Towards a Navigational Logic for Graphical Structures ⋆

Leen Lambers[1], Marisa Navarro[2], Fernando Orejas[3], and Elvira Pino[3]

[1] Hasso Plattner Institut, University of Potsdam, Germany
Leen.Lambers@hpi.de
[2] Universidad del País Vasco (UPV/EHU), San Sebastián, Spain
marisa.navarro@ehu.es
[3] Universitat Politècnica de Catalunya, Barcelona, Spain
{orejas,pino}@cs.upc.edu

Abstract. One of the main advantages of the Logic of Nested Conditions, defined by Habel and Pennemann, for reasoning about graphs, is its generality: this logic can be used in the framework of many classes of graphs and graphical structures. It is enough that the category of these structures satisfies certain basic conditions.
In a previous paper [14], we extended this logic to be able to deal with graph properties including paths, but this extension was only defined for the category of untyped directed graphs. In addition it seemed difficult to talk about paths abstractly, that is, independently of the given category of graphical structures. In this paper we approach this problem. In particular, given an arbitrary category of graphical structures, we assume that for every object of this category there is an associated edge relation that can be used to define a path relation. Moreover, we consider that edges have some kind of labels and paths can be specified by associating them to a set of label sequences. Then, after the presentation of that general framework, we show how it can be applied to several classes of graphs. Moreover, we present a set of sound inference rules for reasoning in the logic.

1 Introduction

Graphs and graphical structures play a very important role in most areas of computer science. For instance, they are used for modeling problems or systems (as done, e.g., with the UML or with other modeling formalisms). Or they are also used as structures to store data in many computer science areas. In particular, in the last few years, in the database area, graph databases are becoming relevant in practice and partially motivate our work. A consequence of this graph ubiquity is that being able to express properties about graphical structures may be interesting in many areas of computer science.

We can use two kinds of approaches to describe graph properties. Obviously, we may use some standard logic, after encoding some graph concepts in the logic. For instance, this is the approach of Courcelle (e.g., [3]), who studied a graph logic defined

⋆ This work has been partially supported by funds from the Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER funds) under grant COMMAS (ref. TIN2013-46181-C2-1-R, TIN2013-46181-C2-2-R) and from the Basque Project GIU15/30, and grant UFI11/45.

Lambers, L., Navarro, M., Orejas, F., Pino, E. Towards a navigational logic for graphical structures. A: Symposium in Memory of Hartmut Ehrig. "Graph transformation, specifications, and nets: in memory of Hartmut Ehrig". Berlín: Springer, 2018, p. 124-141.
The final authenticated version is available online at https://doi.org/10.1007/978-3-319-75396-6_7

in terms of first-order (or monadic second-order) logic. The second kind of approach is based on expressing graph properties in terms of formulas that include graphs (and graph morphisms). The most important example of this kind of approach is the *logic of nested graph conditions* (LNGC), introduced by Habel and Pennemann [9] proven to be equivalent to the first-order logic of graphs of Courcelle. A main advantage of LNGC is its genericity, since it can be used for any category of graphical structures, provided that this category enjoys certain properties. This is not the case of approaches like [3] where, for each class of graphical structures, we would need to define a different encoding.

A main problem of (first-order) graph logics is that it is not possible to express relevant properties like "there is a path from node n to n'", because they are not first-order. As a consequence, there have been a number of proposals that try to overcome this limitation by extending existing logics (like [7, 10, 20]). Along similar lines, in [14] we extended the work presented in [12], allowing us to state properties about paths in graphs and to reason about them. Unfortunately, the work in [14] applies only to untyped unattributed directed graphs. As a continuation, in this paper we show how to overcome this limitation, extending some of the ideas in [14] to deal with arbitrary categories of graphical structures. Moreover, we allow for a more expressive specification of paths, assuming that edges have some kinds of labels and specifying paths using language expressions over these labels. Since this new generic logic allows one to describe properties of paths in graphical structures, we have called it a *navigational logic*.

The paper is organized as follows. In Sect. 2 we present some examples for motivation. In Sect. 3 we introduce the basic elements to define our logic and in the Sect. 4 we see how these elements can be defined in some categories of graphs, implicitly showing that our logic can be used in these categories. In Sect. 5 we introduce the syntax and semantics of our logic, including some proof rules that are shown to be sound. Completeness is not studied, because in our framework we implicitly assume that paths are finite, which means that our inference rules can not be complete [22]. However we conjecture that our rules will be complete in a more complex framework, where graphs may be infinite. Finally, in Sect. 6 we present some related and future work.

2 Motivation

In this section, we present and motivate the basic concepts required to introduce our navigational logic, that is, patterns with paths, and graph properties. In order to give some intuition and motivation, in Subsec. 2.1, we consider a toy example consisting of a network of airports connected by airline companies that operate between them, that is, a graph where nodes are airports and edges are direct flights from an airport to another. The example follows the framework presented in [14]. Then, in Subsec. 2.2 an example of a social network is introduced to motivate the extension of that framework, including labels in paths and edges, allowing us to specify the form of paths.

2.1 A First Navigational Logic Example

The graph in Fig. 1 represents a network with four airports: Barcelona (BCN), Paris (CDG), New York (JFK) and Los Angeles (LAX) and the six directed edges represent

the existing direct flights between these airports. In this scenario, a path (i.e. a concatenation of one or more edges) represents a connection from an airport to another by a sequence of, at least, one direct flight. For instance, BCN is connected to CDG and JFK by direct flights, whereas it is required to concatenate at least two flights to arrive to LAX from BCN. To express basic properties we use patterns, which are graphs extended with a kind of arrows that represent "paths" between nodes. For instance, in Fig. 2, we have two patterns that are present in the airport network in Fig. 1: The first pattern represents a connection from BCN to LAX, and, the second one a direct flight from BCN to CDG followed by a connection from CDG to LAX.

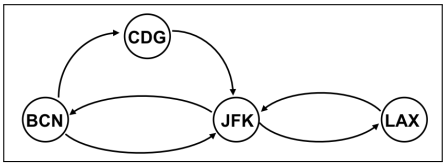


Fig. 1: A graph of connected airports

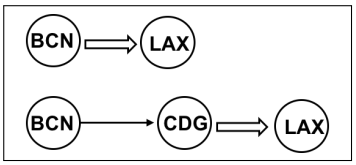


Fig. 2: Two connection patterns

Imagine that we want to state that there should be a connection from BCN to LAX with a stopover either in Paris or in New York and, moreover, that, in the former case, there should be a direct flight from BCN to CDG, whereas in the latter, the flight from JFK to LAX must be direct. The first graph condition in Fig. 3 states those requirements.

