A Logic of Graph Constraints

Fernando Orejas[1], Hartmut Ehrig[2], and Ulrike Prange[2]

[1] Dpto de L.S.I., Universitat Politècnica de Catalunya, Campus Nord, Mòdul Omega, Jordi Girona 1-3, 08034 Barcelona, Spain
orejas@lsi.upc.edu

[2] Fak. IV, Technische Universität Berlin, Franklinstrasse 28/29, 10587 Berlin, Germany
{ehrig,uprange}@cs.tu-berlin.de

Abstract. Graph constraints were introduced in the area of graph transformation, in connection with the notion of (negative) application conditions, as a form to limit the applicability of transformation rules. However, we believe that graph constraints may also play a significant role in the area of visual software modelling or in the specification and verification of semi-structured documents or websites (i.e. HTML or XML sets of documents). In this sense, after some discussion on these application areas, we concentrate on the problem of how to prove the consistency of specifications based on this kind of constraints. In particular, we present proof rules for three classes of (increasingly more powerful) graph constraints and show that our proof rules are sound and (refutationally) complete for each class.

1 Introduction

Graph constraints were introduced in the area of graph transformation, together with the notion of (negative) application conditions, as a form to limit the applicability of transformation rules [7,9,12,6,10,11]. More precisely, a graph constraint is the graphical description of some kind of pattern that must be present (or must not be present) on the graphs that we are transforming. In particular, a transformation would be illegal if the resulting graph would violate any of the given constraints. Graph constraints have been studied mainly in connection with negative application conditions. These conditions are constraints that are associated to the left-hand side or the right-hand side of a graph transformation rule. Then, one such rule would be applicable to a given graph if the left-hand side application conditions are satisfied by the given graph (or rather by the rule matching) and the right-hand side application conditions are satisfied by the result of the transformation. In this context, most of the above-mentioned work has been related to the extension of the basic graph transformation concepts and results to the use of application conditions and constraints and to show how one can transform a set of constraints into application conditions for the given transformation rules. Other work related to these notions has studied the detection of conflicts for graph transformation with application conditions [15], or the expressive power of some kinds of graph constraints [17].

We believe that graph constraints can go beyond their use in connection to graph transformation. More precisely, there are two areas in which we think that graph constraints may play an interesting role. The first one is the area of visual software modelling. The second one is the specification and verification of classes of semi-structured documents, including the specification and verification of websites (i.e. HTML or XML sets of documents).

In the area of visual modelling, especially in the context of UML modelling, models are designed using different kinds of diagrams. However, if we have to impose some specific constraints on the models then we have to use a textual notation as OCL. We consider that this situation is quite inconvenient. Especially in the case when we want to express constraints on the structure of the model, we think that using a graphical notation which is close to the visual description of the model is much more clear and intuitive than using some textual expression where one has to previously code or represent that structure.

On the other hand, we know two kinds of approaches for the specification and verification of semi-structured documents. The first one [2,8] is based on extending a fragment of first-order logic allowing us to refer to the components of the given class of documents (in particular, using XPath notation). This approach, in our opinion, poses two kinds of problems. On one hand, from a technical point of view, the extension of first-order logic to represent XML patterns has to make use of associative-commutative operators. This may make deduction difficult to implement efficiently, since using unification in inference rules may be very costly (in general, two arbitrary atoms may have a doubly exponential amount of most general unifiers). As a consequence, the approaches presented in [2,8] present specification languages that allow us to specify classes of documents and tools that allow us to check if a given document (or a set of documents) follows a specification. However, they do not consider the problem of defining deductive tools to analyze specifications, for instance for looking for inconsistencies. On the other hand, from a pragmatic point of view, XPath expressions can be quite verbose and this may make the resulting specifications unpleasant to read and to write.

The other approach that we know [13], which we consider especially interesting, has a more practical nature. Schematron is a language and a tool that is part of an ISO standard (DSDL: Document Schema Description Languages). The language allows us to specify constraints on XML documents by describing directly XML patterns (using XML) and expressing properties about these patterns. Then, the tool allows us to check if a given XML document satisfies these constraints. However, we consider that there are two problems with this approach. The most important one is that this work lacks proper foundations. The other one is that the kind of patterns that can be expressed in the Schematron language could be a bit limited. On the other hand, as in the approaches mentioned above, Schematron provides no deductive capabilities.

In this paper we start the study of graph constraints as a specification formalism. In particular, we study their underlying logic, providing inference rules that would allow us to prove the consistency (or satisfiability) of specifications. Actually, we show that these rules are sound and refutationally complete for the class of constraints considered. It must be noted that, as it is well-known, the fact that our inference rules are refutationally complete means that we have a complete method to prove consequences of our specifications. In particular, if we want to check if a given property is a consequence of a specification then it is enough to see if the given specification, together with the negation of the property, is inconsistent.

It must also be noted that the results that we present are quite more general than what they actually may seem. Following recent work on algebraic graph transformation (see, e.g., cite [5]), our results apply not only to plain graphs, but generalize to a large class of structures including typed and attributed graphs (we discuss this issue in more detail in the conclusion). In particular, instead of a logic of *graph* constraints we could speak of a logic of *pattern* constraints, since our results would also apply to reasoning about constraints based on other kinds of patterns, like XML patterns. In this sense, we consider that the work that we present in this paper provides the basis for defining the logical foundations of Schematron, and for extending it with more powerful constraints and with deduction capabilities. In particular, the XML patterns that are used in Schematron can be seen just as the textual representation (or, rather, the XML representation) of a subclass of the graph constraints that we consider. In particular, our work could be used to provide deductive capabilities to analyze the consistency of Schematron specifications.

The work that we present is not the first logic to reason about graphs. In particular, with different aims, Courcelle in a series of papers has studied in detail the use of monadic second-order logic (MSOL) to express graph properties (for a survey, see [4]). That logic is quite more powerful than the one that we study in this paper. For instance, we cannot express global properties about graphs (e.g that a graph is connected), but using MSOL we can. Actually, we think that MSOL is too powerful for the kind of applications that we have in mind. On the other hand, in [3] Courcelle's logic is extended with temporal operators. In this case, the intention is to present a logic that can be used for the verification of graph transformation systems. Again, this logic goes far beyond our aims.

The paper is organized as follows. In the following section we present the kind of graph constraints that we consider in this paper and present a small example to motivate their use in connection with visual modelling or website specification. This example will be used as a running example in the rest of the paper. The following section is the core of the paper. It presents inference rules for three classes of graph constraints with increasing expressive power, showing, for all cases, their soundness and completeness. Finally, in the conclusion we discuss several issues concerning the results that we present, in particular, their generality and different issues concerning the possible implementation of a deductive tool.

2 Graphs and Graph Constraints

In this section we present the basic notions that are used in this paper. First we present some notation and terminology needed, and then, in the second subsection we introduce the kind of graph constraints that we consider. For simplicity, we present our definitions in terms of plain directed graphs, although in some examples, for motivation, we deal with typed or attributed graphs. Anyhow, following the approach used in [5], it is not difficult to show that our results generalize to a large class of (graphical) structures, including typed, labelled or attributed graphs. In Section 4 we discuss this issue in more detail.

