A Refutation Procedure for Proving Satisfiability of Constraint Specifications on XML Documents *

Marisa Navarro[1] and Fernando Orejas[2]

[1] Universidad del País Vasco (UPV/EHU), San Sebastián, Spain
marisa.navarro@ehu.es
[2] Universitat Politècnica de Catalunya, Barcelona, Spain
orejas@lsi.upc.edu

Abstract

In this paper we first present three sorts of constraints (positive, negative and conditional ones) to specify that certain patterns must be satisfied in an XML document. These constraints are built on boolean XPath patterns. We define a specification as a set of clauses whose literals are constraints. Then, to reason about these specifications, we study some sound rules which permit to infer, subsume or simplify clauses. The main goal is to design a refutation procedure (based on these rules) to test if a given specification is satisfiable or not. We have formally proven the soundness of our procedure and we study the completeness and termination of the proposed method.

1 Introduction

XPath [12, 16] is a well-known language for navigating an XML document (or XML tree) and returning a set of answer nodes. Since XPath is used in many XML query languages as XQuery, XSLT or XML Schema among others [15, 13, 14], a great amount of papers deal with different aspects on different fragments of XPath. For instance, in [3] an overview of formal results on XPath is presented concerning the expressiveness of several fragments, complexity bounds for evaluation of XPath queries, as well as static analysis of XPath queries. In [4] they study the problem of determining, given a query p (in a given XPath fragment) and a DTD D, whether there exists an XML document conforming to D and satisfying p. They show that the complexity ranges from PTIME to undecidable, depending on the XPath fragment and the DTD chosen. The work presented in [5] deals with the same problem (in a particular case) and it uses Hybrid Modal Logic to model the documents and some class of schemas and constraints. They provide a tableau proof technique for constraint satisfiability testing in the presence of schemas.

Our approach is different than the previous ones in two aspects. On the one hand, we do not consider any DTD or schema, and we use a simple fragment of XPath. In this sense our approach is simpler than previous ones. But, on the other hand, our aim is to define specifications of classes of XML documents as sets of constraints on these documents, and to provide a form of reasoning about these specifications. In this sense, our main question is satisfiability, that is, given a set of constraints S, whether there exists an XML document satisfying all constraints in S. Moreover, we are looking for refutation procedures, based on sound and complete inference rules. In addition to checking satisfiability, these rules can be used to infer other constraints from the given set, which can help us to optimize the given specification.

Some other work, which shares part of our aims, is the approach for the specification and verification of semi-structured documents based on extending a fragment of first-order logic [2, 7]. They 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 given specification. However, they do not consider the problem of defining deductive tools to analyze specifications, for instance to look for inconsistencies.

*This work has been partially supported by the Spanish Project TIN2013-46181-C2-2-R, the Basque Project GIU12/26, and grant UFI11/45.

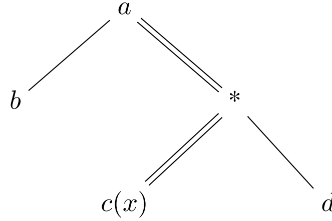


Figure 1: A tree pattern with answer node (marked x)

Schematron [8] has a more practical nature. It 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, as in the previous approach, Schematron provides no deductive capabilities. Finally, we consider the work presented in [10]. It shows how to use graph constraints as a specification formalism, which can be used to specify classes of semi-structured documents, and how to reason about these specifications, providing refutation procedures based on inference rules that are sound and complete.

We follow the main ideas given in [10] trying to apply them to XML documents. Our constraints are based on Xpath queries, as given in [9], which contain branching, label wildcards and can express descendant relationships between nodes. In particular, their *tree patterns* are an alternative representation of XPath queries consisting of node tests, the child axis (/), the descendant axis (//) and wildcards (*). The answer nodes are marked with (x). For instance, Figure 1 shows a tree pattern p that when applied to a given XML document t (which is also represented by a tree but in this case without descendant axis or wildcards), it must check if the root node in t is labelled a, if some child node of the root node in t is labelled b, and if some descendant node of the root node in t has both a child node labelled d and a descendant node labelled c. If all of these conditions are satisfied, the application $p(t)$ will return a set with such last descendant nodes (the nodes marked with x); otherwise, it will return the empty set.

Since our purpose is to reason on XML documents by means of a set of constraints, and not to obtain the answer nodes, we shall consider tree patterns without answer marks (which are called Boolean tree patterns in [9]). The application of such a pattern to a document t will return $true$, if t satisfies the conditions specified by the pattern, or $false$ otherwise.

We consider three kinds of (atomic) constraints. The first one is $\exists p$ where p is a tree pattern. This constraint will be satisfied by a document t if $p(t)$ is $true$. The second one is $\neg\exists p$ that will be satisfied by a document t if $p(t)$ is $false$. The third sort of constraint is written $\forall(c : p \rightarrow q)$, where both p and q are tree patterns (related by c in a special way) and, roughly speaking, it will be satisfied by a document t if $p(t)$ implies $q(t)$. Nevertheless, the application of the ideas in [10] to our setting is not trivial, as discussed in Section 3.

Our aim is to study adequate inference rules to find a sound and complete refutation procedure for checking satisfiability of a given specification. The inference rules take a format similar to the inference rules given in [10], but again the particularization to our setting needs to define appropriate operators and to prove new results. Moreover, these rules allow to infer, subsume and simplify clauses; that is, they permit to manipulate a specification and obtain another optimized specification which is semantically equivalent to the given one.

The paper is organized as follows. Section 2 contains some basic notions and notational conventions we are going to use along the paper. Section 3 introduces the three sorts of constraints that we use as literals of the clauses in a specification. Section 4 presents the main inference rules for our refutation

procedure, proving soundness. We also give an example of refutation for a given specification. Then, in Section 5 we show that some other rules (named unfolding rules) are necessary in order to obtain completeness. We also add some subsumption and simplification rules to produce a more efficient procedure and we study the conditions for the termination of the refutation procedure. Having done this study, we propose in Section 6 a refutation procedure by defining how to apply the rules. Finally, in Section 7 we provide some conclusions and further work.

2 Basic Definitions and Notations

In this section we introduce some basic concepts and notations that are used along the paper.

2.1 XML Documents and Patterns

We consider an *XML document* as an unordered and unranked tree with nodes labelled from an infinite alphabet Σ. The symbols in Σ can represent the element labels, attribute labels, and text values that can occur in documents. A document on $\Sigma = \{a, b, c, d, e, f, g\}$ is given in the right part of Figure 2. By considering that the trees are unordered and unranked, the subtrees can commute (the "sibling ordering" is irrelevant), and there are no restrictions on the number of children a node can have.

Definition 2.1. *Given a signature Σ, a* document on Σ *is a tree t whose nodes are labelled with symbols from Σ and with one sort of edges denoted /. T_Σ denotes the set of all documents on alphabet Σ. Given a document $t \in T_\Sigma$, $Nodes(t)$ and $Edges(t)$ denote respectively the sets of nodes and edges in t; $Root(t)$ denotes its root node; and for each $n \in Nodes(t)$, $Label(n)$ denotes the label of such a node n. $Edges^+(t)$ denotes the transitive closure of $Edges(t)$. Each edge in $Edge(t)$ is represented (x, y) with $x, y \in Nodes(t)$. Each $(x, y) \in Edges^+(t)$ represents a path in t from node x to node y.*

As said above, we use patterns as an alternative representation of queries. In particular, we are interested in boolean patterns. A pattern on $\Sigma = \{a, b, c, d, e, f, g\}$ is given in the left part of Figure 2.

