Keys with Upward Wildcards for XML

Wenfei Fan[1]*, Peter Schwenzer[1], and Kun Wu[1]

Department of Computer and Information Sciences, Temple University
Philadelphia, PA 19122, U.S.A.
{fan, schwenp, kwu}@joda.cis.temple.edu

Abstract. The paper proposes a key constraint language for XML and investigates its associated decision problems. The language is defined in terms of regular path expressions extended with downward and upward wildcards, which can not only move down XML document trees, but also upwards. In a uniform syntax it is capable of expressing both absolute keys and relative keys, which are important for hierarchically structured data, including but not limited to XML documents. In addition to being expressive, keys defined in the language can be reasoned about efficiently. In particular, the paper provides a sound and complete set of inference rules and a cubic time algorithm for determining implication of the keys.

1 Introduction

XML (eXtensible Markup Language [6]) has become the prime standard for data exchange on the Web. XML data typically originates in databases, in which keys are commonly used to convey an essential part of the semantics of the data. In order to use XML to represent data currently residing in databases, one must be able to express the original semantics of the data, in particular, keys. However, the constraint theory developed for traditional structured (e.g., relational and object-oriented) databases no longer applies to XML data [11, 9]. The key specification supported by the XML standard [6] is too weak to fully express database keys or keys for hierarchically structured data. To overcome the limitation, several key languages have been proposed for XML (e.g., [23]). These proposals, however, have a number of technical problems. The absence of a robust key specification for XML has limited its use as a data integration tool, among other things. This demonstrates the need to provide a better key specification language for XML: for specifying the semantics of the data, preserving the original information in data exchange, preventing update anomalies, indexing and archiving the data, and for formulating and optimizing queries.

In a relational database, to define a key we specify the name of a relation (a set of tuples) and a set of attributes of the relation that uniquely identifies tuples in the relation. Along the same lines, to define a key for XML data we specify a pair (Q, S), where

- Q is a path expression that identifies a set of XML elements on which the key is defined, denoted by $[\![Q]\!]$ and called the *target set*; and
- S is a set of path expressions that provides identification for elements in $[\![Q]\!]$, called the *key paths*.

* Contact author. Research Affiliation: Bell Laboratories.

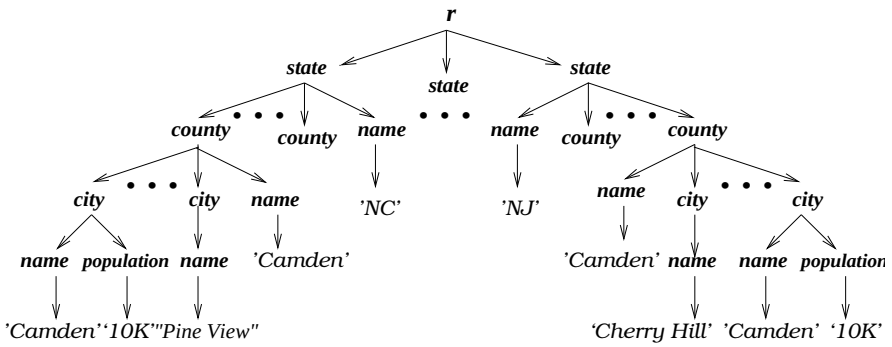

Fig. 1. An XML tree

Since XML data is typically modeled as a tree, $[\![Q]\!]$ represents a set of nodes in an XML document tree, and S specifies another set of nodes whose values identify nodes in $[\![Q]\!]$. As observed by [8], two forms of keys are particularly important for hierarchically structured data, such as XML documents and scientific databases. The first is *absolute key*. Similar to relational database keys, an absolute key identifies a unique node x in $[\![Q]\!]$ with the values of nodes in the subtree rooted at x. In other words, S specifies components of an element in $[\![Q]\!]$. The second is *relative key*, which is analogous to a key for a weak entity in a relational database. Recall that a key of a weak entity is composed of a key of its "parent" entity and some additional identification [21]. In XML data, a relative key uniquely identifies a node x in $[\![Q]\!]$ with two parts: (values of) nodes in the subtrees of some "ancestors" of x, and (values of) nodes in the subtree of x that are additional identification relative to the ancestors. In hierarchically structured data, relative keys are commonly used.

