On the Analysis of Queries with Counting Constraints*

Everardo Bárcenas
INRIA
Everardo.Barcenas-Patino@inria.fr

Pierre Genevès
CNRS
Pierre.Geneves@inria.fr

Nabil Layaïda
INRIA
Nabil.Layaida@inria.fr

ABSTRACT

We study the analysis problem of XPath expressions with counting constraints. Such expressions are commonly used in document transformations or programs in which they select portions of documents subject to transformations. We explore how recent results on the static analysis of navigational aspects of XPath can be extended to counting constraints. The static analysis of this combined XPath fragment allows to detect bugs in transformations and to perform many kinds of optimizations of document transformations. More precisely, we study how a logic for finite trees capable of expressing upward and downward recursive navigation, can be equipped with a counting operator along regular path expressions.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—Query Languages; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—modal logic

General Terms

Algorithms, Languages, Theory, Verification

Keywords

XML, XPath, Type Checking, Counting Constraints, Modal Logics

1. INTRODUCTION

Since its introduction a decade ago, Extensible Markup Language XML, has gained considerable interest from industry and now plays a central role in modern information system infrastructures. Originally introduced to represent document classes, it became the key technology for describing and exchanging a wide variety of data on the Web. The essence of XML consists in organizing information in tree-tagged structures conforming to some constraints expressed using standard type languages such as DTDs, XML schemas and Relax NG.

In the context of document engineering, XML processing can be seen as transforming theses structures using tree-oriented query languages such as XPath expressions and XQuery within full blown transformation languages such as XSLT. The main purpose of such tasks consists in producing formatted views, generate indexes and tables of contents, etc., of such document or converting them from one format to another. Such tasks can also be pipelined using other technologies such as XProc resulting in complex processes where the potential of failures increases.

One of the biggest challenges for such processing chains is to ensure their correctness and efficiency automatically. One kind of correctness is central for document manipulations is the type or schema safety. It consists in providing guarantees that a given transformation yields always valid documents (w.r.t. some schema) when the input is also valid (w.r.t. some schema). Static type safety analysis and optimization consists in providing such guarantees or optimizations at compile time or prior to execution when the language is interpreted.

To this end, there is a need to solve some basic analysis tasks involving very complex constructions such as XML types (regular tree types) and powerful navigational primitives (XPath queries). In particular, every future XML program or XSLT transformation analyzer will have to routinely solve problems such as:

- XPath query emptiness in the presence of a schema: if one can decide at compile time that a query is not satisfiable then subsequent bound computations can be avoided,
- query equivalence, which is important for query reformulation and optimization,
- path type-checking, for ensuring at compile time that invalid documents can never arise as output of XML processing code.

The main difficulty in such decision problems is that the analysis of paths must be performed against a possibly infinite set of trees. This is because document instances against which paths need to be evaluated are not known during program or transformation analysis. In addition, other important features increase the difficulty of solving such problems

*This work has been partially funded by Agence Nationale de la Recherche, decision ANR-08-DEFIS-004.

DocEng'09, September 16–18, 2009, Munich, Germany.

or make them intractable: upward and downward navigation, comparisions of data values of infinite domains such as value joins and general cardinality constraints on node sets.

The computational cost of analysis considering combinations of some of such features or all of them, ranges from polynomial to undecidable [2, 8]. The cost of the analysis of the full navigational aspect of XPath is known [5, 2] to be EXPTIME.

XML type languages such as XML Schema and DTDs imposed certain structure restrictions on the possible sequences of children. Such restrictions are known to be expressible by means of regular expressions, that is, XML types are subsets of regular tree types [7]. Among the structure restrictions of regular tree types we find numerical restrictions on the number of children (direct siblings). Such simple restrictions can be expressed in terms of regular tree types but at the cost of an exponential blow-up in the size of the expressions. For example, to impose that the regular expression $e = a^*ba^*$ contains at least two elements a, one would need to express it as $e = aaa^*ba^*|aa^*baa^*|a^*baaa^*$. The blow-up is even worse when considering regular tree types since element occurrences need to be considered at different locations of the tree.