Definition 2.2. *Given a signature Σ, a* pattern on Σ *is a tree p whose nodes are labelled with symbols from $\Sigma \cup \{*\}$ and with two sorts of edges: the descendant edges denoted // and the child edges denoted /. P_Σ denotes the set of all patterns on alphabet Σ. Given a pattern $p \in P_\Sigma$, we use the same notations as before: $Nodes(p)$, $Edges(p)$ and $Root(p)$ for the nodes, the edges and the root of p, and $Label(n)$ for the label of $n \in Nodes(p)$; but now the edges are distinguished: $Edges(p) = Edges_{//}(p) \cup Edges_{/}(p)$. $Edges^+(p)$ denotes the transitive closure of $Edges(p)$, that is, $(x, y) \in Edges^+(p)$ represents a path in p from node x to node y with edges of type / or // along the path.*

For the sake of simplicity, from now on we omit the signature Σ. Along this paper, patterns will be drawn in the figures as trees, but to write them textually we will use the following format: A pattern p with root labelled a and subtrees $p_1, \ldots, p_n$ will be textually written $p = a(!p_1) \ldots (!p_n)$ where each p_i is recursively written in the same format, and ! being / or // to indicate the edge from the root to each subtree p_i. Some parenthesis can be omitted in the case of having only one subtree.

From the previous definitions, it is clear that documents coincide with a special case of patterns: those patterns with no label $*$ and with no edge of type //. For this reason, in the next section we will define the relation between two patterns, and as a particular case we will have defined the relation between a pattern and a document.

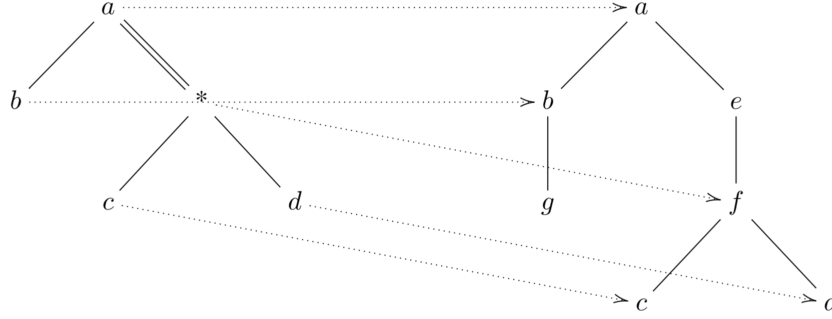


Figure 2: A pattern p, a document t and a monomorphism $h : p \to t$

2.2 Pattern Homomorphisms and Pattern Models

We define here the notion of homomorphism between two patterns and also (as a particular case) the notion of homomorphism between a pattern and a document. The later one will be used to define which documents are the models of a pattern. That is, from a logical point of view, we can see patterns as formulae, documents as structures and we can define a notion of pattern satisfaction.

Definition 2.3. *Given two patterns $p, q \in P$, a* homomorphism *from p to q is a function $h : Nodes(p) \to Nodes(q)$ satisfying the following conditions:*

- *Root-preserving: $h(Root(p)) = Root(q)$;*
- *Label-preserving: For each $n \in (p)$, $Label(n) = *$ or $Label(n) = Label(h(n))$;*
- *Child-edge-preserving: For each $(x, y) \in Edges_/(p)$, $(h(x), h(y)) \in Edges_/(q)$.*
- *Descendant-edge-preserving: For each $(x, y) \in Edges_{//}(p)$, $(h(x), h(y)) \in Edges^+(q)$.*

From now on, we will simply write $h : p \to q$ for $h : Nodes(p) \to Nodes(q)$. The definition of a *homomorphism* from a pattern p to a document t is the previous definition. Note that in this particular case, $Edges(t) = Edges_/(t)$.

Definition 2.4. *Given a pattern $p \in P$ and a document $t \in T$, we say that* t satisfies p*, denoted $t \vDash p$, if there exists a monomorphism (i.e., an injective homomorphism) from p to t. The model set of a pattern p is the set of documents satisfying p: $Mod(p) = \{t \in T \mid t \vDash p\}$*

In Figure 2 there is an example of an injective homomorphism $h : p \to t$ from the pattern $p = a(/b)(//*(/c)(/d))$ to the document $t = a(/e/f(/c)(/d))(/b/g)$. The homomorphism h is drawn with dotted arrows. We can see that the document t satisfies the pattern p because its root is labelled with a, it has a child node labelled b, and it has a descendant node (in the example labelled with f) with two child nodes labelled with c and d respectively. In Figure 3 there is an example of a monomorphism $h : p \to q$ from the pattern $p = *//e$ to the pattern $q = a(/e)(//b/c)$. The monomorphism h is drawn with dotted arrows. The existence of such monomorphism implies that all models of q are also models of p.

The following lemma relates monomorphisms and models for two patterns.

Lemma 2.1. *Given two patterns $p, q \in P$:*

- *If there exists a monomorphism $h : p \to q$ then $Mod(q) \subseteq Mod(p)$.*
- *$Mod(q) \subseteq Mod(p)$ does not imply that there is a monomorphism $h : p \to q$.*

Proof. The first point: Let t be a document in $Mod(q)$, then there exists a monomorphism f from q to t. Then the composition $f \circ h$ is a monomorphism from p to t and therefore the document t is also a model for p. The second point is illustrated with an example in [9] . □

3 Constraints, Clauses and Specifications

We take from [10] the notion of graph constraint to define our notion of pattern constraint. In that paper one sort of constraints is of the form $\exists C$, with C being a graph. Then a given graph G is defined to verify this constraint when G contains C as a subgraph. However, translating the ideas in [10] to our setting is not trivial mainly for two reasons. We deal with patterns that are trees having edges of type //, but the related notion of "path" is not considered for graph constraints. On the other hand, in the setting of graph constraints, models and formulas are both graphs, while in our setting models are documents and formulas are patterns. This difference makes more complicated to apply to our framework results given in [10].

3.1 Constraints and Clauses

We consider three kinds of constraints: positive, negative and conditional constraints. The underlying idea of our constraints is that they should specify that certain patterns must occur (or must not occur) in a given document. For instance, the simplest kind of constraint, $\exists p$, specifies that a given document t should satisfy p. Obviously, $\neg\exists p$ specifies that a given document t should not satisfy p. A more complex kind of constraints is of the form $\forall(c : p \rightarrow q)$ where c is a prefix function. Roughly speaking, this constraint specifies that whenever a document t verifies the pattern p it should also verify the extended pattern q (see Definition 3.3 below). In general we will have clauses formed as disjunctions of these three types of constraints.