Example. To illustrate absolute and relative keys, let us consider an XML document that contains information about states, counties and cities in the U.S. The document is represented as a tree in Fig. 1. A state has a number of counties, and a county in turn has a number of cities. One wants to define keys for `state`, `county` and `city` elements. A key of `state` is specified with the `name` of `state`:

$$(_*.state,\ \{name\}).$$

Here $_*$ is a combination of a (downward) wildcard '$_$' and the Kleene star '$*$', which matches any path and thus can move down an XML tree to an arbitrary depth. The path expression $_*.state$ starts from the root r and identifies the target set, which consists of all `state` nodes in the tree no matter where they occur. Of these `state` nodes, `name` is defined to be a key. It asserts that two distinct `state` nodes cannot have `name` subelements with the same value. That is, the value of a `name` subelement uniquely identifies a `state` node. This is an *absolute key* since `name` subelement of a `state` node is in the subtree of the `state`. Observe that two different notions of equality are used: *value equality* when comparing the values of nodes reached by following the key path `name`, and *node identity* when comparing `state` nodes in the target set. The equality issue is important for XML keys since XML trees do not allow sharing of nodes. In contrast, only value equality is needed when defining relational database keys.

When it comes to a key for `county` nodes, the story is more complicated. The `name` of a `county` uniquely identifies the `county`, but only within the `state` in which the `county` is located. For example, both NJ (New Jersey) and NC (North Carolina) have a unique county named 'Camden', but with only the county name one cannot distinguish between the two different counties in the two states. In other words, `name` of `county` is a key of `county` nodes *relative to* `state`. To uniquely identify a `county` in the document, one needs not only `name` of `county`, but also `name` of `state` that the `county` belongs to. To capture this semantics, we specify a *relative key*:

$$(_ * .state.county,\ \{name, \widehat{up}.name\}).$$

Here $\widehat{up}$ is the upward wildcard symbol that means "moving up to the parent node" in an XML tree. We call the constraint a *relative key* because it needs `name` of the `state`, which is at a level higher than `county` nodes, as opposed to an absolute key specification.

Similarly, to uniquely identify `city` nodes we need to specify a relative key. Indeed, the 'Camden' counties in NJ and NC both have a city also named by 'Camden'. The relative key consists of `name` of `city`, `name` of `county` in which the `city` is, and `name` of `state` where the `county` is located. That is:

$$(_ * .state.county.city,\ \{name, \widehat{up}.name, \widehat{up}.\widehat{up}.name\}).$$

The constraints given above are keys defined in our key constraint language for XML, denoted by $\mathcal{K}$.

There are two important questions associated with keys for XML. One concerns *satisfiability* of keys: given a finite set of keys, is there an XML document that satisfies these keys? In other words, are these keys meaningful? The second question concerns *implication* of keys: suppose it is known that an XML document satisfies a finite set of keys, does it follow that the document must also satisfy some other key? These are the classical decision problems studied in relational dependency theory [2]. They are equally important for XML data. For example, in data integration, it is increasingly common to use XML as a uniform data format for mediators [5]. One may want to know whether a key φ holds in a mediator interface. However, this cannot be verified directly since the mediator interface does not contain data. One way to verify φ is to show that it is implied by keys that are known to hold [17].

Contributions. The main contributions of the paper are the following:

1. We propose a key constraint language $\mathcal{K}$ for XML. The language is defined in terms of regular path expressions extended with two forms of wildcards: a wildcard '_' that can move down XML trees as commonly used in XML query languages [1], and an *upward wildcard* $\widehat{up}$ that moves up XML trees. With these wildcards, $\mathcal{K}$ is capable of expressing both absolute and relative keys uniformly, using the same syntax. The key specification is independent of any DTD and other schema specifications.
2. We show that keys defined in the language $\mathcal{K}$ can be reasoned about efficiently. More specifically, we demonstrate that any keys of $\mathcal{K}$ are always satisfiable, and for implication of keys, we provide a sound and complete set of inference rules and a cubic time algorithm.

