$ *characterized by a type graph* T *(i.e.* $\mathcal{L} = \mathcal{L}(T)$*) or by a restriction graph* R *(i.e.* $\mathcal{L} = \mathcal{L}_R(R)$*) the following problems are decidable:*

1. *Membership, i.e. for each graph* G *it is decidable if* $G \in \mathcal{L}$ *holds.*
2. *Emptiness, i.e. it is decidable if* $\mathcal{L} = \varnothing$ *holds.*

Furthermore, language inclusion is decidable for both classes of languages:

3. *Given type graphs* T_1 *and* T_2 *it is decidable if* $\mathcal{L}(T_1) \subseteq \mathcal{L}(T_2)$ *holds.*
4. *Given restriction graphs* R_1 *and* R_2 *it is decidable if* $\mathcal{L}_R(R_1) \subseteq \mathcal{L}_R(R_2)$ *holds.*

3.2 Closure under Double-Pushout Rewriting

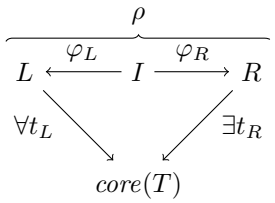
In this subsection we are using the DPO approach with general, not necessarily injective, rules and matches. We discuss how we can show that a graph language $\mathcal{L}$ is a closed under a given graph transformation rule $\rho = (L \leftarrow\!\varphi_L\!- I -\!\varphi_R\!\rightarrow R)$, i.e., $\mathcal{L}$ is an invariant for ρ. This means that for all graphs G, H, where G can be rewritten to H via ρ, it holds that $G \in \mathcal{L}$ implies $H \in \mathcal{L}$.

For both type graph languages and restriction graph languages, separately, we characterize a sufficient and necessary condition which shows that closure under rule application is decidable. The condition for restriction graph languages is related to a condition already discussed in [14].

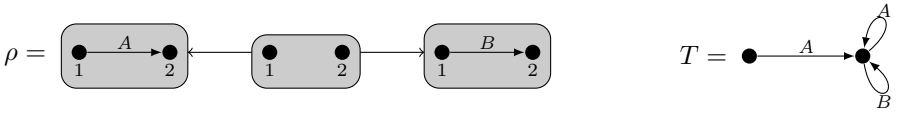
Proposition 11 (Closure under DPO rewriting for restriction graphs). *A restriction graph language $\mathcal{L}_R(S)$ is closed under a rule $\rho = (L \leftarrow\!\varphi_L\!- I -\!\varphi_R\!\rightarrow R)$ if and only if the following condition holds: for every pair of morphisms $\alpha\colon R \to F$, $\beta\colon S \to F$ which are jointly surjective, applying the rule ρ with (co-)match α backwards to F yields a graph E with a homomorphic image of S, i.e., $E \notin \mathcal{L}_R(S)$.*

Proposition 12 (Closure under DPO rewriting for type graphs). *A type graph language $\mathcal{L}(T)$ is closed under a rule $\rho = (L \leftarrow\!\varphi_L\!- I -\!\varphi_R\!\rightarrow R)$ if and only if for each morphism $t_L\colon L \to core(T)$, there exists a morphism $t_R\colon R \to core(T)$ such that $t_L \circ \varphi_L = t_R \circ \varphi_R$.*

$\mathcal{L}(T)$ *is closed under application of* $\rho \quad \Leftrightarrow$



We show that the *only if part* ($\Rightarrow$) of Proposition 12 cannot be weakened by considering morphisms to the type graph T, instead of to $core(T)$. In fact, consider the following type graph T and the rule ρ:



The type graph T contains the flower node, i.e., it has $T^{\{A,B\}}_{\text{✲}}$ as subgraph. This ensures that each graph G, edge-labeled over $\Lambda = \{A, B\}$, is in the language $\mathcal{L}(T)$, and thus by rewriting any graph $G \in \mathcal{L}(T)$ into a graph H using ρ it is guaranteed that $H \in \mathcal{L}(T)$. However there is a morphism $t_L\colon L \to T$, the one mapping the A-labeled edge of L to the left A-labeled edge of T, such that there exists no morphism $t_R\colon R \to T$ satisfying $t_L \circ \varphi_L = t_R \circ \varphi_R$.

3.3 Relating Type graph and Restriction Graph Languages

Both type graph and restriction graph languages specify collections of graphs by forbidding the presence of certain structures. This is more explicit with the use of restriction graphs, though. A natural question is how the two classes of languages are related. A partial answer to this is provided by the notion of *duality pairs* and by an important result concerning their existence, presented in [15].[4]

Definition 13 (Duality pair). *Given two graphs R and T, we call T the* dual *of R if for every graph G it holds that $G \to T$ if and only if $R \not\to G$. In this case the pair (R, T) is called* duality pair.

Clearly, we have that (R, T) is a duality pair if and only if the restriction graph language $\mathcal{L}_R(R)$ coincides with the type graph language $\mathcal{L}(T)$.