Definition 3.1. *Given two patterns p and q, a* prefix function *from p to q is an injective function $c : Nodes(p) \rightarrow Nodes(q)$ that satisfies the following conditions:*

- *Root-identity: $c(Root(p)) = Root(q)$;*
- *Label-identity: For each $n \in Nodes(p)$, $Label(n) = Label(c(n))$;*
- *Child-edge-identity: For each $(x, y) \in Edges_{/}(p)$, $(c(x), c(y)) \in Edges_{/}(q)$;*
- *Descendant-edge-identity: For each $(x, y) \in Edges_{//}(p)$, $(c(x), c(y)) \in Edges_{//}(q)$.*

We will simply write $c : p \rightarrow q$ and we will say that p is a prefix of q. Obviously each prefix function is a monomorphism, but the contrary is not true. See for instance that the monomorphism in Figure 3 is not a prefix function since it violates two conditions: "Label-identity" and "Descendant-edge-identity".

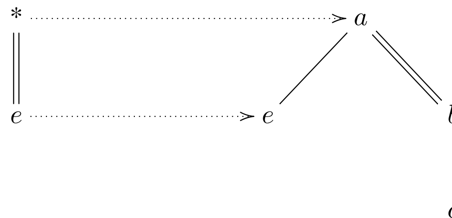


Figure 3: A monomorhism $h : p \rightarrow q$ between two patterns

Definition 3.2. *Given a pattern p, $\exists p$ denotes a* positive constraint *and $\neg\exists p$ denotes a* negative constraint. *A* conditional constraint *is denoted $\forall(c : p \to q)$ where p and q are patterns and $c : p \to q$ is a prefix function.*

A clause *α is a finite disjunction of literals $L_1 \vee L_2 \vee \cdots \vee L_n$, where, for each $i \in \{1, \ldots, n\}$, the literal L_i is a (positive, negative or conditional) constraint. The empty disjunction is called the* empty clause *and it can be represented by $FALSE$.*

Satisfaction of clauses is inductively defined as follows.

Definition 3.3. *A document $t \in T$ satisfies a clause α, denoted $t \models \alpha$, if it holds:*

- *$t \models \exists p$ if $t \vDash p$ (that is, if there exists a monomorphism $h : p \to t$);*
- *$t \models \neg\exists p$ if $t \nvDash p$ (that is, if there does not exist a monomorphism $h : p \to t$);*
- *$t \models \forall(c : p \to q)$ if for every monomorphism $h : p \to t$ there is a monomorphism $f : q \to t$ such that $h = f \circ c$.*
- *$t \models L_1 \vee L_2 \vee \cdots \vee L_n$ if $t \models L_i$ for some $i \in \{1, \ldots, n\}$.*

Let us see what satisfaction of a conditional constraint means. Consider the conditional constraint $\forall(c : p \to q)$ with $p = *//a$, $q = *//a/b$ and c being the obvious prefix function from p to q. By Definition 3.3, a document satisfies this constraint if each node (descendant of the root) labelled a has a child node labelled b. Then the document $t = g(/a/b)(/a/h)$ does not satisfy the constraint. In fact, for the monomorphism $h : p \to t$ that applies the node a in p into the second node a in t, there does not exist a monomorphism $f : q \to t$ such that $h = f \circ c$. However, note that $t \models q$. Therefore, in general, to verify the conditional constraint $\forall(c : p \to q)$ is stronger than to verify the clause $C = \neg\exists p \vee \exists q$.

3.2 Specifications

We assume that a specification consists of a set of clauses. As said in the introduction, our aim is to find a sound and complete refutation procedure for checking satisfiability of specifications consisting of clauses as defined above. Here we give an example of an unsatisfiable specification.

Example 3.1. *Consider the specification $\mathcal{S} = \{C_1, C_2, C_3, C_4\}$ where $C_1 = \exists(*//b) \vee \exists(*//e)$, $C_2 = \forall(c_2 : *//b \to *(//b)(/e))$, $C_3 = \forall(c_3 : *//e \to *(//e)(/b))$, and $C_4 = \neg\exists(*(/b)(/e))$.*
Clause C_1 specifies that the document(s) must have a node labelled b or e; C_2 says that if the document has some node labelled b then its root must have a child node labelled e; similarly, C_3 says that if the document has some node labelled e then the root must have a child node labelled b; and finally, C_4 says that the root cannot have two children nodes labelled b and e. It is easy to test, for instance, that the document $t_1 = a(/b)(/f/e)$ satisfies C_1, C_3 and C_4 but $t_1 \not\models C_2$. Similarly, the document $t_2 = a/e$ satisfies C_1, C_2 and C_4 but $t_2 \not\models C_3$. There is no document satisfying all clauses in $\mathcal{S}$.

4 Rules for a Refutation Procedure

As it is often done in the area of automatic reasoning, the refutation procedure that we present in this paper is defined by means of some inference rules. 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 formulas 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. In our case, we assume that in general the inference rules have the following form, where the premises Γ_1, Γ_2 and the conclusion Γ_3 are clauses [1]:

$$\frac{\Gamma_1 \quad \Gamma_2}{\Gamma_3}$$

A *refutation procedure* for a set of clauses $\mathcal{S}$ is a sequence of inferences: $\mathcal{S}_0 \Rightarrow \mathcal{S}_1 \Rightarrow \cdots \Rightarrow \mathcal{S}_i \Rightarrow \ldots$ where the initial state is the original specification (i.e., $\mathcal{S}_0 = \mathcal{S}$) and where we write $\mathcal{S}_i \Rightarrow \mathcal{S}_{i+1}$ if there is an inference rule such that $\Gamma_1, \Gamma_2 \in \mathcal{S}_i$, and $\mathcal{S}_{i+1} = \mathcal{S}_i \cup \{\Gamma_3\}$.

In general, a refutation procedure for $\mathcal{S}$ is *sound* if whenever the procedure infers the empty clause it holds that $\mathcal{S}$ is unsatisfiable. And a procedure is *complete* if, whenever $\mathcal{S}$ is unsatisfiable, the procedure infers $FALSE$. In this framework, 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 the procedure is *fair*, which means that every inference will eventually be performed (they are not postponed forever).

4.1 Inference Rules

Here we present three inference rules (R1), (R2) and (R3), for our refutation procedure. In our context, the clauses are disjunction of literals where each literal can be of the form $\exists p$, $\neg\exists p$, or $\forall(c : p \rightarrow q)$. We are going to present and explain each rule giving some examples of them.

$$\frac{\exists p_1 \vee \Gamma_1 \quad \neg\exists p_2 \vee \Gamma_2}{\Gamma_1 \vee \Gamma_2} \qquad \textbf{(R1)}$$

if there exists a monomorphism $m : p_2 \rightarrow p_1$

Rule (R1) is similar to the Resolution rule, since the two premises have literals that are, in some sense, “complementary": one is a positive constraint, the other one is a negative one, and the condition about the monomorphism from p_2 to p_1 plays the same role as unification. Note that when Γ_1 and Γ_2 are empty, the rule (R1) infers the empty clause $FALSE$.

