Verification of Tree Updates for Optimization

Michael Benedikt[1], Angela Bonifati[2], Sergio Flesca[3], and Avinash Vyas[1]

[1] Bell Laboratories,
[2] Icar CNR, Italy,
[3] D.E.I.S., University of Calabria

Abstract. With the rise of XML as a standard format for representing tree-shaped data, new programming tools have emerged for specifying transformations to tree-like structures. A recent example along this line are the update languages of [17, 16, 9] which add tree update primitives on top of the declarative query languages XPath and XQuery. These tree update languages use a "snapshot semantics", in which all querying is performed first, after which a generated sequence of concrete updates is performed in a fixed order determined by query evaluation. In order to gain efficiency, one would prefer to perform updates as soon as they are generated, before further querying. This motivates a specific verification problem: given a tree update program, determine whether generated updates can be performed before all querying is completed. We formalize this notion, which we call "Binding Independence". We give an algorithm to verify that a tree update program is Binding Independent, and show how this analysis can be used to produce optimized evaluation orderings that significantly reduce processing time.

1 Introduction

The rise of XML as a common data format for storing structured documents and data has spurred the development of new languages for manipulating tree-structured data, such as XSLT [2] and XQuery [19]. In this work, we deal with a new class of languages for specifying *updates* – programs that describe changes to an input tree. Specification and processing of updates to tree-structured data is a critical data management task. In the XML context, updates have long been implementable in node-at-a-time fashion within navigational interfaces such as DOM, but languages for specifying bulk updates are now emerging. Several language proposals based on extensions of the declarative languages XPath and XQuery have been put forward in the literature [17, 16, 9, 8], and the World Wide Web consortium is well underway in extending XQuery, the XML standard query language, with the capability of expressing updates over XML data. Since XML documents are basically trees, and since the update primitives of these languages cannot violate the tree structure, we refer to these as *tree update languages*. A sample declarative tree update program, in the syntax of [16] is shown below:

```
U1 :    for $i in //open_auction
        insert $i/initial/text into $i/current
        delete $i/bidder
```

//openauction is an XPath expression returning the set of openauction nodes in the tree, hence the opening for loop binds the variable $i to every openauction node in turn. In the body of the loop, the XPath expression $i/initial/text returns the subtree underneath a text node lying below a initial child of a node bound to variable $i at this iteration of the loop. The expression $i/bidder returns all the bidder nodes below the node bound to $i. Informally, example U1 states that below each current child of an openauction node a copy of the subtree rooted at a certain text node is inserted, and that each bidder node lying below a openauction element should be deleted. The effect of this program over an instance, is shown in Figure 1(a) and (b).

Previous tree update language proposals differ in many details, but they agree on a critical semantic issue regarding how program evaluation is to be ordered. The current proposals generally center upon the *snapshot semantics* [16], which specifies the use of two logical phases of processing: in the first all evaluation of query expressions is done, yielding an ordered set of point updates. In the second phase the sequence of updates is applied in the specified order. In the example above, a sequence consisting of a list of insertions and deletions will be generated, based on the ordering of the results of //openauction and the other queries; this sequence will be applied in that order to the tree.

The snapshot semantics has a number of attractive features; it is consistent with the semantics of declarative relational update languages such as SQL, and it averts the possibility of ill-formed reads arising at runtime. The main drawback is that the naive implementation of it is very inefficient. In a straightforward implementation the intermediate results must all be materialized before any writes are performed. To increase performance, one would prefer a more pipelined chaining of reads to subsequent writes – an *interleaved semantics*. Our approach to this problem is to maintain the use of the snapshot semantics, but to verify that an interleaved implementation does not violate the semantics of a given program. We concentrate on determining *statically* whether updates generated from a program can be applied as soon as they are generated. We denote this property *Binding Independence*, and our main contribution is an algorithm for verifying it.

In example U1, our analysis detects that evaluation order of U1 can be rearranged to perform updates as soon as they are generated: that is, U1 is Binding Independent. Intuitively, this is because the insert and delete operations in one iteration do not affect the evaluation of expressions in subsequent iterations. More generally, we formalize a notion of *non-interference* of an update with an expression. We show that non-interference implies Binding Independence, and then present algorithms that decide non-interference properties. In this paper, we concentrate on a subset of the language of [16], but our techniques are applicable to other tree update language proposals that use snapshot semantics.

Optimization based on specification-time verification is particularly attractive for bulk updates in XML, given that they are often defined well in advance of their execution and are used to specify computing-intensive modifications to data that may take minutes or even hours on current update processors. Thus the contributions of the paper are: (*i*) a formalization of Binding Independence, a property of programs that is critical to update optimization, (*ii*) the notion of non-interference of programs, and an algorithm for reducing Binding Independence to a series of non-interference properties, (*iii*) an algorithm for deciding non-interference and (*iv*) experiments investigating the feasibility of the verification.

Related Work. Tree update languages are similar to tree transducers, whose verification is studied in [1, 3]. These works are similar to ours in that they concern capturing the iteration of one transducer with the single action of another transducer. However, the expressiveness of the respective formalisms is incomparable: update languages work on ordered trees with no bound on rank, while [1, 3] deal with fixed-ranked trees; the iteration here is a bounded looping construct, while in [1, 3] it is a transitive closure. The works differ also in the notion of "capturing" (equality up to isomorphism vs. language containment) and the application (optimization vs. model checking). There has been considerable work on static analysis of other tree query and transformation languages. In the XML setting a good summary, focusing on the type-checking problem, can be found in [15]. [11] presents a system for doing static analysis of XSLT. Because XQuery and XSLT cannot perform destructive updates, their analysis is much different than ours. Still, [11, 6] include techniques for performing conservative satisfiability tests on trees that could be used in conjunction with our analysis.