Related work. Keys and the more general functional dependencies have been well studied for relational and object-oriented databases (see [2] for a survey and [18, 20, 22] for recent work). As observed by [9, 8], these constraint specifications are not appropriate for XML data and the complexity results in connection with reasoning about these constraints no longer hold in the XML setting [11, 15].

Key specifications for XML have been proposed in the XML standard (DTD) [6], XML Schema [23] and in a recent proposal [8]. One can view ID attributes in a DTD as keys. This key specification is rather weak. First, IDs can only be specified in DTDs and thus do not help documents without DTDs. Second, in contrast to relational database keys, ID attributes are unique within the entire document rather than among a "class" of elements. Thus one cannot, for example, allow a `student` and a `person` to use the same SSN as an ID. Third, "keys" can only be defined with attributes. In practice, one may want to define keys in terms of subelements or even with path expressions, e.g., to specify SSN as a key of a `person` with a path expression, no matter where SSN occurs in the subtree rooted at the `person`. Fourth, with ID attributes one can only specify unary keys, i.e., keys defined with a single ID attribute. In many situations one wants "composite" keys, e.g., a `student` id and a `course` id as a key of `enrollment` elements, indicating that the `student` is taking the `course`. Finally, at most one ID attribute can be specified for each element type, while in practice, one may want to have more than one key, e.g., both SSN and driver license number as keys of `person` elements. Our key specification overcomes these limitations.

XML Schema specifies keys with XPath [14] expressions. XPath provides a (downward) wildcard and a "parent" function that is similar to our upward wildcard. However, the equivalence and containment of XPath expressions are, as far as the authors are aware, unresolved; and these issues are important if we want to reason about keys. Another technical issue is value equality. XML Schema restricts equality to text, but in many situations keys are not so restricted.

Closer to our work is the key specification recently proposed by [8]. In [8], a general notion of value equality and the notions of absolute and relative keys were introduced. However, [8] did not consider the upward wildcard and thus to identify an element in an XML document, it may require a chain of (relative) keys. In contrast, in our key language $\mathcal{K}$, a single key suffices for this purpose. In addition, although decision problems associated with absolute keys of [8] were studied [7], the problems for relative keys have not been studied, while these are the central technical problems investigated in this paper.

Implication problems associated with a class of simple XML keys (and foreign keys) have been investigated in [16, 15]. The keys considered in those papers are defined with XML attributes and are not as expressive as keys studied here. Implication of path constraints for semistructured data [1] has also been studied [3, 10–12]. These constraints are generalizations of inclusion dependencies and are not capable of expressing keys.

Organization. The rest of the paper is organized as follows. Section 2 defines XML trees, value equality, extended regular path expressions, and the key constraint language $\mathcal{K}$ for XML. Section 3 studies satisfiability and implication of keys in $\mathcal{K}$. Section 4 identifies directions for further research. Proofs are given in the appendix, which may or may not be read at the reviewers' discretion.

2 Regular path expressions and key constraints

In this section, we first present the notions of XML trees and value equality on XML trees. We then introduce two classes of extended regular path expressions and define the key constraint language $\mathcal{K}$ in terms of these path expressions.

2.1 XML trees

A Tree Model. Along the same lines of XML APIs (DOM [4]), query languages (XSL [13, 24]) and schema specifications (XML Schema [23], XPath [14]), we formally define an XML document tree as follows.

Assume a countably infinite set $\mathbf{E}$ of element labels (tags), a countably infinite set $\mathbf{A}$ of attribute names, and a symbol $\mathtt{S}$ indicating text (e.g., PCDATA in XML [6]). Assume that $\mathbf{E}$, $\mathbf{A}$ and $\{\mathtt{S}\}$ are pairwise disjoint.