For instance, for $p_1 = a(/e)(//*(/c)(/b))$ and $p_2 = *//b$, there exists a monomorphism from p_2 to p_1 that applies the root of p_2 (labelled *) into the root of p_1 (labelled a), the node in p_2 labelled b into the node in p_1 labelled b, and the edge // in p_2 into a path in p_1 formed by // followed by /. Then the empty clause $FALSE$ is obtained as the conclusion of rule (R1) from the premises $\exists p_1$ and $\neg\exists p_2$.

$$\frac{\exists p_1 \vee \Gamma_1 \quad \exists p_2 \vee \Gamma_2}{(\bigvee_{s \in p_1 \otimes p_2} \exists s) \vee \Gamma_1 \vee \Gamma_2} \qquad \textbf{(R2)}$$

Rule (R2) builds a disjunction of positive constraints from two positive constraints. It uses the operator $\otimes$ that we define below. Informally speaking, given two patterns p_1 and p_2, $p_1 \otimes p_2$ denotes the set of patterns that can be obtained by “combining" p_1 and p_2 in all possible ways.

For instance, given the patterns $p_1 = a(/b/e)(//c)$ and $p_2 = a//b/x$, the set $p_1 \otimes p_2$ contains the two patterns: $s_1 = a(/b(/e)(/x))(//c)$ and $s_2 = a(/b/e)(//b/x)(//c)$. Each one corresponds with a way of combining p_1 and p_2; the nodes labelled b are shared in s_1 while there are two different nodes b in $s2$.

The underlying idea is that all patterns s in $p_1 \otimes p_2$ must verify that every document that is a model of s must be a model of p_1 and a model of p_2. Conversely, every document that is a model of both p_1 and p_2 must be a model of some s in $p_1 \otimes p_2$. In some cases $FALSE$ can be inferred by rule (R2).

[1] Clauses are seen as sets of literals, i.e. in a clause written $L \vee \Gamma$, L is not necessarily the leftmost literal in the clause.

Definition 4.1. *Given two patterns p_1 and p_2, $p_1 \otimes p_2$ is defined as the following set of patterns: $p_1 \otimes p_2 = \{s \in P \mid$ there exist jointly surjective monomorphisms $inc_1 : p_1 \rightarrow s$ and $inc_2 : p_2 \rightarrow s\}$ where "joinly surjective" means that $Nodes(s) = inc_1(Nodes(p_1)) \cup inc_2(Nodes(p_2))$.*

Lemma 4.1. *Given two patterns p_1 and p_2, the set of patterns $p_1 \otimes p_2$ is the empty set if and only if p_1 and p_2 have different labels in Σ. Then $\bigvee_{p \in p_1 \otimes p_2} \exists p$ is the clause $FALSE$.*

Proof. If the roots of p_1 and p_2 have different labels in Σ (for instance, a and b) then no combination s is possible since $inc_1 : p_1 \rightarrow s$ implies that the root of s must be labelled a and $inc_2 : p_2 \rightarrow s$ implies that the root of s must be labelled b. Conversely, if the roots of p_1 and p_2 have the same label a (or if one of them is a and the other one is $*$) then the document s with root labelled a and whose set of subtrees is the union of the subtrees of p_1 and p_2 is an element in $p_1 \otimes p_2$. □

$$\frac{\exists p_1 \vee \Gamma_1 \quad \forall(c : p_2 \rightarrow q) \vee \Gamma_2}{(\bigvee_{s \in p_1 \otimes_{c,m} q} \exists s) \vee \Gamma_1 \vee \Gamma_2} \quad \textbf{(R3)}$$

if there is a monomorphism $m : p_2 \rightarrow p_1$ that cannot be extended to $f : q \rightarrow p_1$ such that $f \circ c = m$.

Rule (R3) is similar to (R2) in the sense that it builds a disjunction of positive constraints, but now from a positive constraint $\exists p_1$ and a conditional constraint $\forall(c : p_2 \rightarrow q)$. This rule is applied when there is a monomorphism from p_2 to p_1 that cannot be extended to another one from q to p_1 *via* c. That is, there is a monomorphism $m : p_2 \rightarrow p_1$ but there is no monomorphism $f : q \rightarrow p_1$ such that $f \circ c = m$.

Rule (R3) uses the operator $\otimes_{c,m}$ that we define below. Informally speaking, given a pattern p_1, a prefix function $c : p_2 \rightarrow q$ (that is, a pattern q with prefix p_2) and a monomorphism $m : p_2 \rightarrow p_1$, $p_1 \otimes_{c,m} q$ denotes the set of patterns that can be obtained by combining p_1 and q in all possible ways, but maintaining p_2 shared.

For instance, given the patterns: $p_1 = a(/b/e)(//c/i)$, $p_2 = *//b$, and $q = *(//b//a)(//c/d)$, with the unique possible monomorphism $m : p_2 \rightarrow p_1$ and the unique possible prefix function $c : p_2 \rightarrow q$, the set $p_1 \otimes_{c,m} q$ contains the patterns $s_1 = a(/b(/e)(//a))(//c/i)(//c/d)$ and $s_2 = a(/b(/e)(//a))(//c(/i)(/d))$. Note that s_2 is similar to s_1 but with only one node labelled c.

Definition 4.2. *Given a pattern p_1, a prefix function $c : p_2 \rightarrow q$, and a monomorphism $m : p_2 \rightarrow p_1$, $p_1 \otimes_{c,m} q$ is defined as the following set of patterns: $p_1 \otimes_{c,m} q = \{s \in P \mid$ there exist jointly surjective monomorphisms $inc_1 : p_1 \rightarrow s$ and $inc_2 : q \rightarrow s$ such that $inc_1 \circ m = inc_2 \circ c\}$.*

The underlying idea is that all patterns s in $p_1 \otimes_{c,m} q$ must verify that every document t that is a model of s must be a model of p_1 and a model of q. However, such a document t is not necessarily a model of the conditional constraint $\forall(c : p_2 \rightarrow q)$. Conversely, every document that is a model of both p_1 and $\forall(c : p_2 \rightarrow q)$ must be a model of some s in $p_1 \otimes_{c,m} q$, as it is established in Lemma 4.2.

4.2 Example of Refutation

Consider the specification given in Example 3.1, $\mathcal{S} = \{C_1, C_2, C_3, C_4\}$ with $C_1 = \exists(*//b) \vee \exists(*//e)$, $C_2 = \forall(c_2 : *//b \rightarrow *(//b)(/e))$, $C_3 = \forall(c_3 : *//e \rightarrow *(//e)(/b))$, and $C_4 = \neg\exists(*(/b)(/e))$.
We can prove that this set of clauses is unsatisfiable by applying the inference rules until obtaining the empty clause, in the following way:

1.- (R3) applied to C_1 and C_2 gives $C_5 = \exists(*(//b)(/e)) \vee \exists(*//e)$

2.- (R3) applied to C_5 and C_3 gives $C_6 = \exists(*(//b)(/e)(/b)) \vee \exists(*(/e)(/b)) \vee \exists(*//e)$

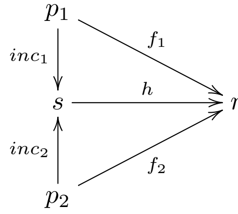
3.- (R1) applied to C_6 and C_4 gives $C_7 = \exists(*(/e)(/b)) \vee \exists(*//e)$

4.- (R1) applied to C_7 and C_4 gives $C_8 = \exists(*//e)$
5.- (R3) applied to C_8 and C_3 gives $C_9 = \exists(*(//e)(/b))$
6.- (R3) applied to C_9 and C_2 gives $C_{10} = \exists(*(//e)(/b)(/e)) \vee \exists(*(/b)(/e))$
7.- (R1) applied to C_{10} and C_4 gives $C_{11} = \exists(*(/b)(/e))$
8.- (R1) applied to C_{11} and C_4 gives $FALSE$.
It may be noted that in step 2, the disjunction $\exists(*(//b)(/e)(/b)) \vee \exists(*(/e)(/b))$ is the result of doing $\bigvee_{s \in p_1 \otimes_{c,m} q} \exists s$ for $p_1 = *(//b)(/e)$ and $\forall(c : p_2 \to q) = C_3$. Similarly in step 6.