The main technique in our analysis is transforming dynamic assertions into static ones. This idea is certainly an ancient one in program analysis (e.g. [4]). Distinctive features in our setting include the fact that a tree pattern query language, XPath, is part of the programming formalism we analyze; also that the update operations stay within the domain of trees, making both the reduction and the final static test simpler.

2 Trees, Queries, and Tree Update Languages

In this section, we review the basics of the data model and the fragment of the XPath query language considered in this work.

Data Model. We deal here with node-labeled trees. Since we think of these as abstractions of XML documents, we refer to labels as *tags*, and generally use D (for document) to range over trees. Each node has additionally a unique identity, referred to as its *node identifier* or nodeId. Moreover, trees are ordered according to the *document order* for nodes, which is the order returned by an in-order depth-first traversal of the tree. An example of one of our trees is given in Figure 1(a).

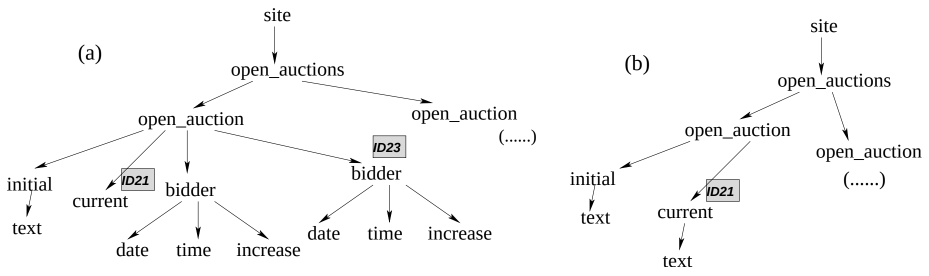

Fig. 1. An example XML tree, before (a) and after (b) the update U1.

XPath. A key component of update languages is the use of patterns or queries to identify nodes. Although our analysis could work for many tree pattern languages, for simplicity we use a subset of the XPath language. XPath consists of *expressions*, each of which defines a map from a node in a tree to an ordered set of output nodes from the same tree, and *filters*, which define predicates on nodes in the tree. The ordering of the output of our XPath expressions will always be via the depth-first traversal ordering of the input tree, so we will not show it explicitly. In this paper, a top-level XPath expression is either a basic expression or a union $E_1 \cup \ldots \cup E_n$ of basic expressions. Basic expressions are built up from rules $E = E/E$, $E = /A$, $E = //A$, $E = /*$, $E = //*$, and $E = [F]$. Here A is any label and F is any filter (defined below). An expression /A returns all the children labeled A of the input node, while //A returns all the A descendants of an input node. The expression /* returns all children of a node, while //* returns all descendants. $/$ is the composition operator on binary relations and [F] is the identity restricted to nodes satisfying filter F. A filter is either a basic expression E, [label=A] for A a tag, or a conjunction of filters. The filter E holds exactly when E returns nonempty at a node, while the label comparison has the obvious meaning. Following convention, we often omit the composition operator for brevity: e.g. writing /A/B rather than ((/A)/(/B)) for the XPath query that returns all B children of A children of an input node. We use XP to denote the XPath language above. As seen in the opening example, our update programs deal with XPath expressions with variables in them. We use XP$(\boldsymbol{x})$ to denote the language built up as above but with the additional rule $E = x_i$, where x_i is a variable in $\boldsymbol{x}$. We let XP_V denote an XPath expression with variables. Such expressions are evaluated relative to an assignment of a nodeId in a tree to each variable, with the expression x_i returning exactly the node associated with the nodeId bound to x_i under the assignment, or $\emptyset$ if there is no such node. An expression with variables in $\boldsymbol{x}$ will be denoted $E(\boldsymbol{x})$. For a tree D, we let $E(\boldsymbol{a})(D)$ be the evaluation of E at the root of D in a context where x_i is assigned a_i.

Schemas and DTDs. Our algorithms can take into account constraints on the input tree. Our most general notion of constraint will be a non-deterministic bottom-up tree automaton on unranked trees. In accordance with XML terminology, we refer to an automaton as a *schema*. Our implementation requires the schema to be given as a Document Type Definition (DTD) from XML. A DTD enumerates a finite set of labels that can be used, and for each label a regular expression over the label alphabet which constrains the children of a node with label. A DTD thus describes a very restricted kind of regular language of unranked trees, where the state depends only on the label. The examples in this paper are based on the XMark DTD [14], used for benchmarking XML tools.

Templates. Finally, in order to present our core update language, we will need functions that construct trees from XP_V expressions. A *template* is a tree whose nodes are labeled with literal tags or XP_V expressions, with the latter appearing only at leaves. A template τ is evaluated relative to a tree D and an assignment b of variables to nodeIds in D. The result of evaluation is $\tau(D, b) = \Theta$, where Θ is the forest formed by replacing each node in τ that is an XP expression E with the sequence of trees resulting from the evaluation of E relative to D, b.

Tree Update Language. We present the syntax of the tree update language TUpdate we use throughout the paper. This language is an abstraction of that presented in [16]. The top-level update constructs we deal with are as follows:

Here **XPathExpr** and **tExpr** are XP_V expressions, while **cExpr** is a template. Intuitively, **tExpr** computes the target location where the update is taking place, while

UpdateStatement ::= SimpleUpdate | **ComplexUpdate**

ComplexUpdate ::= for var in **XPathExpr ComplexUpdate**| **UpdateBlock**

UpdateBlock ::= **SimpleUpdate**$^+$

SimpleUpdate ::= ("insert" **cExpr** ("after" | "before") **tExpr**) | ("insert" **cExpr** "into" **tExpr**) | ("delete" **tExpr**) | ("replace" **tExpr** "with" **cExpr**)

cExpr constructs a new tree which is to be inserted or replaced at the target of the update.