Example 14. Let $\Lambda = \{A, B\}$ be given. The following is a duality pair:

$$(R, T) = \left(\begin{array}{ccccc} \bullet & \xrightarrow{A} & \bullet & \xrightarrow{B} & \bullet \\ 1 & & 2 & & 3 \end{array} \;,\; \begin{array}{ccccc} A\circlearrowright\bullet & & \xleftarrow{A,\,B} & & \bullet\circlearrowright B \\ 1 & & & & 2 \end{array} \right)$$

Since node 1 of T is not the source of a B-labeled edge and node 2 is not the target of an A-labeled edge, for every graph G we have $G \to T$ iff it does not contain a node which is both the target of an A-labeled edge and the source of a B-labeled edge. But it contains such a node if and only if $R \to G$.

One can identify the class of restriction graphs for which a corresponding type graph exists which defines the same graph language. Results from [15] state[5] that given a core graph R, a graph T can be constructed such that (R, T) is a duality pair if and only if R is a tree.

Thus we have a precise characterisation of the intersection of the classes of type and restriction graph languages: $\mathcal{L}$ belongs to the intersection if and only if it is of the form $\mathcal{L} = \mathcal{L}_R(R)$ and $core(R)$ is a tree. It is worth mentioning that the construction of T from R using the results from [15] contains two exponential blow-ups. This can be interpreted by saying that type graphs have limited expressiveness if used to forbid the presence of certain structures.

4 Type Graph Logic

In this section we investigate the possibility to define a language of graphs using a logical formula over type graphs. We start by defining the syntax and semantics of a type graph logic (TGL).

[4] Note that in [15] graphs are simple, but it can be easily seen that for our purposes the results can be transferred straightforwardly.

[5] We refer to Lemma 2.3, Lemma 2.5 and Theorem 3.1 in [15].

Definition 15 (Syntax and semantics of *TGL*). *A TGL formula F over a fixed set of edge labels Λ is formed according to the following grammar:*

$$F := T \mid F \vee F \mid F \wedge F \mid \neg F, \qquad \text{where } T \text{ is a type graph.}$$

Each TGL formula F denotes a graph language $\mathcal{L}(F) \subseteq \mathbf{Gr}_{\mathbf{\Lambda}}$ defined by structural induction as follows:

$$\mathcal{L}(T) = \{G \in \mathbf{Gr}_{\mathbf{\Lambda}} \mid G \to T\} \qquad \mathcal{L}(\neg F) = \mathbf{Gr}_{\mathbf{\Lambda}} \setminus \mathcal{L}(F)$$
$$\mathcal{L}(F_1 \wedge F_2) = \mathcal{L}(F_1) \cap \mathcal{L}(F_2) \qquad \mathcal{L}(F_1 \vee F_2) = \mathcal{L}(F_1) \cup \mathcal{L}(F_2)$$

Clearly, due to the presence of boolean connectives, boolean closure properties come for free.

Example 16. Let the following *TGL* formula F over $\Lambda = \{A, B\}$ be given:

$$F = \neg$$ $$\wedge\ \neg$$ 

The graph language $\mathcal{L}(F)$ consists of all graphs which do not consist exclusively of A-edges or of B-edges, i.e., which contain at least one A-labeled edge and at least one B-labeled edge, something that can not be expressed by pure type graphs.

We now present some positive results for graph languages $\mathcal{L}(F)$ over *TGL* formulas F with respect to decidability problems. Due to the conjunction and negation operator, the emptiness (or unsatisfiability) check is not as trivial as it is for pure type graphs. Note that thanks to the presence of boolean connectives, inclusion can be reduced to emptiness.

Proposition 17. *For a graph language $\mathcal{L}(F)$ characterized by a TGL formula F, the following problems are decidable:*

- *Membership, i.e. for all graphs G it is decidable if $G \in \mathcal{L}(F)$ holds.*
- *Emptiness, i.e. it is decidable if $\mathcal{L}(F) = \varnothing$ holds.*
- *Language inclusion, i.e. given two TGL formulas F_1 and F_2 it is decidable if $\mathcal{L}(F_1) \subseteq \mathcal{L}(F_2)$ holds.*

Such a logic could alternatively also be defined based on restriction graphs. A related logic, for injective occurrences of restriction graphs, is studied in [17], where the authors also give a decidability result via inference rules.

5 Annotated Type Graphs

In this section we will improve the expressiveness of the type graphs themselves, rather than using an additional logic to do so. We will equip graphs with additional annotations. As explained in the introduction, this idea was already used similarly in abstract graph rewriting. In contrast to most other approaches, we will investigate the problem from a categorical point of view.

The idea we follow is to annotate each element of a type graph with pairs of multiplicities, denoting upper and lower bounds. We will define a category of multiply annotated graphs, where we consider elements of a lattice-ordered monoid (short ℓ-monoid) as multiplicities.