2.1 Graphs

As said above, all our notions and results will be presented in terms of plain directed graphs, i.e.:

Definition 1 (Graphs). *A* graph $G = (G_V, G_E, s, t)$ *consists of a set* G_V *of nodes, a set* G_E *of edges, a source function* $s : G_E \to G_V$*, and a target function* $t : G_E \to G_V$*.*

It may be noted that we do not explicitly state that the sets of nodes and edges of a graph are finite sets. That is, according to our definition, unless it is explicitly stated, graphs may be infinite. This issue is discussed in some detail in Section 3.

All over the paper we will have to express that a certain graph G_1 is included into another graph G_2. Obviously, we could have done this through a subgraph relationship. However, G_2 may include several instances of G_1. For this reason, in order to be precise when specifying the specific instance in which we may be interested, we will deal with these inclusions using the notion of graph monomorphism:

Definition 2 (Graph morphisms). *Given the graphs* $G = (G_V, G_E, s, t)$ *and* $G' = (G'_V, G'_E, s', t')$*, a* graph morphism $f : G \to G'$ *is a pair of mappings,* $f_V : G_V \to G'_V, f_E : G_E \to G'_E$ *such that f commutes with the source and target functions, i.e. the diagrams below are commutative.*

$$\begin{tikzcd}
G_E \arrow[r, "s"'] \arrow[d, "f_E"] & G_V \arrow[d, "f_V"] & G_E \arrow[r, "t"'] \arrow[d, "f_E"] & G'_V \arrow[d, "f_V"] \\
G'_E \arrow[r, "s'"'] & G'_V & G'_E \arrow[r, "t'"'] & G'_V
\end{tikzcd}$$

A graph morphism $f : G \to G'$ *is a* monomorphism *if* f_V *and* f_E *are injective mappings.*

In several results of the paper, given two graphs G, G' we will need to put them together in all possible ways. This will be done using the construction $G \otimes G'$:

Definition 3 (Jointly surjective morphisms). *Two graph morphisms* $m : H \to G$ *and* $m' : H' \to G$ *are* jointly surjective *if* $m_V(H_V) \cup m'_V(H'_V) = G_V$ *and* $m_E(H_E) \cup m'_E(H'_E) = G_E$*.*

Given two graphs G and G′, the set of all pairs of jointly surjective monomorphisms from G and G′ is denoted $G \otimes G'$*, that is:*

$$G \otimes G' = \{m : G \to H \leftarrow G' : m' \mid \text{m and m}' \text{ are jointly surjective monomorphisms}\}.$$

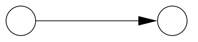
The definition of $G \otimes G'$ in terms of sets of pairs of monomorphisms may look a bit more complex than needed but, as in the case of the inclusions, we often need to identify the specific instances of G and G' inside H. However, in many occasions it is enough to consider that $G \otimes G'$ is the set of all graphs that can be seen as the result of putting together G and G'.

Note that if G and G' are finite graphs then $G \otimes G'$ is a also finite set. This is needed because in several inference rules (see Section 3) the result is a clause involving a disjunction related to a set of this kind. In particular, if $G \otimes G'$ is infinite so would be the corresponding disjunction.

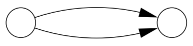
The above operation can be extended for putting together arbitrary (finite) sequences of graphs.

2.2 Graph Constraints

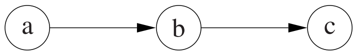
The underlying idea of a graph constraint is that it should specify that certain structures must be present (or must not be present) in a given graph. For instance, the simplest kind of graph constraint, $\exists C$, specifies that a given graph G should include (a copy of) C. For instance, the constraint:

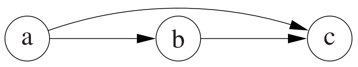
$\exists \big($  $\big)$

specifies that a graph should include at least one edge. Obviously, $\neg\exists C$ specifies that a given graph G should not include (a copy of) C. For instance, the constraint:

$\neg \exists \big($  $\big)$

specifies that a given graph G should not include two different edges between any two nodes. A slightly more complex kind of graph constraints are atomic constraints of the form $\forall(c : X \to C)$ where c is a monomorphism (or, just, an inclusion). This constraint specifies that whenever a graph G includes (a copy of) the graph X it should also include (a copy of) its extension C. However, in order to enhance readability (the monomorphism arrow may be confused with the edges of the graphs), in our examples we will display this kind of constraints using an **if - then** notation, where the two graphs involved have been labelled to implicitly represent the given monomorphism. For instance, the constraint:

if 

then 

specifies that a graph must be transitive, i.e. the constraint says that for every three nodes, a, b, c if there is an edge from a to b and an edge from b to c then there should be an edge from a to c.

Obviously, graph constraints can be combined using the standard connectives $\vee$ and $\neg$ (as usual, $\wedge$ can be considered a derived operation). In addition, in [6,17] a more complex kind of constraints, namely nested constraints, is defined, but we do not consider them in this paper.

Definition 4 (Syntax of graph constraints). *An* atomic graph constraint $\forall(c : X \to C)$ *is a graph monomorphism* $c : X \to C$*. An atomic graph constraint* $\forall(c : X \to C)$*, where* $X = \emptyset$*, is called a basic atomic constraint (or just a basic constraint) and will be denoted* $\exists C$*.*

Graph constraints *are logic formulas defined inductively as usual:*

– *Every atomic graph constraint is a graph constraint.*

– *If α is a graph constraint then $\neg\alpha$ is also a graph constraint.*
– *If α_1 and α_2 are graph constraints then $\alpha_1 \vee \alpha_2$ is also a graph constraint.*

Satisfaction of constraints is defined inductively following the intuitions described above.

Definition 5 (Satisfaction of graph constraints). *A graph G satisfies a constraint α, denoted $G \models \alpha$ if:*

– *$G \models \forall(c : X \to C)$ if for every monomorphism $h : X \to G$ there is a monomorphism $f : C \to G$ such that $h = f \circ c$.*
– *$G \models \neg\alpha$ if G does not satisfy α.*
– *$G \models \alpha_1 \vee \alpha_2$ if $G \models \alpha_1$ or $G \models \alpha_2$.*

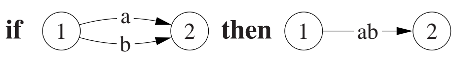
In this paper, for simplicity, we will assume that our specifications consist only of atomic constraints and negated atomic constraints (negative constraints). However, our results should extend straightforwardly to clausal specifications, i.e. sets of formulas of the form $L_1 \vee \cdots \vee L_n$, where each *literal* L_i is either an atomic constraint (a *positive literal*) or a negative atomic constraint (a *negative literal*). Actually, this would mean that we could deal with arbitrary formulas since they could always be transformed into clausal form.

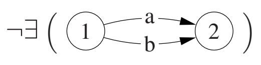
It may be noted that in the case of basic constraints the above definition specializes as expected:

Fact 1 (Satisfaction of basic constraints)
$G \models \exists C$ if there is a monomorphism $f : C \to G$.

It may also be noted that, according to these definitions, the constraint $\exists\emptyset$, where $\emptyset$ denotes the empty graph, is satisfied by any graph, i.e. $\exists\emptyset$ may be considered the trivial *true* constraint.