We now review the semantics of tree update programs in the style of [17, 16]. The semantics of all existing update proposals [17, 16, 9, 13] consists first of a description of how individual updates apply, and secondly how query evaluation is used to generate an ordered sequence of individual updates. Following this, our semantics will be via a transition system, with two kinds of transitions, one for application of individual updates and the other for reducing complex updates to simpler ones based on queries. We will give both these transitions, and then discuss the order in which transitions fire.

Concrete Update API. Let D be a tree, f be a forest (ordered sequence of trees), and n a node identifier. A *concrete update* u, is one of the following operations: *(i)* $u = \text{InsAft}(n,\ f)$ or $u = \text{InsBef}(n,\ f)$: when applied to a tree D the operation returns a new tree, such that, if $n \in D$, the trees in f are inserted immediately after (before) the node with id n in the parent node of n, in the same order as in the forest f. If $n \notin D$, the operation just returns D (we omit the similar requirement on the updates below); *(ii)* $u = \text{InsInto}(n,\ f)$: when applied to D, the operation returns a new tree such that, if $n \in D$, the trees in f are inserted after the last child of node n; *(iii)* $u = \text{Del}(n)$: the operation returns a new tree obtained from D by removing the sub-tree rooted at n; *(iv)* $u = \text{Replace}(n,\ f)$: the operation returns a new tree such that, if $n \in D$, the trees in f replace the sub-tree rooted in the node n (in the ordering given by f). In all cases above, fresh nodeIds are generated for inserted nodes.

Single-step processing of programs. We now present the next main component of update evaluation, the operator that reduces a single partially-evaluated update to a sequence of simpler ones.

An *expression binding* for an update u is a mapping associating a set of tuples to occurrences of XPath expressions in u. A tuple will be either a nodeId in the original tree or a tree constructed from the original one (e.g., a copy of the subtree below a node). A *bound update* is a pair (b, u) where u is an **UpdateStatement** and b is an expression binding for u. The *update reduction operator* $[\cdot]$ takes a bound update and produces a sequence of bound updates and concrete updates. We refer to such a sequence as an *update sequence*. We define $[p]$ for a bound update $p = (b, u)$ as follows. If $p =$ (for var in $E\ u', b$), we form $[p]$ by evaluating E to get nodes $n_1 \dots n_k$, and return the sequence whose i^{th} element is (b_i, u') , where b_i extends b by assigning var to n_i. If u is an update block $u_1 \dots u_l$, then $[p]$ returns $(b, u_1) \dots (b, u_l)$. If u is a simple update with no bindings for the expressions in u, $[p]$ is formed by first evaluating the template in u (in case of replace or insert) to get a forest, and evaluating the target expressions in u to get one or more target node identifiers. We then proceed as follows: for an insert or replace if the target expression evaluated to more than one target node, $[p]$ is the empty sequence, otherwise $[p]$ is (b', u_0), where b' extends b by binding the remaining variables according to the evaluation just performed. For a delete, let nodeIds $n_1 \dots n_j$ be the result of evaluation of the target expression. $[p]$ is $(b_1, u) \dots (b_j, u)$, where b_i

extends b by assigning the target expression of the delete to n_i. Finally, if $p = (b, u)$ is a bound update in which u is simple and every expression is already bound, then $[p]$ is simply the concrete update formed by replacing the expressions in u with the corresponding nodeId or forest given by the bindings.

Processing Order for Complex Updates. We are now ready to define the semantics of programs, using two kinds of transitions acting on a *program state*, which consists of a tree and an update sequence.

An *evaluation step* on a program state $(D, \mathrm{us} = p_1 \dots p_n)$ is a transition to (D, us') where the new update sequence us' is formed by picking a bound update $p = (b, u)$ from the update sequence and replacing p by $[p]$ in the sequence. If ps is the program state before such an evaluation step and ps' is the result of the step, we write $\mathrm{ps} \rightsquigarrow^e_p \mathrm{ps}'$. For example, the processing of the update U = for \$x in /A/B insert \$x/C into /B at program state $\mathrm{ps}_0 = (D, \mathrm{p}_0 = (\emptyset, U))$ would begin with the step: $\mathrm{ps}_0 \rightsquigarrow^e_{p_0} \mathrm{ps}_1$ where ps_1 is: D, $\{p_1$=(⟨ \$x:i$_1$ ⟩, insert \$x/C into /B); $p_2 =$ (⟨ \$x:i$_2$ ⟩, insert \$x/C into /B) $\}$ and where the nodeIds $\{i_1, i_2\}$ are the result of evaluating $/A/B$. An *application step* simply consumes a concrete update u from the update sequence and replaces the tree D by the result of applying u to D. We write $\mathrm{ps} \rightsquigarrow^a_u \mathrm{ps}'$

An *evaluation sequence* is any sequence of steps $\rightsquigarrow^e$ and $\rightsquigarrow^a$ as above, leading from the initial tree and update statement to some tree with empty update sequence. The final tree is the *output of the sequence*. In general, different evaluation sequences may produce distinct outputs. As mentioned in the introduction, all existing proposals use a snapshot semantics which restricts to evaluation sequences such that *(i)* (*snapshot rule*) all available evaluation transitions $\rightsquigarrow^e$ must be applied before any application step $\rightsquigarrow^a$ is performed, and *(ii)* (*ordering rule*) the application steps $\rightsquigarrow_a$ must then be applied in exactly the order given in the update sequence - that is, we always perform $\rightsquigarrow^a_p$ starting at the initial concrete update in the update sequence. It is easy to see that this results in a unique output for each update. We say $(D, U) \rightsquigarrow^{snap} D'$ if $(D, \{(\emptyset, U)\}$ rewrites to $(D', \emptyset)$ via a sequence of $\rightsquigarrow^e$ and $\rightsquigarrow^a$ transitions, subject to the conditions above.