Definition 18 (Lattice-ordered monoid). *A lattice-ordered monoid (ℓ-monoid) $(\mathcal{M}, +, \leq)$ consists of a set $\mathcal{M}$, a partial order $\leq$ and a binary operation $+$ such that*

- *$(\mathcal{M}, \leq)$ is a lattice.*
- *$(\mathcal{M}, +)$ is a monoid; we denote its unit by 0.*
- *It holds that $a + (b \vee c) = (a + b) \vee (a + c)$ and $a + (b \wedge c) = (a + b) \wedge (a + c)$, where $\wedge, \vee$ are the meet and join of $\leq$.*

*We denote by ℓ**Mon** the category having ℓ-monoids as objects and as arrows monoid homomorphisms which are monotone.*

Example 19. Let $n \in \mathbb{N}\backslash\{0\}$ and take $\mathcal{M}_n = \{0, 1, \ldots, n, m\}$ (zero, one, ..., n, many) with $0 \leq 1 \leq \cdots \leq n \leq m$ and addition as monoid operation with the proviso that $\ell_1 + \ell_2 = m$ if the sum is larger than n. Clearly, for all $a, b, c \in \mathcal{M}_n$ $a \vee b = \max\{a, b\}$ and $a \wedge b = \min\{a, b\}$. From this we can infer distributivity and therefore $(\mathcal{M}_n, +, \leq)$ forms an ℓ-monoid.

Furthermore, given a set S and an ℓ-monoid $(\mathcal{M}, +, \leq)$, it is easy to check that also $(\{a \colon S \to \mathcal{M}\}, +, \leq)$ is an ℓ-monoid, where the elements are functions from S to $\mathcal{M}$ and the partial order and the monoidal operation are taken pointwise.

In the following we will sometimes denote an ℓ-monoid by its underlying set.

Definition 20 (Annotations and multiplicities for graphs). *Given a functor $\mathcal{A} \colon$ **Graph** $\to \ell$**Mon**, an* annotation based on $\mathcal{A}$ *for a graph G is an element $a \in \mathcal{A}(G)$. We write $\mathcal{A}_\varphi$, instead of $\mathcal{A}(\varphi)$, for the action of functor $\mathcal{A}$ on a graph morphism φ. We assume that for each graph G there is a* standard annotation *based on $\mathcal{A}$ that we denote by s_G, thus $s_G \in \mathcal{A}(G)$.*
*Given an ℓ-monoid $\mathcal{M}_n = \{0, 1, \ldots, n, m\}$ we define the functor $\mathcal{B}^n$: **Graph** $\to$ ℓ**Mon** as follows:*

- *for every graph G, $\mathcal{B}^n(G) = \{a \colon (V_G \cup E_G) \to \mathcal{M}_n\}$;*
- *for every graph morphism $\varphi \colon G \to G'$ and $a \in \mathcal{B}^n(G)$, we have $\mathcal{B}^n_\varphi(a) \colon V_{G'} \cup E_{G'} \to \mathcal{M}_n$ with:*

$$\mathcal{B}^n_\varphi(a)(y) = \sum_{\varphi(x)=y} a(x), \quad \text{where } x \in (V_G \cup E_G) \text{ and } y \in (V_{G'} \cup E_{G'})$$

Therefore an annotation based on a functor $\mathcal{B}^n$ associates every item of a graph with a number (or the top value m). We will call such kind of annotations multiplicities. *Furthermore, the action of the functor on a morphism transforms a multiplicity by summing up (in $\mathcal{M}_n$) the values of all items of the source graph that are mapped to the same item of the target graph.*

For a graph G, its standard multiplicity $s_G \in \mathcal{B}^n(G)$ *is defined as the function which maps every node and edge of G to 1.*

Some of the results that we will present in the rest of the paper will hold for annotations based on a generic functor $\mathcal{A}$, some only for annotations based on functors $\mathcal{B}^n$, i.e. for multiplicities.

The type graphs which we are going to consider are enriched with a set of pairs of annotations. The motivation for considering multiple annotations rather than a single one is mainly to ensure closure under union. Each pair can be interpreted as establishing a lower and an upper bound to what a graph morphism can map to the graph.

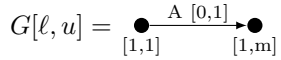
Definition 21 (Multiply annotated graphs). *Given a functor* $\mathcal{A}\colon$ **Graph** $\to$ ℓ**Mon**, *a* multiply annotated graph $G[M]$ (over $\mathcal{A}$) *is a graph* G *equipped with a finite set of pairs of annotations* $M \subseteq \mathcal{A}(G) \times \mathcal{A}(G)$, *such that* $\ell \leq u$ *for all* $(\ell, u) \in M$.

An arrow $\varphi\colon G[M] \to G'[M']$, *also called a* legal morphism, *is a graph morphism* $\varphi\colon G \to G'$ *such that for all* $(\ell, u) \in M$ *there exists* $(\ell', u') \in M'$ *with* $\mathcal{A}_\varphi(\ell) \geq \ell'$ *and* $\mathcal{A}_\varphi(u) \leq u'$. *We will write* $G[\ell, u]$ *as an abbreviation of* $G[\{(\ell, u)\}]$. *In case of annotations based on* $\mathcal{B}^n$, *we will often call a pair* (ℓ, u) *a* double multiplicity.