Related work and contributions

Among the recent and most expressive formalisms to reason about finite trees, we find [10, 5]. [10] proposes a modal logic capable of expressing both, downward and upward recursive navigation with a computational cost of $2^{O(n \log m)}$, where n is the size of a formula and m is the number of modal cycles in the scope of least fixpoints. In [5], this bound is improved to $2^{O(n)}$ and a linear translation of XPath and regular tree types into the logic is also presented. In addition, an efficient BDD-based implementation of the satisfiability-checking algorithm together with XPath analysis task performance evaluations are given. The extensions explored in this paper are partly based on the results of [5].

Close in spirit to this work, a modal logic is introduced in [4], called sheaves logic, where cardinality constraints can be set on children nodes, that is, restrictions like *p_1 nodes have no more "children" than p_2 nodes*, are expressible by this approach. [9] introduces a fixpoint presburger logic, such that, besides cardinality constraints on children, recursive downward navigation is also allowed, that is, expressions like *the descentants of p_1 nodes have no more "children" than the descendants of p_2 nodes*, are allowed. These approaches provide highly expressive logics but, on the other hand, counting is limited to children nodes. More specifically, the notion of context is limited to the parent node of counted nodes.

In [1], the logic in [5] is extended to express cardinality constraints, w.r.t a constant, on the whole tree rather than from particular nodes in the document, without an extra computational cost. Recursive upward and downward navigation can be imposed succinctly in numerical constraints, for example, *there are no more than* 5 *p_1 nodes with either a descendant or an ancestor named p_2*, is succintly expressible in such logic.

The more general problem which consists in considering full presburger arithmetic [6] in path expressions and recursive upward/downward navigation in trees would lead to undecidability [6]. In order to balance expressivity and efficiency, in Section 2, we propose a modal logic with the following distinctive features:

$\Phi \ni \phi$	$:=$		Formulas
		p	propositions
	$\mid$	x	variable
	$\mid$	$\top$	true symbol
	$\mid$	$\neg\phi$	negation
	$\mid$	$\phi_1 \wedge \phi_2$	conjunction
	$\mid$	$\phi_1 \vee \phi_2$	disjunction
	$\mid$	$\langle m \rangle \phi$	modality
	$\mid$	$\mu x.\phi$	fixpoint
	$\mid$	$(\phi_1 \xrightarrow{\alpha} \phi_2) \leq k$	counting

Figure 1: Formula syntax

- recursive upward/downward navigation;
- cardinality constraints, w.r.t. constants, using recursive multidirectional paths, that is, expressions like *the preceding siblings of the p_2 descendants of p_1 nodes are more than* 5, are allowed.

In Section 2, we sketch a tableau-based satisfiability-checking algorithm with an optimal simple exponential time complexity for such a logic. In Section 3, by means of a linear translation of XPath expressions into the logic, we identify a decidable fragment of the XPath language where cardinality constraints are allowed. Regular tree types, with numerical restrictions on children, are also supported (Section 3), leading to a highly expressive and efficient reasoning framework for XPath decision problems under type constraints.

2. DEEP COUNTING TREE LOGIC

We consider a μ–calculus-based logic, as first introduced in [5], extended with a counting operator to restrict node cardinalities w.r.t. to a constant. The syntax of the logic is given on Figure 1. A formula is interpreted as a set of nodes in a finite binary tree (there is a well known bijective encoding between n–ary unranked trees and binary trees [5]). Propositions denote the nodes where they occur. Negations is interpreted as set complement, conjunction and disjunction of formulas are interpreted as union and intersection of sets, respectively. Modal formulas denote the transition relations (first child, sibling, parent) among nodes. The least fixpoint operator performs finite recursive navigation. Counting formulas $(\phi_1 \xrightarrow{\alpha} \phi_2) \leq k$ denote the nodes satisfying ϕ_1, such that there is a trail α from them leading to no more than k nodes satisfying ϕ_2.