Definition 2.1: An *XML (document) tree* T is defined by $(V, lab, ele, att, val, r)$, where V is a set of *nodes*; lab is a function from V to $\mathbf{E} \cup \mathbf{A} \cup \{\mathtt{S}\}$; ele is a partial function from V to sequences of V nodes such that for any $v \in V$, if $ele(v)$ is defined then $lab(v) \in \mathbf{E}$; att is a partial function from $V \times \mathbf{A}$ to V such that for any $v \in V$ and $l \in \mathbf{A}$, if $att(v, l) = v'$ then $lab(v) \in \mathbf{E}$ and $lab(v') = l$; val is a partial function from V to string values such that for any node $v \in V$, $val(v)$ is a string iff either $lab(v) = \mathtt{S}$ or $lab(v) \in \mathbf{A}$; r is a distinguished node in V and is called the *root* of T. Without loss of generality, assume $lab(r) = r$, and that there is a unique node in T labeled r.

For any node $v \in V$, if $ele(v)$ is defined then the nodes v' in $ele(v)$ are called the *subelements* of v. For any attribute $l \in \mathbf{A}$, if $att(v, l) = v'$ then v' is called *an attribute* of v. In either case we say that there is a *parent-child edge* from v to v'. The subelements and attributes of v are called the *children* of v. An XML tree has a tree structure, i.e., for each node $v \in V$, there is a unique path of parent-child edges from the root r to v. ∎

Intuitively, V is the set of nodes of the tree T. Nodes in V can be classified into three types: text nodes, attribute nodes, and element nodes. Text nodes have no name but carry text, attribute nodes both have a name and carry text, and element nodes have a name but do not carry text. Attribute and text nodes are leaves, while elements are internal nodes. More specifically, if a node x is an element node, i.e., a node labeled with an element tag in $\mathbf{E}$, then the functions ele and att define the children of x, which are partitioned into *subelements* and *attributes*. The subelements of node x are ordered. In contrast, the attributes of node x are unordered and are identified by their labels (names). It should be noted that more than one subelements of x may have the same tag. In other words, a subelement cannot be identified by its tag. The function val assigns string values to attribute and text nodes, i.e., nodes labeled with attributes in $\mathbf{A}$ or with label $\mathtt{S}$. Observe that T has a tree structure and thus sharing of nodes is not allowed in T. For example, Fig. 1 depicts an XML tree.

An XML tree is called *finite* if V is finite. We only consider *finite* XML trees.

Value equality. The notion of equality is central to a definition of keys. In relational databases, this is not a problem: one needs only to compare values of

atomic types, e.g., integer, real and string values, when checking the satisfaction of a key. An XML tree has a hierarchical structure and it is no longer trivial to compare the values of two XML trees (subtrees).

Let $T = (V, lab, ele, att, val, r)$ be an XML tree. Two nodes v, v' in V are *value equal*, denoted by $v =_v v'$, iff their subtrees are isomorphic by an isomorphism that is the identity on string values. Formally, $v =_v v'$ iff the following conditions are satisfied: (1) $lab(v) = lab(v')$, i.e., they have the same tag (label); (2) if $lab(v) = \mathtt{S}$ or $lab(v) \in \mathbf{A}$, then $val(v) = val(v')$, i.e., if they are text or attribute nodes then they have the same string value; (3) if $lab(v) \in \mathbf{E}$, i.e., if they are element nodes, then their children are pairwise value equal, i.e.,

- for any $l \in \mathbf{A}$, $val(att(v, l)) = val(att(v', l))$;
- if $ele(v) = [v_1, ..., v_k]$, then $ele(v') = [v'_1, ..., v'_k]$ and for all $i \in [1, k]$, $v_i =_v v'_i$.