Remark 1. Atomic constraints can be generalized by allowing its definition in terms of arbitrary morphisms. That is, we could have defined atomic graph constraints $\forall(c : X \to C)$ where c is an arbitrary morphism. However, with our notion of satisfaction, this generalization does not add any additional power to our logic, since it can be proved [10] that if c is not a monomorphism then the constraint $\forall(c : X \to C)$ is logically equivalent to the constraint $\neg\exists X$. For instance, the two constraints below are equivalent. In particular, both constraints specify that there can not be two different edges between any two nodes.

(1)


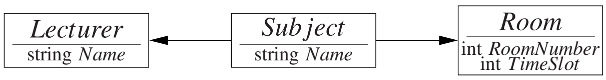
(2)


Analogously, we could have also generalized our notion of satisfaction by allowing h and f to be also arbitrary morphisms and not just monomorphisms. This generalized form of satisfaction has been studied in [11], where it is called $\mathcal{A}$-satisfaction in contrast with the notion of satisfaction that we use, which is called $\mathcal{M}$-satisfaction in that paper.

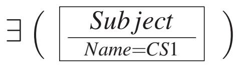
In particular, in [11], it is shown how to transform constraints such that $\mathcal{A}$-satisfiability for a certain constraint is equivalent to $\mathcal{M}$-satisfiability for the transformed constraint (and vice versa). This would mean that, again, working with $\mathcal{A}$-satisfaction would not add any additional power. Moreover, we believe that $\mathcal{A}$-satisfaction is not very intuitive implying that it may be not very appropriate for specification purposes.

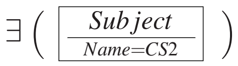
As pointed out above, these notions can be defined not only for plain graphs but for other classes of structures. In this sense, in the example below we will use typed attributed graph constraints.

Example 1. Let us suppose that we want to model an information system describing the lecturing organization of a department. Then the type graph of (part of) our system could be the following one:

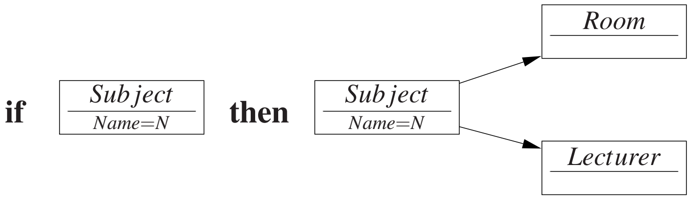


This means that in our system we have three types of nodes: Rooms including two attributes, the room number and a time slot, and Subjects and Lecturers, having its name as an attribute. We also have two types of edges. In particular, an edge from a Subject S to a Lecturer L means, obviously, that L is the lecturer for S. An edge from a Subject S to to a Room means that the lecturing for S takes place on that room for the given time slot. Now for this system we could include the following constraints:

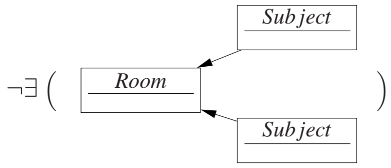
(1)


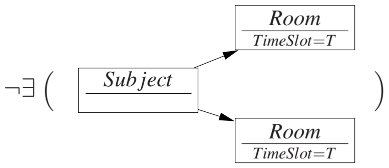
(2)


meaning that the given system must include the compulsory subjects Computer Science 1 and Computer Science 2. Moreover we may have a constraint saying that every subject included in the system must have some lecturer assignment and some room assignment:

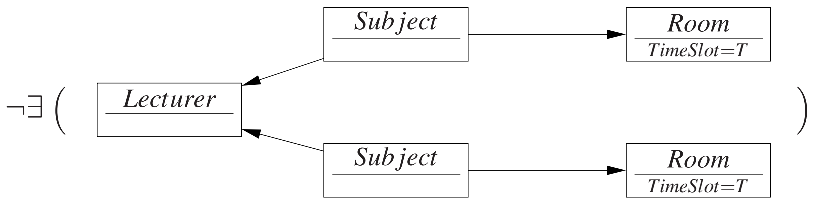
(3)


Then, we may also have constraints expressing some negative conditions. For instance, that a room is not assigned at the same time to two subjects or that two different rooms are assigned at the same time to the same subject:

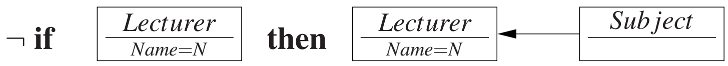
(4)


(5)


or, similarly, that a lecturer does not have to lecture on two different subjects in two different rooms at the same time:

(6)


Finally, perhaps we may want to specify that not every lecturer has a teaching assignment, so that every semester there may be someone on sabbatical:

(7)


It may be noticed that the system that we are describing with these graphical constraints may be not an information system, but the set of web pages of a department, where an arrow from a node of type t_1 to a node of type t_2 may mean that there is a link between two web pages (for instance from the web page of a subject to the web pages of the lecturers), or it may mean that the information of type t_2 is a subfield of the information of type t_1 (for instance the room assignment may be a field of the subjects web pages). In this case, we could have displayed our constraints not in terms of graphs, but as HTML or XML expressions.

3 Satisfiability of Sets of Graph Constraints

In this section we will present several inference systems that provide sound and complete refutation procedures for checking satisfiability for different classes of graph constraints. More precisely, after defining some standard basic concepts about refutation procedures, we will study the case of (positive and negative) basic constraints. In particular, in this case the inference rules define a sound and complete procedure for checking satisfiability that always terminate. Then, in the third subsection we will study the case where we have positive and negative basic constraints and positive atomic constraints. Finally, we consider the satisfiability problem for specifications consisting of arbitrary positive and negative atomic constraints. In this case, the inference rules deal with a more general notion of constraint, which we call *contextual literals* consisting of a positive literal with an associated context. In particular, this context is a finite set of negative atomic constraints together with monomorphisms binding the conditional part of each constraint to the corresponding literal. Using these contextual clauses, we prove that our inference system defines a refutation procedure which is sound and complete. In the last two cases, our refutation procedures may not terminate, which means that the procedures are just refutationally complete. Moreover, in these two cases our procedures check satisfiability with respect to the class of finite and infinite graphs. In fact, we show an example of a specification whose only models are infinite graphs. As a consequence, we guess that satisfiability for this class of constraints is already undecidable (but semi-decidable).

3.1 Basic Concepts

As it is often done in the area of automatic reasoning, the refutation procedures that we present in this paper are defined by means of some inference rules. More precisely, as usual, each rule tells us that if certain premises are satisfied then a given consequence will also hold. In this context, a refutation procedure can be seen as a (possibly nonterminating) nondeterministic computation where the current state is given by the set of formula that have been inferred until the given moment and where a computation step means adding to the given state the result of applying an inference rule to that state.