4.3 Soundness of the Inference Rules

For proving soundness of a refutation procedure it is enough to prove the soundness of its rules. We present first a property and then prove the soundness of each rule.

Proposition 4.1. ***(Pair Factorization Property)***
Given three patterns p_1, p_2, r, and two monomorphisms $f_1 : p_1 \to r$ and $f_2 : p_2 \to r$, there exists a pattern $s \in p_1 \otimes p_2$, with monomorphisms $inc_1 : p_1 \to s$ and $inc_2 : p_2 \to s$, and there exists a monomorphism $h : s \to r$ such that $h \circ inc_1 = f_1$ and $h \circ inc_2 = f_2$. In the particular case when r is a document, this property means that r is a model of s . Graphically:



Proof. Since f_1, f_2 are monomorphisms, the roots of p_1 and p_2 cannot have different labels in Σ. Moreover, some $s \in p_1 \otimes p_2$ holds this property. Such pattern s must be chosen such that, for every $m \in Nodes(p_1)$ and $n \in Nodes(p_2)$: if $f_1(m) = f_2(n)$ then $inc_1(m) = inc_2(n)$ and if $f_1(m)$ is an ancestor (respectively descendant) of $f_2(n)$, $inc_1(m)$ must not be a descendant (respectively ancestor) of $inc_2(n)$. Then h is well-defined. □

Lemma 4.2. *Rules (R1), (R2), and (R3) are sound.*

Proof. A rule with premises Γ_1 and Γ_2 and conclusion Γ_3 is sound if for every document t: if $t \models \Gamma_1$ and $t \models \Gamma_2$ then $t \models \Gamma_3$.

Rule (R1). Let t be a document and suppose that $t \models \exists p_1 \vee \Gamma_1$, $t \models \neg\exists p_2 \vee \Gamma_2$, and that there exists a monomorphism $m : p_2 \to p_1$. It cannot happen that $t \models \exists p_1$ and $t \models \neg\exists p_2$, since if $t \models \exists p_1$ then there exists a monomorphism $h : p_1 \to t$ and it implies that $h \circ m : p_2 \to t$ is also a monomorphism, meaning that $t \models \exists p_2$. Therefore, $t \models \Gamma_1 \vee \Gamma_2$.

Rule (R2). Let t be a document such that $t \models \exists p_1 \vee \Gamma_1$ and $t \models \exists p_2 \vee \Gamma_2$. The cases where $t \models \Gamma_1$ or $t \models \Gamma_2$ are trivial. Suppose that $t \models \exists p_1$ and $t \models \exists p_2$. It means that there are two monomorphisms $h_1 : p_1 \to t$ and $h_2 : p_2 \to t$. By Proposition 4.1, there exists some $s \in p_1 \otimes p_2$ verifying the *pair factorization property* with $h : s \to t$ being a monomorphism. Then $t \models \exists s$ and therefore $t \models \bigvee_{s \in p_1 \otimes p_2} \exists s$.

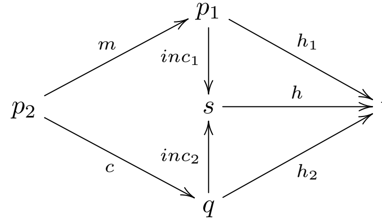
Rule (R3). Let t be a document such that $t \models \exists p_1 \vee \Gamma_1$ and $t \models \forall(c : p_2 \to q) \vee \Gamma_2$, and suppose that the condition of the rule is fulfilled for the monomorphism $m : p_2 \to p_1$. The cases where $t \models \Gamma_1$ or $t \models \Gamma_2$ are again trivial. Suppose that $t \models \exists p_1$ and $t \models \forall(c : p_2 \to q)$, and let us see that $t \models \exists s$ for some s in $p_1 \otimes_{c,m} q$. Since $t \models \exists p_1$, there exists a monomorphism $h_1 : p_1 \to t$. Then $h_1 \circ m$ is also a monomorphism from p_2 to t. From here, since $t \models \forall(c : p_2 \to q)$, there is a monomorphism

$h_2 : q \to t$ such that $h_1 \circ m = h_2 \circ c$. On other hand, we now that for each element s in $p_1 \otimes_{c,m} q$ it holds that $inc_1 \circ m = inc_2 \circ c$. Choose s such that, for each pair of nodes x in p_1 and y in q, the following properties hold:

a) If $h_1(x) = h_2(y)$ then $inc_1(x) = inc_2(y)$.

b) If $h_1(x)$ is an ancestor (respectively descendant) of $h_2(y)$ in t then $inc_1(x)$ is not a descendant (repectively an ancestor) of $inc_2(y)$ in pattern s.

Then we can build a monomorphism $h : s \to t$ verifying $h \circ inc_1 = h_1$ and $h \circ inc_2 = h_2$. Such monomorphism h is defined as follows: For each node z in $inc_1(p_1)$: $h(z) = h_1({inc_1}^{-1}(z))$. For each node z in $inc_2(q)$: $h(z) = h_2({inc_2}^{-1}(z))$. By property a), h is well-defined for the nodes in $inc_1(p_1) \cap inc_2(q)$; by property b), $h : s \to t$ is a monomorphism. Therefore $t \models \exists s$. Graphically:



□

5 Looking for Completeness

We have seen that our refutation procedure consisting of the three inference rules (R1), (R2) and (R3) is sound, that is, whenever the procedure infers the empty clause from a set of clauses $\mathcal{S}$, we have proven that $\mathcal{S}$ is unsatisfiable. However, the procedure is not complete: It may happen that the clause $FALSE$ is not infered by the procedure although $\mathcal{S}$ is unsatisfiable, as the following example shows.

Example 5.1. *Consider the specification $\mathcal{S}$ = {C_1, C_2, C_3} which contains the clauses: $C_1 = \exists(a//b)$, $C_2 = \neg\exists(a/*//b)$, and $C_3 = \neg\exists(a/b)$. Obviously, rules (R2) and (R3) cannot be used here. Rule (R1) cannot be applied to C_1 and C_2, because there is no monomorphism from $(a/*//b)$ to $(a//b)$, and (R1) cannot be applied to C_1 and C_3, because there is no monomorphism from (a/b) to $(a//b)$. However C_1 is semantically equivalent to the clause $C_1' = \exists(a/*//b) \vee \exists(a/b)$, since for every document t it holds: $t \models C_1$ if and only if $t \models C_1'$. Therefore $\mathcal{S}$ is unsatisfiable but our procedure does not infer $FALSE$.*

5.1 Unfold Rules

The problem detected in the previous example can be solved by adding some new rules to our refutation procedure to allow *unfolding* a pattern like $a//b$ in the two cases a/b and $a/*//b$. Then, by transforming C_1 into C_1', the procedure can infer from the set $\{C_1', C_2, C_3\}$ by applying twice the rule (R1).

