A Logic of Graph Conditions Extended with Paths [*]

Marisa Navarro
Universidad del País Vasco (UPV/EHU)
San Sebastián, Spain
marisa.navarro@ehu.es

Fernando Orejas
Universitat Politècnica de Catalunya
Barcelona, Spain
orejas@cs.upc.edu

Elvira Pino
Universitat Politècnica de Catalunya
Barcelona, Spain
pino@cs.upc.edu

Leen Lambers
Hasso Plattner Institut
University of Potsdam, Germany
Leen.Lambers@hpi.de

In this paper we tackle the problem of extending the logic of nested graph conditions with paths. This means, for instance, that we may state properties about the existence of paths between some given nodes. As a main contribution, a sound and complete tableau method is defined for reasoning about this kind of properties.

1 Introduction

Being able to state properties about graphs and to reason about them is important in many areas of computer science, where graphs play a relevant role. For instance, in software and system modeling, where models are described using different graphical notations, graph properties may be used to describe properties of the given models, and reasoning tools may be used for model validation. Similarly, in the context of graph databases, for instance, graph properties could be used to express integrity constraints or just to express queries to the database. In that context reasoning tools may allow us check these constraints or to formally define the search engine to satisfy these queries.

Two kinds of approaches can be used to describe graph properties. On the one hand, we may use some standard logic, after encoding some graph concepts in the logic. For instance, this is the approach of Courcelle [1], who studied a graph logic defined in terms of first-order (or monadic second-order) logic, extended with a predicate $node(n)$ for stating that n is a node, and with a predicate $edge(n,n')$, for stating that there is an edge from node n no n'. 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 [4], which was proven to be equivalent to the first-order logic of graphs of Courcelle. Moreover Pennemann [12] showed that a specialized prover for their logic outperformed some standard provers, like Darwin or Vampire, when applied to graph formulas using Courcelle's logic.

A main problem of the LNGC is that we can only express *local* properties, but it is not possible to express relevant properties like "there is a path from node n to n'", or "the given graph is connected", which are second-order properties with respect to LNGC. In this sense, we extend LNGC with the possibility of stating that there are paths between some nodes. Moreover, we present a tableau method, extending the work in [8, 9], that is shown to be sound and complete for this new logic. It must be said that this

[*]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.

extension was not straightforward. First, as explained in Sect. 2.2, we have to consider the existence of infinite paths, then, it was not obvious to find the right notion of infinite path. Also, we had to deal with the problem that, in [8] a set of negative literals is always satisfiable, but not when dealing with paths. Finally, our formulas are more general than the formulas used in [8], not only because of paths, but also because we allow for the use of arbitrary morphisms, and not only monomorphisms and, the same happens with satisfaction, defined also in terms of arbitrary morphisms.

The paper is organized as follows. In Sect. 2, we introduce graphs, patterns and how they are related, and we also discuss the need to deal with infinite graphs with infinite paths as models of our logic. Then, in Sect. 3, we introduce the syntax and semantics of our logic (GPL), including the class of formulas in Conjunctive Normal Form, and some basic results that are needed in the rest of the paper. In Sect. 4, we present our tableau reasoning method and we show its soundness and completeness. Finally, in Sect. 5, we describe related work and we present some conclusions.

2 Preliminaries

2.1 Graphs, Patterns, and Graph Properties

Roughly, the idea of the graph logics that are based on the notion of graph constraints [7] is that basic properties state if a given pattern is present in a graph, where patterns are graphs themselves. For instance, the graph on the left of Fig. 1 describes the existence of three nodes, where 1 is connected to 2, 2 is connected to 3, and 3 is connected to 1. More precisely, if we work with a certain category of graphical objects (e.g. directed graphs), then a pattern P may be just an object in that category, and we consider that this pattern occurs in an object G if there is a morphism from P to G. However, in this paper, we work with patterns that may include the specification of the existence of paths between nodes of a graph. For instance, the pattern on the right of Fig. 1 describes the existence of three nodes, where 1 is connected to 2, 2 is connected to 3, and there is a path ($\Rightarrow$) from 3 to 1. That is, in this paper patterns are not exactly graphs, implying that they would belong to different categories, adding some complication. We solve this problem by defining a notion of pattern, where graphs can be seen as a special case.

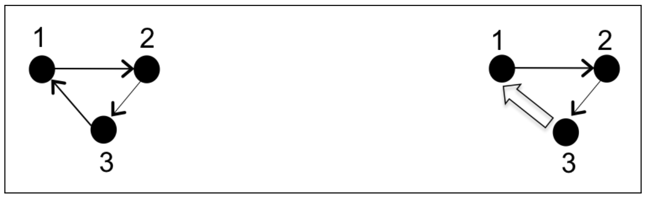

Figure 1: Patterns

Definition 1 (Graph Patterns, Complete Patterns, Graphs, and Pattern Morphisms) *A* graph pattern *P is a tuple $P = (Nodes_P, Edges_P, s_P, t_P, \Rightarrow_P)$, where*

- *$Nodes_P$ is a set of nodes,*
- *$Edges_P$ is a set of edges,*
- *$s_P : Edges_P \rightarrow Nodes_P$ and $t_P : Edges_P \rightarrow Nodes_P$ are the source and target functions, and*
- *$\Rightarrow_P \subseteq Nodes_P \times Nodes_P$ is the path relation.*

A pattern is complete *if $\Rightarrow_P$ includes the transitive closure $\rightarrow_P^+$ of the relation $\rightarrow_P \subseteq Nodes_P \times Nodes_P$, defined as $\langle n, n' \rangle \in \rightarrow_P$, if there is an edge $e \in Edges_P$ such that $s_P(e) = n$ and $t_P(e) = n'$. P^* denotes the* completion *of P, i.e. the smallest complete pattern that includes P.*

A graph *G is a complete pattern such that $\Rightarrow_G = \rightarrow_G^+$.*

A pattern morphism $f : P_1 \rightarrow P_2$, $f = (f_N, f_E)$ consists of two functions $f_N : Nodes_{P_1} \rightarrow Nodes_{P_2}$, $f_E : Edges_{P_1} \rightarrow Edges_{P_2}$ such that $n \Rightarrow_{P_1} n'$ implies $f_N(n) \Rightarrow_{P_2} f_N(n')$, $f_N \circ s_{P_1} = s_{P_2} \circ f_E$ and $f_N \circ t_{P_1} = t_{P_2} \circ f_E$.