More precisely, in our case, we assume that the inference rules have the form:

$$\frac{\Gamma_1 \quad \alpha}{\Gamma_2}$$

where Γ_1 and Γ_2 are clauses of the form $\exists G_1 \vee \cdots \vee \exists G_n$, where $\exists G_1$, $\exists G_n$ are basic positive constraints and where α is an atomic constraint. Moreover, Γ_1 is assumed to belong to the current set of inferred clauses and α is assumed to belong to the original specification. Then a *refutation procedure* for a set of constraints $\mathcal{C}$ is a sequence of inferences:

$$\mathcal{C}_0 \Rightarrow_{\mathcal{C}} \mathcal{C}_1 \Rightarrow_{\mathcal{C}} \cdots \Rightarrow_{\mathcal{C}} \mathcal{C}_i \Rightarrow_{\mathcal{C}} \cdots$$

where the initial state just includes the *true* clause (i.e. $\mathcal{C}_0 = \{\exists\emptyset\}$) and where we write $\mathcal{C}_i \Rightarrow_{\mathcal{C}} \mathcal{C}_{i+1}$ if there is an inference rule like the one above such that $\Gamma_1 \in \mathcal{C}_i$, $\alpha \in \mathcal{C}$, and $\mathcal{C}_{i+1} = \mathcal{C}_i \cup \{\Gamma_2\}$. Moreover, we will assume that $\mathcal{C}_i \subset \mathcal{C}_{i+1}$, i.e. $\Gamma_2 \notin \mathcal{C}_i$, to avoid useless inferences.

It may be noted that our refutation procedures are *linear*, which means that no inferences from derived clauses are needed. If we would generalize our approach, allowing $\mathcal{C}$ to be a set of arbitrary graph constraints, then we would lose the linearity of the refutation procedures.

In this framework, proving the unsatisfiability of a set of constraints means inferring the *false* clause (which is represented by the empty clause, i.e. the empty disjunction, denoted $\square$), provided that the procedure is sound and complete. Since the procedures are nondeterministic, there is the possibility that we never apply some key inference. To avoid this problem we will always assume that our procedures are *fair*, which means that if at any moment i, there is a possible inference $\mathcal{C}_i \Rightarrow_{\mathcal{C}} \mathcal{C}_i \cup \{\Gamma\}$, for some clause Γ, then at some moment j we have that $\Gamma \in \mathcal{C}_j$.

Then, a refutation procedure for $\mathcal{C}$ is *sound* if whenever the procedure infers the empty clause we have that $\mathcal{C}$ is unsatisfiable. And a procedure is *complete* if, whenever $\mathcal{C}$ is unsatisfiable, we have that the procedure infers. $\square$

It may be noted that if a refutation procedure is sound and complete then we may know in a finite amount of time if a given set of constraints is unsatisfiable. However, it may be impossible to know in a finite amount of time if the set of constraints is satisfiable. For this reason, sometimes the above definition of completeness is called refutational completeness, using the term completeness when both satisfiability and unsatisfiability are decidable.

As usual, for proving soundness of a refutation procedure it is enough to prove the soundness of the inference rules. This means that for every rule as the one above and every graph G, if $G \models \Gamma_1$ and $G \models \alpha$ then $G \models \Gamma_2$.

3.2 Basic Constraints

In this section we study the case where specifications consist only of positive and negative basic constraints. This means that specifications are assumed to be sets of literals of the form $\exists C_1$ or $\neg\exists C_1$. However, as said above, in the deduction process we will deal with clauses of the form $\exists G_1 \vee \cdots \vee \exists G_n$. Now, for this case, satisfiability will be based on two rules that can be found below (obviously, we assume disjunction to be commutative and associative, which means that the literal $\exists C_1$ in the premises of the rules is not necessarily the leftmost literal in the given clause).

$$\frac{\exists C_1 \vee \Gamma \quad \neg\exists C_2}{\Gamma} \quad \textbf{(R1)}$$

if there exists a monomorphism $m : C_2 \to C_1$

$$\frac{\exists C_1 \vee \Gamma \quad \exists C_2}{(\bigvee_{G \in \mathcal{G}} \exists G) \vee \Gamma} \quad \textbf{(R2)}$$

if there is no monomorphism $m : C_2 \to C_1$, where $\mathcal{G} = \{G \mid \langle f_1 : C_1 \to G \leftarrow C_2 : f_2\rangle \in (C_1 \otimes C_2)\}$ and where $(\bigvee_{G \in \mathcal{G}} \exists G)$ denotes the (finite) disjunction $\exists G_1 \vee \cdots \vee \exists G_n$, if $\mathcal{G} = \{G_1, \ldots, G_n\}$.

The first rule is, in some sense, similar to resolution and is the rule that may allow us to infer the empty clause. The reason is that it is the only rule that generates clauses with fewer literals. The second one can be seen as a rule that, given two constraints, builds a new constraint that subsumes them. More precisely, the graphs involved in the new literals in the clause, i.e. the graphs $G \in \mathcal{G}$ satisfy both constraints $\exists C_1$ and $\exists C_2$. This means that if we apply this rule repeatedly, using all the positive constraints in the original set $\mathcal{C}$, we would build (minimal) graphs that satisfy all the positive constraints in $\mathcal{C}$.

Example 2. If we consider the basic constraints that are included in the Example 1 (i.e. the constraints (1), (2), (4), (5), and (6)) then it would only be possible to infer the constraint below as follows. First, using rule (R2) on the trivial clause $\exists\emptyset$ and constraint (1)we obtain a clause including only constraint (1). Then, using again rule (R2) on this clause and constraint(2) we obtain the clause:

$$\textbf{(8)} \quad \exists\left(\boxed{\begin{array}{c}\mathit{Subject}\\ \hline \mathit{Name{=}CS1}\end{array}} \qquad \boxed{\begin{array}{c}\mathit{Subject}\\ \hline \mathit{Name{=}CS2}\end{array}}\right)$$

meaning that the graph representing the system must include at least two Subject nodes (with attributes CS1 and CS2). No further inference rule can be applied, which means that these constraints are satisfiable, and indeed this constraint represents a (minimal) model satisfying the constraints (1), (2), (4), (5), and (6).

These two rules are sound and complete. The soundness of the first rule is quite obvious. If a graph G satisfies both premises and in addition we know that C_2 is included in C_1 then G cannot satisfy $\exists C_1$. Therefore, it should satisfy Γ. The soundness of the second rule is based on the so-called *pair factorization property*: Given two (mono)morphisms, $f_1 : G_1 \to G$, $f_2 : G_2 \to G$, with the same codomain G there exists a graph H and monomorphisms $g_1 : G_1 \to H$, $g_2 : G_2 \to H$ and $h : H \to G$ such that g_1 and g_2 are jointly surjective and the diagram below commutes:

$$\begin{tikzcd} G_1 \arrow[d, "g_1"'] \arrow[dr, "f_1"] & \\ H \arrow[r, "h"] & G \\ G_2 \arrow[u, "g_2"] \arrow[ur, "f_2"'] & \end{tikzcd}$$

Then, if a graph G satisfies $\exists C_1$ and $\exists C_2$ the pair factorization property will tell us that G will also satisfy $\exists H$, where H is a graph in $\mathcal{G}$.

It may be noted that in the proof of soundness of rule (R2) we have not used the condition that there is no monomorphism $m : C_2 \to C_1$. The reason for such condition is to limit the number of possible inferences, i.e. to make the refutation procedure a bit more efficient since we do not lose completeness with this restriction.