3 Optimized Evaluation and Verification

Naturally, an implementation of the snapshot semantics will differ from the conceptual version above: e.g. multiple evaluation or concrete update steps can be folded into one, and cursors over intermediate tables containing query results will be used, rather than explicit construction of the update sequence. However, even a sophisticated implementation may be inefficient if it respects the snapshot rule *(i)* above. The snapshot rule forces the evaluation of all embedded expressions to occur: this can be very expensive in terms of space. Furthermore, it may be more efficient to apply delete operations as soon as they are generated, since these can dramatically reduce the processing time in further evaluations.

The *eager evaluation* of an update u is the evaluation formed by dropping the snapshot rule and replacing it with the requirement *(i')* that whenever the update sequence has as initial element a concrete update p, we perform the application step $\rightsquigarrow^a_p$. It is easy to see that *(i')* also guarantees that there is at most one outcome of every evaluation. We denote the corresponding rewriting relation by $\rightsquigarrow^{eager}$.

We say that a program U is *Binding Independent* (BI) if any evaluation sequence for u satisfying requirement the ordering rule *(ii)* produces the same output modulo an isomorphism preserving any nodeIds from the input tree. Note that if two trees are isomorphic in this sense, then no user query can distinguish between them. Clearly, if an update is BI we can use $\rightsquigarrow^{eager}$ instead of $\rightsquigarrow^{snap}$. Similarly, we say that a program U is BI with respect to a schema (automaton or DTD) if the above holds for all trees D satisfying the schema. The example U1 is BI with respect to the XMark DTD (a fact which our analysis verifies). A simple example of an update that is not BI is:

```
U2 :    for $i in //openauction,for $ j in //openauction,
        insert $i into $j
```

Indeed, the eager evaluation of U2 can increase the size of the tree exponentially, since at each $i element we duplicate every openauction element in the tree. A simple argument shows that snapshot evaluation can increase the size of the tree only polynomially. Unfortunately, one cannot hope to decide whether an arbitrary TUpdate program is BI. That is, we have:

Theorem 1. *The problem of deciding whether a TUpdate program is Binding Independent with respect to an automaton is undecidable.*

The proof is by a reduction to solvability of diophantine equations, and is omitted for space reasons. We thus turn to the main goal of the paper: a static analysis procedure that gives sufficient conditions for BI. These conditions will also guarantee that the number of evaluation steps needed to process the update under eager evaluation is no greater than its time under snapshot evaluation.

Binding Independence Verification Algorithm. We give a conservative reduction of BI testing to the decidable problem of satisfiability testing for *XPath equations*. Given $\boldsymbol{x}$ a sequence of variables, a *system of XPath equations in* $\boldsymbol{x}$ is a conjunction of statements either of the form $x_i : E_i$, where E_i is in XP$(x_1 \ldots x_{i-1})$, or of one of the forms $lab(x) = A$, $lab(x) \neq A$, $x_i \neq x_j$ for $i \neq j$. Given a tree D and sequence of nodeIds $a_1 \ldots a_n$, we say $\boldsymbol{a}$ satisfies the equations if a_i is in the result of E_i evaluated in a context where $\boldsymbol{x}$ interpreted by $\boldsymbol{a}$, and if the label and inequality conjuncts are satisfied. A system of equations $H(\boldsymbol{x})$ is unsatisfiable iff for all trees D there is no $\boldsymbol{a}$ satisfying H in D. We define the notion of unsatisfiability with respect to a schema analogously. Our goal is an algorithm that, given a TUpdate program U will produce a set of systems of XPath equations $S_1 \ldots S_n$ such that: each S_i is unsatisfiable implies U is BI.

Non-interference. Intuitively, an update is BI if performing concrete updates that are generated from a program does not impact the evaluation of other XPath expressions. For example U1, we can see that in order to verify BI it suffices to check that: i) for each a_0 in //openauction, the update ($i:a$_0$, insert $i/initial/text into $i/current) does not change the value of the expression $i/initial/text, $i/current, or $i/bidder, where in the last expression $i can be bound to any a_1 in //openauction, and in the first two to any $a_1 \neq a_0$ ii) for each a$_0$ in //openauction, ($i:a$_0$, delete $i/bidder) does not effect $i/initial/text, or $i/current for any $i, and does not effect $i/bidder when $i is bound to $a_1 \neq a_0$. The above suffices to show BI, since it implies that performing any evaluation step after an update gives the same result as performing the evaluation before the update.

We say a TUpdate program U is *non-interfering* if: for every D, for each two distinct tuples $\boldsymbol{a}$ and $\boldsymbol{a}'$ that can be generated from binding the for loops in U on D, for each simple update u in U and every XPath expression E in U, $E(\boldsymbol{a}')(D)$ returns the same set as $E(\boldsymbol{a}')(u(\boldsymbol{a})(D))$, and if the above holds for $\boldsymbol{a} = \boldsymbol{a}'$ if E is not in

u. Note that both eager and snapshot semantics agree on what $u(\boldsymbol{a})(D)$ means for a simple update u. The discussion previously is summarized in the observation: *if U is non-interfering, then U is BI*. Furthermore, if U is non-interfering, the eager evaluation terminates in at most the number of steps needed to evaluate U under snapshot evaluation. The condition is not necessary. Thus far we have found that BI updates arising in practice (i.e. in the uses of our update engine within projects at Lucent Technologies) are non-interfering. Moreover, non-interference can be tested effectively; however, to gain additional efficiency, we provide only a conservative test in our implementation.