Graph properties can be described by using certain diagrams including patterns, morphisms and logical symbols. For instance, we may consider that the property on the left of Fig. 2 states that there must not exist cycles in a graph (i.e. there is no node having a path to itself); and that the property on the right, where the pattern morphisms $h_1 : P_1 \rightarrow P_2$ and $h_2 : P_2 \rightarrow P_3$ are the obvious inclusions, states that there must exist a node with a loop, such that for all pairs of edges connected to that node, there exist two paths into some node completing a rectangle. Since we consider arbitrary (not necessary injective) morphisms, in some models these two paths may overlap. More precisely, a graph G would satisfy the latter condition if there exists a morphism $f : P_1 \rightarrow G$ such that for every morphism $f' : P_2 \rightarrow G$, with $f = f' \circ h_1$, there exists a morphism $f'' : P_3 \rightarrow G$, such that $f' = f'' \circ h_2$.

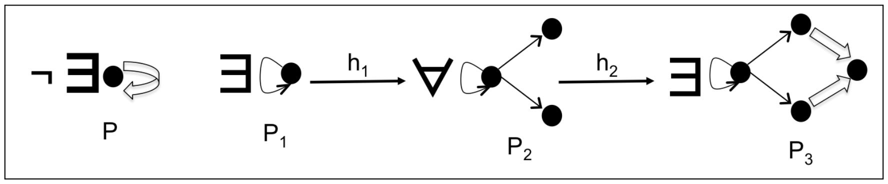

Figure 2: Graph properties

2.2 Infinite Graphs and Infinite Paths

According to Def. 1, graphs and patterns may be infinite, even if we are only interested in finite graphs. But conditions, like the ones depicted in Fig. 3, may specify infinite graphs[1]. In Fig. 3, the first two properties state that there must exist a node, let us call it 1, and that every node must be connected to another node. This means that 1 must be connected to a node 2, and 2 must be connected to a node 3, and so on. Moreover, all these nodes must be different. For instance, if 3 and 1 are the same node, there would be a path from 1 to itself, contradicting the third property. So there is no finite graph that satisfies these three properties, but a graph consisting of infinite nodes $0, 1, 2, \ldots, n, \ldots$, where for every node i there is an edge to node $i + 1$, would satisfy these properties.

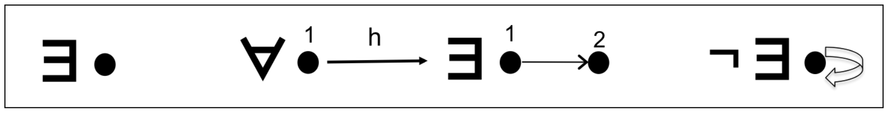

Figure 3: A set of properties having no finite models

We may consider that this set of properties is correct and assume that the models of a set of properties include infinite graphs, or that this example is erroneous (like a non-terminating program), because it

[1]Or, equivalently, they may have only infinite models.

has no finite models. The problem in the latter case is that finite graph satisfiability is not even semi-decidable, which means that no complete refutation (deduction) method can exist, if models are just finite graphs, as shown by Trakhtenbrot [15]. As a consequence, we consider that infinite graphs may be valid models in our logic.

Obviously, if we think that only finite graphs should be considered valid, this would be a limitation of our results. In particular, if a set of conditions is only satisfied by infinite graphs, we would be unable to refute it. We must claim, nevertheless, that working with infinite graphs has almost no consequences in practice, since the completeness of our deduction method implies that we can detect all unsatisfiable sets of conditions, whose inconsistency can be proven finitely.

Also according to Def. 1, paths are assumed to be finite, since we require in graphs that $\Rightarrow_G$ must be the transitive closure of $\rightarrow_G$. Unfortunately, in this case, satisfiability is not semi-decidable either. Hence, we also consider that the path relation may be interpreted by (finite or infinite) sequences of edges.

To end this section, we provide a definition of what an infinite path is, based on the idea that we are not interested in all kinds of infinite graphs or patterns, but only on those that can be built as the limit (technically, the colimit) of a sequence of finite patterns $\{P_i \xrightarrow{a_i} P_{i+1}\}_{i\geq 0}$. Then, roughly, an infinite path from a node n to a node n' would be the limit of two sequences of edges, $\{e_i\}_{i\geq 0}, \{e'_i\}_{i\geq 0}$, where the first sequence starts in n, the second sequence ends in n', and we could think that both sequences meet somewhere in the middle. Moreover, for each i, e_i, e'_i must be the image of edges o_i, o'_i, respectively, from pattern P_i, so that there is a path from the target of o_i to the source of o'_i. This is an unusual definition of infinite paths. A standard one would see an infinite path as a sequence of edges that starts in n and approaches n' infinitely. This definition is not adequate for us, because we need to be able to say, for instance, that a given (possibly infinite) path starts by some edges and finishes by some other edges.

Definition 2 (Infinite Paths, Graphs with Infinite Paths) *An* infinite path *from nodes n to n' in a pattern P consists of two sequences of edges in P, $\{e_i\}_{i\geq 0}, \{e'_i\}_{i\geq 0}$, with $s_P(e_0) = n, t_P(e'_0) = n'$, such that P, together with the collection of morphisms $\{P_i \xrightarrow{f_i} P\}_{i\geq 0}$, is the colimit of a sequence of morphisms $\{P_i \xrightarrow{a_i} P_{i+1}\}_{i\geq 0}$,*

$$\begin{tikzcd}
P_0 \arrow[r, "a_0"] \arrow[drr, "f_0"'] & \dots \arrow[r, "a_{i-1}"] & P_i \arrow[r, "a_i"] \arrow[d, "f_i"] & P_{i+1} \dots \arrow[dl, "f_{i+1}"] \\
 & & P &
\end{tikzcd}$$

where each P_i is finite, and for every i there are edges o_i, o'_i in P_i, $i \geq 0$, with:

- *$f_i(o_i) = e_i$ and $f_i(o'_i) = e'_i$.*
- *$a_i(t_{P_i}(o_i)) = s_{P_{i+1}}(o_{i+1})$ and $a_i(s_{P_i}(o'_i)) = t_{P_{i+1}}(o'_{i+1})$.*
- *$t_{P_i}(o_i) \Rightarrow_{P_i} s_{P_i}(o'_i)$.*

A complete pattern P is a graph with infinite paths *if whenever $n \Rightarrow_P n'$ either $n \rightarrow^+_P n'$ or there is an infinite path in P from n to n'.*

From now on, graphs with infinite paths will be just called graphs and they will be assumed to be the models of our logic.