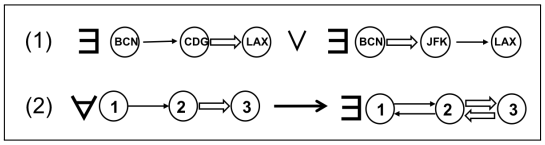


Fig. 3: Properties on airports networks

The second graph condition in Fig. 3 states that if there is a connection from an airport 1 to an airport 3 with a first stopover at an airport 2, it must be possible to go back from 3 to 1 with a similar flight plan that also stops at 2 but as the last stopover. Our network in Fig. 1 does not satisfy this requirement, since there is a connection from BCN to LAX with a first stopover in CDG but there are no direct flights from CDG to BCN.

2.2 Path Expressionss

In the framework described in the previous subsection [14], we can specify the existence of a path between two nodes, but we cannot provide any description of such path. For instance, suppose that edges are labelled with the name of the airline company operating that flight. In the described framework it is impossible to specify that there is a connection between BCN and LAX consisting of flights from the same company.

A simple way of dealing with this situation is to label paths with language expressions over an alphabet of edge labels. For instance, in most approaches (e.g., [1, 2, 4, 13,

23]), paths are labeled by regular expressions. The idea is that, if a pattern includes a path from node 1 to node 2 labelled with a language expression denoting a language L, then if a graph G includes that pattern, it must include some sequence of edges labelled with $l_1, \ldots, l_k$, such that $l_1 \cdot l_2 \cdot \ldots \cdot l_k \in L$.

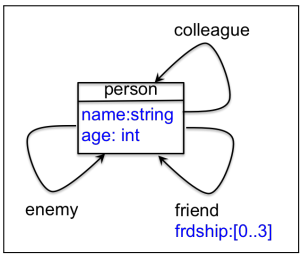


Fig. 4: A social network type graph

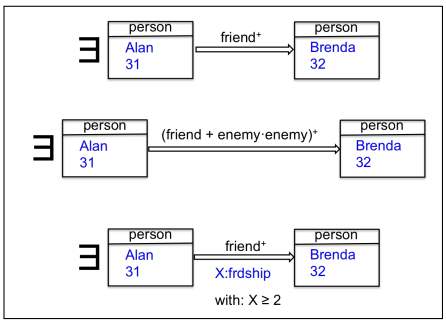


Fig. 5: Patterns of labeled connections

For example, in Fig. 4 we depict the type graph of a social network including nodes of type person and edges of type friend, enemy, and colleague. Then, in Fig. 5 we depict some conditions over this type graph. In the first two conditions we (implicitly) assume that edges are labeled with the name of their types. So, the first condition describes the existence of a path, consisting only of a sequence of edges of type friend between nodes having Alan and Brenda as their name attributes, while the second condition describes the existence of a path between Alan and Brenda, consisting of edges of type friend or two consecutive edges of type enemy. In the third condition, we implicitly assume that edges are labeled not only with types, but also with the values of their attributes and it describes the existence of a path between Alan and Brenda consisting of edges of type friend, whose friendship attribute is greater or equal to 2.

3 Patterns with Paths for Arbitrary Graphical Structures

In this paper our aim is to define a general framework that will allow us to express properties about arbitrary graphical structures and their paths, and to reason about them. A main problem is how to cope with this level of generality. In particular, given a specific class of graphs, like directed graphs (as in [14]) the notion of a path is clear. However, when working with an arbitrary category, that is supposed to represent any kind of graphical structure (e.g. graphs, Petri Nets or automata), we need some abstract notion of path that can accommodate the notion of path that we would have in each of these categories.

In principle, a path is a sequence of edges, but not all kinds of categories that we may consider have a proper notion of edge, although they may have something that we may consider to be similar. For instance, in a Petri Net we may consider that transitions play

the role of edges[4]. So our first step is to consider that we can associate to every category of graphical structures an associated category including an explicit edge relation. Then, it will be simple to define paths in these categories.

We assume that edges are labelled, so that we can use these labels to describe paths, as seen above. Moreover, we will assume that all edges in our graphical structures are defined over a universal set of nodes and a universal set of labels.

Definition 1 (Edges). *Given a set of labels Σ and a set of nodes V, the set of all possible Σ-labeled edges over V is* $\texttt{Edges}_{\Sigma,V} = V \times \Sigma \times V$.

Definition 2 (Edge-Labelled Structures). *Given a set of labels Σ and a set of nodes V, and given a category of graphical structures* $\underline{\texttt{Struct}}$ *with pushouts and initial objects, we say that* $\underline{\texttt{StructEdges}}_{\Sigma,V}$ *is its associated category of edge-labeled structures over V and Σ, EL-structures in short, if the following conditions hold:*

1. *The objects in* $\underline{\texttt{StructEdges}}_{\Sigma,V}$ *are pairs (S,E), where S is an object in* $\underline{\texttt{Struct}}$ *and E is a set of Σ-labeled edges over V.*
2. *A morphism $f : (S,E) \to (S',E')$ in* $\underline{\texttt{StructEdges}}_{\Sigma,V}$*, consists of functions $f = (f_s, f_v, f_e)$ such that*
   - *$f_s : S \to S'$ is a morphism in* $\underline{\texttt{Struct}}$*, and*
   - *$f_v : V \to V$ and $f_e : E \to E'$ satisfy for every $\langle n,l,n' \rangle \in E$ that $f_e(\langle n,l,n' \rangle) = \langle f_v(n), l, f_v(n') \rangle$.*
3. $\underline{\texttt{Struct}}$ *and* $\underline{\texttt{StructEdges}}_{\Sigma,V}$ *are isomorphic. Specifically, there must exist an isomorphism $\Psi : \underline{\texttt{Struct}} \to \underline{\texttt{StructEdges}}_{\Sigma,V}$.*

We will write just $\texttt{Edges}$ *and* $\underline{\texttt{StructEdges}}$ *whenever Σ and V are clear.*

Intuitively, the idea is that for each object S of an arbitrary category of graphical structures, we can associate a set of labelled edges that we assume that are implicit in S. Notice that if S is a graph this does not mean that it is a labelled graph. It only means that we can associate some kind of labels to its edges. For instance, if S is a typed graph, then we may consider that edges are labeled by their types. Similarly, as said above, if S is not a graph, like in the case of Petri Nets, this does not mean that S must include a proper notion of edge, but that we may consider that its edges are some of its elements.