Non-interference breaks down into a number of assertions about the invariance of expressions under updates, each of which needs to be verified separately. A *delete non-interference assertion* is of the form $(C(\boldsymbol{x}), u(\boldsymbol{x}), M(\boldsymbol{x}, \boldsymbol{y}))$ where $C(\boldsymbol{x})$ is a system of XPath equations, M is a system of equations in XP$(\boldsymbol{x}, \boldsymbol{y})$, and u is a SimpleUpdate. The system M is the *monitored system* of the assertion while C is the *context system*. An *insert non-interference assertion* has the same form.

A delete non-interference assertion is valid iff for all trees D $\boldsymbol{a}$ satisfying $C(\boldsymbol{x})$ in D, for every $\boldsymbol{b}$ in D, we have $D \models M(\boldsymbol{a}, \boldsymbol{b}) \rightarrow u(\boldsymbol{a})(D) \models M(\boldsymbol{a}, \boldsymbol{b})$. That is, u does not delete anything from the monitored system. An insert non-interference assertion is valid iff for all trees D, $\boldsymbol{a}$ satisfying $C(\boldsymbol{x})$ in D, and for every $\boldsymbol{b}$ in $u(\boldsymbol{a})D$, $u(\boldsymbol{a})(D) \models M(\boldsymbol{a}, \boldsymbol{b}) \rightarrow D \models M(\boldsymbol{a}, \boldsymbol{b})$. That is, u does not add anything to the monitored system. Note that if u is a delete, then we need only consider delete non-interference assertions, since the XPath queries we deal with are monotone; similarly, if u is an insert, we consider only insert non-interference assertions. Hence we drop the word "insert" or "delete" before non-interference assertions for these simple updates, implicitly assuming the non-vacuous case. We write non-interference assertions in tabular form. For example, the non-interference assertion below, generated from U1, states that a delete in U1 for an index \$i does not effect the expression \$i/current for any other value of \$i:

C= $\$i_1$://openauction	u= delete $\$\mathrm{i}_1$/bidder	M= $\$\mathrm{i}_1 \neq \$\mathrm{i}_2$; $\$i_2$://openauction; \$z:$\$\mathrm{i}_2$/current

Non-interference of U amounts to verifying a set of non-interference assertions, one for each triple consisting of a simple update in U, an XPath expression in U, and an index that witnesses that the context variables are distinct. To increase precision, we can exclude considering non-interference assertions where u is a delete and E is the target expression of u: this broadening of the definition of non-interference is sound, since if $u(\boldsymbol{a})$ deletes from $E(\boldsymbol{a}')$ this does not effect the final evaluation but merely accelerates it.

From Non-interference to Satisfiability. Validity of non-interference assertions still requires reasoning about updates over multiple trees, while we wish to reason about XPath satisfiability over a single tree. Our main result is a reduction of non-interference assertions to satisfiability tests. This reduction makes use of a fundamental property of the snapshot semantics: under this semantics elements in the output tree are in one-to-one correspondence with tuples of elements in the input tree.

Consider the non-interference assertion generated from U1 in the table above. To check this it suffices to confirm that the deleted items do not overlap with the monitored expression $\$\mathrm{i}_2$/current. So the non-interference assertion is equivalent to the joint unsatisfiability of the equations: i_1$://openauction; i_2$://openauction; i_1 \neq$ i_2$; $z: i_1$/bidder//*; $z:$i$_2$/current. This system is unsatisfiable because $z cannot be both a descendant of i_1$ and a child of i_2 \neq$ i_1$, and this is easily detected by our satisfiability test. In general, for delete operations, a non-interference assertion requires checking whether the system Γ is unsatisfiable, where Γ contains the context and monitored equations, and equations $o:$te//*$. Here te is the target expression of the delete, and $o is the variable appearing in an XPath equation in the monitored system (for a delete, this will consist of inequalities plus one XPath equation).

The analysis for inserts requires a much more complex transformation. We give the intuition for this by example, leaving details for the full paper. Consider the non-interference assertion generated from U1, which states that the insert into $i/current does not effect $i/initial/text:

C=$i://openauction	u = insert $i/initial/text into $i/current	M = $i′://openauction; $i′ ≠ $i; $z:$i′/initial/text;

We want to derive a collection of systems of equations such that they are all unsatisfiable iff this assertion holds. We start by normalizing the assertion so that all equations are *basic*: a basic equation is either a label test, an inequality, or of the form $x:$y/* or $x:$y//*. That is, all the XPath expressions consist of just a single step. We do this by introducing additional variables for intermediate steps in all path expressions.

C	lab($i)=openauction; $k:$i/*; lab($k)=initial; $l:$k/*; lab($l)=text; $m:$i/*; lab($m)=current;
u	insert $l into $m
M	$i′ ≠$i; lab($i′)=openauction; $k′:$i′/*; lab($k′)=initial; $l′:$k′/*; lab($l′)=text;

So this assertion says that for any values of i, k, l, m satisfying the equations at the top, the operation in the center does not insert any new witness for the equations on the bottom. Note that there is a further approximation being done here, since instead of checking whether the output of an XPath expression changes, we check whether there is a change to a vector of variables that projects onto that output. This approximation allows us to reduce non-interference to a satisfiability test, while an exact non-interference test would require a (EXPTIME complete [10]) containment test in the presence of a schema.