3 Graph Properties expressed in GPL

In previous sections, we were informally writing graph conditions in examples to provide some intuition about our *Graph Pattern Logic* (GPL). In this section, we will define precisely their syntax and semantics. In the first subsection, we adapt the nested notation defined in [4], and in the second one we study the transformation of arbitrary conditions into Conjunctive Normal Form (CNF), and some other constructions that are used in our tableau method.

3.1 Graph Properties as Nested Conditions

For our convenience, we express graph properties using a nested notation [4] and avoiding the use of universal quantifiers.

Definition 3 (Conditions over Patterns, Satisfaction of Conditions) *Given a finite pattern P, 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 $a : P \to Q$ and condition c_Q over a finite pattern Q with nesting level $n \geq 0$, $\exists(a, c_Q)$ is a condition over P with nesting level $n+1$.*
- *If c_P is a condition over P with nesting level n, then $\neg c_P$ is a condition over P with nesting level n.*
- *If c_P and c'_P are conditions over P with nesting level n and n', respectively, then $c_P \wedge c'_P$ is a condition over P with nesting level $max(n, n')$. We restrict ourselves to finite conditions, i.e. each conjunction of conditions is finite.*

If G is a graph, we inductively define when a morphism $f : P \to G$ satisfies *a condition c_P over P, denoted $f \models c_P$:*

$$\begin{tikzcd}
P \arrow[rr, "a"] \arrow[dr, "f"'] & & Q \lhd c_Q \arrow[dl, "f' \models c_Q"] \\
& G &
\end{tikzcd}$$

- $f \models$ `true`.
- $f \models \exists(a, c_Q)$ *if there exists $f' : Q \to G$ 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 the pattern P, we also say that P is the context *of c_P.*

In many approaches (e.g. [8]), morphisms in conditions are assumed to be injective, but we consider (as in [4]) that they may be non-injective, since this gives more expressive power to the formalism. Similarly, in most approaches, it is assumed that the morphism f in the above definition is injective. We assume that f may be any morphism (a-satisfaction, according to [4]) and not only injective (m-satisfaction). There are two reasons for this. First, injectivity of f is similar to requiring, in classical logic, that variable assignments should be injective. Then, working with arbitrary morphisms simplifies the construction for the shifting lemma (cf. Lemma 1), replacing pair factorizations by pushouts. In particular, the use of pushouts, instead of pair factorizations, reduces considerably the size of proofs.

It is often argued that m-satisfaction is more intuitive than a-satisfaction. This is considered a practical argument in favour of m-satisfaction. However, in [4] it is proved that both forms of satisfaction are equivalent. In particular, given a set S of conditions, there is a transformation tr such that a graph G m-satisfies S if and only if G a-satisfies $tr(S)$. Hence, working with a-satisfaction or m-satisfaction is not relevant in practice.

Nested conditions are more general than needed, since they define properties on graph morphisms, rather than on graphs. Graph properties in our graph pattern logic *GPL* are conditions over the empty pattern, since a morphism $\emptyset \to G$ can be considered equivalent to the graph G. However, we must notice that if $\exists(a,c)$ is a graph property, in general c is an arbitrary condition over graph patterns.

Definition 4 (GPL Syntax, GPL Semantics) *The language of graph properties with paths consists of all conditions in GPL over the empty pattern $\emptyset$. Given an element $\exists(a,c_P)$ of GPL with $a : \emptyset \to P$, we also denote it by $\exists(P,c_P)$. A graph G satisfies a graph property c of GPL if the unique morphism $i : \emptyset \to G$ satisfies c.*

Notice that, if $a : P \to Q$ is a *split morphism* [2], then $\exists(a,\texttt{true})$ is equivalent to $\texttt{true}$. The reason is that every morphism $h : P \to G$ satisfies $\exists(a : P \to Q,\texttt{true})$, because the morphism $h \circ a^{-1} : Q \to G$ satisfies $h \circ a^{-1} \circ a = h$. But $\exists(a,c)$, with $a : P \to Q$ split, is not equivalent to c. In fact, $\exists(a,c)$ is a condition over P, while c is a condition over Q.

In [8], sets of negative literals $\neg\exists(a,c_Q)$ are always satisfiable if a is not a split morphism, since $id_P \models \neg\exists(a,c_Q)$ because there is not a morphism $b : Q \to P$ such that $b \circ a = id_P$. But, in our logic with paths, this is not true, as the following example shows.

Example 1 *Consider the following two negative literals:*

$$\ell_1 = \neg\exists(\overset{1}{\bullet}\Rightarrow\overset{2}{\bullet} \xrightarrow{b_1} \overset{1}{\bullet}\to\overset{2}{\bullet},\texttt{true}) \qquad \ell_2 = \neg\exists(\overset{1}{\bullet}\Rightarrow\overset{2}{\bullet} \xrightarrow{b_2} \overset{1}{\bullet}\to\bullet\Rightarrow\overset{2}{\bullet},\texttt{true})$$

b_1 and b_2 are not split but the condition $c = \ell_1 \wedge \ell_2$ is obviously unsatisfiable because for every graph G such that $\overset{1}{\bullet}\Rightarrow_G\overset{2}{\bullet}$, it is either satisfied that $\overset{1}{\bullet}\to_G\overset{2}{\bullet}$ or $\overset{1}{\bullet}\to_G\bullet\Rightarrow_G\overset{2}{\bullet}$.

3.2 Conjunctive Normal Form, Shifting, and Unfolding

In this section, we introduce the notion of clause and conjunctive normal form in GPL that is needed in the following section to present tableau reasoning [6] efficiently.

Definition 5 (Literals, CNF-conditions) *A* positive *(resp.* negative*)* literal *ℓ is a condition of the form $\exists(a,d)$ (resp. $\neg\exists(a,d)$), and a* clause *is a disjunction of literals.*

A condition c is in conjunctive normal form (CNF) *if it is either* $\texttt{true}$*, or* $\texttt{false}$*, or a conjunction of clauses $c = \wedge_{j\in J} c_j$, with $c_j = \vee_{k\in K_j} \ell_{jk}$, where for each literal $\ell_{jk} = \exists(a_{jk},d_{jk})$ or $\ell_{jk} = \neg\exists(a_{jk},d_{jk})$, a_{jk} is not a split morphism and d_{jk} is in CNF.*

In [12], Pennemann describes a procedure for transforming any condition into CNF. Since our framework is slightly more general than [12], a slight adaptation of that procedure is needed in our case. Essentially, we have to consider the case when a literal includes a split morphism, $a : P \to Q$. In this case, we use the following equivalences:

$$\exists(a,\texttt{true}) \equiv \texttt{true} \qquad \text{and} \qquad \exists(a,\exists(b,c)) \equiv \exists(b \circ a,c)$$

that allow us to eliminate split morphisms from conditions. In the rest of the paper, the notation $[c]$ will stand for the transformation of the condition c into its CNF form.

