Proving satisfiability of constraint specifications on XML documents [*]

Marisa Navarro
Departamento de LSI
Universidad del País Vasco, San Sebastián
marisa.navarro@ehu.es

Fernando Orejas
Departament de LSI
Universitat Politècnica de Catalunya, Barcelona
orejas@lsi.upc.edu

Abstract

In this paper we study a set of logical rules to prove the satisfiability for a class of specifications on XML documents. Specifications are sets of constrains built on XPath patterns. In [8], it is shown 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 inference rules that are sound and complete. Our aim is to formulate similar notions and to obtain similar results in the framework of XML documents. Nevertheless, the difference between both frameworks makes not easy the adaptation of the graph constraints into our setting. We present here a primer study on how approaching this problem.

1 Introduction

XPath [10, 14] 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 [13, 11, 12], a great amount of papers deal with different aspects on different fragments of XPath. For instance, in [2] 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. More concretely, in [3] it is studied the problem of determining, given a query p (in a given XPath fragment) and a DTD D, whether or not there exists an XML document conforming to D and satisfying p. They show that the complexity ranges from PTIME to undecidible, depending on the XPath fragment and the DTD chosen. The work presented in [4] 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 the following two points. On the one hand, we do not consider any DTD or schema, and we use a simple fragment of XPath, where patterns can be represented by means of simple "tree patterns". In this sense our approach is simpler than the previous ones. However, on the other hand, our aim is to define specifications of classes of XML documents as sets of constraints (of some specific class) 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 or not 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 deduce 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 [1, 5] allowing us to refer to the components of a given class of documents (in particular, using XPath notation). 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,

[*] This work has been partially supported by the spanish CICYT project FORMALISM (ref. TIN2007-66523)

they do not consider the problem of defining deductive tools to analyze specifications, for instance to look for inconsistencies.

Another approach that we know [6] 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, as in the previous approach, Schematron provides no deductive capabilities.

Finally, we consider the work presented in [8]. 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.

Our aim is to follow the main ideas given in [8] and try to apply them to XML documents. To define the constraints on some XPath notation, we select the representation of Xpath queries given in [7]. Miklau and Suciu study the containment and equivalence problems for a class of XPath queries that contain branching, label wildcards and can express descendant relationships between nodes. In particular, they introduce *tree patterns* as an alternative representation of XPath queries consisting of these usual elements: node tests, the child axis (/), the descendent axis (//) and wildcards (*). The answer nodes are marked with (x). For instance, Figure 1 shows a tree pattern p that when is applied to a given XML document t (which is also represented by a tree but in this case without descendent axis or wildcards), it must check: if the root node in t is labeled a, if some child node of the root node in t is labeled b, and if some descendent node of the root node in t has both a child node labeled d and a descendent node labeled c. If all of these conditions are satisfied, the application $p(t)$ will return a set with such last descendents (the marked nodes with x); in other case, 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

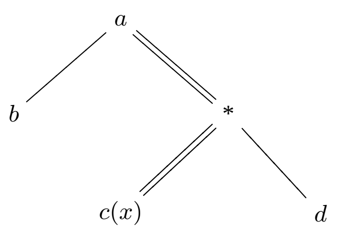

Figure 1: A tree pattern with answer node (marked x)

Boolean tree patterns in [7]). The application of such a pattern to a document t will return $true$, if t satisfies the conditions specified by the pattern, or $false$ in other case.

As we said above, in this paper we follow the main ideas given in [8] and try to apply them to XML documents by defining three sorts of 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 particularization of graph constraints to our setting is not trivial, mainly for two reasons that will be explained in Section 3.

We assume that a specification consists of a set of clauses, where a clause is a disjunction of constraints. To know if an XML document satisfies a specification means to check if it satisfies each of its clauses. However a set of clauses can be unsatisfiable, that is, it may be happen that no document can satisfy all clauses in the specification. To avoid useless work, we consider the task to check if a given specification is satisfiable. 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 similar format than the inference rules given in [8], but again the particularization to our setting needs to define appropriate operators and to prove new results.

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 constraints and clauses that we are going to use to define our specifications. 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 work in progress we are doing in order to obtain completeness for our refutation procedure. Finally, in Section 6 we provide some conclusions.