Now, before defining the notion of pattern, we must first define the notion of path expressions, that is, the specification of a set of paths between two nodes. Moreover, we also define the notion of closure of a set of path expressions under composition and decomposition. Intuitively, a path is in the closure of a set of path expressions if its existence is a consequence of these expressions.

Definition 3 (Paths, Path Expressions and their Closure). *We define the set of path expressions over Σ, V,* $\texttt{PathExpr}_{\Sigma,V} = V \times 2^{\Sigma^*} \times V$. [5]

[4] But in Petri Nets we may also consider that both places and transitions play the role of the nodes in a graph and that the edges in a Petri Net are the arrows in the graphical representation of the net going from places to transitions or from transitions to places.

[5] Even if we may consider that empty paths are not really paths, assuming that every node is connected to itself through an empty path provides some technical simplifications.

A path *specified by a path expression* $pe = \langle n, L, m\rangle$*, is any triple* $\langle n, s, m\rangle \in V \times \Sigma^+ \times V$ *such that* $s \in L$*. Then,* $paths(pe)$ *denotes the set of paths specified by pe.*

If $R \subseteq \texttt{PathExpr}_{\Sigma,V}$ *is a set of path expressions, then the* closure *of R, written* R^+*, is the set of path expressions defined inductively:*

1. $R \subseteq R^+$.
2. Empty paths*: For every node n,* $\langle n, \varepsilon, n\rangle \in R^+$.
3. Composition*: If* $\langle n, L_1, m\rangle, \langle m, L_2, n'\rangle \in R^+$ *then* $\langle n, L_1L_2, n'\rangle \in R^+$.

Now, we can define what patterns are:

Definition 4 (Patterns). *Given a category* $\underline{\texttt{StructEdges}}_{\Sigma,V}$ *of EL-structures, its associated category of patterns,* $\underline{\texttt{StructPatterns}}_{\Sigma,V}$*, is defined as follows:*

1. *Objects are triples* $P = (S, E, PE)$ *where*
   - (S, E) *is in* $\underline{\texttt{StructEdges}}_{\Sigma,V}$ *and*
   - $PE \subseteq \texttt{PathExpr}_{\Sigma,V}$.
2. *A* pattern morphism $f : (S, E, PE) \to (S', E', PE')$*, is a morphism* $f : (S, E) \to (S', E')$ *in* $\underline{\texttt{StructEdges}}_{\Sigma,V}$ *such that, for every* $\langle n, L, m\rangle \in PE$*, there is a path expression* $\langle f_v(n), L', f_v(m)\rangle \in (E' \cup PE')^+$ *with* $L' \subseteq L$.

We will write just PathExpr *and* $\underline{\texttt{StructPatterns}}$ *whenever* Σ*, and V are clear.*

Notice that a Σ-labeled edge $\langle n, l, n'\rangle \in \texttt{Edges}$ can be considered a special kind of *unit* path expression $\langle n, \{l\}, n'\rangle$. As we may see in the definition above, even if there is an abuse of notation, given a set of edges E we will consider that E also denotes its associated set of unit path expressions.

Intuitively, a structure S can be considered a trivial pattern that is always present in S itself. However, technically, following the above definition S is not a pattern, but we can define a pattern, $Pattern(S)$, that intuitively represents S. Given S and its associated set of edges E, $Pattern(S) = (S, E, E^+)$, i.e. the path expressions in $Pattern(S)$ are precisely the paths defined by the edges in E. Conversely, any pattern (S, E, PE) where $PE = E^+$ can be considered equivalent to the structure S [6]. As a consequence, even if it is an abuse of notation, we will identify structures with their associated patterns. For instance, if we write that there is a pattern morphism $f : P \to S$, we really mean $f : P \to (S, E, E^+)$.

As we will see in Sect. 5.3, an (important) inference rule for reasoning in our logic is the unfolding rule that roughly says that if a pattern in a given condition includes the path expression $\langle n, L, n'\rangle$, then we may replace this pattern by another one that includes some of its possible decompositions. For instance, if $\langle n, a(c^*), n'\rangle$ is a path expression in P, we should be able to infer that the structures that satisfy the pattern either have an edge $\langle n, a, n'\rangle$, or an edge $\langle n, a, n_0\rangle$ followed by a path from n_0 to n' consisting of edges labelled by c. More precisely, from the condition $\exists P$ we should be able to infer $\exists P_1 \vee \exists P_2$, with $P_1 = P + \{\langle n, a, n'\rangle\}$ and $P_2 = P + \{\langle n, a, n_0\rangle, \langle n_0, c^+, n'\rangle\}$, where n_0 is a node that is not present in P and $P + s$ denotes the pattern obtained adding to P the paths and edges in the set s. The problem is how can we define formally P_1 and P_2. If

[6] That is $\underline{\texttt{Struct}}$ is embedded in $\underline{\texttt{Patterns}}$ via the functor *Pattern*.

$P = (S, E, PE)$, it would be wrong to define $P_1 = (S, E \cup \{\langle n, a, n' \rangle\}, PE)$, because E is the set of edges (implicitly) included in S, and $E \cup \{\langle n, a, n' \rangle\}$ can not also be the set of edges of S (unless S already included $\langle n, a, n' \rangle$, which in general will not be the case). Instead, we will assume that every specific framework is equipped with a procedure to define the structure S' that includes S and whose set of edges is $E \cup \{\langle n, a, n' \rangle\}$. This procedure is the mapping called `Unfold` in Def. 6, which actually does not return S', but the morphism $u : S \to S'$ that represents the inclusion of S in S'.

Before defining the unfolding construction, we define the decompositions that are associated to these unfoldings. First, a subdecomposition *sd* of a path expression $pe = \langle n, L, n' \rangle$ can be seen as a refinement of *pe*, in the sense that we may consider that *sd* defines a path expression $\langle n, L', n' \rangle$, where $L' \subseteq L$. For instance, in the example above $\{\langle n, a, n' \rangle\}$ and $\{\langle n, a, n_0 \rangle, \langle n_0, c^+, n' \rangle\}$ are subdecompositions of $\langle n, L, n' \rangle$. In the former case $L' = \{a\}$ and in the later case $L' = \{ac, acc, accc, \ldots\}$. Then, a decomposition of $\langle n, L, n' \rangle$ is a set of subdecompositions such that L coincides with the union of the languages associated to its subdecompositions. For, instance $\{\langle n, a, n' \rangle\}$ and $\{\langle n, a, n_0 \rangle, \langle n_0, c^+, n' \rangle\}$ are a decomposition of $\langle n, L, n' \rangle$, since $ac^* \equiv a|ac^+$.