For example, the formula ϕ denoting p_1 nodes with at least one ancestor named p_2 is written $p_1 \wedge \langle\overline{1}\rangle \mu x.p_2 \vee \langle\overline{1}\rangle x \vee \langle\overline{2}\rangle x$. Now, if we want to express the nodes satisfying ϕ with no more than 2 descendants named p_3, we can write $(\phi \xrightarrow{1,(1|2)^\star} p_3) \leq 2$.

It is not hard to observe that the *greater than* operator ($>$) and the *equality* operator can be easily defined from the *less or equal* operator ($\leq$).

One may also be interested in a slightly different manner of counting called global counting (as studied in [1]). Global counting simply consists in counting the number of nodes where a formula ϕ holds, independently from any context.

$SS \leftarrow N^\phi$
$ST \leftarrow \emptyset$
repeat
  $AUX \leftarrow \{(n, \Gamma_1, \Gamma_2) \mid R^\phi(n, i) = root(\Gamma_i), n \in SS, \Gamma_i \in ST, i = 1, 2\}$
  if $AUX = \emptyset$ **then**
    return 0
  end if
  $ST \leftarrow ST \cup AUX$
  $SS \leftarrow SS \setminus nodes(root(AUX))$
until $ST \vdash \phi$
return 1

Figure 2: Satisfiability-Checking Algorithm

A global counting formula $\phi \leq k$ is intended to denote all the nodes in the tree where ϕ holds whenever the number of nodes satisfying ϕ is less or equal than k; otherwise, the formula denotes the empty set. It is worth noticing that any global counting formula can be expressed in terms of a local counting formula as introduced on Figure 1. For this purpose, we denote the whole set of nodes by the formula $\top$, from each of them we navigate to any other node, by means of the trail $(\overline{1} \mid \overline{2})^\star, (1,2)^\star$, in order to count the number of nodes where the formula in question holds. Hence, we define for $\# \in \{\leq, >, =\}$:

$$\phi \# k \quad \equiv \quad ((\top \xrightarrow{(\overline{1}|\overline{2})^\star,(1,2)^\star} \phi) \# k) \wedge \phi$$

Formulas like $(\phi_1 \xrightarrow{\alpha} \phi_2) \leq k$ give additional expressivity over global counting, since they can take any context into account. Nodes in a specific region of the tree can be counted from any other location in the tree. This additional expressive power – very useful in practice – makes the decision procedure for the extended logic even more challenging. In the next section, we report on our research in progress, toward obtaining an efficient satisfiability-testing algorithm.

Satisfiability

The major difficulty consists in extending the satisfiability algorithm proposed in [5] to deal with a satisfaction relation for counting formulas. The approach we take is to define this relation inductively from the satisfaction relation of non-counting formulas and a compatibility transition relation among nodes. We sketch below a tableau-based satisfiability-checking algorithm for the logic, based on this idea.

There are two main tasks for such an algorithm: first, enumerating all candidate satisfying trees, and second, evaluating each tree against the formula in question. The candidate trees are built in a bottom-up manner, that is, from the leaves to the root. We begin by considering all possible leaves, then in the iteration step, we find a parent for the trees considered in previous steps. After each iteration, we check if we obtain a satisfying tree. When no more parents can be added, if no satisfying tree was found, then we say the formula is unsatisfiable. A sketch of the satisfiability algorithm is given in Figure 2 as a boolean function, with a formula as input. The algorithm outputs 1 whenever the formula is satisfiable, and 0 otherwise. A major difference with the algorithm of [5] is that multisets are involved in order to perform counting. In Figure 2: N^ϕ is the multiset of all possible nodes needed to build a satisfying tree, ST is a multiset

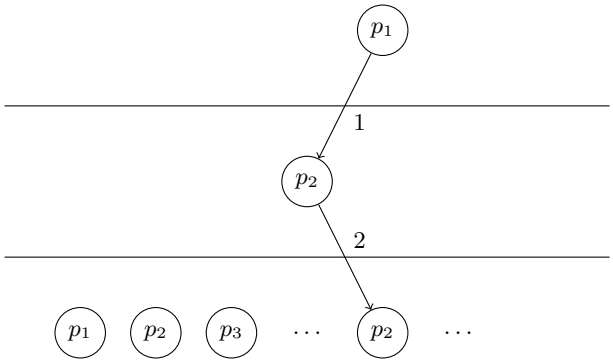

Figure 3: Graphical run of the satisfiability algorithm for $(p_1 \xrightarrow{(1,2^\star)^\star} p_2) > 1$

of candidate trees (triples (r, T_1, T_2), where r is the root, and T_1 and T_2 are its left and right subtree, resp.), $R^\phi(n, n') =$ is a compatibility transition relation among nodes, and $\vdash$ is a satisfaction relation of formulas against a tree.