Now, in this case, as said above, we can ensure not only the completeness of any fair refutation procedure, but also its termination. The key property for proving completeness and termination, related to the observation made above about the second inference rule, is that if $\mathcal{C}_0 \Rightarrow^*_{\mathcal{C}} \mathcal{C}'$ and $\exists C$ is an atom (different from the trivial constraint $\exists\emptyset$) included in a clause in $\mathcal{C}'$ then $C \in \{G \mid \langle f_1 : H_1 \to G \leftarrow H_2 : f_2\rangle \in (\bigotimes_{C_i \in \mathcal{L}} C_i)\}$, where $\mathcal{L}$ is a subset of $\mathcal{C}$ consisting only of positive constraints. Then, since there is a finite number of these graphs, we can be sure that there is a finite number of clauses involving these graphs and, hence, only a finite number of sets of clauses can be inferred using the two rules. This ensures termination. On the other hand, if we know that the refutation procedure terminates on the set of clauses $\mathcal{C}_k$ it is easy to see that either it includes the empty clause or there is a clause involving a graph G that satisfies all the constraints in $\mathcal{C}$. Therefore, we have:

Theorem 1 (Termination, Soundness and Completeness). *Given a set of basic (positive and negative) constraints $\mathcal{C}$, any fair refutation procedure defined over $\mathcal{C}$ based on the rules (R1) and (R2) will always terminate, i.e. there is an k such that $\mathcal{C} \Rightarrow_{\mathcal{C}} \mathcal{C}_1 \Rightarrow_{\mathcal{C}} \mathcal{C}_2 \Rightarrow_{\mathcal{C}} \cdots \Rightarrow_{\mathcal{C}} \mathcal{C}_k$ and no rule can be applied on $\mathcal{C}_k$. Moreover, $\mathcal{C}$ is unsatisfiable if and only if the empty clause is in $\mathcal{C}_k$.*

3.3 Basic Constraints and Positive Atomic Constraints

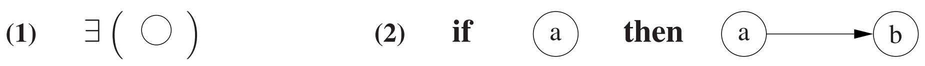
In this section we extend the case studied in the previous subsection by allowing specifications including positive atomic constraints. This means that the given specifications are assumed to consist of literals of the form $\exists C_1$, $\neg\exists C_1$, or $\forall(c : X \to C_2)$. In this case, satisfiability is based on the two rules presented in the previous subsection plus the following new rule:

$$\frac{\exists C_1 \vee \Gamma \quad \forall(c : X \to C_2)}{(\bigvee_{G \in \mathcal{G}} \exists G) \vee \Gamma} \qquad \textbf{(R3)}$$

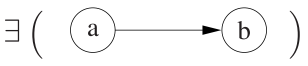
if there is a monomorphism $m : X \to C_1$ such that there is no monomorphism $h : C_2 \to C_1$ such that $m = h \circ c$ and where $\mathcal{G}$ is the set consisting of all the graphs G such that there are two jointly surjective monomorphisms $f_1 : C_1 \to G$ and $f_2 : C_2 \to G$ such that the diagram below commutes:

$$\begin{tikzcd}
& C_1 \arrow[dr, "f_1"] & \\
X \arrow[ur, "m"] \arrow[dr, "c"'] & & G \\
& C_2 \arrow[ur, "f_2"'] &
\end{tikzcd}$$

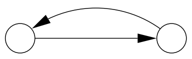
This new rule is similar to rule (R2) in the sense that given a positive basic constraint and a positive atomic constraint it builds a disjunction of literals representing graphs that try to satisfy both constraints. However, in this case the satisfaction of the constraint $\forall(c : X \to C_2)$ is not ensured. In particular, the idea of the rule is that if we know that X is included in C_1 then we build all the possible extensions of C_1 which also include C_2 (each G would be one of such extensions). However, in this case we cannot be sure that G satisfies $\forall(c : X \to C_2)$, because G may include an instance of X which was not included in C_1. For instance, suppose that we have the following constraints:



where the first one specifies that the given graph must include a node and where the second one specifies that every node must have an outgoing edge. Then applying rule (R3) to these constraints would yield a clause one of whose subterms is the constraint:



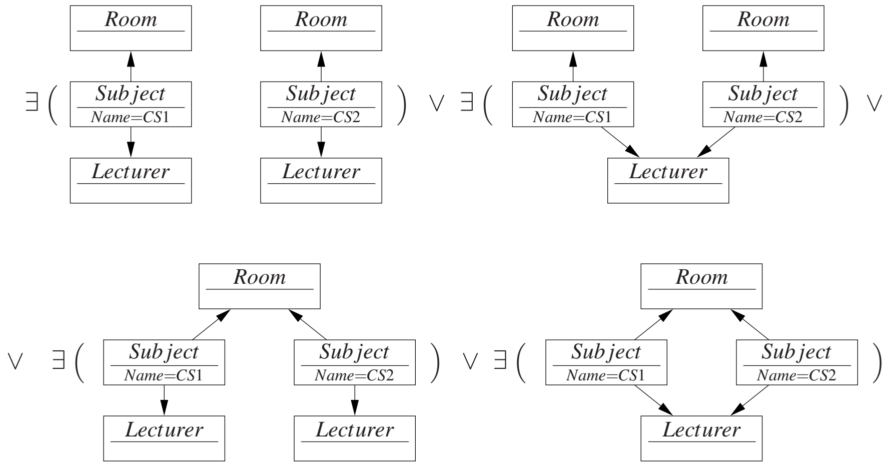
Now, in this graph node a has an outgoing edge, but node b does not have it, so the graph still does not satisfy the second constraint. If we would apply again the third rule, then we would infer a clause including a graph with three nodes and two edges, and so on. This is the reason why, in this case, a refutation procedure may not terminate. Moreover, as we will also see, if the procedure does not refute the given set of constraints then the completeness proof ensures that there will be a model that satisfies this set of constraints, but this model may be an infinite graph built by an infinite colimit. One may wonder whether there will also exist a finite model of that specification. In the case of this example such a finite graph exists. Actually, the resulting clause after applying for the second time the third rule to the graph above, would also include the graph below that satisfies both constraints.



However, in general, we do not know if an arbitrary set of basic constraints and positive atomic constraints which is satisfiable by an infinite graph, is also satisfied by some finite graph. Nevertheless, in the general case (when dealing with positive and negative atomic constraints) there are sets of constraints whose only models are infinite graphs, as we will see in the following subsection. For this reason we conjecture that in this case the answer to this question will also be negative.