Definition 5 (Path Expression Decomposition). *If $pe = \langle n, L, n' \rangle$ is a path expression, a* subdecomposition *sd of pe is a pair (L', s), where $L' \subseteq L$ and s is a set of edges and path expressions such that one of the following conditions holds:*

1. *$s = \{\langle n, l, n' \rangle\}$ and $L' = \{l\}$.*
2. *$s = \{\langle n, L_1, n_1 \rangle, \langle n_1, l, n_2 \rangle, \langle n_2, L_2, n' \rangle\}$ and $L' = L_1\{l\}L_2$, with $L_1, L_2 \subseteq \Sigma^*$* [7].

A decomposition *d of $pe = \langle n, L, n' \rangle$ is a finite set of subdecompositions, $d = \{sd_1, \ldots, sd_k\}$, with $sd_i = (L_i, s_i)$, for $i \in \{1, \ldots, k\}$, such that $(L_1 \cup \cdots \cup L_k) = L$.*

Definition 6 (Unfolding Morphisms). *We say that the category* `StructPatterns` *has* unfolding morphisms *if it is equipped with a function* `Unfold` *that given a pattern $P = (S, E, PE)$, a path expression $pe = \langle n, L, n' \rangle \in PE$, and a subdecomposition $sd = (L', s)$ of pe, it returns a morphism $u : P \to P'$, where $P' = (S', E', PE')$, such that:*

1. *$E' = E \cup \{\langle n_1, l, n_2 \rangle\}$ if $\langle n_1, l, n_2 \rangle \in s$, with $l \in \Sigma$, and*
2. *$PE' = PE \cup \{\langle n_1, L_i, n_2 \rangle | \langle n_1, L_i, n_2 \rangle \in s\}$.*
3. *For every morphism $f : P \to P_0$, with $P_0 = (S_0, E_0, PE_0)$, if there is a path expression $\langle f(n), L_0, f(n') \rangle \in PE_0$, with $E' \subseteq E_0$ and $PE' \subseteq PE_0$, then there is a morphism $h : P' \to P_0$, such that $f = h \circ u$.*

That is, if $u : P \to P' = \texttt{Unfold}(P, \langle n, L, n' \rangle, sd)$ then P' contains an unfolding of $\langle n, L, n' \rangle$ in P, built by adding new edges and paths to P. Notice that the component $u_s : S \to S'$ of every $u = \texttt{Unfold}(P, \langle n, L, n' \rangle, sd)$ must build the proper unfolded version S' of S so that the isomorphism $\Psi : \underline{\texttt{Struct}} \to \underline{\texttt{StructEdges}}$ is preserved.

From now on, we assume that our categories of patterns have unfolding morphisms.

[7] Notice that L_1 or L_2 may just consist of the empty string, in which case $n = n_1$ or $n' = n_2$, respectively.

4 Instantiation to Different Classes of Graphs

In this section we present how our general framework works in the context of some classes of graphs. We assume that the reader knows the (more or less) standard definitions in the literature of these classes of graphs (see, e.g., [5]). For simplicity, we assume that the languages used to label paths are defined by means of a regular expression. It should be clear that the three categories of graphs have pushouts and an initial object (the empty graph). Moreover, it is trivial to define `Unfold` for the three classes of graphs. In particular, given a pattern $P = (G, E, PE)$ where G is a graph of any of the three classes considered below, $\texttt{Unfold}(P, pe, sd)$ would return the inclusion $(G, E, PE) \hookrightarrow (G', E', PE')$, where G' is the graph obtained after adding to G the edges and new nodes in sd, E' is E plus these edges and PE' is PE plus the path expressions in sd. In all cases, V will be the class of all nodes of the given class of graphs.

4.1 Untyped Directed Graphs

The category of untyped directed graphs can be seen as an instance of our general framework, where:

- Σ is a set with a single label l.
- The isomorphism Ψ between this category and `StructEdges` that defines how a graph G is seen as an object in `StructEdges` is defined as follows:
  - For every graph $G = (V_G, E_G, s_G, t_G)$, $\Psi(G)$ is the EL-graph (G, E) where E is the set implicitly defined by E_G, that is, $E = \{\langle n, l, m\rangle \mid n, m \in V_G$, such that there exists $e \in E_G$ with $s_G(e) = n$ and $t_G(e) = m\}$.
  - For every morphism $f_s : G \to G'$, the corresponding EL-morphism $\Psi(f_s)$ is defined as (f_s, f_v, f_e) with $f_e(\langle n, l, m\rangle) = \langle f_v(n), l, f_v(m)\rangle$ for each $\langle n, l, m\rangle \in E$.

In this context, the only path expressions $PE \subseteq \texttt{PathExpr}$ are of the form $\langle n, L, m\rangle$, where L is a regular expression over the single label l. In particular, $\langle n, l^+, m\rangle$ would mean that there is a path from node n to node m formed by a non specified number of edges. For instance, the patterns in Fig. 2 could be seen as patterns in our framework if we consider that the paths depicted in those patterns are labeled with l^+.

4.2 Typed Graphs

To see that the category of typed graphs over a given type graph TG is an instance of our general framework, we assume that types have unique names.

- $\Sigma = \{t_1, t_2, \dots\}$ is a set of names for the types in TG.
- The isomorphism Ψ between this category and `StructEdges`, that defines how a typed graph $(G, type_G)$ is seen as an object in `StructEdges`, is defined as follows:
  - For every typed graph $(G, type_G)$, with $G = (V_G, E_G, s_G, t_G)$, $\Psi((G, type_G))$ is the EL-graph (G, E) where $E = \{\langle n, t, m\rangle \mid n, m \in V_G$, and $t = type_G(e)$ for some edge $e \in E_G$ with $s_G(e) = n$ and $t_G(e) = m\}$.
  - For every morphism f, the corresponding EL-morphism $\Psi(f)$ is defined as (f_s, f_v, f_e) with $f_e(\langle n, t, m\rangle) = \langle f_v(n), t, f_v(m)\rangle$ for each $\langle n, t, m\rangle$ in E.

For instance, the first two conditions in Fig. 5 include examples of patterns for the given type graph.

We may notice that a different category StructEdges (and consequently a different category StructPatterns) can be associated to typed graphs, if we consider that an edge e is labeled not by the name of its type, but by the pair (t_1,t_2) where t_1 and t_2 are the names of the types of the source and target nodes of e, respectively.