As for the logic proposed in [5], the current logic enjoys the *small* model property, requiring at most 2^n nodes (where n is the formula size) in a satistying tree, turning out to be the key factor in the computational cost of the whole algorithm.

Figure 3 illustrates a run of the algorithm on a simple formula which is found satisfiable at step 3 of the algorithm.

3. XML PATHS AND TYPES

This section presents a large fragment of the XPath recommendation [3], as well as a linear translation of XPath expressions into the logic. The principles for translating regular tree types (that capture most XML schemas, DTDs and Relax NGs in use today) into the logic are also explained.

XPath

XPath is a highly expressive language for describing queries with a simple and elegant syntax. Basic XPath expressions have the form $a :: p$, such expressions can be composed, leading to expressions with the form ρ_1/ρ_2. XPath expressions can also be filtered by the so-called qualifiers, forming expressions like $\rho[q]$. The qualifiers are themselves path expressions additionally composed by logical connectives (conjunction, disjunction, negation) and counting operators.

The expressions $a :: p$ denote the nodes named p accessible by a navigation axis denoted by a. Such a navigation allows to reach not only neighbor nodes (parent or children), but also descendants or even ancestor nodes. The composition of expressions denotes consecutive navigation steps. For example, the expression self :: p_1/descendant :: p_2 denotes, in a given tree, all the nodes labeled by p_2 which are descendants of nodes labeled by p_2. Qualifiers act as filters: they can be viewed as boolean functions. If a path expression evaluates to at least one node then, when interpreted as a qualifier, it is true. For example, the expression self :: p_1[child :: p_2] denotes the p_1 nodes which have at least one p_2 child. When the qualifiers are conjuncted, disjuncted or negated, they are interpreted as expected. Also, when a path is restricted by an occurrence number, by means of a counting operator, its semantics behaves as naturally expected. For example, the expression self :: p_1[child :: $p_2 \leq 5$] denotes the p_1 nodes

with no more than 5 children named p_2.

We now introduce a linear translation of XPath expressions into the logic. A basic XPath expression $a :: p$ is translated as the proposition conjuncted with a fixpoint formula that navigates as required by a. For example, child $:: p$ is translated as $p \wedge \mu x.\langle\overline{1}\rangle\xi \vee \langle\overline{2}\rangle x$, where ξ is an arbitrary formula denoting a context. The composition of paths ρ_1/ρ_2 is translated as the translation of ρ_2 with the translation of ρ_1 as context. As for qualified paths $\rho[q]$, when the qualifier is path ρ', such path is translated as its inverse, that is, instead of denoting the nodes reachable from a context by means of ρ', it will denote such context nodes. For example, the expression self $:: p_1$[child $:: p_2$] is translated as $\xi \wedge p_1 \wedge \langle 1\rangle\mu x.\xi \vee \langle 2\rangle x$, for a context ξ. The conjunction, disjunction and negation of qualifiers is translated as expected. When the qualifier is a constrained path $\rho \leq k$, it is translated as the counting formula $(\xi \xrightarrow{\alpha} \phi^\rho) \leq k$, where ξ is the context, α contains the navigational information of ρ, that is, the composition of the interpretation of axis as trails, and ϕ^ρ is the translation of a path resulting after the extraction of navigational information of ρ. For example, the expression self $:: p_1$[child $:: p_2 \leq 5$] is translated as $p_1 \wedge (\top \xrightarrow{1,2^*} p_2) \leq 5$. For further details on the translation, we refer the reader to [5, 1].