In the sections below, we make extensive use of the following shifting result that allows us to move a condition along a morphism.

[2] a is a split morphism if it is mono and has a left inverse. That is, there is a morphism a^{-1} such that $a^{-1} \circ a = id_P$.

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

$$\begin{tikzcd}
P \arrow[r, "b"] \arrow[d, "a"'] \arrow[dr, phantom, "(1)"] & P' \arrow[d, "a'"] \\
Q \arrow[r, "b'"'] & Q' \\
\triangle & \triangle \\
c_Q & c_{Q'}
\end{tikzcd}$$

- $\mathit{Shift}(b, \mathtt{true}) = \mathtt{true}$.
- $\mathit{Shift}(b, \exists(a, c_Q)) = \exists(a', c_{Q'})$ *with* $c_{Q'} = \mathit{Shift}(b', c_Q)$ *such that (1) is a pushout.*
- $\mathit{Shift}(b, \neg c_P) = \neg \mathit{Shift}(b, c_P)$
- $\mathit{Shift}(b, \wedge_{i\in I} c_{P_i}) = \wedge_{i\in I} \mathit{Shift}(b, c_{P_i})$.

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

In [11, 12], Pennemann proves 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.

Lemma 2 (Lift of Literals [11, 12]) *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: $\mathit{Lift}(\exists(a_1, c_1), \ell_2) = \exists(a_1, c_1 \wedge [\mathit{Shift}(a_1, \ell_2)])$. Then, $f \models \ell_1 \wedge \ell_2$ if, and only if, $f \models \mathit{Lift}(\ell_1, \ell_2)$.*

In our case, in addition to a lifting rule based on that 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.

Definition 6 (Unfolding) *If $\langle n, n' \rangle \in \Rightarrow_P$, we define the following inclusion morphisms:*

- $u^0_{P[n,n']} : P \hookrightarrow P[n \to n']$
- $u^1_{P[n,n']} : P \hookrightarrow P[n \to m_1 \Rightarrow n']$
- $u^2_{P[n,n']} : P \hookrightarrow P[n \Rightarrow m_2 \to n']$

where $P[\dots]$ is the least pattern including P, such that any node inside the brackets $[\dots]$ which is different from n and n' is assumed to be a fresh new node, and any relation inside $[\dots]$ holds in $P[\dots]$.

It is easy to see that if $\langle n, n' \rangle \in \Rightarrow_P$, the condition $\exists(u^0_{P[n,n']}, \mathtt{true}) \vee (\exists(u^1_{P[n,n']}, \mathtt{true}) \wedge \exists(u^2_{P[n,n']}, \mathtt{true}))$ is a tautology.

4 Tableaux Reasoning for Graph Properties with Paths

Tableaux are a standard refutation technique for theorem proving that is used in the context of many logics (see, e.g. [6]). A tableau is a tree that represents the set of formulas that we want to refute. A branch in a tableau is the representation of the conjunction of formulas in the branch, and a tableau represents the disjunction of all the formulas represented by its branches. Tableaux are constructed by some given rules. Some of these rules allow us to decompose the given formulas into subformulas that are placed in the tableau. And we also have inference rules whose results are also placed in the tableau. When we detect a contradiction in a branch of a tableau, we *close* the branch. If at some point all the branches of the tableau are closed, we consider that the given set of formulas has been refuted. In this sense, the role of the inference rules is to generate enough consequences, so that contradictions are made

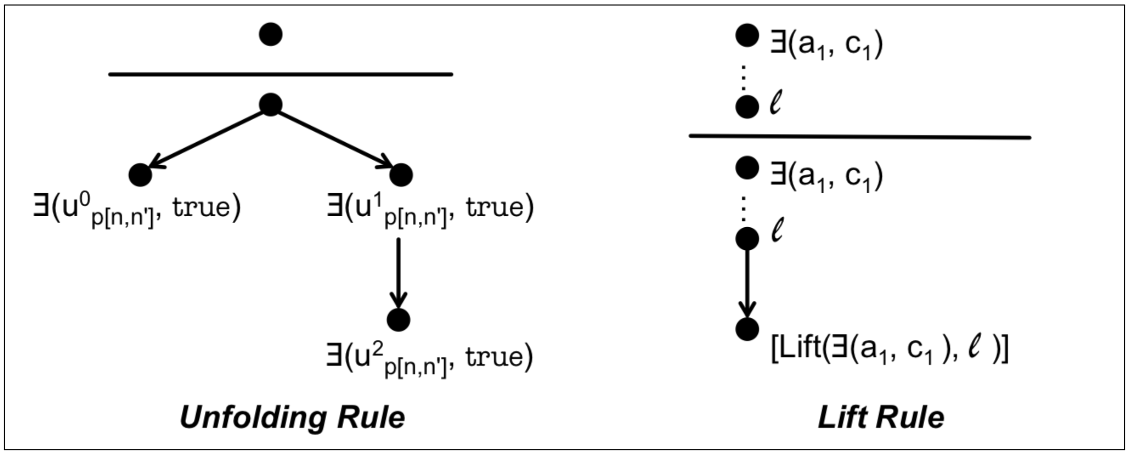

Figure 4: Tableau Rules

explicit. Conversely, if there are open branches and we have not postponed indefinitely the application of some inferences, we consider that the given set of formulas is satisfiable. Obviously, if satisfiability is undecidable, the construction of a tableau may never end. However, soundness and completeness would ensure that, the given formulas are unsatisfiable if and only if their associated tableau would be closed in finite time.

In our case, the nested structure of conditions makes it difficult to check satisfiability using standard tableaux. For this reason, we have developed a notion of nested tableaux that fits adequately in our framework. More precisely, in the first subsection of this section, we present the basic tableaux that we use, together with the inference rules that we use to build them; in the second subsection, we study our notion of nested tableaux; finally, in the third subsection we present our soundness and completeness results.