For example, in Fig. 1 the leftmost `city` node (Camden in NC) and the rightmost `city` node (Camden in NJ) are value equal: they have the same tag (`city`) and all their children (`name`, `population`) are pairwise value equal.

2.2 Extended regular path expressions

A key constraint φ of $\mathcal{K}$ is specified with a pair (Q, S), where Q is a regular path expression called the *target path* of φ, and S is a set of path expressions called the *key paths* of φ. The target path identifies a set of nodes in an XML tree on which the key is defined, denoted by $[\![Q]\!]$. The key paths provide identification for nodes in $[\![Q]\!]$. As mentioned earlier, XPath is too complex for one to reason about keys. As pointed out by [8], the language used to specify key paths could be simpler than the language for describing target paths. Here we present two classes of regular path languages, denoted by TL and KL, for specifying target paths and key paths, respectively. These path languages are both powerful enough to specify important keys and simple enough to yield low complexity when it comes to reasoning about keys defined with these languages.

The path language TL. This language will be used to specify target paths in $\mathcal{K}$ constraints. A path expression of TL is syntactically defined as follows:

$$Q \quad ::= \quad \epsilon \;\mid\; l \;\mid\; Q.Q \;\mid\; _*$$

where ϵ is the empty path, node label $l \in \mathbf{E} \cup \mathbf{A} \cup \{\mathtt{S}\}$, "." is the concatenation operator, "_" is the wildcard symbol that matches any label and "_*" is the combination of the wildcard and Kleene closure that represents any path. For example, $_*.state$ and $_*.state.county$ are path expressions in TL.

A *(simple) path* ρ of TL is a sequence of node labels, i.e., a path expression of TL without $_*$. In an XML tree T, ρ represents a parent-child path. We write $T \models \rho(v_1, v_2)$ iff there is a parent-child path from node v_1 to v_2 whose sequence of node labels is ρ. Recall that attribute and text nodes (S) are leaves in a tree and thus do not have outgoing edges. As a result, we assume that if a path ρ contains attribute or S, then the label must be the last symbol in ρ.

A path expression Q of TL defines a regular language of paths. We use $\rho \in Q$ to denote that path ρ is in the language defined by Q. Let T be an XML tree, and v_1, v_2 be nodes in T. We say that v_2 is *reachable by following Q from* v_1,

denoted by $T \models Q(v_1, v_2)$, iff there is a path $\rho \in Q$ such that $T \models \rho(v_1, v_2)$. In particular, $T \models _* (v_1, v_2)$ if there is a (possibly empty) parent-child path ρ from v_1 to v_2, no matter what the path is. Intuitively, this means moving down from a node to an arbitrary depth. We use $[\![Q]\!]$ to denote the set of nodes in T that is reachable by following Q from the *root* r:

$$[\![Q]\!] = \{v \mid T \models Q(r, v)\}.$$

The path language KL. This is used to express key paths. A path expression P of KL has the form $\varrho.\rho$, where ϱ is a (possibly empty) sequence of $\widehat{up}$ symbols and ρ is a *nonempty* (simple) path. Here $\widehat{up}$ is the *upward wildcard* that matches any node label upwards. We call ϱ the *upward prefix* of P and denote it by P^u. For example, $\widehat{up}.name$ and $\widehat{up}.\widehat{up}.name$ are path expressions in KL.

Intuitively, in an XML tree T, the upward wildcard $\widehat{up}$ indicates moving from a child node up to its parent. More specifically, let v_1 and v_2 be nodes in T. We write $T \models \widehat{up}(v_2, v_1)$ iff v_2 is a child of v_1 no matter what node label v_1 has. In general, given any path expression $P = \varrho.\rho$ in KL with ϱ being the upward prefix of P, we say that v_1 is *reachable by following P from v_2*, denoted by $T \models P(v_2, v_1)$, iff there is a node v in T such that v is reached by moving up k levels from v_2 and $T \models \rho(v, v_1)$, where k is the number of $\widehat{up}$ symbols in ϱ. Observe that because T is a tree, there exists at most one v reachable by moving up k levels from v_2. Note that (simple) paths are also path expressions of KL, i.e., when their upward prefixes are the empty path.