2 Basic definitions and notation

In this section we introduce some basic concepts and notation that will be used along the paper. Most of them are taken from [7].

2.1 XML documents and patterns

We consider an *XML document* as an unordered and unranked tree with nodes labeled from an infinite alphabet Σ. The symbols in Σ represent the element labels, attribute labels, and text values that can occur in XML documents. T_Σ denotes the set of all trees on alphabet Σ. We also call each element in T_Σ a *document tree*. Given a document tree $t \in T_\Sigma$, $Nodes(t)$ and $Edges(t)$ denote respectively the sets of nodes and edges in the tree 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 $x, y \in Nodes(t)$. If $(x, y) \in Edges^+(t)$ then it represents a path in t from node x to node y.

As said in the Introduction, we use tree patterns as an alternative representation of queries. In particular, we are interested in tree patterns without answer nodes to build constraints. Now we give the definition of a tree pattern and the definition of a function, called *embedding*, that will serve us to define when a document satisfies a pattern.

Definition 2.1 *Given a signature Σ, a* tree pattern *on Σ is a tree p whose nodes are labeled with symbols from $\Sigma \cup \{*\}$ and with two sorts of edges: the descendent edges denoted $//$ and the child edges denoted $/$. P_Σ denotes the set of all patterns on alphabet Σ. We use the same notations as before: $Nodes(p)$, $Edges(p)$, $Root(p)$ and $Label(n)$ for each $n \in Nodes(p)$; but now the edges are distinguished, $Edges(p) = Edges_{//}(p) \cup Edges_{/}(p)$.*

For the sake of simplicity, from now on we omit the signature Σ , and tree patterns are simply called $patterns$. Along this paper, patterns will be drawn in the figures as trees, but to write them textually (in the examples) we will use the following format: A pattern p with root labeled 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. For instance, the pattern given in Figure 1 can be textually written $a(/b)(//*(//c)(/d))$.

Definition 2.2 *Given a pattern $p \in P$ and a document tree $t \in T$, an* embedding *from p to t is a function $e : Nodes(p) \to Nodes(t)$ satisfying the following conditions:*

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

From now on, we will also write $e : p \to t$ for $e : Nodes(p) \to Nodes(t)$. Miklau and Suciu [7] define, for a pattern p and a document tree t, that p(t) is true if there exists an embedding e from p to t. They give an algorithm to decide whether $p(t)$ is true. This algorithm runs in lineal time. In Figure 2 there is an example of an embedding $e : p \to t$ from the pattern $p = a//*(/c)(/d)$ to the document tree $t = a(/e/f(/c)(/d))(/b/g)$. The embedding e is drawn with dotted arrows.

2.2 Pattern satisfaction and pattern homomorphism

In the following we define the notion of (pattern) satisfaction, and as it is usual in logic, from this notion we can define the notion of (pattern) model.

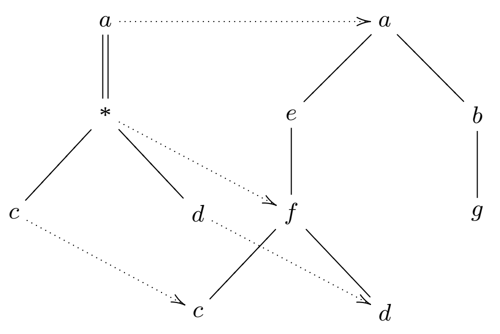
Figure 2: An embedding $e : p \rightarrow t$

Definition 2.3 *Given a pattern $p \in P$ and a document tree $t \in T$, we say that* t satisfies p, *denoted $t \vDash p$, if there exists an injective embedding from p to t. The model set of a pattern p is the set of document trees satisfying p:*

$$Mod(p) = \{t \in T \ / \ t \vDash p\}$$

We must point out here a main difference between our approach and the approach given in [7]. Our notion of satisfaction ask the embedding to be injective; that is, $p(t)$ is $true$ if there exists an *injective* embedding from p to t. However, in [7], the embedding is defined not necessarily injective.