As $a/*//b$ and $a//*/b$ are equivalent patterns, we will need to have two different ways of unfolding a descendant edge (or //-edge). The two unfolding rules are the following (to indicate a specific edge // in a tree T to be unfolded we will write T[//]):

$\dfrac{\exists p \vee \Gamma}{\exists p_1 \vee \exists p_2 \vee \Gamma}$ **(Unfold1)**	$\dfrac{\exists p \vee \Gamma}{\exists p_1 \vee \exists p_2 \vee \Gamma}$ **(Unfold2)**
for $p = T[//]$: $p_1 = T[/]$ and $p_2 = T[/,*,//]$	for $p = T[//]$: $p_1 = T[/]$ and $p_2 = T[//,*,/]$

The rule (Unfold1) substitutes inside a clause the positive constraint $\exists p$ by $\exists p_1 \vee \exists p_2$, where p_1 (respectively p_2) is obtained from p by substituting an edge // in p by the edge / in p_1 (respectively by the sequence /,*,// in p_2). The rule (Unfold2) is similar, but substituting // by the sequence //,*,/ in p_2. Both rules are sound: for every document t, it holds that $t \models \exists p$ if and only if $t \models \exists p_1 \vee \exists p_2$.

With the two unfolding rules added to our refutation procedure, it is possible to infer the empty clause in more cases than without them, as we have seen in the previous example. Nevertheless, the repeated application of the unfolding rules can be infinite, giving rise to a termination problem. To avoid such a problem, the idea would be to apply the unfolding rules finitely and only in the necessary cases. We show this idea in the following example.

Example 5.2. *Consider the specification $\mathcal{S}$ = {C_1, C_2, C_3} with clauses: $C_1 = \exists(a//c/d)$, $C_2 = \neg\exists(a/*/d)$, and $C_3 = \neg\exists(a/*//*/c)$.*
We can see that is not possible to apply the rule (R1) to C_1 and C_3 since there is no monomorphism from the pattern $q = (a/*//*/c)$ to the pattern $p = a//c/d$. However, it can be detected that a monomorphism would be possible if the edge // from a to c in p, is unfolded until matching with the sequence ///*/ from a to c in q. The form of this sequence tells us to apply first (Unfold1) and then (Unfold2). More precisely: Rule (Unfold1) is applied to C_1 giving $C_1' = \exists(a/c/d) \vee \exists(a/*//c/d)$. Rule (Unfold2) is applied to C_1' giving $C_1'' = \exists(a/c/d) \vee \exists(a/*/c/d) \vee \exists(a/*//*/c/d)$. Now, the rule (R1) can already be applied to C_1'' and C_3 giving $C_4 = \exists(a/c/d) \vee \exists(a/*/c/d)$. To finish, the rule (R1) can be applied to C_4 and C_2 giving $C_5 = \exists(a/*/c/d)$.*

5.2 Subsumption and Simplification Rules

Finally, we can consider another classical notion, the *subsumption* of clauses, to build a more efficient refutation procedure. Given two clauses C and D, C subsumes D (or equivalently D is subsumed by C) if $Mod(C) \subseteq Mod(D)$. For instance, Γ_1 subsumes $\Gamma_1 \vee \Gamma_2$. Subsumed clauses are redundant and it seems obvious that they must be deleted as soon as possible in the refutation procedure. However, we must have into account that, in some cases, introducing deleting rules may cause that a different strategy is needed to prove the completeness of the procedure [11]. Following with the example 5.2, we show now the subsumed clauses that can be deleted in each step of our procedure.

Example 5.3. *Following the previous example we have that: C_1' replaces C_1 after the application of (Unfold1), therefore C_1 is deleted; C_1'' replaces C_1' after the application of (Unfold2), therefore C_1' is deleted; C_4 subsumes C_1'', so C_1'' can be deleted after the first application of (R1); and C_5 subsumes C_4, so C_4 can be deleted after the second application of (R1). Taking into account these subsumptions, the sequence of inferences from the specification $\mathcal{S}$ = {C_1, C_2, C_3} can be then summarized as follows (by applying subsumption as soon as possible):*

$$\mathcal{S} = \{C_1, C_2, C_3\} \Rightarrow \{C_1', C_2, C_3\} \Rightarrow \{C_1'', C_2, C_3\} \Rightarrow \{C_4, C_2, C_3\} \Rightarrow \{C_5, C_2, C_3\}.$$

In this step of the refutation procedure, the set of clauses is $\{\exists(a/*/c/d), \neg\exists(a/*/d), \neg\exists(a/*//*/c)\}$. Now no rule can be applied (note that the unfolding rules are only applied on positive constraints), therefore the procedure finishes. As $FALSE$ has not been inferred, the actual set of clauses (and then also the initial state) is satisfiable. Moreover, the last set of clauses constitutes a new specification that is equivalent to but simpler than $\mathcal{S}$.

Here we present all subsumption and simplification rules that are used in our procedure. Apart from the general subsumption rule we mentioned before (named S1 below), we have defined other three subsumption rules (S2, S3 and S4). The meaning of such rules is that one premise disappears because it

is subsumed by the other premise under some conditions.

$$\frac{\Gamma_1 \vee \Gamma_2 \quad \Gamma_1}{\Gamma_1} \quad \textbf{(S1)}$$

$$\frac{\exists q \vee \Gamma \quad \exists p \vee \Gamma}{\exists q \vee \Gamma} \quad \textbf{(S2)}$$

if there exists a monomorphism $m : p \to q$

$$\frac{\neg\exists p \vee \Gamma \quad \neg\exists q \vee \Gamma}{\neg\exists p \vee \Gamma} \quad \textbf{(S3)}$$

if there exists a monomorphism $m : p \to q$

$$\frac{\forall(c_1 : x_1 \to q_1) \vee \Gamma \quad \forall(c_2 : x_2 \to q_2) \vee \Gamma}{\forall(c_1 : x_1 \to q_1) \vee \Gamma} \quad \textbf{(S4)}$$

if there exist monomorphisms $f : x_1 \to x_2$ and $g : q_2 \to q_1$

Soundness of rules S2 and S3 can be proven by using Lemma 2.1. For S4 note that if $f : x_1 \to x_2$ and $g : q_2 \to q_1$ are monomorphisms then every model of $\forall(c_1 : x_1 \to q_1)$ is a model of $\forall(c_2 : x_2 \to q_2)$.