4.1 Basic Tableaux for Graph Conditions

As often done, the formulas in our tableaux are literals. The construction of a tableau for a condition c_P in CNF is roughly as follows. We start with a tableau consisting of the single node `true`, and for every clause $c_1 \vee \ldots \vee c_n$ in c_P we extend all the leaves in the tableau with n branches, one for each condition c_i. The rules that are specific for our logic are the lift rule based in Lemma 2 [11, 12], and the unfolding rule, based on the construction described in Def. 6. See Fig. 4 for both rules. In the lift rule, given two literals $\ell_1 = \exists(a_1, c_1)$ and ℓ_2 in the same branch, we add the literal $\ell_3 = \mathit{Lift}(\ell_1, \ell_2)$ to that branch. In the other case, if the context is P and $\langle n, n' \rangle \in \Rightarrow_P$, the unfolding rule allows us to extend this branch with two new branches, one branch with the literal $\exists(u^0_{P[n,n']}, \texttt{true})$, and the other branch with the two literals $\{\exists(u^i_{P[n,n']}, \texttt{true})\}_{i=1,2}$

Definition 7 (Tableau and branch of context *P*) *Given a finite pattern P, a* tableau of context *P is a finitely branching tree whose nodes are literals over P in CNF. A* branch *in a tableau T is a maximal path in T.*

Definition 8 (Tableau rules) *Given a condition c_P over P in CNF, a tableau of context P for c_P is constructed using the following rules:*

- **Initial rule:** *A tree consisting of the single node* `true` *is a tableau.*
- **Extension-rule** ($\vee$)**:** *If the clause $\ell_1 \vee \ldots \vee \ell_n$ is in c_P, we can extend all branches B with n descendants $\ell_1, \ldots \ell_n$.*
- **Lift rule** (*Lift*)**:** *If a given branch B includes the literals $\ell_1 = \exists(a_1, c_1)$ and ℓ_2 then we can extend B with the literal $[Lift(\ell_1, \ell_2)]$.*
- **Unfolding rule** (U)**:** *If $\langle n, n' \rangle \in \Rightarrow_P$, we can extend all branches B with 2 descendants, the first one with literal $\exists(u^0_{P[n,n']}, \texttt{true})$, and the second one with literal $\exists(u^1_{P[n,n']}, \texttt{true})$ followed by literal $\exists(u^2_{P[n,n']}, \texttt{true})$.*

Definition 9 (Open/closed branch) *In a tableau T a branch B is* closed *if B contains $\exists(a, \texttt{false})$ or* `false`*; otherwise, it is* open.

For instance, consider the condition $c_P = \ell_1 \wedge \ell_2$ with context P being the pattern $\overset{1}{\bullet}\Rightarrow\overset{2}{\bullet}$, and the literals ℓ_1, ℓ_2 from Example 1. Then, in Fig. 5 we have a closed tableau for c_P. Notice that the tableau can be closed because:

- $Shift(u^0_P, \ell_1) = \neg\exists(id_{\overset{1}{\bullet}\rightarrow\overset{2}{\bullet}}, \texttt{true}) \equiv \texttt{false}$.
- $Shift(u^1_P, \ell_2) = \neg\exists(id_{\overset{1}{\bullet}\rightarrow\bullet\Rightarrow\overset{2}{\bullet}}, \texttt{true}) \equiv \texttt{false}$.

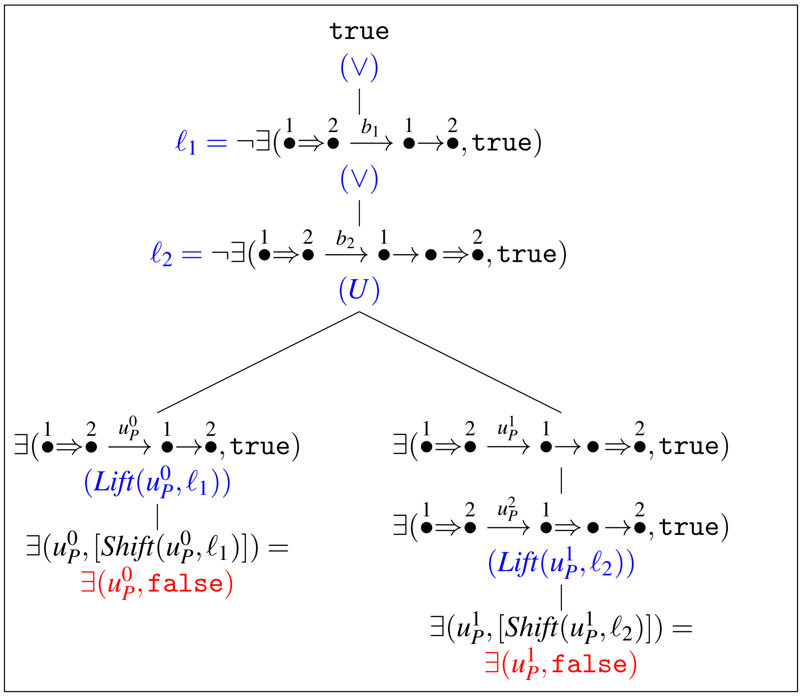

Figure 5: Closed Tableau.

The above rules may generate contradictions at the outer level of nesting for the literals in the given condition c_P, as seen in the example in Fig. 5. This is not enough, because contradictions may occur at inner levels of nesting. Instead of defining additional rules that would do something similar at any nesting level, what we do is to associate additional tableaux for each level. Our procedure can be roughly described as follows. First, we apply the extension rule until no new literals can be added. Next, if $\langle n,n'\rangle \in \Rightarrow_P$, we apply the unfolding rule in connection to, at least, one such pair of nodes on every branch of the tableau. Then, for every branch B, either we are able to close it, or using the lifting rule as many times as needed, we produce a literal that represents the conjunction of literals in B (see Lemma 2). If $\ell_1,\ldots,\ell_n$ are the literals in the branch, it is enough to choose a positive literal[3], say ℓ_1, that we call the *hook* of the branch, and to successively apply the lift rule, first to ℓ_1 and ℓ_2, next to the result and ℓ_3 and so on, until we have applied the lift rule to all the literals in the branch. Hence, at the end, the leaf of the branch will be the literal $\exists(a_1, c_1 \wedge_{\ell\in\{\ell_2,..,\ell_n\}} [\mathit{Shift}(a_1,\ell)])$. Finally, as we will see in the following section, we build a new tableau associated to the condition $c_1 \wedge_{\ell\in\{\ell_2,..,\ell_n\}} [\mathit{Shift}(a_1,\ell)]$ in the following nesting level, and so on.