Our definition is useful to distinguish repeated nodes as different ones. For example: Let p be the pattern textually written $a(/b)(/b)(/c)$, that is, the pattern with root labelled a and three children labelled b, b and c respectively. Let q be the pattern textually written $a(/b)(/c)$, that is, the pattern with root labelled a and two children labelled b and c. In our approach, each model t for p must be a document tree with a root labelled a and at least three children labelled b, b and c respectively; therefore each model t for p is also a model for q. On the other hand, there exist trees that are models for q but not models for p, for instance: $t = a(/b)(/c)$. That is, $Mod(p) \subset Mod(q)$ and $Mod(q) \not\subset Mod(p)$. However, in [7], such patterns p and q would have the same models.

Now we define the notion of homomorphism between patterns, similar to the notion of embedding between a pattern and a document tree. Again we shall use only *injective* homomorphisms.

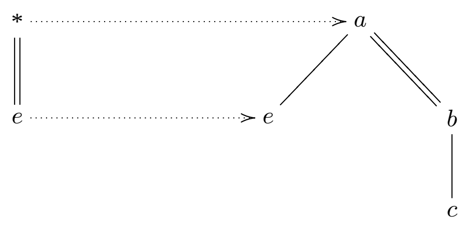
Figure 3: A monomorhism $h : p \rightarrow q$

Definition 2.4 *Since each pattern can be seen as a tree with possible labels * and edges of type / or // , we can define a* homomorphism *from a pattern p to a pattern q as a function $h : Nodes(p) \rightarrow Nodes(q)$ that preserves all conditions of an embedding but with the additional condition that each edge / in p must be applied into an edge / in q. That is, we change only the following condition:*

- *Child-edge-preserving: For each $(x, y) \in Edges_/(p)$, $(e(x), e(y)) \in Edges_/(q)$.*

Note that the condition of "Descendent-edge-preserving" remains as in Definition 2.2. Now $Edges^+(q)$ denotes the transitive closure of $Edges(q) = Edges_{//}(q) \cup Edges_/(q)$; that is, $(x, y) \in Edges^+(q)$ represents a path in q from node x to node y with edges of type / or // .

Again we will simply write $h : p \rightarrow q$ for $h : Nodes(p) \rightarrow Nodes(q)$. As usual, an injective homomorphism is called a *monomorphism*. Similarly, an injective embedding will be called a *mono-embedding*. In Figure 3 there is an example of a monomorphism $h : p \rightarrow q$ from the pattern $p = *//e$ to the pattern $q = a(/e)(//b/c)$. The monomorphism h is drawn with dotted arrows.

The following result relates monomorphisms and models. The first point is easy to prove. The second point is illustrated with an example in [7] .

Lemma 2.1 *For p, q patterns:*

- *If there exists a monomorphism $h : p \rightarrow q$ then $Mod(q) \subseteq Mod(p)$.*
- *It may happen that $Mod(q) \subseteq Mod(p)$ but there is not any monomorphism $h : p \rightarrow q$.*

3 Constraints and clauses

We take from [8] 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. Here we are dealing with constraints as some sort of formulas that we want that a document verifies. The particularization to our setting is not trivial mainly for two reasons: On the one hand, although a tree is a particular case of a graph, we deal with patterns that are trees having edges of type $//$. For instance, with a pattern constraint like $\exists p$ being p the pattern textually written $a//b$, we can specify that there must exist a descendent node labelled b in our document with root labelled a. However, graphs constraints do not work with the relation "descendent". On the other hand, in the setting of graph constraints, the models and the formulas (or constraints) are both graphs, while in our setting the models are documents and the formulas are patterns. This second difference makes more complicated to instance the results given in [8] into our framework.

3.1 Constraints

The underlying idea of our constraints is that they should specify that certain patterns must be satisfied (or must not be satisfied) 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 the pattern p is a prefix tree of the pattern q, indicated by the monomorphism c. Roughly speaking, this constraint specifies that whenever a document t verifies the pattern p it should also verify the extended pattern q (see the formal definition 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 function $c : Nodes(p) \rightarrow Nodes(q)$ is a* prefix function *if 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 $(x, y) \in Edges_/(p)$, $(c(x), c(y)) \in Edges_/(q)$;*
- *Descendent-edge-identity: For each $(x, y) \in Edges_{//}(p)$, $(c(x), c(y)) \in Edges_{//}(q)$.*

We will simply write $c : p \rightarrow q$. Obviously each prefix function is a monomorphism.