Multiply annotated graphs and legal morphisms form a category.

Lemma 22. *The composition of two legal morphisms is a legal morphism.*

Example 23. Consider the following multiply annotated graphs (over $\mathcal{B}^2$) $G[\ell, u]$ and $H[\ell', u']$, both having one double multiplicity.

$G[\ell, u] =$ 

$H[\ell', u'] =$ 

As evident from the picture, multiplicities are represented by writing the lower and upper bounds next to the corresponding graph elements. Note that there is a unique, obvious graph morphism $\varphi\colon G \to H$, mapping both nodes of G to the only node of H. Concerning multiplicities, by adding the lower and upper bounds of the two nodes of G, one gets the interval $[2, m]$ which is included in the interval of the node of H, $[1, m]$. Similarly, the double multiplicity $[0, 1]$ of the edge of G is included in $[0, m]$. Therefore, since both $\mathcal{B}^2_\varphi(\ell) \geq \ell'$ and $\mathcal{B}^2_\varphi(u) \leq u'$ hold, we can conclude that $\varphi\colon G[\ell, u] \to H[\ell', u']$ is a legal morphism.

We are now ready to define how a graph language $\mathcal{L}(T[M])$ looks like.

Definition 24 (Graph languages of multiply annotated type graphs). *We say that a graph* G *is represented by a multiply annotated type graph* $T[M]$ *whenever there exists a legal morphism* $\varphi\colon G[s_G, s_G] \to T[M]$, *i.e., there exists* $(\ell, u) \in M$ *such that* $\ell \leq \mathcal{A}_\varphi(s_G) \leq u$. *We will write* $G \in \mathcal{L}(T[M])$ *in this case. Whenever* $M = \varnothing$ *for a multiply annotated type graph* $T[M]$ *we get* $\mathcal{L}(T[M]) = \varnothing$.

An extended example can be found in [6].

5.1 Decidability Properties for Multiply Annotated Graphs

We now address some decidability problems for languages defined by multiply annotated graphs. We get positive results with respect to the membership and emptiness problems. However, for decidability of language inclusion we only get partial results.

For the membership problem we can simply enumerate all graph morphisms $\varphi\colon G \to T$ and check if there exists a legal morphism $\varphi\colon G[s_G, s_G] \to T[M]$.

The emptiness check is somewhat more involved, since we have to take care of "illegal" annotations.

Proposition 25. *For a graph language $\mathcal{L}(T[M])$ characterized by a multiply annotated type graph $T[M]$ over $\mathcal{B}^n$ the emptiness problem is decidable: $\mathcal{L}(T[M]) = \varnothing$ iff $M = \varnothing$ or for each $(\ell, u) \in M$ there exists an edge $e \in E_T$ such that $\ell(e) \geq 1$ and $(u(src(e)) = 0$ or $u(tgt(e)) = 0)$.*

Language inclusion can be deduced from the existence of a legal morphism between the two multiply annotated type graphs.

Proposition 26. *The existence of a legal morphism $\varphi\colon T_1[M] \to T_2[N]$ implies $\mathcal{L}(T_1[M]) \subseteq \mathcal{L}(T_2[N])$.*

We would like to remark that this condition is sufficient but not necessary, and we present the following counterexample. Let the following two multiply annotated type graphs $T_1[M_1]$ and $T_2[M_2]$ over $\mathcal{B}^1$ be given where $|M_1| = |M_2| = 1$:

$$T_1[M_1] = \underset{[1,m]}{\bullet} \qquad\qquad T_2[M_2] = \underset{[1,1]}{\bullet}\ \underset{[0,m]}{\bullet}$$

Clearly we have that the languages $\mathcal{L}(T_1[M_1])$ and $\mathcal{L}(T_2[M_2])$ are equal as both contain all discrete non-empty graphs. Thus $\mathcal{L}(T_1[M_1]) \subseteq \mathcal{L}(T_2[M_2])$, but there exists no legal morphism $\varphi\colon T_1[M_1] \to T_2[M_2]$. In fact, the upper bound of the first node of T_2 would be violated if the node of T_1 is mapped by φ to it, while the lower bound would be violated if the node of T_1 is mapped to the other node.

5.2 Deciding Language Inclusion for Annotated Type Graphs

In this section we show that if we allow only bounded graph languages consisting of graphs up to a fixed pathwidth, the language inclusion problem becomes decidable for annotations based on $\mathcal{B}^n$. Pathwidth is a well-known concept from graph theory that intuitively measures how much a graph resembles a path.

The proof is based on the notion of recognizability, which will be described via automaton functors that were introduced in [4]. We start with the main result and explain step by step the arguments that will lead to decidability.