4.3 Attributed Graphs

Roughly, an attributed graph can be seen as some kind of labelled graph whose labels (the values of attributes) consist of values from a given data domain. There are several approaches to formalize this kind of graphs. In this paper we use the notion of *symbolic graph* ([15, 16]), because it is the most adequate approach to define patterns that include conditions on the attribute values. Symbolic graphs are defined using the notion of E-graphs, introduced in [5] as a first step to define attributed graphs. Intuitively, an E-graph is a kind of labelled graph, where both nodes and edges may be decorated with labels from a given set E. Being more precise, a symbolic graph G consists of an E-graph EG_G whose labels are seen as variables that represent the values of the given attributes, together with a formula Φ_G over these variables, used to constrain the possible values of the associated attributes. In general, a symbolic graph G can be considered a specification of a class of attributed graphs, since every model of Φ_G can be considered a graph specified by G. However, we can identify attributed graphs with *grounded symbolic graphs*, i.e. symbolic graphs G, where Φ_G is satisfied by just one graph (up to isomorphism).[8]

Then the category of attributed graphs (grounded symbolic graphs) can be seen as an instance of our general framework, where:

- Labels in Σ consist of the types of the edges together with their attributes and the variables associated to these attributes, i.e. labels are tuples $\langle t, x_1 : att_1, \ldots, x_k : att_k \rangle$, where $att_1, \ldots, att_k$ are the attributes of type t and $x_1, \ldots, x_k$ are their associated variables.
- The isomorphism ψ between this category and StructEdges is defined as follows.
  - For every symbolic graph G, $\psi(G)$ is the $\overline{EL}$-graph (G, E) where $E = \{\langle n, \langle t, x_1 : att_1, \ldots, x_k : att_k \rangle, m \rangle \mid$ there is an edge from n to m of type t with attributes $att_1, \ldots, att_k$ and $x_1, \ldots, x_k$ are their associated variables$\}$.
  - For every attributed morphism $f_s : G \to G'$, the corresponding EL-morphism $\psi(f_s)$ is defined as (f_s, f_v, f_e) with $f_e(\langle n, l, m \rangle) = \langle f_v(n), l, f_v(m) \rangle$ for each label l and each $\langle n, l, m \rangle$ in E.

In this case, if we want to put conditions on paths, as in the third pattern in Fig. 5, a path expression pe could be a triple $pe = \langle n, (exp, \Phi_{pe}), m \rangle$, where exp could be a regular expression over labels of the form $\langle t, x_1 : att_1, \ldots, x_k : att_k \rangle$ and Φ_{pe} would be a formula on the variables in exp. In this case, the third pattern in Fig. 5, the path from Alan to Brenda would be labelled with the regular expression $\langle friend, X : friendship \rangle^+$ together with the condition $X > 2$.

[8] In particular, we may consider that in a grounded symbolic graph G we have $\Phi_G \equiv (x_1 = v_1 \wedge \cdots \wedge x_k = v_k)$, for some values $v_1, \ldots, v_k$.

5 Reasoning about Navigational Properties

In this section we introduce in detail our logic. In the first subsection, we define its syntax and semantics. In the next one we show some properties that are used in the third subsection to define our inference rules and to show their soundness.

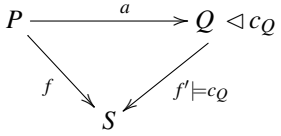
5.1 Nested Pattern Conditions, Models and Satisfaction

For our convenience, we express our properties using a nested notation [9] and avoiding the use of universal quantifiers.

Definition 7 (Conditions over Patterns, Satisfaction of Conditions). *Given a pattern P in* `StructPatterns`*, a* condition *over P is defined inductively as follows:*

- `true` *is a condition over P. We say that* `true` *has nesting level* 0*.*
- *For every morphism $a : P \to Q$ in* `StructPatterns`*, and every condition c_Q over Q with nesting level $j \geq 0$, $\exists(a, c_Q)$ is a condition over P, called literal, with nesting level $j+1$.*
- *If c_P is a condition over P with nesting level j, then $\neg c_P$ is a condition over P with nesting level j.*
- *If c_P and c'_P are conditions over P with nesting level j and j', respectively, then $c_P \wedge c'_P$ is a condition over P with nesting level $max(j, j')$.*

Given a structure S, we inductively define when the pattern morphism $f : P \to S$ satisfies *a condition c_P over P, denoted $f \models c_P$:*



- $f \models$ `true`.
- $f \models \exists(a, c_Q)$ *if there exists $f' : Q \to S$ such that $f' \circ a = f$ and $f' \models c_Q$.*
- $f \models \neg c_P$ *if* $f \not\models c_P$
- $f \models c_P \wedge c'_P$ *if* $f \models c_P$ *and* $f \models c'_P$.

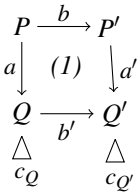
If c_P is a condition over P, we also say that P is the context *of c_P.*

Definition 8 (Navigational Logic: Syntax and Semantics). *The language of our Navigational Logic (NL) consists of all conditions over the initial pattern, $\emptyset$, in the category of patterns. Given a literal $\exists(a : \emptyset \to P, c_P)$ of NL, we also denote it by $\exists(P, c_P)$. A structure S satisfies a property c of NL if the unique morphism $i : \emptyset \to S$ satisfies c.*

5.2 Transformation by Lift and Unfolding

In this section we introduce some constructions that are used in our inference rules. The first one is the shift construction (introduced in [18, 19]) that allows us to translate conditions along morphisms.

Lemma 1 (Shift of Conditions over Morphisms). *Let Shift be a transformation of conditions inductively defined as follows:*

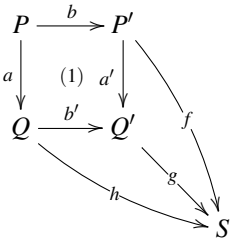


- $Shift(b, \texttt{true}) = \texttt{true}$.
- $Shift(b, \exists(a, c_Q)) = \exists(a', c_{Q'})$ *with* $c_{Q'} = Shift(b', c_Q)$ *such that (1) is a pushout.*
- $Shift(b, \neg c_P) = \neg Shift(b, c_P)$
- $Shift(b, c_P \wedge c'_P) = Shift(b, c_P) \wedge Shift(b, c'_P)$.

Then, for each condition c_P over P and each morphism $b : P \to P'$, $c_{P'} = Shift(b, c_P)$ is a condition over P' with smaller or equal nesting level, such that for each morphism $f : P' \to S$ we have that $f \models Shift(b, c_P) \Leftrightarrow f \circ b \models c_P$.