Now we formally define the constraints we are going to use: positive and negative basic constraints and conditional constraints. A constraint clause is a disjunction of constraints.

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

Definition 3.3 *A* constraint clause *(or simply* clause*) α is a finite disjunction of literals $L_1 \vee L_2 \vee \ldots \vee L_n$, where, for each $i \in \{1, \ldots, n\}$, the literal L_i is a (positive or negative) basic constraint or a conditional constraint. The empty disjunction is called the* empty clause *and it can be represented by* FALSE.

Satisfaction of constraint clauses is defined inductively, following the intuitions described above.

Definition 3.4 *A document tree $t \in T$ satisfies a constraint clause α, denoted $t \models \alpha$, if it holds:*

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

3.2 Example of specification

We assume that our specifications consist of constraint clauses. To know if an XML document t satisfies an specification $\mathcal{C}$ means to check if $t \models \alpha$, for every clause $\alpha \in \mathcal{C}$. However a set of clauses can be unsatisfiable, that is, it may be happen that no document satisfies all clauses in $\mathcal{C}$. To avoid useless work, we should consider to check first if a given specification is satisfiable. Our aim is to find a sound and complete refutation procedure for checking satisfiability of specifications consisting of constraint clauses as defined above.

First let us see with some examples what does the satisfaction of a conditional constraint mean. Then we give an example of an unsatisfiable specification.

We consider the conditional constraint $\forall(c : p \rightarrow q)$ with $p = *//a$, $q = *//a/b$ and c being the obvious prefix function from p to q. By Definition 3.4, a document tree t satisfies this constraint if each node (descendent of the root) labeled a has a child node labeled b. For instance, the document tree t = $g(/a/b)(/a/h)$ does not satisfy the constraint: For the mono-embedding $e : p \rightarrow t$, that applies the node a in p into the second node a in t, there does not exist a mono-embedding $f : q \rightarrow t$ such that $e = f \circ c$. In words: "this mono-embedding e from p to t can not be extended to another one from q to t". However, note that $t \models p$ and $t \models q$. Therefore, in general, to verify the conditional constraint $\forall(c : p \rightarrow q)$ is stronger than to verify the clause C = $\neg\exists p \vee \exists q$, that may be seen as a conditional clause.

Example 3.1 *Consider the specification $\mathcal{C}$ = {C_1, C_2, C_3, C_4} where 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 = $\neg\exists(*(/b)(/e))$.*
Clause C_1 specifies that the document tree must have a node b or e; C_2 says that if the document tree has some node b then its root must have a node child e; similarly, C_3 says that if the document tree has some node e then the root must have a node child b; and finally, C_4 says that the root cannot have two children b and e.
The document $t_1 = a(/b)(/f/e)$ satisfies C_1, C_3 and C_4 but $t_1 \not\models C_2$. 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 $\mathcal{C}$.

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 form:

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

where the premises Γ_1, Γ_2 and the conclusion Γ_3 are (constrained) clauses. Clauses are seen as sets of literals. This means that if we write that a clause has the form $L \vee \Gamma$, this does not necessarily imply that L is the leftmost literal of the given clause. $L \vee \Gamma$ denotes a clause with literal L and Γ the rest of the disjunction. Similarly, we consider that the clause $\Gamma \vee L$ is the same as the clause $\Gamma \vee L \vee L$.

Then, a *refutation procedure* for a set of constraint clauses $\mathcal{C}$ is a sequence of inferences:

$$\mathcal{C}_0 \Rightarrow \mathcal{C}_1 \Rightarrow \ldots \Rightarrow \mathcal{C}_i \Rightarrow \ldots$$

where the initial state is the original specification (i.e., $\mathcal{C}_0 = \mathcal{C}$) and where we write $\mathcal{C}_i \Rightarrow \mathcal{C}_{i+1}$ if there is an inference rule such that $\Gamma_1, \Gamma_2 \in \mathcal{C}_i$, and $\mathcal{C}_{i+1} = \mathcal{C}_i \cup \{\Gamma_3\}$. Moreover, we will assume that $\mathcal{C}_i \subset \mathcal{C}_{i+1}$, i.e. $\Gamma_3 \notin \mathcal{C}_i$, to avoid useless inferences.