Proposition 27. *The language inclusion problem is decidable for graph languages of bounded pathwidth characterized by multiply annotated type graphs over $\mathcal{B}^n$. That is, given $k \in \mathbb{N}$ and two multiply annotated type graphs $T_1[M_1]$ and $T_2[M_2]$ over $\mathcal{B}^n$, it is decidable whether $\mathcal{L}(T_1[M_1])^{\leq k} \subseteq \mathcal{L}(T_2[M_2])^{\leq k}$, where $\mathcal{L}(T[M])^{\leq k} = \{G \in \mathcal{L}(T[M]) \mid G$ has pathwidth $\leq k\}$.*

Our automaton model, given by automaton functors, reads cospans (i.e., graphs with interfaces) instead of single graphs. Therefore in the following, the category under consideration will be $Cospan_m(\mathbf{Graph})$, i.e. the category of cospans of graphs where the objects are discrete graphs J, K and the arrows are cospans $c\colon J \to G \leftarrow K$ where both graph morphisms are injective. We will refer to the graph J as the *inner interface* and to the graph K as the *outer interface* of the graph G. In addition we will sometimes abbreviate the cospan $c\colon J \to G \leftarrow K$ to the short representation $c\colon J \nrightarrow K$.

According to [3] a graph has pathwidth k iff it can be decomposed into cospans where each middle graph of a cospan has at most $k+1$ nodes. Hence it is easy to check that a path has pathwidth 1, while a clique of order k has pathwidth $k-1$.

Our main goal is to build an automaton which can read all graphs of our language step by step, similar to the idea of finite automata reading words in formal languages. Such an automaton can be constructed for an unbounded language, where the pathwidth is not restricted. However, we obtain a *finite* automaton only if we restrict the pathwidth. Then we can use well-known algorithms for finite automata to solve the language inclusion problem. Note that, if we would use tree automata instead of finite automata, our result could be generalized to graphs of bounded *treewidth*.

We will first introduce the notion of automaton functor (which is a categorical automaton model for so-called recognizable arrow languages) and which is inspired by Courcelle's theory of recognizable graph languages [9].

Definition 28 (Automaton functor [4]). *An automaton functor* $\mathcal{C}\colon \mathrm{Cospan}_m(\mathbf{Graph}) \to \mathbf{Rel}$ *is a functor that maps every object* J *(i.e., every discrete graph) to a finite set* $\mathcal{C}(J)$ *(the set of states of* J*) and every cospan* $c\colon J \nrightarrow K$ *to a relation* $\mathcal{C}(c) \subseteq \mathcal{C}(J) \times \mathcal{C}(K)$ *(the transition relation of* c*). In addition there is a distinguished set of initial states* $I \subseteq \mathcal{C}(\varnothing)$ *and a distinguished set of final states* $F \subseteq \mathcal{C}(\varnothing)$*. The* language $\mathcal{L}_\mathcal{C}$ *of* $\mathcal{C}$ *is defined as follows:*

> *A graph* G *is contained in* $\mathcal{L}_\mathcal{C}$ *if and only if there exist states* $q \in I$ *and* $q' \in F$ *which are related by* $\mathcal{C}(c)$*, i.e.* $(q, q') \in \mathcal{C}(c)$*, where* $c\colon \varnothing \to G \leftarrow \varnothing$ *is the unique cospan with empty interfaces and middle graph* G*.*

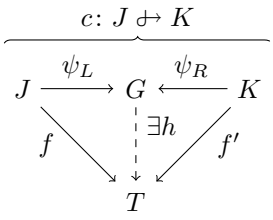
Languages accepted by automaton functors are called recognizable*.*

We will now define an automaton functor for a type graph $T[M]$ over $\mathcal{B}^n$.

Definition 29 (Counting cospan automaton). *Let* $T[M]$ *be a multiply annotated type graph over* $\mathcal{B}^n$*. We define an automaton functor* $\mathcal{C}_{T[M]}\colon \mathrm{Cospan}_m(\mathbf{Graph}) \to \mathbf{Rel}$ *as follows:*

- *For each object* J *of* $\mathrm{Cospan}_m(\mathbf{Graph})$ *(thus* J *is a finite discrete graph),* $\mathcal{C}_{T[M]}(J) = \{(f, b) \mid f\colon J \to T, b \in \mathcal{B}^n(T)\}$ *is its finite set of states*
- $I \subseteq \mathcal{C}_{T[M]}(\varnothing)$ *is the set of* initial *states with* $I = \{(f\colon \varnothing \to T, 0)\}$*, where* 0 *is the constant* 0*-function*
- $F \subseteq \mathcal{C}_{T[M]}(\varnothing)$ *is the set of* final *states with* $F = \{(f\colon \varnothing \to T, b) \mid \exists (\ell, u) \in M : \ell \leq b \leq u\}$

– *Let* $c\colon J \to G \leftarrow K$ *be an arrow in the category* $\mathrm{Cospan}_m(\mathbf{Graph})$ *with discrete interface graphs* J *and* K *where both graph morphisms* $\psi_L\colon J \to G$ *and* $\psi_R\colon K \to G$ *are injective. Two states* $(f\colon J \to T, b)$ *and* $(f'\colon K \to T, b')$ *are in the relation* $\mathcal{C}_{T[M]}(c)$ *if and only if there exists a morphism* $h\colon G \to T$ *such that the diagram to the right commutes and for all* $x \in V_T \cup E_T$ *the following equation holds:*