To reason about these assertions, we consider the possible ways in which a new witness set for the monitored equations can occur. Continuing with our example, let D be some tree, i, k, l, m be witnesses for the context equations in D, and $D'(i, k, l, m)$ be the tree resulting from the insert above. Every node in D' is either a node from the initial tree D (an "old" witness) or was inserted by the operation insert $l into $m. In general, nodes n arising from an insert or replace can be classified by which node of the insert or replace template they arose from. They are either matched by a literal node tn of a template, or they are matched by a node p that lies inside a copy of the subtree t_a of D rooted at node a, where a was matched by some variable v associated with template node tn of the insert operation. In either of the last two cases, we say that n is *generated by template node* tn. In the second case we say the node p is the *pre-witness of* n: that is, p is the element of the old tree that was copied to get n. We write $p = \mathsf{pre}(n)$. In the case n is an old witness, we say $\mathsf{pre}(n) = n$, and in the case n

is associated with a literal template node, we set $\mathsf{pre}(n)$ to be the insertion point where the constructed witness was appended. We can classify our witness tuple by means of a *witness map*: this is a function F assigning to each variable in the monitored equations either an element of the insert or replace template, or the keyword old.

In the example above, one of the witness maps is: $F(\mathrm{i}') = F(\mathrm{k}') = \mathsf{old}, F(\mathrm{l}') = tn$ where tn is the only template node of the insert. We reduce the number of maps considered by enforcing some simple consistency conditions needed for a map to have the possibility of generating a witness. For example, not all variables can be mapped to old, if there is equation $lab(x) = A$ in M, and x is mapped by F to a literal node of a template, then the label of that node must be the literal A.

In the case above, these rules imply that the map above is the only witness map that could yield a witness violating the non-interference assertion, because the template node is a text node and hence cannot witness a non-text node. Figure 2 illustrates the situation: a) shows the initial tree before the insert, with the inserted tree and insert point highlighted. b) shows the tree resulting from the insert, and c) shows the pattern needed to witness the interference assertion in the new tree, according to this witness map. Note

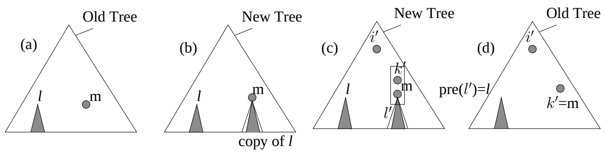

Fig. 2. A pattern and its precondition relative to a witness map.

that the monitored equations assert that k' is the parent of l'. So we have an old node k' that is the parent of a newly-inserted node l'. From the picture, it is clear that this can happen only if k' is actually the target of the insert, namely m, while l' is a copy of the root l of the inserted tree. Hence in the original tree D, the pre-witnesses for $\mathrm{i}', \mathrm{k}', \mathrm{l}'$ are shown in Figure 2d); since k', i' are old witnesses, we write i' instead of $\mathsf{pre}(\mathrm{i}')$ in the figure. The pre-witnesses of i', k' are a pair satisfying the context equations, but with $\mathsf{pre}(\mathrm{k}')$ constrained to be m; the pre-witness of l' will be l. Figure 2d) thus shows what the old tree must look like. The set of equations corresponding to this is shown below. The equations are unsatisfiable, since $m = \mathsf{pre}(\mathrm{k}')$ but they have distinct labels.

lab(\$i)=openauction; \$k:\$i/*; lab(\$k)=initial; \$l:\$k/*; lab(\$l)=text; \$m:\$i/*; lab(\$m)=current;
\$i′ $\neq$ \$i; lab(\$i′)=openauction; pre(\$k′):\$i′/*; lab(pre(\$k′))=initial; pre(\$k′) =\$m; pre(\$l′)=\$l;

Given a witness map F and assertion $A = (H(\boldsymbol{x}), u(\boldsymbol{x}), M(\boldsymbol{x}, \boldsymbol{y}))$, we can now state more precisely our goal: to get a set of equations $K(\boldsymbol{x}, \boldsymbol{y}')$ such that for any tree D and $\boldsymbol{a}$ satisfying H in D, for any $\boldsymbol{b}'$ in D, $K(\boldsymbol{a}, \boldsymbol{b}')$ holds iff there is $\boldsymbol{b}$ such that $H(\boldsymbol{a}, \boldsymbol{b})$ holds in $D' = u(\boldsymbol{a})(D)$ and $\mathsf{pre}(\mathrm{b}_i) = \mathrm{b}'_i$. As the example shows, we get this set of equations by unioning the context equations with a rewriting of the monitored equations. Details are given in the full paper.

Theorem 2. *For every non-interference assertion A we can generate a collection of systems of equations $S_1 \ldots S_n$ such that A is valid iff each of the S_i are unsatisfiable.*

Accounting for a schema. The analysis above checks that an update program U is Binding Independent when run on *any* tree D. Of course, since many programs are not BI on an arbitrary input, it is essential to do a more precise analysis that verifies BI only for input trees satisfying a given automata or DTD S. It is tempting to think that relativizing to a schema requires one only to do the satisfiability test relative to the schema. That is, one might think that a program U is BI w.r.t. schema S if for every tree D satisfying S, for any concrete update u generated from U and any path expression E with parameters from D, E has the same value on D as it does on $u(D)$. However, this is not the case. Consider the update U: for $x in //A insert $x into $x//C insert $x into $x//B delete A/[/B and /C]. Suppose that we wish to consider whether or not U is BI with respect to a given schema S. It is clear that we need to know that instances of the insert do not produce a new witness to A/[/B and /C]. Thus, we need to prove that under eager evaluation there is no evaluation sequence adding a new witness pattern consisting of A,B, and C nodes. But the final witness to this pattern will result from an update to some intermediate tree D', which may not satisfy S. Hence to reduce BI analysis to non-interference of concrete updates *over trees D satisfying the schema*, we must deal with the impact of *sequences* of concrete updates on D.