In this framework, proving the unsatisfiability of a set of constraints means inferring the empty clause (*FALSE*), 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 procedure is *fair*, which means that, if at any moment i, there is a possible inference $\mathcal{C}_i \Rightarrow \mathcal{C}_i \cup \{\Gamma\}$, for some clause Γ, then at some moment j we have that $\Gamma \in \mathcal{C}_j$. This means that inferences are not postponed forever, i.e. every inference will eventually be performed.

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 *FALSE*.

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.

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.

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

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 basic constraint, the other one is a negative one, and the condition about the monomorphism from p_2 to p_1 plays the same role than unification. Note that when Γ_1 and Γ_2 are empty, the rule (R1) infers the empty clause.

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

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

Rule (R2) builds a disjunction of positive basic constraints from two positive basic 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 labeled 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 tree that is a model of s must be a model of p_1 and a model of p_2. Conversely, every document tree that is a model of both p_1 and p_2 must be a model of some s in $p_1 \otimes p_2$. It must be noted that if the roots of p_1 and p_2 have different labels, for instance are labeled a and b, then no combination is possible. This implies that, in some cases, the empty clause can also be produced by rule (R2).

Now we formalize these ideas within the following definitions.

Definition 4.1 *Given two patterns p and q, $p \otimes q$ is defined as the set of patterns: $p \otimes q = \{s \in P$ / there exist jointly surjective monomorphisms $inc1 : p \rightarrow s$ and $inc2 : q \rightarrow s\}$ where “joinly surjective" means that $Nodes(s) = inc1(Nodes(p)) \cup inc2(Nodes(q))$.*

Definition 4.2 *$join : (\Sigma \cup \{*\}) \times (\Sigma \cup \{*\}) \rightarrow \Sigma \cup \{*\}$ is a partial function which returns a label as the result of joining two labels:*

- *$join(a, a) = join(*, a) = join(a, *) = a$, for each label $a \in \Sigma$;*
- *$join(*, *) = *$;*
- *$join(a, b) = undefined$, for $a, b \in \Sigma$ and $a \neq b$.*

Lemma 4.1 *Given two patterns p and q, the set of patterns $p \otimes q$ is the empty set if and only if join(Label(Root(p)),Label(Root(q))) = undefined.*

Note that if the set $p_1 \otimes p_2$ is the empty set then $\bigvee_{p \in p_1 \otimes p_2} \exists p$ is the clause $FALSE$.

Proposition 4.1 ***(Pair Factorization Property)*** *Given three patterns p, q, r, and two monomorphisms f1: p→r and f2: q→r, there exists a pattern s∈p⊗q and monomorphisms inc1: p→s, inc2: q→s, and h: s→r such that $h \circ inc1 = f1$ and $h \circ inc2 = f2$. In the particular case when r is a document tree, $f1$, $f2$ and h are mono-embeddings. Graphically:*

$$\begin{tikzcd}
p \arrow[d, "inc1"'] \arrow[dr, "f1"] & \\
s \arrow[r, "h"] & r \\
q \arrow[u, "inc2"] \arrow[ur, "f2"'] &
\end{tikzcd}$$

Proof. Since $f1$, $f2$ are monomorphisms, $join(Label(Root(p)), Label(Root(q)))$ is defined. Moreover, some $s \in p \otimes q$ holds this property. Such pattern s must be chosen such that, for every $m \in Nodes(p)$ and $n \in Nodes(q)$: if $f1(m) = f2(n)$ then $inc1(m) = inc2(n)$ and if $f1(m)$ is an ancestor (respectively descendent) of $f2(n)$, $inc1(m)$ must not be a descendent (respectively ancestor) of $inc2(n)$. Then h is well-defined. ■

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

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

Rule (R3) is similar to rule (R2) in the sense that given a positive basic constraint $\exists p_1$ and a conditional constraint $\forall(c : p_2 \rightarrow q)$, it builds a disjunction of positive basic constraints. 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 two patterns p_1, p_2, a prefix function $c : p_2 \rightarrow q$, 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.