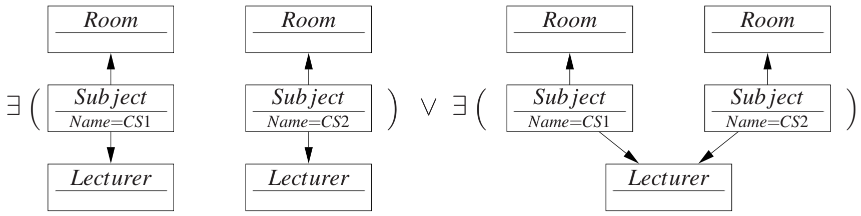
Example 3. Let us consider the basic constraints and the positive atomic constraints that are included in Examples 1 and 2 (i.e. the constraints (1), (2), (3), (4), (5), (6), and (8)). If we apply the third rule on constraints (8) and (3), and again on the resulting clause and on constraint (3) then we would infer the following clause:

(9)



This clause states that the graph should include two subjects (CS1 and CS2) and these subjects may be assigned to two different rooms and to either two different lecturers, or to the same lecturer, or they may be assigned to the same room, and to either different lecturers, or the same lecturer. Obviously, the last two constraints in this clause violate constraint (4), which means that we can eliminate them using twice rule (R1), yielding the following clause:

(10)



Then, no further inferences can be applied, which means that this set of constraints is satisfiable. Actually, the two graphs occurring in clause (10) would be (minimal) models of the set of constraints (except constraint (7) which is considered in the following subsection).

The proof of soundness of the new rule (R3) is very similar to the proof for rule (R2). If G satisfies $\exists C_1$ and $\forall(c : X \to C_2)$, using pair factorization we get the diagram below:

$$\begin{tikzcd}
 & C_1 \arrow[dr, "f_1"] \arrow[drr, "h_1"] & & \\
X \arrow[ur, "m"] \arrow[dr, "c"'] & & H \arrow[r] & G \\
 & C_2 \arrow[ur, "f_2"] \arrow[urr, "h_2"'] & &
\end{tikzcd}$$

then, G will also satisfy $\exists H$, where H is a graph in $\mathcal{G}$.

In this case, the proof of completeness is a bit more involved as a consequence of the possible nontermination of the refutation procedure. The underlying idea is inspired by a technique called *model construction* used for proving completeness of some inference systems for first-order logic with equality [16]. According to this technique, we see the inference rules as steps for building a model of the given set of constraints. In particular, given a set of constraints $\mathcal{C}$, we consider sequences of graphs $\emptyset \prec C_1 \prec \cdots \prec C_i \prec \ldots$, where $C_i \prec C_{i+1}$ if (a) $\exists C_{i+1}$ is a literal in the clause inferred after applying rules (R2) or (R3) to $\exists C_i$ and to some positive constraint in $\mathcal{C}$ (this means that C_i is included in C_{i+1}) ; and (b) C_{i+1} satisfies all the negative constraints in $\mathcal{C}$. Each of these sequences can be seen as a path for building a possible model of $\mathcal{C}$. In this sense, condition (b) is needed because if C_{i+1} does not satisfy a negative constraint in $\mathcal{C}$ then it is useless to continue this path. Moreover, we require these sequences to be fair, which means that if we can have $C_i \prec C'_{i+1}$ via some inference an the sequence is infinite, then there would be some j such that C_j includes C'_{i+1}. Then, we have three cases:

- All maximal sequences of this kind are finite, and in all cases the last graph C_i in each sequence does not satisfy all positive constraints in $\mathcal{C}$. This means that no path is useful for building a model. Then we can show that a fair procedure would generate the empty clause for $\mathcal{C}$.
- There is a finite sequence whose last graph C_i satisfies all the constraints in $\mathcal{C}$. Then we have built a finite model for $\mathcal{C}$.
- There is an infinite fair sequence. Then we show that if we take the union of all the graphs in the sequence (the colimit of the sequence) the result is a model for $\mathcal{C}$.

As a consequence, we have:

Theorem 2 (Soundness and Completeness). *Let $\mathcal{C} \Rightarrow_C \mathcal{C}_1 \Rightarrow_C \cdots \Rightarrow_C \mathcal{C}_k \ldots$ be a fair refutation procedure defined over a set of basic constraints and positive atomic constraints$\mathcal{C}$, based on the rules (R1), (R2), and (R3). Then, $\mathcal{C}$ is unsatisfiable if and only if there is a j such that the empty clause is in $\mathcal{C}_j$.*

3.4 Atomic Constraints

In this section we study the case of general specifications including arbitrary positive and negative atomic constraints. In this case, in order to ensure completeness, the inference rules are defined over a generalized notion of clause, consisting of *contextual* *literals*, which are positive basic constraints with an associated context. More precisely, given a constraint $\exists C$, a context for this constraint is a set of negative atomic constraints $\neg\forall(g : X \to C_1)$ such that X is included in C. Actually, C has to satisfy this negative constraint. However, as usual, we need to know not only that X is a subgraph of C, but also to identify the specific instance X that cannot be extended to C_1. For this reason we consider that a context is a finite set of negative atomic constraints together with monomorphisms binding the conditional part of each constraint to the corresponding literal. Below, when sketching the completeness proof, we will explain the need for these contexts.

Definition 6 (Contextual Constraints). *A contextual constraint $\exists C[Q]$ is a pair consisting of a basic constraint, $\exists C$, and a set Q consisting of pairs $\langle\neg\forall(g : X \to C_1), h : X \to C\rangle$, where $\neg\forall(g : X \to C_1)$ is a negative atomic constraint and h is a monomorphism such that there is no monomorphism $h' : C_1 \to C$ such that $h = h' \circ g$.*

Now, we have to define satisfaction for this kind of contextual constraints. The idea is that a graph satisfies a contextual constraint $\exists C[Q]$ if it satisfies $\exists C$ and all the constraints in its context:

Definition 7 (Satisfaction of Contextual Constraints). *A graph G satisfies a contextual constraint $\exists C[Q]$ via a monomorphism $f : C \to G$, $G \models_f \exists C[Q]$, if for every $\langle\neg\forall(g : X \to C_1), h : X \to C\rangle \in Q$ there is no monomorphism $h' : C_1 \to G$ such that $f \circ h = h' \circ g$. G satisfies $\exists C[Q]$, $G \models \exists C[Q]$, if there is a monomorphism $f : C \to G$ such that $G \models_f \exists C[Q]$.*

Finally, given a contextual constraint $\exists C[Q]$ and a graph G that satisfies it via a monomorphism $f : C \to G$, in our inference rules we need to be able to build a contextual constraint whose left-hand side is $\exists G$ and whose context includes the same negative constraints as $[Q]$. In order to do this we need to define the new binding of the negative constraints in $[Q]$ with G:

Definition 8. *Given a contextual constraint $\exists C[Q]$ and a monomorphism $g : C \to G$, we define the context $g\langle Q\rangle$ as the set $\{\langle\neg\forall(g : X \to C_1), g \circ h : X \to G\rangle \mid \langle\neg\forall(g : X \to C_1), h : X \to C\rangle \in Q\}$.*

In this case, satisfiability is based on four rules. The first three rules are a reformulation (in terms of contextual constraints) of the rules defined in the previous sections. In addition, a new rule describes the kind of inferences that can be done using negative atomic constraints. The four rules are:

$$\frac{\exists C_1[Q] \vee \Gamma \quad \neg\exists C_2}{\Gamma} \quad \textbf{(R1')}$$

if there exists a monomorphism $m : C_2 \to C_1$

$$\frac{\exists C_1[Q] \vee \Gamma \quad \exists C_2}{(\bigvee_{G \in \mathcal{G}} \exists G[f_1\langle Q\rangle]) \vee \Gamma} \quad \textbf{(R2')}$$

if there is no monomorphism $m : C_2 \to C_1$ and where $\mathcal{G} = \{G \mid \langle f_1 : C_1 \to G \leftarrow C_2 : f_2\rangle \in (C_1 \otimes C_2)$, such that $G \models_{id} \exists G[f_1\langle Q\rangle]\}$.