For integer k, a TUpdate U is *k non-interfering with respect to a schema S* if for every D satisfying S *i)* no delete or replace operation generated from U deletes a witness to an XPath expression in U for a distinct binding, and *ii)* for every sequence $u_1(\boldsymbol{a}^1) \ldots u_k(\boldsymbol{a}^k)$, where u_i are simple inserts or replaces in U and $\boldsymbol{a}^i$ are tuples satisfying the for loop bindings in D, for every expression E, and every $\boldsymbol{b}$ satisfying the bindings and distinct from all $\boldsymbol{a}^i$, $E(\boldsymbol{b})(D) = E(\boldsymbol{b})(D')$, where D' is the result of applying each $u_i^+(\boldsymbol{a}_i)$ to D. Here u^+ for $u =$ replace E with τ is defined to be insert τ into E, and $u^+ = u$ for other simple updates. Informally, k non-interfering means that no single update deletes a witness to an XPath expression in U, and no sequence of k updates inserts a witness. It is easy to see that programs with this property for *every* k are Binding Independent w.r.t. S. The following result gives a bound on k: its proof is given in the full paper.

Theorem 3. *A program U is BI with respect to S if it is k-non-interfering, with respect to S, where k is the maximum of the number of axis steps in any expression E.*

From the theorem, we can see that Binding Independence of U w.r.t. schema S can be reduced to polynomially many non-interference assertions, where this notation is extended to allow a collection of updates. If we consider the update U1 with regard to schema-based BI verification, we need to check assertions such as:

C	$\$i_1$://openauction;	$\$i_2$://openauction;	$\$i_3$://openauction;
$\mathcal{U}$	insert $\$i_1$/initial/text into $\$i_1$/current	insert $\$i_2$/initial/text into $\$i_2$/current	insert $\$i_3$/initial/text into $\$i_3$/current
M	$\$i'$://openauction;	$\$i' \neq \$i_1,i_2,i_3$;	$z:$\i'/initial/text;

These new non-interference assertions are reduced to satisfiability through the use of witness maps and rewriting as previously – we now map nodes in the monitored equations to template nodes in any of the updates.

(Un)Satisfiability Test. The previous results allow us to reduce BI analysis to a collection of unsatisfiability tests of systems of XPath equations on a single tree. Unsatisfiability can be seen decidable via appeal to classical decidability results on trees [18]. Although we could have made use of a third party satisfiability test for logics on trees (e.g. the MONA system [5], although this deals only with satisfiability over ranked trees), we found it more convenient to craft our own test. Our satisfiability problem is in CO-NP; in contrast MONA implements a satisfiability test for a much more powerful logic, whose worst-case complexity is non-elementary. For the quantifier-free XPath equations considered here, the satisfiability problem is known to be NP-complete [7]. From this, we can show that non-interference is CO-NP hard.

Since an exact test is CO-NP, we first perform a conservative unsatisfiability test which simply extracts the descendant relations that are permitted by the schema (for a DTD, this is done by taking the transitive closure of the dependency graph) and then checks each axis equation for consistency with these relations: this test is linear in the size of the equations, and if all equations generated by the reduction are unsatisfiable, we have verified Binding Independence. Our exact test uses a fairly standard automata-theoretic method. The basic idea is to "complete" a system of equations S to get a system S' in which: i) the dependency relation between variables forms a tree, ii) distinct variables are related by inequations, and variables that are siblings within the tree are related by a total sibling ordering iii) if x is a variable then there is at most one variable y that is related to x by an equation y: x//*. The significance of complete systems is that they can be translated in linear time into a tree automaton. There are many completions of a given system, and we can enumerate them by making choices for the relations among variables. Although the number of completions is necessarily exponential in the worst case, in the presence of a schema we can trim the number significantly by making only those choices consistent with the schema dependency graph. In the absence of a schema, any propositionally consistent complete equation system is satisfiable; thus we can test satisfiability by checking for existence of a complete consistent extension. In the presence of a schema, we translate complete systems into automata, and then take a product. Consider the standard encoding of an ordered unranked tree T by a binary tree T': in this encoding a node $n' \in T'$ represents a node $n \in T$, with the left child c' of n' corresponding to the first child of n, and the right child of n' corresponding to the following sibling of n in T (this sibling ordering is given as part of the complete system). Relative to this coding, we translate a complete system into a system of equations over binary trees. Our translation maps an XPath equation ρ to a "binary XPath equation" ρ' , where these equations mention the axes $\downarrow_{\mathsf{left}}, \downarrow_{\mathsf{right}}, \downarrow^{*}_{\mathsf{right}} \dots$. For example, an equation of the form $x:$y/* maps to an equation $x:$y/$\downarrow_{\mathsf{left}}/\downarrow^{*}_{\mathsf{right}}$. The resulting binary equations are again complete, and can thus be translated easily into a bottom-up non-deterministic tree automaton over finite trees.

We have thus arrived at a collection of automata A_i representing complete binary systems extending our original system S. We can likewise transform the schema S into an automaton A_S accepting exactly the binary encodings of trees conforming to S. A standard product construction then yields an automaton A'_i accepting exactly those trees conforming to D for which S returns a result, and a simple fixed point algorithm is used to see if the language returned by an A'_i is nonempty [12].