Definition 10 (Semi-saturation, hook for a branch) *Given a* tableau *T for a condition c_P over P, we say that T is* semi-saturated *if:*

- *No new literals can be added to any branch in T using the extension rule,*
- *At least an unfolding rule associated to some pair of nodes $\langle n,n'\rangle \in \Rightarrow_P$, if any, has been applied and*
- *For every branch B in T one of the following conditions hold:*
  - *B is closed.*
  - *All the literals in B are negative and $\Rightarrow_P = \emptyset$*[4]
  - *There is a positive literal $\ell = \exists(a : P \to Q, c)$ in B, such that the literal in the leaf of B is $\ell_{leaf} = \exists(a,\ c \wedge_{\ell'\in B\setminus\{\ell\}} [\mathit{Shift}(a,\ell')])$. Then, we say that ℓ is the* hook *for the branch B in T.*

It should be obvious that, following the procedure described above, for any condition in CNF we can build a finite semi-saturated tableau.

To end this section, we show the soundness of the tableau rules.

Lemma 3 (Tableau soundness) *Given a condition c_P in CNF and a tableau T for this condition, if c_P is satisfiable then, so is T.*

The proof is by induction on the structure of the tableau. The base case is trivial. If a node has been added by using the extension rule, then satisfiability of the given condition implies satisfiability of the tableau. The case of the unfolding rule is a direct consequence of the fact that this rule just adds to the tableau a tautology. Finally, the case of the lift rule is a consequence of Lemma 2.

4.2 Nested Tableaux for Graph Properties with Paths

The idea of *nested tableaux* is that, for each open branch of a tableau T whose literal in the leaf is $\exists(a : P \to Q, c_Q)$, we open a new tableau T' to try to refute condition c_Q. Then, we say that $\exists(a, c_Q)$ is the *opener* for T'.

Nested tableaux have nested branches consisting of sequences of branches of a sequence of tableaux in the given nested tableau. While our basic tableaux are assumed to be finite, nested tableaux and nested

[3]If all the literals are negative, this would mean that the context P does not include any pair of nodes $\langle n,n'\rangle \in \Rightarrow_P$, since, otherwise, an unfolding rule would have generated a positive literal. In that case, no rule can be applied, but we can conclude that the given condition c_P is satisfiable. The reason is that the identity would be a model for all the literals in the branch.

[4]If $\Rightarrow_P$ would not be empty, the application of the unfolding rule would imply that the branch includes a positive literal.

branches may be infinite. As said above, we assume that the condition to (dis)prove is a graph property with paths, i.e. the given condition c is a condition over the empty graph $\emptyset$ and the initial tableau has context $\emptyset$.

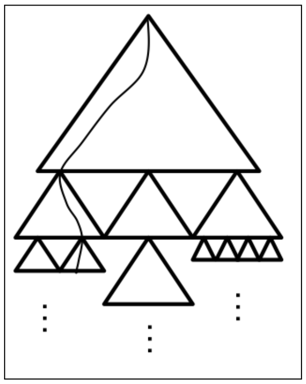

Figure 6: Nested tableau with nested branch

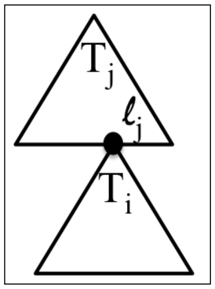

Figure 7: Opener

Definition 11 (Nested tableau, opener, nested branch, semi-saturation) *Let $(I, \leq, i_0)$ be a poset with minimal element i_0. A* nested tableau *NT is a family of triples $\{\langle T_i, j, \ell_j\rangle\}_{i,j\in I}$, where T_i is a tableau and $\ell_j = \exists(a_j, c_j)$, called the* opener *of T_i, is the literal of an open branch in T_j with $j < i$ (see Fig. 6 and 7).*

Moreover, we assume that there is an initial tableau T_{i_1} with $\langle T_{i_1}, i_0, true\rangle \in NT$ with context $\emptyset$. For any other $\langle T_i, j, \exists(a_j : P_j \to P_{j+1}, c_j)\rangle \in NT$, T_i has context P_{j+1}.

A nested branch NB *in a nested tableau $NT = \{\langle T_i, j, \ell_j\rangle\}_{i\in I}$ is a maximal sequence of branches $B_{i_1}, \ldots, B_{i_k}, B_{i_{k+1}}, \ldots$ from tableaux $T_{i_1}, \ldots, T_{i_k}, T_{i_{k+1}}, \ldots$ in NT starting with a branch B_{i_1} in the initial tableau T_{i_1}, such that if B_{i_k} and $B_{i_{k+1}}$ are two consecutive branches in the sequence then the leaf in B_{i_k} is the opener for $T_{i_{k+1}}$.*

Finally, NT is semi-saturated *if each tableau in NT is semi-saturated.*

Definition 12 (Nested tableau rules) *Given a graph property c in CNF, a* nested tableau *for c is constructed with the following rules:*

- **Initialization rule (I)***: Let c be a condition over $\emptyset$ and T_{i_1} be a tableau constructed for c following the rules in Def. 8, then $\{\langle T_{i_1}, i_0, true\rangle\}$ is a nested tableau for c.*
- **Nesting rule (N)** *If $NT = \{\langle T_i, j, \exists(a_j, c_j)\rangle\}_{i\in I}$ is a nested tableau for the condition c then $NT' = NT \cup \{\langle T_k, n, \exists(a_n, c_n)\rangle\}$ is a nested tableau for c, if $\exists(a_n, c_n)$, with $a_n : P_n \to P_{n+1}$, is a literal in a leaf of a tableau T_n in NT such that it is not the opener for any other tableau in NT, $k \notin I$, $k > n$ and T_k is a tableau for c_n.*

As in the case of standard tableaux, a closed nested branch represents an inconsistency detected between the literals in the branch, and an open branch represents, under adequate assumptions, a model of the original condition.