Definition 4.3 *Given two patterns p_1, p_2, 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:*
$p_1 \otimes_{c,m} q$ = {$s \in P$ / there exist jointly surjective monomorphisms $inc1 : p_1 \rightarrow s$ and $inc2 : q \rightarrow s$ such that $inc1 \circ m$ = $inc2 \circ c$}. Graphically:

$$\begin{tikzcd}
 & p_1 \arrow[dr, "inc1"] & \\
p_2 \arrow[ur, "m"] \arrow[dr, "c"'] & & s \\
 & q \arrow[ur, "inc2"'] &
\end{tikzcd}$$

Since c is a prefix function (and m is a monomorphism), $join(Label(Root(p_1)), Label(Root(q)))$ is defined. Moreover, $p_1 \otimes_{c,m} q$ is always a non-empty set. The patterns s in $p_1 \otimes_{c,m} q$ are obtained by adding to p_1 all nodes (and edges) in $q - c(p_2)$ *in the place where m indicates*. In particular, at least one s in $p_1 \otimes_{c,m} q$ can be obtained giving the following steps:

First let $s = p_1$ and $inc1 = identity$. Second, let $inc2(n) = inc1(m(c^{-1}(n)))$ for each node n in $c(p_2)$. Finally, extend $inc2$ by adding into s the following subtrees of q: If n is a node in $c(p_2)$ and $subtr$ is a maximal subtree of n in q formed with nodes all in $q - c(p_2)$ then add $subtr$ as a maximal subtree of the node $inc2(n)$.

For instance, given the following patterns: $p_1 = a(/b/e)(//c/i)$, $p_2 = *//b$, and $q = *(//b//a)(//c/d)$, the unique monomorphism $m : p_2 \rightarrow p_1$ and the prefix function $c : p_2 \rightarrow q$, the pattern obtained by following the steps explained above is $s_1 = a(/b(/e)(//a))(//c/i)(//c/d)$. However, in this case, the set $p_1 \otimes_{c,m} q$ also contains the pattern $s_2 = a(/b(/e)(//a))(//c(/i)(/d))$ that is similar to s_1 but with only one node labeled c.

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

4.2 Soundness of the inference rules

For proving soundness of a refutation procedure it is enough to prove the soundness of the inference rules.

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 tree t: if $t \models \Gamma_1$ and $t \models \Gamma_2$ then $t \models \Gamma_3$.

(Rule R1). Let t be a document tree and suppose that $t \models \exists p_1 \vee \Gamma_1$, $t \models \neg\exists p_2 \vee \Gamma_2$, and 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 mono-embedding $h : p_1 \to t$ and this implies that $h \circ m : p_2 \to t$ is also a mono-embedding, meaning that $t \models \exists p_2$. Therefore, $t \models \Gamma_1 \vee \Gamma_2$.

Rule (R2). Let t be a document tree 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 $t \models \exists p_2$. This means that there are two mono-embedings $e_1 : p_1 \to t$ and $e_2 : p_2 \to t$. 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 mono-embedding. Then $t \models \exists s$ and therefore $t \models \bigvee_{p \in p_1 \otimes p_2} \exists p$.

Rule (R3). Let t be a document tree 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 mono-embedding $e_1 : p_1 \to t$. Then $e_1 \circ m$ is also a mono-embedding from p_2 to t. From here, since $t \models \forall(c : p_2 \to q)$, there is a mono-embedding $e_2 : q \to t$ such that $e_1 \circ m = e_2 \circ c$. On other hand, we now that for each element s in $p_1 \otimes_{c,m} q$ it holds that $inc1 \circ m = inc2 \circ c$. Now we can choose one s such that, for each pair of nodes, x in p_1 and y in q, the following properties hold:

a) If $e_1(x) = e_2(y)$ then $inc1(x) = inc2(y)$.

b) If $e_1(x)$ is an ancestor (respectively descendent) of $e_2(y)$ in t then $inc1(x)$ is not a descendent (repectively an ancestor) of $inc2(y)$ in pattern s.