Other rules that are convenient for a refutation procedure are the simplification rules. We implicitly consider that the clause $\Gamma \vee L \vee L$ is the same as the clause $\Gamma \vee L$, and we add three simplification rules that correspond with the previous subsumption rules (each Sim with the corresponding S):

$$\frac{\exists p \vee \exists q \vee \Gamma}{\exists p \vee \Gamma} \quad \textbf{(Sim2)}$$

if there exists a monomorphism $m : p \to q$

$$\frac{\neg\exists p \vee \neg\exists q \vee \Gamma}{\neg\exists q \vee \Gamma} \quad \textbf{(Sim3)}$$

if there exists a monomorphism $m : p \to q$

$$\frac{\forall(c_1 : x_1 \to q_1) \vee \forall(c_2 : x_2 \to q_2) \vee \Gamma}{\forall(c_2 : x_2 \to q_2) \vee \Gamma} \quad \textbf{(Sim4)}$$

if there exist monomorphisms $f : x_1 \to x_2$ and $g : q_2 \to q_1$

As before, soundness of the simplification rules can be easily proved. To avoid unnecessary steps, the simplifications must be applied as soon as possible in the refutation procedure.

5.3 Termination

Let us study the termination of the refutation procedure: Are there examples of non-termination? Can we obtain the termination under some conditions? To answer these questions we will try to separate the different cases which we must face down.

Example 5.4. *Consider the specification $\mathcal{S}$ = {C_1, C_2, C_3} with clauses $C_1 = \exists(a/b)$, $C_2 = \forall(c_2 : *//b \to *//b/c)$, and $C_3 = \forall(c_3 : *//c \to *//c/b)$.*

C_2 says that each node labelled b must have a child node labelled c and C_3 says that each node labelled c must have a child node labelled b. Since $C_1 = \exists(a/b)$, then a document satisfying the three clauses must necessarily be a document with a infinite branch $a/b/c/b/c/\ldots$. Therefore, rule (R3) can be infinitely applied: First to C_1 and C_2 giving $C_4 = a/b/c$, then to C_4 and C_3 giving $C_5 = a/b/c/b$, etc. Moreover, by the subsumption rule (S2), C_1 is subsumed by C_4, and C_4 is subsumed by C_5, ..., and so on. Note that no more rules can be applied in this example.

The previous example shows that "non-termination" is intrinsically linked with specifications that have only infinite models. As it could expect, the procedure will not stop for such class of specifications.

However, in the case of a satisfiable specification with a finite model, the refutation procedure will stop in a finite number of steps. This result will be obtained if two conditions hold. On one side, as commented before, the procedure must be "fair", which means that inferences are not postponed forever. For instance, suppose we add the clause $C_4 = \neg\exists(*//c)$ in the Example 5.4. Then, we need to apply (R1) eventually to obtain $FALSE$.

The second point to study is more intricate and refers to the application of the unfolding rules. These rules change a positive constraint with a //-edge into the disjunction of two positive constraints, one of which also containing a //-edge. Then obviously these rules can be infinitely applied and they can cause non-termination for a satisfiable specification. We need to control the application of the unfolding rules, using them only if necessary and in a controlled way, as viewed in Examples 5.2 and 5.3. However, to be sure that we will not need to unfold later a concrete //-edge that has already been unfolded, we must unfold it until a maximum size. The following definition help us to formalize this maximum size.

Definition 5.1. *Given a pattern p, a *-sequence of p is a sequence of the form !*!...!*! belonging to (a branch of) p, with all its nodes (labelled *) without more children in p, and each ! being either a /-edge or a //-edge. The length of a *-sequence is the number of nodes in the sequence. Let L_S be the set of all *-sequences that appear either in a negative constraint $\neg\exists q$ or in the premise p of a conditional constraint $\forall(c : p \rightarrow q)$ in a specification S. The star-length of the specification S is defined as $m + 1$ where m = maximum of the lengths of the elements in L_S .*

For instance, for the specification in the Example 5.2, the star-length is 3. This means that we do not need to apply more than 3 times the rules to unfold each //-edge (in a positive constraint). But in general we do not know a priori what is the appropriate combination of unfolding rules (we may need more than one combination), for this reason the number of times will be extended to cover all cases. In concrete, we will proceed as follows: Once we have calculated the star-length l of a given specification S, all the //-edges in each positive constraint $\exists p$ will be unfolded by applying l times the rule (Unfold 1) followed by applying l times the rule (Unfold2). Indeed, these applications can be done in one summarized step.

Again in Example 5.2: C_1 is unfolded (in one step) into the clause $C = \exists(a/c/d) \vee \exists(a/*/c/d) \vee \exists(a/*/*/c/d) \vee \exists(a/*/*/*/c/d) \vee \exists(a/*/*/*/*/c/d) \vee \exists(a/*/*/*/*/*/c/d) \vee \exists(a/*/*/*//*/*/*/c/d)$. Now the rule (R1) can be repeatedly applied to C and the negative constraint C_3, on the third to seventh literals in C, obtaining as a resovent the clause $C_4 = \exists(a/c/d) \vee \exists(a/*/c/d)$ as we expected.

This case study can be extended to any star-length. It is worth mentioning that the definition of the star-length (Definition 5.1) also takes into account the *-sequences appearing in the premises of a conditional constraint because we may also need to unfold the //-edges in the positive constraints to see explicitly all the particular cases that may be extended by the conditional constraints by means of the rule (R3).

6 The Refutation Procedure

Once defined all the rules, we classify them in three groups: the first group $\mathcal{IR}$ consists of the inference rules (R1), (R2) and (R3); the second group $\mathcal{SR}$ consists of the subsumption rules (S1),(S2),(S3), (S4) and the simplification rules (Sim2), (Sim3) and (Sim4); the third group $\mathcal{UR}$ consists of the unfolding rules (Unfold1) and (Unfold2).

Starting from the general definition given in Section 4 and considering now these groups of rules, we can say that a *refutation procedure* for a specification $\mathcal{S}$ is a sequence of states:

$$\mathcal{S}_0 \Rightarrow \mathcal{S}_1 \Rightarrow \cdots \Rightarrow \mathcal{S}_i \Rightarrow \ldots$$

where the initial state is the original specification (i.e., $\mathcal{S}_0 = \mathcal{S}$) and where each state $\mathcal{S}_{i+1}$ is obtained from the previous state $\mathcal{S}_i$ by applying a rule. If this rule is in $\mathcal{IR}$, a new clause Γ is added to $\mathcal{S}_i$ to obtain $\mathcal{S}_{i+1}$. Otherwise, if the rule is in $\mathcal{SR}$ then $\mathcal{S}_{i+1}$ is obtained either deleting a clause in $\mathcal{S}_i$ (by a subsumption rule) or replacing a clause in $\mathcal{S}_i$ by another equivalent clause (by a simplification rule).

We can add the application of a rule in $\mathcal{UR}$ into this scheme, but we would need to keep in $\mathcal{S}_{i+1}$ both the premise and the conclusion of the rule or, alternatively, apply both unfolding rules simultanously and then replace the premise by the two conclusions. However, the application of the rules in $\mathcal{UR}$ must be controlled, therefore we will proceed as explained in the previous section:

Calculate the star-length l of the specification $\mathcal{S}_0$ and apply the rules in $\mathcal{UR}$ in a "great summarized step", by unfolding all //-edges in each positive constraint $\exists p$ by applying l times the rule (Unfold 1) followed by applying l times the rule (Unfold2) and marking the remaining //-edges which are not going to be unfolded anymore. After this preprocess, we obtain a specification $\mathcal{S'}_0$ semantically equivalent to $\mathcal{S}_0$ where all necessary unfoldings (for the //-edges in the present positive constraints) are already done.