Let n be a node in an XML tree T and P be a path expression of KL. We use $n[\![P]\!]$ to denote the set of nodes reachable from n by following P, i.e.,

$$n[\![P]\!] = \{v \mid T \models P(n, v)\}.$$

Notations. The *length* of a path ρ, denoted by $|\rho|$, is the number of node labels in ρ. Let ϱ be a sequence of upward wildcard symbols. We use $|\varrho|$ to denote the number of the upward wildcard symbols in ϱ. For any path expression P of KL, we use $|P^u|$ to denote the number of the $\widehat{up}$ symbols in the upward prefix of P. For example, let $P = \widehat{up}.\widehat{up}.name$ then $|P^u| = 2$.

Let Q_1, Q_2 be path expressions in TL (resp. KL). We say that Q_1 is a *prefix* of Q_2, denoted by $Q_1 \preceq_p Q_2$, if there exists path expression Q in TL (resp. KL) such that $Q_2 = Q_1.Q$. Analogously, Q_1 is called a *suffix* of Q_2, denoted by $Q_1 \preceq_s Q_2$, if there exists Q such that $Q_2 = Q.Q_1$. For example, we have $_*.state \preceq_p _*.state.county$ and $county \preceq_s _*.state.county$.

2.3 A key constraint language for XML

In terms of TL and KL, we define key constraints of $\mathcal{K}$.

Definition 2.2: A *key constraint* φ for XML is an expression of the form

$$(Q, \{P_1, \ldots, P_k\}),$$

where Q is a path expression in TL and is called the *target path* of φ; P_1, ..., P_k are path expressions in KL and are called the *key paths* of φ, with $k \geq 1$. In addition, Q has a suffix Q_s such that Q_s is a path (i.e., it does not contain '$_*$') and for each $i \in [1, k]$, $|Q_s| \geq |P_i^u|$, where P_i^u is the upward prefix of P_i.

An XML tree T *satisfies* φ, denoted by $T \models \varphi$, iff for any n_1, n_2 in $[\![Q]\!]$, if for all $i \in [1,k]$ there are $x \in n_1[\![P_i]\!]$ and $y \in n_2[\![P_i]\!]$ such that $x =_v y$, then $n_1 = n_2$; that is, $\forall\ n_1\, n_2 \in [\![Q]\!]\ (\bigwedge_{1 \le i \le k} \exists\ x \in n_1[\![P_i]\!]\ \exists\ y \in n_2[\![P_i]\!](x =_v y) \rightarrow n_1 = n_2)$.
The set of all key constraints is denoted by $\mathcal{K}$.

The key φ is called an *absolute key* if for each $i \in [1,k]$, P_i does not contain an upward wildcard, i.e., $|P_i^u| = 0$. Otherwise it is called a *relative key*. ∎

For example, the three constraints we have seen in Sect. 1 are keys of $\mathcal{K}$. More specifically, the key for `state` nodes is an absolute key, and the other two are relative keys. Other examples of $\mathcal{K}$ constraints include:

$$(_*,\ \{ID\}), \quad (_*.book,\ \{name\}), \quad (_*.book.chapter,\ \{number,\ \widehat{up}.name\}).$$

The first two are absolute keys, and the last one is a relative key. The first key in fact describes the semantics of ID attributes in DTDs: for any node, no matter where it occurs in a document, if it has an ID, then the ID value uniquely identifies the node in the whole document. The second key states that `name` is a key for the collection of all `book` nodes in a document. The third key states that `number` is a key for `chapter`, but only within the `book` that contains the `chapter`. To uniquely identify a `chapter` in the document, one needs the `number` of the `chapter` as well as the `name` of the `book` that contains the `chapter`.

It should be noted that two notions of equality is used to define keys: *value equality* $(=_v)$ when comparing nodes reached by following key paths, and *node identify* $(=)$ when comparing two nodes in the target set. This is different from keys in relational databases, in which only value equality is needed.