Definition 13 (Open/closed nested branch, nested tableau proof) *A nested branch NB, in a nested tableau NT for a graph property c in CNF, is* closed *if NB contains $\exists(a, false)$ or* `false`*; otherwise, it is* open. *A nested tableau is closed if* all *its nested branches are closed.*

A nested tableau proof *for (the unsatisfiability of) a graph property c in CNF, is a closed nested tableau NT for c according to the rules given in Def. 12.*

Example 2 (Closed nested tableau) *Let ℓ_1 and ℓ_2 be the literals in Ex. 1, and consider the new one $\ell_3 = \exists(\overset{1}{\bullet}\xrightarrow{b_3}\overset{1}{\bullet}\Rightarrow\overset{2}{\bullet}, true)$. Consider the condition $c_\emptyset$ over the $\emptyset$ context consisting in the conjunction of the following literals: $\exists(\emptyset \xrightarrow{a_1} \overset{1}{\bullet}\Rightarrow\overset{2}{\bullet}, \ell_1 \wedge \ell_2) \wedge \exists(\emptyset \xrightarrow{a_2} \overset{1}{\bullet}, \ell_3)$.*

Then, the tableau in Fig. 8 is the result of first applying the extension rule on $c_\emptyset$ and, then, the lift rule where the literal $\exists(a_1, \ell_1 \wedge \ell_2)$ has been chosen as hook. The tableau is open but semi-saturated as it satisfies Def. 10. That is, no path can be unfolded in the context $\emptyset$, and a leaf $\exists(a_1, \ell_1 \wedge \ell_2 \wedge [Shift(a_1, \exists(a_2, \ell_3))])$ has been generated taking into account all the literals in the branch.

Then, from that leaf, a new tableau with context $\overset{1}{\bullet}\Rightarrow\overset{2}{\bullet}$ is opened as in Fig. 9. Finally, the new tableau can be closed basically as we did for tableau in Fig. 5, since the contradiction arises from literals ℓ_1 and ℓ_2 if we apply the unfolding rule to $\overset{1}{\bullet}\Rightarrow\overset{2}{\bullet}$ and choose the unfolding literals as hooks.

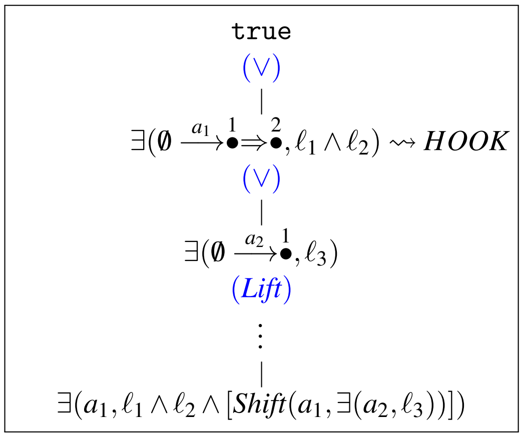
true
(∨)
$\exists(\emptyset \xrightarrow{a_1} \overset{1}{\bullet}\Rightarrow\overset{2}{\bullet}, \ell_1 \wedge \ell_2) \rightsquigarrow HOOK$
(∨)
$\exists(\emptyset \xrightarrow{a_2} \overset{1}{\bullet}, \ell_3)$
(*Lift*)
$\exists(a_1, \ell_1 \wedge \ell_2 \wedge [Shift(a_1, \exists(a_2, \ell_3))])$

Figure 8: Tableau of context $\emptyset$

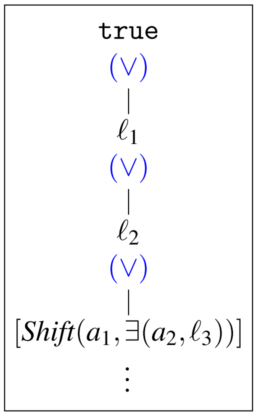

Figure 9: Tableau of context $\overset{1}{\bullet}\Rightarrow\overset{2}{\bullet}$

4.3 Soundness and Completeness

In this section we state that our tableau method is sound and complete. In particular, soundness means that if we are able to construct a nested tableau where all its branches are closed then we may be sure that our original condition c is unsatisfiable. Completeness means that if a *saturated* tableau includes an open branch, where the notion of saturation is defined below, then the original condition is satisfiable. Actually, the open branch provides the model that satisfies the condition. Due to lack of space, we only present here sketches of the proofs.

Theorem 1 (Soundness) *Given a graph property c in CNF, if there is a nested tableau proof for c then c is unsatisfiable.*

The proof uses Lemma 3 that states the soundness of the rules for constructing (basic) tableaux and the fact that if all branches of the nested tableau are closed then it is finite. In particular, we can prove by induction on the structure of NT that if c is satisfiable, then it must include an open branch. The base case is a consequence of Lemma 3. For the general case, assuming that the given nested tableau NT_i has an open nested branch NB, and using again Lemma 3, it can be shown that NB can be extended by a branch of the new tableau.

For the completeness proof, a notion of *saturation* of nested tableaux is needed, describing some kind of fairness that ensures that we do not postpone indefinitely some inference step. In this case, this means two issues: the choice of the hook for each tableau and that (in the limit) all possible unfoldings are performed. Roughly, if a (positive) literal is never chosen as a hook we will be unable to make inferences between that literal and other literals, especially, negative literals. Similarly, if some possible unfolding is never performed, we may fail to see a contradiction between some conditions.

Hence, to construct a saturated tableau, we have to use a fair strategy in the selection of hooks. This can be done by having, for each nested branch, a queue that includes the literals that are pending to be chosen as hooks. Similarly, to ensure that all unfoldings are performed, we may also keep queues of pending unfoldings. So, when opening a new tableau for a given nested branch, we would choose the hook for that tableau and we would perform an unfolding according to the given queues.

To prove the completeness theorem we use a key Lemma that shows that we can associate a graph G to any open nested branch in a nested tableau NT, so that, if NT is saturated, G is a model for NT. In particular, G is defined as the colimit of the morphisms associated to the sequence of tableau openers on the branch.