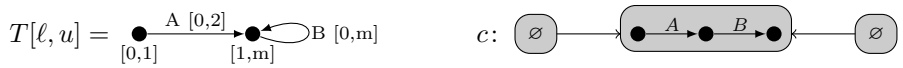


$$b'(x) = b(x) + |\{y \in (G \setminus \psi_R(K)) \mid h(y) = x\}|$$

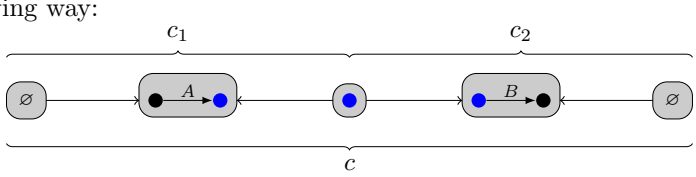
The set $G \setminus \psi_R(K)$ *consists of all elements of* G *which are not targeted by the morphism* ψ_R*, e.g.* $G \setminus \psi_R(K) = (V_G \setminus \psi_R(V_K)) \cup (E_G \setminus \psi_R(E_K))$*. Instead of* $\mathcal{L}_{\mathcal{C}_{T[M]}}$ *and* $\mathcal{C}_{T[M]}$ *we just write* $\mathcal{L}_{\mathcal{C}}$ *and* $\mathcal{C}$ *if* $T[M]$ *is clear from the context.*

The intuition behind this construction is to count for each item x of T, step by step, the number of elements that are being mapped from a graph G (which is in the form of a cospan decomposition) to x, and then check if the bounds of a pair of annotations $(\ell, u) \in M$ of the multiply annotated type graph $T[M]$ are satisfied. We give a short example before moving on to the results.

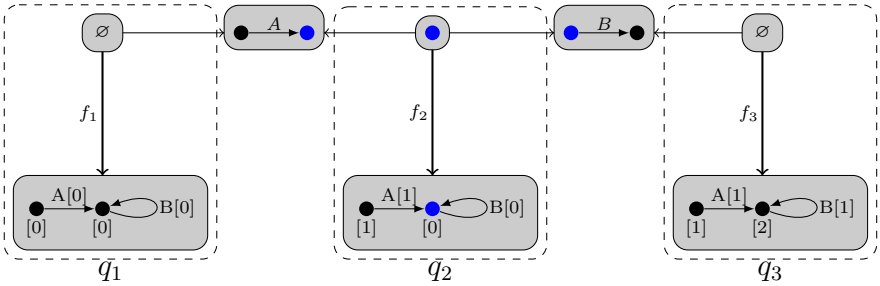
Example 30. Let the following multiply annotated type graph (over $\mathcal{B}^2$) $T[\ell, u]$ and the cospan $(c\colon \varnothing \to G \leftarrow \varnothing)$ with $G \in \mathcal{L}(T[\ell, u])$ be given:



We will now decompose the cospan c into two cospans c_1, c_2 with $c = c_1; c_2$ in the following way:



We let our counting cospan automaton parse the cospan decomposition $c_1; c_2$ step by step now to show how the annotations for the type graph T evolve during the process. According to our construction, every element in T has multiplicity 0 in the initial state of the automaton. We then sum up the number of elements within the middle graphs of the cospans which are *not* part of the right interface. Therefore we get the following parsing process:



We visited three states q_1, q_2 and q_3 in the automaton with $(q_1, q_2) \in \mathcal{C}(c_1)$ and $(q_2, q_3) \in \mathcal{C}(c_2)$. Since $\mathcal{C}$ is supposed to be a functor we get that $\mathcal{C}(c_1); \mathcal{C}(c_2) = \mathcal{C}(c)$ and therefore $(q_1, q_3) \in \mathcal{C}(c)$ also holds. In addition we have $q_1 \in I$ and since the annotation function $b \in \mathcal{B}^2(T)$ in $q_3 = (f_3, b)$ satisfies $\ell \leq b \leq u$ we can infer that $q_3 \in F$. Therefore we can conclude that $G \in \mathcal{L}_\mathcal{C}$ holds as well.

We still need to prove that $\mathcal{C}$ is indeed a functor. Intuitively this shows that acceptance of a graph by the automaton is not dependent on its specific decomposition.

Proposition 31. *Let $c_1 \colon J \to G \leftarrow K$ and $c_2 \colon K \to H \leftarrow L$ be two arrows and let $id_G \colon G \to G \leftarrow G$ be the identity cospan.*

The mapping $\mathcal{C}_{T[M]} \colon \mathrm{Cospan}_m(\mathbf{Graph}) \to \mathbf{Rel}$ is a functor:

1. $\mathcal{C}_{T[M]}(id_G) = id_{C_{T[M]}(G)}$
2. $\mathcal{C}_{T[M]}(c_1; c_2) = \mathcal{C}_{T[M]}(c_1); \mathcal{C}_{T[M]}(c_2)$

The language accepted by the automaton $\mathcal{L}_\mathcal{C}$ is exactly the graph language $\mathcal{L}(T[M])$.