$$\frac{\exists C_1[Q] \vee \Gamma \quad \forall(c : X \to C_2)}{(\bigvee_{G\in\mathcal{G}} \exists G[f_1\langle Q\rangle]) \vee \Gamma} \quad \textbf{(R3')}$$

if there is a monomorphism $m : X \to C_1$ and there is no monomorphism $h : C_2 \to C_1$ such that $m = h \circ c$ and where $\mathcal{G}$ is the set consisting of all the graphs G such that there are two jointly surjective monomorphisms $f_1 : C_1 \to G$ and $f_2 : C_2 \to G$ such that $G \models_{id} \exists G[f_1\langle Q\rangle]\}$ and the diagram below commutes:

$$\begin{tikzcd}
& C_1 \arrow[dr, "f_1"] & \\
X \arrow[ur, "m"] \arrow[dr, "c"'] & & G \\
& C_2 \arrow[ur, "f_2"'] &
\end{tikzcd}$$

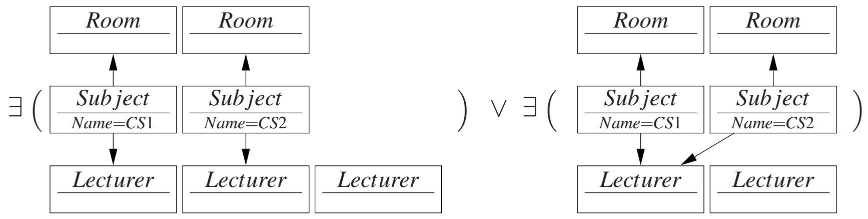
$$\frac{\exists C_1[Q] \vee \Gamma \quad \neg\forall(g : X \to C_2)}{(\bigvee_{G\in\mathcal{G}} \exists G[Q']) \vee \Gamma} \quad \textbf{(R4)}$$

if $(\neg\forall(g : X \to C_2)) \notin Q$ and where $\mathcal{G} = \{G \mid \langle f_1 : C_1 \to G \leftarrow X : f_2\rangle \in (C_1 \otimes X)$, such that $G \models_{id} \exists G[Q']\}$, and $Q' = f_1\langle Q\rangle \cup \{\langle\neg\forall(g : X \to C_2), f_2\rangle\}$.

This new rule is similar to (the reformulation of) rule (R2). The reason is that a negative atomic constraint $\neg\forall(c : X \to C_2)$ (partly) specifies that there must be a copy of X in the given graph, as it happens with the constraint $\exists X$. The main difference to rule (R2') is that, in the new rule, the negative constraint is added to the context of the new constraints introduced in the clause inferred by the rule.

It may be noticed that with any of these four rules we may generate the empty clause, since all the rules may delete a literal from a rule because of the contexts. In the previous cases only rule (R1) would eliminate literals. However, in this case, in rules (R2'), (R3') and (R4) it may happen that no $G \in \mathcal{G}$ satisfies the resulting context. As a consequence, in this situation, the resulting clause would be Γ.

Example 4. Let us consider all the constraints and clauses from Examples 1, 1, and 3. If we apply twice the fourth rule on clause (10) and constraint (7) then we would infer the following clause:

(11)  where the context associated to both literals (not displayed above) would consist of constraint (7). Again, no further inferences can be applied, which means that this set of constraints is satisfiable, and the two graphs occurring in clause (11) would be (minimal) models of the set of constraints.

Now, with this new formulation, again we are able to show soundness and completeness of our inference rules. In particular, the proofs of soundness for rules (R1')-(R3') are very similar to the proofs for rules (R1)-(R3). The only difference is that we have to take into account the contexts. Then the proof of soundness for rule (R4) also follows the same pattern. In particular, if $G \models_{h_1} \exists C_1[\mathcal{Q}]$ and $G \models \neg\forall(c : X \to C_2)$ then using again the property of pair factorization we have:

$$\begin{tikzcd}
C_1 \arrow[d, "f_1"'] \arrow[dr, "h_1"] & \\
G' \arrow[r, "f"] & G \\
X \arrow[u, "f_2"] \arrow[ur, "h_2"'] &
\end{tikzcd}$$

where $f_1 : C_1 \to G'$ and $f_2 : X \to G'$ are jointly surjective. In this context, it is routine to prove that, on one hand, $G' \models_{id} \exists G'[\mathcal{Q}']$ (so, $G' \in \mathcal{G}$) and, on the other, $G \models_f G'[\mathcal{Q}']$.

The proof of completeness in this case is very similar to the previous completeness proof. The main difference is in the key role played by the contexts. The idea in the previous proof was to consider sequences of graphs $\emptyset \prec C_1 \prec \cdots \prec C_i \prec \ldots$, where every C_i is included in C_{i+1}, that could be seen as the construction of a model for $\mathcal{C}$, if the empty clause was never inferred. In particular, these sequences were associated to the given inferences. Moreover, an important property in that proof is that it was assumed that every graph in these sequences would satisfy all the negative constraints in $\mathcal{C}$. In particular, given a graph C_i, if a possible successor C_{i+1} does not satisfy a negative constraint $\neg\exists C$ in $\mathcal{C}$ then we know that a sequence $\emptyset \prec C_1 \prec \cdots \prec C_i \prec C_{i+1} \prec \ldots$ would never yield a model of $\mathcal{C}$. The reason is that any graph including C_{i+1} will neither satisfy $\neg\exists C$. However, this is not true for negative atomic constraints. If C_{i+1} does not satisfy $\neg\forall(g : X \to C)$ then some graphs G including C_{i+1} may satisfy $\neg\forall(g : X \to C)$. For this reason it would be wrong to prune a sequence $\emptyset \prec C_1 \prec \cdots \prec C_i \prec \ldots$, if we know that C_i does not satisfy a constraint $\neg\forall(g : X \to C)$, because some graph C_k, with $k > i$, may satisfy it. However, in this situation it is impossible to say if this sequence, in the limit (or, rather, in the colimit) would yield a model of $\mathcal{C}$ and, especially, if it would satisfy that constraint.

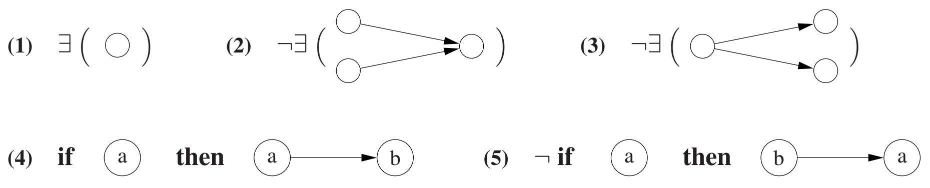
The use of contexts solves this problem. In particular, if $C_i[\mathcal{Q}]$ does not satisfy a constraint $\neg\forall(g : X \to C)$ in its context $\mathcal{Q}$ then no larger graph would satisfy it. Then, in a similar manner as in the previous completeness proof, we can define sequences $\emptyset[\emptyset] \prec C_1[\mathcal{Q}_1] \prec \cdots \prec C_i[\mathcal{Q}_i] \prec C_{i+1}[\mathcal{Q}_{i+1}] \prec \ldots$, where each C_i satisfies all the negative basic constraints in $\mathcal{C}$ and all the negative constraints in $\mathcal{Q}_i$. Then, fairness of the sequences ensure that for every sequence there is an i such that $\mathcal{Q}_i$ includes all the negative atomic constraints in $\mathcal{C}$. This ensures that a fair sequence will yield a model of $\mathcal{C}$, provided that the empty clause cannot be inferred from $\mathcal{C}$.