Proof. The proof uses double induction on the structure and the nesting level of conditions. The base case is trivial since $Shift(b, \texttt{true}) = \texttt{true}$, so they have the same nesting level $j = 0$, and every morphism satisfies $\texttt{true}$.

If c_P is not $\texttt{true}$, we proceed by induction on the nesting level of conditions. The base case is proven. Let c_P be $\exists(a, c_Q)$ of nesting level $j+1$ and suppose there is a morphism $f : P' \to S$ such that $f \models Shift(b, \exists(a, c_Q))$. That is, $f \models \exists(a', Shift(b', c_Q))$, according to the definition and diagram (1) below. This means there exists a morphism $g : Q' \to S$ such that $g \models Shift(b', c_Q)$ and $f = g \circ a'$. Then, since (1) is a pushout, we know that $f \circ b = g \circ a' \circ b = g \circ b' \circ a$ and, by induction, we have that $g \circ b' \models c_Q$. Therefore, $f \circ b \models c_P$.



Conversely, if $f \circ b \models c_P$ there exists $h : Q \to S$ such that $f \circ b = h \circ a$ and $h \models c_Q$. By the universal property of pushouts, there exists $g : Q' \to S$ such that $f = g \circ a'$ and $h = g \circ b'$ and, by induction, $g \models Shift(b', c_Q)$. Hence, $f \models Shift(b, \exists(a, c_Q))$. In addition, $\exists(a', Shift(b', c_Q))$ has nesting level smaller or equal to $j+1$ since, again as a consequence of the induction hypothesis $Shift(b', c_Q)$ has nesting level smaller or equal to j.

The rest of the cases easily follow from the induction hypothesis and the satisfaction and nesting level definitions. ■

In [18, 19], it is proved that, given two literals ℓ_1 and ℓ_2, a new literal ℓ_3 can be built (pushing ℓ_2 inside ℓ_1) that is equivalent to the conjunction of ℓ_1 and ℓ_2. Again, the following lemma is our version of that result:

Lemma 2 (Lift of Literals). *Let $\ell_1 = \exists(a_1, c_1)$ and ℓ_2 be literals with morphisms $a_i : P \to Q_i$, for $i = 1, 2$. We define the lift of literals as follows:*

$$Lift(\exists(a_1, c_1), \ell_2) = \exists(a_1, c_1 \wedge Shift(a_1, \ell_2))$$

Then, $f \models \ell_1 \wedge \ell_2$ if, and only if, $f \models Lift(\ell_1, \ell_2)$.

Proof. Assume $f : P \to S$ such that $f \models \exists(a_1, c_1 \wedge Shift(a_1, \ell_2))$. That is, there exists a morphism $g : Q_1 \to S$ such that $f = g \circ a_1$ and $g \models c_1 \wedge Shift(a_1, \ell_2)$. Then, this is equivalent to $f \models \ell_1$ and $f \models \ell_2$, since by Lemma 1 we have that $g \circ a_1 \models \ell_2$. ■

Note that when pushing ℓ_2 inside ℓ_1, the literal ℓ_2 can be positive or negative. But we will also need a special way of pushing a negative literal ℓ_2 inside a positive one ℓ_1 under some conditions, as shown in next lemma. In this case, the literal resulting from the lifting is just a consequence of the conjunction of ℓ_1 and ℓ_2.

Lemma 3 (Partial Lift of Literals). *Let $\ell_1 = \exists(a_1 : P \to Q_1, c_1)$ and $\ell_2 = \neg\exists(a_2 : P \to Q_2, c_2)$ such that there exists a morphism $g : Q_2 \to Q_1$ satisfying $a_1 = g \circ a_2$. We define the partial lift of literals as follows:*

$$\mathit{PLift}(\exists(a_1, c_1), \ell_2) = \exists(a_1, c_1 \wedge \mathit{Shift}(g, \neg c_2))$$

Then, $f \models \ell_1 \wedge \ell_2$ implies $f \models \mathit{PLift}(\ell_1, \ell_2)$.

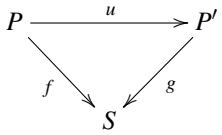
Proof. On the one hand, since $f \models \ell_1$, there exists a morphism $h_1 : Q_1 \to S$ such that $f = h_1 \circ a_1 = h_1 \circ g \circ a_2$, and $h_1 \models c_1$. On the other hand, since $f \models \ell_2$, it cannot exist a morphism $h_2 : Q_2 \to S$ satisfying both conditions $f = h_2 \circ a_2$, and $h_2 \models c_2$. Now, we consider the morphism $h_1 \circ g : Q_2 \to S$, which satisfies the first condition. Then necessarily $h_1 \circ g \models \neg c_2$ which implies $h_1 \models \mathit{Shift}(g, \neg c_2)$ by Lemma 1. Since $h_1 \models c_1 \wedge \mathit{Shift}(g, \neg c_2)$ we conclude that $f \models \mathit{PLift}(\ell_1, \ell_2)$. ■

Moreover, in addition to the lifting and partial lifting rules based on the *Shift* operation, we also need a rule that allows us to unfold the paths occurring in the contexts of conditions. For this purpose, in the rest of this subsection, we formalize the unfolding mechanism that we will use in the rest of the paper.

The following proposition establishes a key tautology in our logic with paths:

Proposition 1 (Unfolding Tautology). *Given a pattern $P = (S, E, PE)$, a pattern expression $pe = \langle n, L, n' \rangle \in PE$, and a decomposition d of pe, we have that the condition $\bigvee_{sd \in d} \exists(\texttt{Unfold}(P, \langle n, L, n' \rangle, sd), \texttt{true})$ is a tautology over P.*

Proof. We have to prove that every $f : P \to S \models \bigvee_{sd \in d} \exists(\texttt{Unfold}(P, \langle n, L, n' \rangle, sd), \texttt{true})$, where S is a structure. That is, we have to prove that there exist $sd \in d$ and $g : P' \to S$ such that $g \circ u = f$, where $u : P \to P' = \texttt{Unfold}(P, \langle n, L, n' \rangle, sd)$.