Then we can build a mono-embedding $e : s \to t$ verifying $e \circ inc1 = e_1$ and $e \circ inc2 = e_2$. Such an embedding e is defined as follows: For each node z in $inc1(p_1)$: $e(z) = e_1(inc1^{-1}(z))$. For each node z in $inc2(q)$: $e(z) = e_2(inc2^{-1}(z))$.
By property a), e is well-defined for the nodes in $inc1(p_1) \cap inc2(q)$; by property b), $e : s \to t$ is a mono-embedding. Therefore $t \models \exists s$. Graphically:

$$\begin{tikzcd}
& p_1 \arrow[d, "inc1"'] \arrow[dr, "e_1"] & \\
p_2 \arrow[ur, "m"] \arrow[dr, "c"'] & s \arrow[r, "e"] & t \\
& q \arrow[u, "inc2"] \arrow[ur, "e_2"'] &
\end{tikzcd}$$

■

4.3 Example of refutation

Consider the specification given in Example 3.1, $\mathcal{C} = \{C_1, C_2, C_3, C_4\}$ with constraint clauses: $C_1 = \exists(*//b) \vee \exists(*//e)$, $C_2 = \forall(c_2 : *//b \to *(//b)(/e))$, $C_3 = \forall(c_3 : *//e \to *(//e)(/b))$, and $C_4 = \neg\exists(*(/b)(/e))$.

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 $\bigvee_{p \in p_1 \otimes_{c,m} q} \exists p$ for $p_1 = *(//b)(/e)$ and $\forall(c : p_2 \to q) = C_3$. Similarly in step 6.

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 constrained clauses $\mathcal{C}$, we have proven that $\mathcal{C}$ is unsatisfiable. However, our procedure is not complete. It cannot infer *FALSE* for some unsatisfiable set $\mathcal{C}$ as the following example shows.

Example 5.1 *Consider the specification* $\mathcal{C}$ = {C_1, C_2, C_3} *with constraint 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 does not exist a monomorphism from* $(a/*//b)$ *to* $(a//b)$*. And (R1) cannot be applied to* C_1 *and* C_3*, because there does not exist a monomorphism from* (a/b) *to* $(a//b)$*. However it is clear that* C_1 *is equivalent to the clause* $C'_1 = \exists(a/*//b) \vee \exists(a/b)$*, since for every document tree* t *it holds:* $t \models C_1$ *if and only if* $t \models C'_1$*. Therefore* $\mathcal{C}$ *is unsatisfiable but our procedure does not infer* FALSE.

The problem detected in the previous example can be resolved by adding to our refutation procedure some new rules 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 *FALSE* 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 descendent edge //. To indicate the specific edge // in a tree T to be unfolded we will write T[//]. The two unfolding rules are the following:

$$\frac{\exists p \vee \Gamma}{\exists p_1 \vee \exists p_2 \vee \Gamma} \quad \textbf{(Unfold1)}$$

for $p = T[//] : p_1 = T[/]$ and $p_2 = T[/, *, //]$

$$\frac{\exists p \vee \Gamma}{\exists p_1 \vee \exists p_2 \vee \Gamma} \quad \textbf{(Unfold2)}$$

for $p = T[//] : p_1 = T[/]$ and $p_2 = T[//, *, /]$

The rule (Unfold1) substitutes inside a clause the positive basic 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 tree t, it holds that if $t \models \exists p$ then $t \models \exists p_1 \vee \exists p_2$.

5.1 About termination and completeness

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.

The idea is to apply finitely the unfolding rules, only in the necessary cases. In concrete, if we have the two premises of Rule (R1) with complementary basic constraints $\exists p$ and $\neg\exists q$ and there is no monomorphism from q to p, then we are able to unfold an edge // in p so many times as q indicates. We need to look for sequences in q of the form !*!...!*! with n nodes *, each ! is either / or // but at least one of them must be //. Then we unfold the edge // in p exactly n times. Moreover, the sequence !*!...!*! in q tells us which unfolding rule must be applied. We show this idea in the following example.

Example 5.2 *Consider the specification* $\mathcal{C}$ = {C_1, C_2, C_3} *with constraint clauses:* $C_1 = \exists(a//c/d)$, $C_2 = \neg\exists(a/*/d)$, *and* $C_3 = \neg\exists(a/*//*/c)$.