Lemma 4 (Canonical model for an open nested branch) *Let NB be an open nested branch in a saturated nested tableau NT for a graph property c in CNF. Let* $\emptyset \overset{a_0}{\to} \dots P_i \overset{a_i}{\to} \cdots \overset{a_{j-1}}{\to} P_j, \dots$ *be the corresponding sequence of contexts for NB, and let G be the colimit (cf. Def. 2) of the sequence* $\emptyset \overset{a_0}{\to} \dots P_i^* \overset{a_i}{\to} \cdots \overset{a_{j-1}}{\to} P_j^*, \dots$*, where* P^* *denotes the completion of P (cf. Def. 1). Then, G is a graph that satisfies all literals in NB.*

The proof of this Lemma consists of three parts. First, prove that G is indeed a graph (with infinite paths). Second, prove that G satisfies all the positive literals in NB. Finally, in the third part, we can show that if $\ell = \neg\exists(a : P_i \to Q, c)$ is a literal in NB, then there is a successor of ℓ in NB, $\ell' = \neg\exists(a' : P_j \to Q', c')$, obtained by means of the application of the lifting rule, such that a' is a split morphism. This property is the basis for showing that negative literals are also satisfied by G. Along the process, we need to track how conditions evolve along a nested branch. More precisely, if ℓ_i is a literal in a branch B_i that is part of a nested branch NB, and ℓ_i is not the hook for that branch, it will be transformed, via a lift rule, into an equivalent literal that will be part of the next branch B_j in NB, and so on. In this sense, the successor relation would tell us which is the literal ℓ_j that is equivalent to ℓ_i in any following branch B_j.

Theorem 2 (Completeness) *Given a graph property c in CNF, if c is unsatisfiable then there is a tableau proof for c.*

A proof can be done by showing that if there is no tableau proof for c, then c is satisfiable. More precisely, first, we can prove that there exists a saturated nested tableau NT for c that must include at least one open nested branch NB. Then, Lemma 4 implies that c is satisfiable.

5 Related Work and Conclusion

The idea of expressing graph properties by means of graphs and graphs morphisms has its origins in the notions of graph constraints and application conditions [2, 7, 3]. In [14], Rensink presented a logic for expressing graph properties, closely related with the Logic of Nested Graph Conditions (LNGC) of Habel and Penneman [4]. Moreover, in [5], 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. First approaches to provide deductive methods to this kind of logics were presented in [10] for a fragment of LNGC, and by Pennemann [11, 12] for the whole logic. Unfortunately, Penneman was unable to show the completeness of his approach. In [13], Poskitt and Plump propose an extension of nested conditions with monadic second-order (MSO) properties over nodes and edges, In particular, they can define path predicates that allow for the direct expression of properties about arbitrary-length paths between nodes. They also define a weakest precondition system (a la Pennemann) for verification. However, again this formalism lacks a deduction method. Lambers and Orejas [8] defined the nested tableaux method used in this paper and were able to show the completeness of Pennemann's inference rules. Recently, in [9], Navarro, Orejas and Pino, presented a complete proof system for reasoning about XML patterns, including paths.

Our work extends [8, 9], but this extension is far from straightforward. First, we had to find the right notions of graphs and patterns including a non-standard notion of infinite paths. These notions are quite more complex than the ones used in [9], where we were dealing with trees. Second, we had to deal with the fact that, in the new logic, sets of negative conditions may be unsatisfiable. Finally, the use of arbitrary morphisms, instead of monomorphisms, in formulas and satisfaction added some more difficulties.

In this paper, we have presented an extension of the LNGC including the possibility of specifying the existence of paths between nodes, and we have presented a sound and complete tableau proof method for this logic. Moreover, the formulas in this logic include arbitrary morphisms, and not only monomorphisms, which gives the logic additional expressive power. Similarly, satisfaction is also defined in terms of arbitrary morphisms, which makes proofs shorter, since the shift construction is defined just in terms of a pushout, instead of using pair factorization. More precisely, the result of shifting a literal in our context is, in general, just another literal. However, when using pair factorization, the result of shifting a literal is, in general, a disjunction of literals, which will cause additional branching in tableau proofs. In the future, we plan to relate this logic with the query languages that are used in graph databases.

References

[1] Bruno Courcelle (1997): *The Expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic*. In Grzegorz Rozenberg, editor: *Handbook of Graph Grammars*, World Scientific, pp. 313–400.

[2] Hartmut Ehrig & Annegret Habel (1986): *Graph Grammars with Application Conditions*. In G. Rozenberg & A. Salomaa, editors: *The Book of L*, Springer, Berlin, pp. 87–100.

[3] Annegret Habel, Reiko Heckel & Gabriele Taentzer (1996): *Graph Grammars with Negative Application Conditions*. *Fundamenta Informaticae* 26, pp. 287–313.

[4] Annegret Habel & Karl-Heinz Pennemann (2009): *Correctness of high-level transformation systems relative to nested conditions*. *Mathematical Structures in Computer Science* 19(2), pp. 245–296.

[5] Annegret Habel & Hendrik Radke (2010): *Expressiveness of graph conditions with variables*. *ECEASST* 30.

[6] Reiner Hähnle (2001): *Tableaux and Related Methods*. In John Alan Robinson & Andrei Voronkov, editors: *Handbook of Automated Reasoning*, Elsevier and MIT Press, pp. 100–178.

[7] Reiko Heckel & Annika Wagner (1995): *Ensuring consistency of conditional graph rewriting - a constructive approach*. *Electr. Notes Theor. Comput. Sci.* 2, pp. 118–126.

[8] Leen Lambers & Fernando Orejas (2014): *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* 8571, Springer, pp. 17–32.

[9] Marisa Navarro, Fernando Orejas & Elvira Pino (2015): *Satisfiability of Constraint Specifications on XML Documents*. In Narciso Martí-Oliet, Peter Csaba Ölveczky & Carolyn L. Talcott, editors: *Logic, Rewriting,*

and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday, *Lecture Notes in Computer Science* 9200, Springer, pp. 539–561.

[10] Fernando Orejas, Hartmut Ehrig & Ulrike Prange (2010): *Reasoning with graph constraints*. *Formal Asp. Comput.* 22(3-4), pp. 385–422.

[11] Karl-Heinz Pennemann (2008): *Resolution-Like Theorem Proving for High-Level Conditions*. In: *Graph Transformations, 4th International Conference, ICGT 2008*, *Lecture Notes in Computer Science* 5214, Springer, pp. 289–304.

[12] Karl-Heinz Pennemann (2009): *Development of Correct Graph Transformation Systems, PhD Thesis*. Dept. Informatik, Univ. Oldedburg.

[13] Christopher M. Poskitt & Detlef Plump (2014): *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.

[14] Arend Rensink (2004): *Representing First-Order Logic Using Graphs*. In: *Graph Transformations, Second International Conference, ICGT 2004*, *Lecture Notes in Computer Science* 3256, Springer, pp. 319–335.

[15] B. A. Trakhtenbrot (1963): *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), pp. 1–5.