Since f is a pattern morphism and S is a structure, we have that $\langle f_v(n), L_0, f_v(n') \rangle \in E_S^+$, where L_0 includes only the sequence of labels of the path from $f_v(n)$ to $f_v(n)$, i.e., $L_0 = \{l_1 \dots l_k\} \subseteq L$. Then, since d is a decomposition of $\langle n, L, n' \rangle$, there is a sub-decomposition $sd \in d$, with $sd = (L', s)$ such that $L_0 \subseteq L'$. This means there exists $u = \texttt{Unfold}(P, \langle n, L, n' \rangle, sd)$ and, as a consequence of (3) in Def. 6, we have that the morphism g exists, such that $g \circ u = f$. ■

5.3 Inference Rules

We consider the following set of rules, where ℓ_2 means any (positive or negative) literal condition, c_P is any condition over $P = (S, E, PE)$, and $d \in D(pe)$ denotes that d is a decomposition of pe. Without loss of generality[9], we will assume that our conditions are in clausal form, that is, they are sets of disjunctions of *literals*, where a literal is either `true` or a condition $\exists(a, c_Q)$ or $\neg\exists(a, c_Q)$, where c_Q is again in clausal form.

$$(\texttt{Lift})\ \frac{\exists(a_1, c_1) \quad \ell_2}{\exists(a_1, c_1 \wedge \mathit{Shift}(a_1, \ell_2))}$$

$$(\texttt{Partial Lift})\ \frac{\exists(a_1, c_1) \quad \neg\exists(a_2, c_2)}{\exists(a_1, c_1 \wedge \mathit{Shift}(g, \neg c_2))}\ \text{if } a_1 = g \circ a_2$$

$$(\texttt{Unfolding})\ \frac{c_P}{\bigvee_{sd \in d} \exists(\texttt{Unfold}(P, pe, sd), \texttt{true})}\ \text{if } d \in D(pe) \text{ for } pe = \langle n, L, n' \rangle \in PE$$

$$(\texttt{Split Introduction})^{10}\ \frac{\neg\exists(a, c)}{\exists(a, \texttt{true})}\ \text{if } a \text{ is a split mono } ^{11}$$

$$(\texttt{False})\ \frac{\exists(a_1, \texttt{false})}{\texttt{false}}$$

Let us prove the soundness of the inference rules.

Theorem 1 (Soundness of Rules). *The above rules are sound.*

Proof. Let S be a structure and $f : P \to S$ be a pattern morphism. We need to prove that whenever f is a model of the premise(s) of a rule, it is also a model of the conclusion.

Lemmas 2 and 3 respectively prove the soundness of the Lift and Partial Lift rules, whereas soundness of the Unfolding rule is obtained from Proposition 1.

Soundness of Split Introduction is a consequence of the following property: If $a : P \to Q$ is a *split mono* then $\exists(a, \texttt{true})$ is equivalent to `true`. The reason is that every morphism $h : P \to S$ satisfies $\exists(a : P \to Q, \texttt{true})$, because the morphism $h \circ a^{-1} : Q \to S$ satisfies $(h \circ a^{-1}) \circ a = h \circ (a^{-1} \circ a) = h$, and $h \circ a^{-1}$ trivially satisfies `true`.

Finally, soundness of False is trivial, because there is no structure that satisfies $\exists(a_1, \texttt{false})$. ■

As a very simple example, let us now show that the set of three conditions in Fig. 6 is unsatisfiable.

Applying the Lift rule to conditions 1 and 2 we get condition 4 in Fig. 7, and applying again Lift to condition 4 and condition 3 we get condition 5 also in Fig. 7. Now, let us consider the inner conditions in condition 5, i.e. conditions 6 and 7 in Fig. 7. Applying Unfolding to the path expression labelled with a^+ in condition 6, we get condition

[9] In [18, 19] it is proved that we can transform any condition into a clausal form.

[10] This rule may seem not very useful, however in [14] it was needed to achieve completeness.

[11] A morphism $a : P \to Q$ is a split mono if it has a left inverse, that is, if there is a morphism a^{-1} such that $a^{-1} \circ a = id_P$.

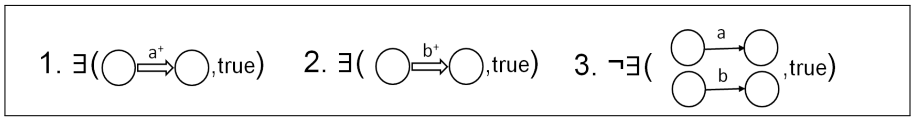


Fig. 6: Example of insatisfiable properties

8, and applying unfolding to the path expression labelled with b^+ in condition 8, we get condition 9. Now, applying Partial Lift to condition 7 and, successively, to the four conditions in the disjunction in condition 9, we get condition 10. Then applying four times the rule False to condition 10, we get `false`, which means that if we replace the inner conditions of condition 5, we get condition 11. Finally, if we apply the rule False to that condition we get `false`.