Proposition 32. *Let the multiply annotated type graph $T[M]$ (over $\mathcal{B}^n$) and the corresponding automaton functor $\mathcal{C} \colon \mathrm{Cospan}_m(\mathbf{Graph}) \to \mathbf{Rel}$ for $T[M]$ be given. Then $\mathcal{L}_\mathcal{C} = \mathcal{L}(T[M])$ holds, i.e. for a graph G we have $G \in \mathcal{L}(T[M])$ if and only if there exist states $i \in I \subseteq \mathcal{C}(\varnothing)$ and $f \in F \subseteq \mathcal{C}(\varnothing)$ such that $(i, f) \in \mathcal{C}(c)$, where $c \colon \varnothing \to G \leftarrow \varnothing$.*

Therefore we can construct an automaton for each graph language specified by a multiply annotated type graph $T[M]$, which accepts exactly the same language. In case of a bounded graph language this automaton will have only finitely many states. Furthermore we can restrict the label alphabet, i.e., the cospans by using only atomic cospans, adding a single node or edges (see [2]). Once these steps are performed, we obtain conventional non-deterministic finite automata over a finite alphabet and we can use standard techniques from automata theory to solve the language inclusion problem directly on the finite automata.

5.3 Closure Properties for Multiply Annotated Graphs

Extending the expressiveness of the type graphs by adding multiplicities gives us positive results in case of closure under union and intersection. Here we use constructions that rely on products and coproducts in the category of graphs.
Closure under intersection holds for the most general form of annotations. From $T_1[M_1]$, $T_2[M_2]$ we can construct an annotated type graph $(T_1 \times T_2)[N]$, where N contains all annotations which make both projections $\pi_i \colon T_1 \times T_2 \to T_i$ legal.

Proposition 33. *The category of multiply annotated graphs is closed under intersection.*

We can prove closure under union for the case of annotations based on the functor $\mathcal{B}^n$. Here we take the coproduct $(T_1 \oplus T_2)[N]$, where N contains all annotations of M_1, M_2, transferred to $T_1 \oplus T_2$ via the injections $i_j : T_j \to T_1 \oplus T_2$. Intuitively, graph items not in the original domain of the annotations receive annotation $[0, 0]$. This can be generalized under some mild assumptions (see proof in [6]).

Proposition 34. *The category of multiply annotated graphs over functor $\mathcal{B}^n$ is closed under union.*

Closure under complement is still an open issue. If we restrict to graphs of bounded pathwidth, we have a (non-deterministic) automaton (functor), as described in Section 5.1, which could be determinized and complemented. However, this does not provide us with an annotated type graph for the complement. We conjecture that closure under complement does not hold.

6 Conclusion

Our results on decidability and closure properties for specification languages are summarized in the following table. In the case where the results hold only for bounded pathwidth, the checkmark is in brackets.

		Pure TG	Restr. Gr.	TG Logic	Annotated TG
Decidability	$G \in \mathcal{L}$?	✓	✓	✓	✓
	$\mathcal{L} = \varnothing$?	✓	✓	✓	✓
	$\mathcal{L}_1 \subseteq \mathcal{L}_2$?	✓	✓	✓	(✓)
Closure Properties	$\mathcal{L}_1 \cup \mathcal{L}_2$	✗	✓	✓	✓
	$\mathcal{L}_1 \cap \mathcal{L}_2$	✓	✗	✓	✓
	$\mathbf{Gr}_{\mathbf{\Lambda}} \setminus \mathcal{L}$	✗	✗	✓	?

One open question that remains is whether language inclusion for annotated type graphs is decidable if we do not restrict to bounded treewidth. Similarly, closure under complement is still open.

Furthermore, in order to be able to use these formalisms extensively in applications, it is necessary to provide a mechanism to compute weakest preconditions and strongest postconditions. This does not seem feasible for pure type graphs or the type graph logic. Hence, we are currently working on characterizing weakest preconditions and strongest postconditions in the setting of annotated type graphs. This requires a materialisation construction, similar to [23], which we plan to characterize abstractly, exploiting universal properties in category theory.

Note that our annotations are global, i.e., we count *all* items that are mapped to a specific item in the type graph. This holds also for edges, as opposed to UML multiplicities, which are local wrt. the classes which are related by an edge (i.e., an association). We plan to study the possibility to integrate this into our framework and investigate the corresponding decidability and closure properties.

Related work: As already mentioned there are many approaches for specifying graph languages. One can not say that one is superior to the other, usually there is a tradeoff between expressiveness and decidability properties, furthermore they differ in terms of closure properties.