Note that our key specification is independent of DTDs and other schema specifications. In a key, the target path Q starts at the root of T. We require $|Q_s| \ge |P_i^u|$ to ensure that the sequence P_i^u of $\widehat{up}$ symbols does not seek the "parent" of the root of an XML tree. Intuitively, φ is a key for nodes in $[\![Q]\!]$, and the (nonempty) set of key paths of φ provides identification for these nodes. If two nodes n_1, n_2 have all the key paths and agree on them, then they must be the same. In other words, the key requires that if two nodes in $[\![Q]\!]$ are distinct, then the two sets of nodes reached on some P_i must be disjoint up to value equality. Note that we do not require $n_1[\![P_i]\!]$ and $n_2[\![P_i]\!]$ to be singleton sets. That is, there may be multiple nodes reachable from n_1 or n_2 by following P_i. As long as there exist some $x_i \in n_1[\![P_i]\!]$ and $y_i \in n_2[\![P_i]\!]$ such that x_i and y_i are value equal for all $i \in [1,k]$, then n_1 and n_2 must be the same node. In particular, if P_i is missing at either n_1 or n_2 then $n_1[\![P_i]\!]$ and $n_2[\![P_i]\!]$ are disjoint up to value equality and the key is satisfied. This definition of keys takes into account the semistructured nature of XML data [1]. Observe that P_i starts from nodes in $[\![Q]\!]$. Recall that attribute and text nodes do not have outgoing edges. Thus we only consider paths $\rho \in Q$ such that ρ does not contain any attribute or `S`.

For example, $(_*.state.county.city,\ \{name, \widehat{up}.name, \widehat{up}.\widehat{up}.name\})$ is satisfied by the XML tree in Fig. 1 since no two distinct `city` nodes agree on all the key paths up to value equality. However, the tree does not satisfy the key $(_*.state.county.city,\ \{name\})$, because the `city` of Camden in the `county` of Camden of NC and the `city` of Camden in the `county` of Camden of NJ agree on `name`, but they are different cities.

3 Decision problems for $\mathcal{K}$ constraints

We next investigate the satisfiability and implication problems associated with key constraints of $\mathcal{K}$. We should remark that we consider *finite* XML trees only.

Let Σ be a finite set of $\mathcal{K}$ constraints and T an XML tree. We use $T \models \Sigma$ to denote that T *satisfies* Σ. That is, for any $\psi \in \Sigma$, $T \models \psi$. Let $\Sigma \cup \{\varphi\}$ be a finite set of $\mathcal{K}$ constraints. We use $\Sigma \models \varphi$ to denote that Σ *implies* φ, that is, for any XML tree T, if $T \models \Sigma$, then $T \models \varphi$.

The *satisfiability problem* for $\mathcal{K}$ is to determine, given any finite set of key constraints Σ of $\mathcal{K}$, whether there exists an XML tree T such that $T \models \Sigma$.

The *implication problem* for $\mathcal{K}$ is to determine, given any finite set of key constraints $\Sigma \cup \{\varphi\}$ in $\mathcal{K}$, whether $\Sigma \models \varphi$.

For example, let $\Sigma_0 = \{(_*, \{ID\})\}$ and $\varphi_0 = (_* .person, \{ID, name\})$. Then $\Sigma_0 \models \varphi_0$ is an instance of the implication problem for $\mathcal{K}$.

3.1 Satisfiability of keys

Given a key constraint language for XML, it is important to be able to decide whether keys expressed in the language can be satisfied by any XML trees at all. Better still, it is desirable if all keys are meaningful, i.e., given any finite set of keys, one can always find an XML tree that satisfies the keys. Some key specifications do not have this property. For example, keys proposed in XML Schema [23] and the strong keys defined in [8] do not have the finite model property, i.e., some keys expressed in those languages may not have any finite XML tree that satisfies them (see [8] for examples).