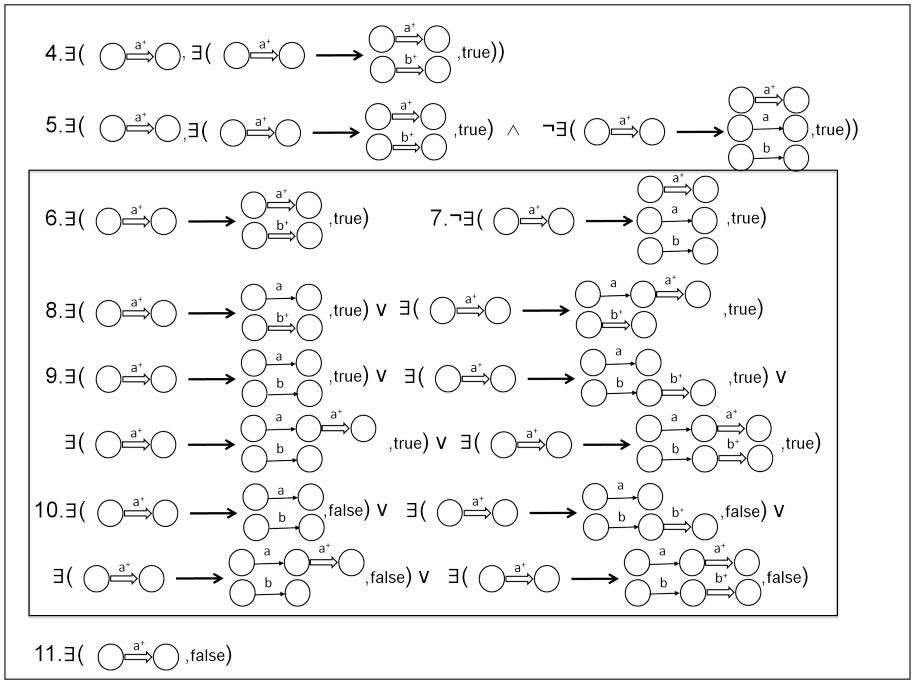


Fig. 7: Example of inferences

6 Related Work, Conclusion and Future Work

The idea of expressing graph properties by means of graphs and graph morphisms has its origins in the notions of graph constraints and application conditions [6, 8, 11]. In [21], Rensink presented a logic for expressing graph properties, closely related with the Logic of Nested Graph Conditions (LNGC) defined by Habel and Penneman [9]. First approaches to provide deductive methods to this kind of logics were presented in [17] for a fragment of LNGC, and by Pennemann [18, 19] for the whole logic. Among the extensions allowing us to state path properties, in [10], Habel and Radke presented a notion of HR^+ conditions with variables that allowed them to express properties about paths, but no deduction method was presented. Also, in [20], Poskitt and Plump proposed an extension of nested conditions with monadic second-order (MSO) properties over nodes and edges. Within this extension, they can define path predicates that allow for the direct expression of properties about paths between nodes, but without defining any deduction method. Finally, in [7], Flick extended the LNGC with recursive definitions using a μ notation and presented a proof calculus showing its partial correctness.

In [14] we presented an extension of LNGC, restricted to the case of directed graphs, including the possibility of specifying the existence of paths between nodes, together with a sound and complete tableau proof method for this logic.

The specification of paths by means of language expressions (in particular, regular expressions) is a usual technique in query languages for graph databases (e.g., [1, 2, 4, 13, 23]), but no associated logic is defined.

In this paper we have shown how to generalize the approach presented in [14] to arbitrary categories of graphical structures, including attributed typed graphs. In this sense, the results presented in this paper can be seen as a first step to define a logic underlying graph databases. The next obvious step will be showing the completeness of our inference rules.

Acknowledgements We are grateful to the anonymous reviewers for their comments that have contributed to improve the paper.

References

1. Barceló, P.: Querying graph databases. In: Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013. pp. 175–188 (2013), https://doi.org/10.1145/2463664.2465216
2. Barceló, P., Libkin, L., Reutter, J.L.: Querying graph patterns. In: Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece. pp. 199–210 (2011), https://doi.org/10.1145/1989284.1989307
3. Courcelle, B.: The expression of graph properties and graph transformations in monadic second-order logic. In: Rozenberg, G. (ed.) Handbook of Graph Grammars. pp. 313–400. World Scientific (1997), http://dl.acm.org/citation.cfm?id=278918.278932
4. Cruz, I.F., Mendelzon, A.O., Wood, P.T.: A graphical query language supporting recursion. In: Proceedings SIGMOD 1987 Annual Conference, San Francisco, California, May 27-29, 1987. pp. 323–330 (1987), http://doi.acm.org/10.1145/38713.38749

5. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph transformation. Springer-Verlag (2006)
6. Ehrig, H., Habel, A.: Graph grammars with application conditions. In: Rozenberg, G., Salomaa, A. (eds.) The Book of L, pp. 87–100. Springer, Berlin (1986)
7. Flick, N.E.: On correctness of graph programs relative to recursively nested conditions. In: Workshop on Graph Computation Models (GCM 2015), volume 1403, pages 97-112. CEUR-WS.org (2015)
8. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae 26, 287–313 (1996), https://doi.org/10.3233/FI-1996-263404
9. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science 19(2), 245–296 (2009), https://doi.org/10.1017/S0960129508007202
10. Habel, A., Radke, H.: Expressiveness of graph conditions with variables. ECEASST 30 (2010), http://dx.doi.org/10.14279/tuj.eceasst.30.404
11. Heckel, R., Wagner, A.: Ensuring consistency of conditional graph rewriting - a constructive approach. Electr. Notes Theor. Comput. Sci. 2, 118–126 (1995), https://doi.org/10.1016/S1571-0661(05)80188-4
12. Lambers, L., Orejas, F.: Tableau-based reasoning for graph properties. In: Graph Transformation - 7th International Conference, ICGT 2014, Held as Part of STAF 2014, York, UK, July 22-24, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8571, pp. 17–32. Springer (2014)
13. Libkin, L., Vrgoc, D.: Regular path queries on graphs with data. In: 15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012. pp. 74–85 (2012), https://doi.org/10.1145/2274576.2274585
14. Navarro, M., Orejas, F., Pino, E., Lambers, L.: A logic of graph conditions extended with paths. In: Workshop on Graph Computation Models (GCM 2016), Vienna (2016)
15. Orejas, F.: Attributed graph constraints. In: Graph Transformations, 4th International Conference, ICGT 2008. Lecture Notes in Computer Science, vol. 5214, pp. 274–288. Springer (2008)
16. Orejas, F.: Symbolic graphs for attributed graph constraints. J. Symb. Comput. 46(3), 294–315 (2011), https://doi.org/10.1016/j.jsc.2010.09.009
17. Orejas, F., Ehrig, H., Prange, U.: Reasoning with graph constraints. Formal Asp. Comput. 22(3-4), 385–422 (2010), https://doi.org/10.1007/s00165-009-0116-9
18. Pennemann, K.H.: Resolution-like theorem proving for high-level conditions. In: Graph Transformations, 4th International Conference, ICGT 2008. Lecture Notes in Computer Science, vol. 5214, pp. 289–304. Springer (2008)
19. Pennemann, K.H.: Development of Correct Graph Transformation Systems, PhD Thesis. Department of Computing Science, Univ. of Oldenburg (2009)
20. Poskitt, C.M., Plump, D.: Verifying monadic second-order properties of graph programs. In: Graph Transformation - 7th International Conference, ICGT 2014, Held as Part of STAF 2014, York, UK, July 22-24, 2014. Proceedings. pp. 33–48 (2014)
21. Rensink, A.: Representing first-order logic using graphs. In: Graph Transformations, Second International Conference, ICGT 2004. Lecture Notes in Computer Science, vol. 3256, pp. 319–335. Springer (2004)
22. Trakhtenbrot, B.A.: The impossibility of an algorithm for the decision problem on finite classes (in russian). Doklady Akademii Nauk SSSR, 70:569-572, 1950. English translation in: Nine Papers on Logic and Quantum Electrodynamics, AMS Transl. Ser. 2(23), 1–5 (1963)
23. Wood, P.T.: Query languages for graph databases. SIGMOD Record 41(1), 50–60 (2012), http://doi.acm.org/10.1145/2206869.2206879