4 Experimental results

The overall flow of the verification and its use is shown in Figure 3. At verification time, a program is parsed and then goes through the stages of: i) generating non-interference assertions, ii) for each assertion generating the witness maps, iii) for each assertion and witness map, performing rewriting to get a system of equations which needs to be found unsatisfiable. Each system produced is tested for satisfiability with respect to the DTD, if one is present. If a system is found satisfiable, analysis terminates; in this case, nothing is known about the program. If all systems are unsatisfiable, the analysis outputs that the program is verified to be BI. At runtime, the program is processed by our modification of the Galax-based XML update engine. A flag is passed with the program telling whether the program is BI; if so, the eager evaluation is used. The verification

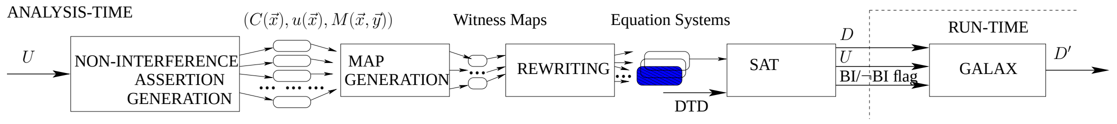

Fig. 3. Architecture of Static Analysis-time and Run-time in Galax.

algorithms are in Java, while the runtime, like the rest of Galax, is in OCAML.

We ran our analysis algorithms on a testbed of sample updates, corresponding in size and complexity to those used in our application of the XML update language of [16] at Lucent Technologies. In Table 1 we show the verification times as the number of steps in the update increases. The times are based on a Pentium 4 with a 1GB RAM and 2.80GHz CPU running XP: in each case, the verification runs in seconds. The times are an average over 1-4 updates, 90% BI and 10% non-BI. We were interested also in tracking the two potential sources of combinatorial blow-up in our algorithms: the first is the number of witness maps to be considered, which determines the number of equation systems that need to be generated. The second is the complexity of the satisfiability test. The first is controlled by the consistency rules for witness map assignment, and our preliminary results (also shown in the figure) show that our rules keep the number of equations systems, and hence calls to satisfiability, low, generally in single digits. The complexity of the satisfiability test is controlled principally by filtering using an approximate test and secondly by using the DTD dependency graph to limit the number of completions. The latter is particularly useful given that DTDs tend to give strong restrictions on the tags that can appear in a parent/child relationship. Currently, our approximate test is quite crude, and eliminates only a small percentage of the equation systems. However, the XMark DTD reduces the number of completions significantly – in our sample updates, to at most several hundred. This results in low time for the aggregate test, since for complete systems the satisfiability test is linear.

Acknowledgements: We thank Jérôme Siméon for many helpful discussions on update semantics, and Andrea Pugliese for his support in implementation of the static analysis routines. We are also very grateful to Glenn Bruns and Patrice Godefroid for invaluable comments on the draft.

References

1. A. Bouajjani and T. Touili. Extrapolating Tree Transformations. In *Proceedings of CAV*, 2002.
2. James Clark. *XSL Transformations (XSLT)*. W3C Recommendation, November 1999. `http://www.w3.org/TR/xslt`.

Query Steps	Verification Time(ms)	Nr. of Equation Systems
9	344	7
10	2390	15
11	5125	15
12	6953	17
13	8250	17
14	10984	17
15	8719	17
16	15701	17
17	25046	17
18	29781	17
19	31281	17

Table 1. Results of Analysis on a query with variable number of steps.

3. D. Dams, Y. Lakhnech, and M. Steffen. Iterating Transducers for Safety of Data-Abstractions. In *Proceedings of CAV*, 2001.
4. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
5. Jacob Elgaard, Nils Klarlund, and Anders Moller. Mona 1.x: new techniques for WS1S and WS2S. In *Computer Aided Verification, CAV '98*, volume 1427 of *LNCS*, 1998.
6. C. Kirkegaard, A. Moller, and M. I. Schwartzbach. Static Analysis of XML Transformations in Java. *IEEE Transactions on Software Engineering*, 2004.
7. L. V. S. Lakshmanan, G. Ramesh, H. Wang, and Z. Zhao. On Testing Satisfiability of Tree Pattern Queries. In *Proc. of VLDB*, 2004.
8. A. Laux and L. Martin. XUpdate - XML Update Language., 2000. `http://www.xmldb.org/xupdate/xupdate-wd.html`.
9. P. Lehti. Design and Implementation of a Data Manipulation Processor. Diplomarbeit, Technische Universitat Darmstadt, 2002. `http://www.lehti.de/beruf/diplomarbeit.pdf`.
10. F. Neven and T. Schwentick. XPath Containment in the Presence of Disjunction, DTDs, and Variables. In *Proc. of ICDT*, pages 315–329, 2003.
11. Mads Olesen. Static validation of XSLT, 2004. `http://www.daimi.au.dk/ mad-man/xsltvalidation`.
12. Grzegorz Rozenberg and Arto Salomaa. *Handbook of formal languages, volume 3: beyond words*. Springer Verlag, 1997.
13. M. Rys. Bringing the Internet to Your Database: Using SQLServer 2000 and XML to Build Loosely-Coupled Systems. In *Proc. of ICDE*, pages 465–472, 2001.
14. A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proc. of VLDB*, 2002.
15. Michael Schwartzbach and Anders Moller. The design space of type checkers for XML transformation languages. In *Proc. of ICDT*, 2005.
16. G. Sur, J. Hammer, and J. Siméon. An XQuery-Based Language for Processing Updates in XML. In *PLAN-X*, 2004. See http://www.cise.ufl.edu/research/mobility.
17. I. Tatarinov, Z. Ives, A.Y. Halevy, and D.S. Weld. Updating XML. In *Proc. of ACM SIGMOD*, 2001.
18. J.W. Thatcher and J.B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Math. Systems Theory*, 2:57–82, 1968.
19. Website. XQuery 1.0: An XML Query Language, 2004. `http://www.w3.org/TR/xquery`.