Recognizable graph languages [8, 9], which are the counterpart to regular word languages, are closely related with monadic second-order graph logic. If one restricts recognizable graph languages to bounded treewidth (or pathwidth as we did), one obtains satisfactory decidability properties. On the other hand, the size of the resulting graph automata is often quite intimidating [2] and hence they are difficult to work with in practical applications. The use of nested application conditions [13], equivalent to first-order logic [20], has a long tradition in graph rewriting and they can be used to compute pre- and postconditions for rules [18]. However, satisfiability and implication are undecidable for first-order logic.

A notion of grammars that is equivalent to context-free (word) grammars are hyperedge replacement grammars [12]. Many aspects of the theory of context-free languages can be transferred to the graph setting.

In heap analysis the representation of pointer structures to be analyzed requires methods to specify sets of graphs. Hence both the TVLA approach by Sagiv, Reps and Wilhelm [23], as well as separation logic [16, 10] face this problem. In [23] heaps are represented by graphs, annotated with predicates from a three-valued logics (with truth values *yes*, *no* and *maybe*).

A further interesting approach are forest automata [1] that have many interesting properties, but are somewhat complex to handle.

In [22] the authors study an approach called Diagram Predicate Framework (DPF), in which type graphs have annotations based on generalized sketches. This formalism is intended for MOF-based modelling languages and allows more complex annotations than our framework.

References

1. Parosh Aziz Abdulla, Lukás Holík, Bengt Jonsson, Ondrej Lengál, Cong Quy Trinh, and Tomás Vojnar. Verification of heap manipulating programs with ordered data by extended forest automata. In *Proc. of ATVA '13*, pages 224–239, 2013. LNCS 8172.
2. Christoph Blume, H.J. Sander Bruggink, Dominik Engelke, and Barbara König. Efficient symbolic implementation of graph automata with applications to invariant checking. In *Proc. of ICGT '12*, pages 264–278. Springer, 2012. LNCS 7562.
3. Christoph Blume, H.J. Sander Bruggink, Martin Friedrich, and Barbara König. Treewidth, pathwidth and cospan decompositions with applications to graph-accepting tree automata. *Journal of Visual Languages & Computing*, 24(3):192–206, 2013.
4. H.J. Sander Bruggink and Barbara König. On the recognizability of arrow and graph languages. In *Proc. of ICGT '08*, pages 336–350. Springer, 2008. LNCS 5214.
5. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

6. Andrea Corradini, Barbara König, and Dennis Nolte. Specifying graph languages with type graphs, 2017. arXiv:1704.05263.
7. Andrea Corradini, Ugo Montanari, and Francesca Rossi. Graph processes. *Fundamenta Informaticae*, 26(3/4):241–265, 1996.
8. Bruno Courcelle. The monadic second-order logic of graphs I. Recognizable sets of finite graphs. *Information and Computation*, 85:12–75, 1990.
9. Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic, A Language-Theoretic Approach*. Cambridge University Press, June 2012.
10. Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Proc. of TACAS '06*, pages 287–302. Springer, 2006. LNCS 3920.
11. Jörg Endrullis and Hans Zantema. Proving non-termination by finite automata. In *RTA '15*, volume 36 of *LIPIcs*, pages 160–176. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.
12. Annegret Habel. *Hyperedge Replacement: Grammars and Languages*. Springer-Verlag, 1992. LNCS 643.
13. Annegret Habel and Karl-Heinz Pennemann. Nested constraints and application conditions for high-level structures. In *Formal Methods in Software and Systems Modeling. Essays Dedicated to Hartmut Ehrig, on the Occasion of His 60th Birthday*, pages 294–308. Springer, 2005. LNCS 3393.
14. Reiko Heckel and Annika Wagner. Ensuring consistency of conditional graph rewriting – a constructive approach. In *Proc. of the Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, volume 2 of *ENTCS*, 1995.
15. Jaroslav Nešetřil and Claude Tardif. Duality theorems for finite structures (characterising gaps and good characterisations). *Journal of Combinatorial Theory, Series B*, 80:80–97, 2000.
16. Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, May 2007. Reynolds Festschrift.
17. Fernando Orejas, Hartmut Ehrig, and Ulrike Prange. A logic of graph constraints. In *Proc. of FASE '08*, pages 179–198. Springer, 2008. LNCS 4961.
18. Karl-Heinz Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, Universität Oldenburg, May 2009.
19. Arend Rensink. Canonical graph shapes. In *Proc. of ESOP '04*, pages 401–415. Springer, 2004. LNCS 2986.
20. Arend Rensink. Representing first-order logic using graphs. In *Proc. of ICGT '04*, pages 319–335. Springer, 2004. LNCS 3256.
21. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*. World Scientific, 1997.
22. Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A diagrammatic formalisation of MOF-based modelling languages. In *Proc. of TOOLS EUROPE '09*, pages 37–56. Springer, 2009. LNBIP 33.
23. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS (ACM Transactions on Programming Languages and Systems)*, 24(3):217–298, 2002.
24. Dominik Steenken, Heike Wehrheim, and Daniel Wonisch. Sound and complete abstract graph transformation. In *Proc. of SBMF '11*, pages 92–107. Springer, 2011. LNCS 7021.