From this new specification, the algorithm proceeds by applying the rules in $\mathcal{IR}$ and $\mathcal{SR}$. This chain of states will stop within a state $\mathcal{S}_i$ if the clause $FALSE$ appears in $\mathcal{S}_i$ or when no more rules can be applied to $\mathcal{S}_i$. In the former case, the procedure is finished. In the second case, $\mathcal{S}_i$ is the new input specification and the whole process must be repeated: apply the rules in $\mathcal{UR}$ in a "great summarized step" but now only unfold the //-edges that are non-marked; $\mathcal{S'}_i$ is obtained; then the algorithm will continue by applying the rules in $\mathcal{IR}$ and $\mathcal{SR}$ if such $\mathcal{S'}_i$ is different from $\mathcal{S}_i$. Otherwise, the procedure is finished. Note that, from the aplication of the rule (R3), $\mathcal{S}_i$ may contain new positive constraints with some //-edges marked and some //-edges non-marked.

Graphically, by using the arrow $\longrightarrow$ to express a $\mathcal{UR}$ summarized step, our refutation procedure is:

$$\mathcal{S}_0 \longrightarrow \mathcal{S'}_0 \Rightarrow \mathcal{S}_1 \longrightarrow \mathcal{S'}_1 \Rightarrow \cdots \Rightarrow \mathcal{S}_i \longrightarrow \mathcal{S'}_i \Rightarrow \mathcal{S}_{i+1} \longrightarrow \mathcal{S'}_{i+1} \Rightarrow \cdots \Rightarrow \mathcal{S}_n \ldots$$

Recall that the procedure must be fair (as said in Section 5.3). A suitable implementation will also apply the rules in $\mathcal{SR}$ as soon as possible to get a better performance. Although without a formal proof yet, we claim that this procedure is sound and complete: The specification is unsatisfiable if and only if the clause $FALSE$ is obtained in the process. Moreover, we think that the final outcome is one of the three following results:
a) If the procedure finishes with the clause $FALSE$ in some state, then the specification is unsatisfiable.
b) If the procedure finishes and $FALSE$ does not belong to any state, then the specification is satisfiable and it has a finite model.
c) If the procedure does not stop, then the specification is satisfiable and only has infinite models.

Finally, note that when the specification has no conditional constraints, the algorithm always finishes giving a result in cases a) or b).

7 Conclusion and Further Work

As said in the Introduction, our aim was to define a class of specifications on XML documents and to reason about these specifications. To this extent we have first proposed the specifications as sets of clauses, where a clause is a disjunction of constraints built on boolean XPath-patterns. In particular, we have defined three sorts of constrains: positive, negative, and conditional constraints. We have also defined when a document satisfies a constraint and therefore when a specification is satisfiable.

In order to reason about these specifications, we have studied adequate inference rules to build a sound and complete refutation procedure for checking the satisfiability of a given specification. In particular, we consider three inference rules (R1), (R2) and (R3), which take a similar format than the inference rules for graphs given in [10] but defining the appropriate operators ($p \otimes q$ and $p \otimes_{c,m} q$)

for our setting. Also some subsumption and simplification rules are added. We have proven soundness of all the rules. Then we show that some other rules are needed in order to obtain completeness. In concrete, we introduce the unfolding rules and study a way of apply them finitely. Then, assuming that the procedure is fair, the termination of the procedure rests within the own specification.

We plan to prove formally that the refutation procedure is complete; and with respect to implementation, we are building a prototype [1]. In fact, a first prototype implementing the previously described refutation procedure is available at `http://www.sc.ehu.es/jiwnagom/PaginaWebLorea/SpecSatisfiabilityTool.html`, where we also explain the application's requirements. The algorithm of the refutation procedure is written in Prolog [6] but it also has a Java interface for an easy and friendly use. The main goal of this tool is to test if a given specification is satisfiable or not, showing the history of the execution. It can also be used to test if a given document is a model of a given specification and, as a subproduct, it permits to look for all the relations (monomorphisms) between two patterns or the result of the operations $p \otimes q$ and $p \otimes_{c,m} q$. The results are visually shown and therefore the tool makes these operations to be more understandable.

References

[1] Albors, J., and Navarro, M. *SpecSatisfiabilityTool: A tool for testing the satisfiability of specifications on XML documents*, Proceedings of PROLE 2014, to appear in eptcs (Elect. Proc. in Theoretical Computer Science).

[2] Alpuente. M., Ballis, D., and Falaschi, M. *Automated Verification of Web Sites Using Partial Rewriting*, Software Tools for Technology Transfer, 8 (2006), 565-585.

[3] Benedikt, M., and Koch, C. *XPath Leashed*, ACM Computing Surveys 41, 1 (2008).

[4] Benedikt, M., Fan, W., and Geerts, F. *XPath satisfiability in the presence of DTDs*. JACM 55, 2 (2008).

[5] Bidoit, N., and Colazzo D. *Testing XML constraint satisfiability*. Proceedings of the International Workshop on Hybrid Logic (HyLo 2006). ENTCS 174, 6 (2007), 45-61.

[6] Clocksin, W. F., Mellish, C.S. Programming in Prolog. Springer-Verlag (2003).

[7] Ellmer, E., Emmerich, W., Finkelstein, A., and Nentwich, C. *Flexible Consistency Checking*, ACM Transaction on Software Engineering and Methodology, 12(1) (2003), 28–63.

[8] Jelliffe, R. *Schematron*, Internet Document, http://xml.ascc.net/resource/ schematron/.

[9] Miklau, G., and Suciu, D. *Containment and equivalence for a fragment of XPath*, JACM, 51, 1 (2004).

[10] Orejas, F., Ehrig, H., and Prange, U. *A Logic of Graph Constraints*, Fundamental Approaches to Software Engineering, 11th Int. Conference, FASE 2008. LNCS 4961 (2008) 179-198.

[11] Pichler, R. *Completeness and Redundancy in Constrained Clause Logic*, LNCS 1761 (2000), 221-235.

[12] WORLD WIDE WEB CONSORTIUM. 1999a. *XML path language (XPath) recommendation*, http://www.w3c.org/TR/xpath/.

[13] WORLD WIDE WEB CONSORTIUM. 1999b. *XSL transformations (XSLT). W3C recommendation version 1.0*, http://www.w3.org/TR/xslt.

[14] WORLD WIDE WEB CONSORTIUM. 2001. *XML schema part 0: Primer. W3C recommendation*, http://www.w3c.org/XML/Schema.

[15] WORLD WIDE WEB CONSORTIUM. 2002. *XQuery 1.0 and XPath 2.0 formal semantics. W3C working draft*, http://www.w3.org/TR/query-algebra/.

[16] WORLD WIDE WEB CONSORTIUM. 2007. *XML path language (XPath) 2.0.*