Regular Tree Types

In this work we consider the binary representation of unranked trees and use the corresponding binary version of regular tree types, as introduced in [5]. In their basic form, binary regular tree types are defined by labels (propositions), empty sequences, and variables. From such basic forms we can build tree types by composition, alternation and an operator for recursion. Detailed formal syntax and restrictions can be found in [5].

The interpretation of types is a set of finite trees w.r.t. a set of propositions and variables. The variables are also interpreted as sets of trees. Empty sequences denote the absence of subtrees. A labeled composition $p(x_1, x_2)$ denotes the trees rooted by p nodes with the interpretation of x_1 and x_2 as left and right subtrees, resp. The alternation of types is interpreted as set union. A specific operator is intended to allow recursive definition of types. A n–ary version of this operator was chosen so that several type variables can be bound at a time, in order to avoid blow-ups in the size of the types when mutually recursive definitions occur.

In some schema languages there are operators for constraining the number of type occurrences.[1] For example, in order to fix the occurrence of labeled composition we may define:

$$\begin{aligned} p(x_1, x_2)^{=0} &= () \\ p(x_1, x_1)^{=k} &= p(x_1, p(x_1, x_2)^{=k-1}) \quad \text{where } k > 0 \end{aligned}$$

In order to avoid the blow-up in the size of type expressions where cardinality constraints are present, we extend the translation of regular tree types (provided in [5]) by directly translating counting type expressions into formulas $(\top \xrightarrow{2^*} \phi)\#k$, where ϕ is the translation of the type we want to constraint. Since the translations of both XPath expressions and regular tree types are linear, we ended-up with an efficient decision procedure for XPath problems where XML types may be present.

[1]The W3C XML Schema recommendation notably defines two attributes `minoccur` and `maxoccur` for this purpose.

4. CONCLUSION

We have proposed a modal logic to reason about finite trees where cardinality constraints on nodes (expressed by regular recursive paths) can be used in path expressions. We sketched a satisfiability-checking algorithm for the logic, with no increase in computational complexity compared to the logic without the counting operator. This allows to address the static analysis of XML transformations on a larger fragment of XPath than prior work on the subject. In addition, we take into account cardinality constraints on regular tree type expressions extending the scope of the analysis to paths under such constraints. We are currently implementing the satisfiability algorithm and generalizing this counting approach to other types of counting expressions.

5. REFERENCES

[1] E. Bárcenas, P. Genevès, and N. Layaïda. Counting in trees along multidirectional regular paths. In *PLAN-X'09*, January 2009.

[2] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. *J. ACM*, 55(2):1–79, 2008.

[3] J. Clark and S. DeRose. XML path language (XPath) version 1.0. W3C recommendation, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[4] S. Dal-Zilio, D. Lugiez, and C. Meyssonnier. A logic you can count on. *SIGPLAN Not.*, 39(1):135–146, 2004.

[5] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *PLDI '07*, pages 342–351, New York, NY, USA, 2007. ACM Press.

[6] F. Kladtke and H. Ruess. Parikh automata and modanic second-order logics with linear cardinality constraints. In *Technical Report 177*. Institute of CS at Freiburg University, 2002.

[7] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.

[8] F. Neven and T. Schwentick. Xpath containment in the presence of disjunction, dtds, and variables. In D. Calvanese, M. Lenzerini, and R. Motwani, editors, *ICDT*, volume 2572 of *LNCS*, pages 312–326. Springer, 2003.

[9] H. Seidl, T. Schwentick, A. Muscholl, and P. Habermehl. Counting in trees for free. In J. Díaz, J. Karhumäki, A. Lepistö, and D. Sannella, editors, *ICALP*, volume 3142 of *LNCS*, pages 1136–1149. Springer, 2004.

[10] Y. Tanabe, K. Takahashi, M. Yamamoto, A. Tozawa, and M. Hagiya. A decision procedure for the alternation-free two-way modal mu-calculus. In B. Beckert, editor, *TABLEAUX*, volume 3702 of *LNCS*, pages 277–291. Springer, 2005.