Rule (R3) is not used here because there is no conditional constraint. Rule (R2) can not be applied since $\mathcal{C}$ *has only one positive constraint. 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 first to apply (Unfold1) to* $(a//c)$ *to obtain* $(a/*//c)$*, and then to apply (Unfold2) to* $(a/*//c)$ *to obtain* $(a/*//*/c)$*. More precisely: Rule (Unfold1) is applied to* C_1 *giving* $C_4 = \exists(a/c/d) \vee \exists(a/*//c/d)$*. Rule (Unfold2) is applied to* C_4 *giving* $C_5 = \exists(a/c/d) \vee \exists(a/*/c/d) \vee \exists(a/*//*/c/d)$.

Now, the rule (R1) can already be applied to C_5 and C_3 giving $C_6 = \exists(a/c/d) \vee \exists(a/*/c/d)$. To finish, the rule (R1) can be applied to C_6 and C_2 giving $C_7 = \exists(a/*/c/d)$.

Finally, we can consider another classical notion, the *subsumption* of clauses, to build a more efficient refutation procedure. 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 [9]. Following with the previous example, we show now the subsumed clauses that can be deleted in each step of our procedure.

Example 5.3 *Taking into account that, given two clauses C and D, C subsumes D (or equivalently, D is subsumed by C) if $Mod(C) \subseteq Mod(D)$, in the previous example we have that: C_4 replaces C_1 after the application of (Unfold1), therefore C_1 is deleted; C_5 replaces C_4 after the application of (Unfold2), therefore C_4 is deleted; C_6 subsumes C_5, so C_5 can be deleted after the first application of (R1); and C_7 subsumes C_6, so C_6 can be deleted after the second application of (R1). Taking into account these subsumptions, the sequence of inferences from the specification $\mathcal{C} = \{C_1, C_2, C_3\}$ can be then summarized as follows:*

$$\mathcal{C} \Rightarrow \{C_4, C_2, C_3\} \Rightarrow \{C_5, C_2, C_3\} \Rightarrow \{C_6, C_2, C_3\} \Rightarrow \{C_7, C_2, C_3\}.$$

In this step of the refutation procedure, the actual state is the set of clauses $\{C_7, C_2, C_3\} = \{\exists(a/*/c/d), \neg\exists(a/*/d), \neg\exists(a/*//*/c)\}$. In this moment, no rule can be applied (note that the unfolding rules are only applied on positive basic constraints) and therefore the procedure finishes. As FALSE *has not been inferred, the actual set of clauses (and then also the initial state) is satisfiable.*

6 Conclusion

As said in the Introduction, our aim is to define a class of specifications on XML documents and to reason about these specifications. In this paper, we first propose 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 and negative basic constraints, and conditional constraints. We define when a document satisfies a constraint and therefore when a specification is satisfiable.

In order to reason about these specifications, we study adequate inference rules to find 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 [8] but defining the appropriate operators ($p \otimes q$ and $p \otimes_{c,m} q$) for our setting. We prove soundness of the refutation procedure. Then we show that some other inference rules are needed in order to obtain completeness. In concrete, we introduce two unfolding rules and also the idea of using subsumption rules. This part of the paper shows work in progress. It is informally presented by means of examples, where we can observe that the unfolding rules must be applied in some specific way to avoid termination problems of the procedure. We need to set up clearly the use of the unfolding rules and to define the subsumption rules. Then we plan to define formally the refutation procedure, using all the above inference rules, and to prove that it is complete.

Finally, as regards termination, we think that our refutation procedure may not terminate, which means that the procedure would be just refutationally complete. However, if we restrict our logic to the basic constraints then we think that the refutation procedure would terminate.

References

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

[2] Benedikt, M., and Koch, C. *XPath Leashed*, ACM Computing Surveys, Vol 41, N^{o} 1, Article 3 (2008).

[3] Benedikt, M., Fan, W., and Geerts, F. *XPath satisfiability in the presence of DTDs*. Proceedings of the 24th Symposium on Principles of Database Systems (PODS 2005). Journal of the ACM 55, n. 2 (2008).

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

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

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

[7] Miklau, G., and Suciu, D. *Containment and equivalence for a fragment of XPath*, Journal of the ACM, Vol. 51, N^o 1, (2004) 2-45.

[8] 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.

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

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

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

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

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

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