As a consequence, we have:

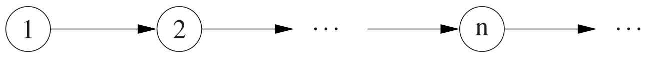
Theorem 3 (Soundness and Completeness). *Let $C \Rightarrow_C C_1 \Rightarrow_C \cdots \Rightarrow_C C_k \ldots$ be a fair refutation procedure defined over a set of atomic constraints C, based on the rules (R1'), (R2'), (R3') and (R4). Then, C is unsatisfiable if and only if there is a j such that the empty clause is in C_j.*

As discussed above, our completeness results show that a set of constraints is satisfiable then a fair refutation procedure will never infer an empty clause from the given set of constraints. However, in the proof of completeness, the model constructed to show the satisfiability of the constraints is an infinite graph. One could wonder whether in this situation it would always be possible to find an alternative finite model for these constraints. The answer is no. As we can see in the counter-example below, there are sets of atomic constraints which do not have finite models.

Example 5. The set of constraints below is not satisfied by any finite graph, but only by infinite graphs:



Let n be the number of nodes of a graph satisfying the constraints and e its number of edges. The first constraint specifies that the graph must have at least a node, i.e. $n \geq 1$. The second and third constraints specify that every node must have at most one incoming edge and one outgoing edge, i.e. $n \geq e$. The fourth constraint specifies that every node has an outgoing edge, , i.e., $n \leq e$ and, finally, the fifth constraint specifies that not every node has an incoming edge, i.e., $n > e$. Now, obviously no finite graph would satisfy these constraints. However the graph below does satisfy them:



4 Conclusion

In this paper we have shown how we can use graph constraints as a specification formalism to define constraints associated to visual modelling formalisms or to specify classes of semi-structured documents. In particular, we have shown how we can reason about these specifications, providing inference rules that are sound and complete. Moreover, as can be seen in our examples, the completeness proofs (only sketched in the paper) show that our inference rules can also be used for the construction of minimal models for the given sets of constraints.

As pointed out above, our results apply not only to plain graphs, but generalize to a large class of structures including typed and attributed graphs. In this sense, in [6,5] the constraints that we consider have been defined for any adhesive HLR-category [14,5]. However, to be precise, to generalize our results we would need that the underlying category of structures satisfies two properties that are not considered in [6,5]. On one hand, we would need that $G_1 \otimes G_2$ is finite provided that G_1 and G_2 are finite. It may be noticed that this may be not the case if G_1 and G_2 are arbitrary attributed graphs. However, if we restrict ourselves to the class of monomorphisms which are the identity on the attributes then the property would hold. On the other hand, the second condition that we need is the existence of infinite colimits (together with some additional technical property).

We have not yet implemented these techniques, although it would not be too difficult to implement them on top of the AGG system [1], given that the basic construction that we use in our inference rules (i.e. building $G_1 \otimes G_2$) is already implemented there. However we think that, before doing this kind of implementation, it may be worth to make our approach more "efficient". In particular, in this paper we have defined our rules caring mainly about completeness. However there are some other rules that, although they do not help in ensuring completeness, may help to prove more rapidly the (un)satisfiability of a set of constraints. In a different sense, in our refutation procedures we never eliminate any clause from the given proof state. However, we know that we can use some other rules for eliminating clauses, which means that we would reduce the search space of the procedure.

Acknowledgements. This work has been partially supported by the CICYT project FORMALISM (ref. TIN2007-66523) and by the AGAUR grant to the research group ALBCOM (ref. 00516).

References

1. AGG.: The AGG, `httptfs.cs.tu-berlin.de/agg`
2. Alpuente, M., Ballis, D., Falaschi, M.: Automated Verification of Web Sites Using Partial Rewriting. Software Tools for Technology Transfer 8, 565–585 (2006)
3. Baldan, P., Corradini, A., Kšnig, B., Lluch-Lafuente, A.: A Temporal Graph Logic for Verification of Graph Transformation Systems. In: Recent Trends in Algebraic Development Techniques, 18th International Workshop, WADT 2006. LNCS, vol. 4409, pp. 1–20. Springer, Heidelberg (2007)
4. Courcelle, B.: The expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic. In: [18], pp. 313–400 (1997)
5. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006)
6. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.-H.: Constraints and Application Conditions: From Graphs to High-Level Structures. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 287–303. Springer, Heidelberg (2004)
7. Ehrig, H., Habel, A.: Graph Grammars with Application Conditions: From Graphs to High-Level Structures. In: Rozenberg, G., Salomaa, A. (eds.) The Book of L, pp. 87–100. Springer, Heidelberg (1986)
8. Ellmer, E., Emmerich, W., Finkelstein, A., Nentwich, C.: Flexible Consistency Checking. ACM Transaction on Software Engineering and Methodology 12(1), 28–63 (2003)
9. Habel, A., Heckel, R., Taentzer, G.: Graph Grammars with Negative Application Conditions. Fundam. Inform. 26(3/4), 287–313 (1996)

10. Habel, A., Pennemann, K.-H.: Nested Constraints and Application Conditions for High-Level Structures. In: Kreowski, H.-J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) Formal Methods in Software and Systems Modeling. LNCS, vol. 3393, pp. 293–308. Springer, Heidelberg (2005)
11. Habel, A., Pennemann, K.-H.: Satisfiability of High-Level Conditions. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 430–444. Springer, Heidelberg (2006)
12. Heckel, R., Wagner, A.: Ensuring Consistency of Conditional Graph Grammars - A Constructive Approach. In: Proceedings SEGRAGRA 1995. Electr. Notes Theor. Comput. Sci., vol. 2, pp. 118–126 (1995)
13. Jelliffe, R.: "Schematron", Internet Document (May 2000), `http://xml.ascc.net/resource/schematron/`
14. Lack, S., Sobocinski, P.: Adhesive Categories. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 273–288. Springer, Heidelberg (2004)
15. Lambers, L., Ehrig, H., Orejas, F.: Conflict Detection for Graph Transformation with Negative Application Conditions. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 61–76. Springer, Heidelberg (2006)
16. Nieuwenhuis, R., Rubio, A.: Paramodulation-Based Theorem Proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, Elsevier Science and MIT Press (2001)
17. Rensink, A.: Representing First-Order Logic Using Graphs. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) Graph Transformations, Second International Conference, ICGT 2004. LNCS, vol. 3256, pp. 319–335. Springer, Heidelberg (2004)
18. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1. World Scientific, Singapore (1997)