In relational databases, given any finite set of keys (and foreign keys) over a schema, there is always a nonempty finite instance of the schema that satisfies the constraints. Keys of $\mathcal{K}$ are also always satisfiable. Indeed, given any finite set Σ of keys in $\mathcal{K}$, one can always find an XML tree T such that $T \models \Sigma$. In particular, the tree consisting of a single node (the root) satisfies any keys in $\mathcal{K}$.

3.2 Implication of keys

For implication of $\mathcal{K}$ constraints we establish the following:

Theorem 3.1: Implication of $\mathcal{K}$ constraints is finitely axiomatizable and is cubic-time decidable. ∎

The proof is a little involved. It needs a finite set of inference rules and a cubic-time algorithm for determining $\mathcal{K}$ constraint implication.

Containment of path expressions. As will be seen shortly, for inference of $\mathcal{K}$ constraints we need to determine *containment of path expressions*.

Let Q_1, Q_2 be path expressions in TL (resp. KL). We say Q_1 is *contained* in Q_2, denoted by $Q_1 \subseteq Q_2$, if for any XML tree T and any nodes x and y in T, if $T \models Q_1(x, y)$ then $T \models Q_2(x, y)$. We write $Q_1 \equiv Q_2$ if $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$.

Given the definition of KL expressions one can show the following: for any path expressions P_1, P_2 in KL, $P_1 \subseteq P_2$ iff P_1 and P_2 are (syntactically) equal (i.e., $P_1 = P_2$). This can be easily verified by induction on the number of upward wildcard symbols in P_1, i.e., on $|P_1^u|$. This simplifies our inference rules

for $\mathcal{K}$ constraint implication. As a result, containment of KL expressions can be decided in linear time. As an example, observe that $\widehat{up}.name \not\subseteq \widehat{up}.\widehat{up}.foo.name$ and $\widehat{up}.\widehat{up}.foo.name \not\subseteq \widehat{up}.name$ for any label foo.

It should be noted that TL is a star-free regular language (see, e.g., [25]). As shown in [19], the containment problem for star-free languages is coNP-complete in general. In contrast, we have shown the following in [7].

Lemma 3.2: Containment of TL expressions is decidable in quadratic time. ∎

A sound and complete set of rewriting rules has been developed for TL expressions in [7]. In particular, for any TL expressions Q_1, Q_2 and Q, the following holds: $Q_1.Q.Q_2 \subseteq Q_1._*.Q_2$. As an immediate result of this rule we have $Q_1._*.Q_2 \equiv Q_1._*._*.Q_2$. See the appendix for a proof sketch of the lemma.

Finite axiomatizability. We next present a set of inference rules for implication of key constraints of $\mathcal{K}$, denoted by $\mathcal{I}$, as follows:

– superkey:
$$\frac{(Q,\, S), \quad P \in KL}{(Q,\, S \cup \{P\})}$$

– subnode:
$$\frac{\begin{array}{c}(Q.\rho,\, \{\varrho.P_1, ..., P'_l, ..., \varrho.P_k\}),\\ P'_l \text{ is either } \rho' \text{ or } \varrho.P_l\\ \rho, \rho' \text{ are paths,}\\ \varrho \text{ is a sequence of } \widehat{up}\text{'s and } |\varrho| = |\rho|,\\ P_l = \rho.\rho'\end{array}}{(Q,\, \{P_1, ..., P_l, ..., P_k\})}$$

– containment:
$$\frac{(Q,\, S), \quad Q' \subseteq Q}{(Q',\, S)}$$

– empty-path:
$$\frac{S \subseteq KL, \quad S \text{ is nonempty}}{(\epsilon,\, S)}$$

We briefly illustrate these rules as follows.
(1) *superkey*. It should be mentioned that this rule also holds for keys in relational databases. It states that if S is a key for $[\![Q]\!]$, then so is any superset of S.
(2) *subnode*. Consider two nodes $n_1, n_2 \in [\![Q]\!]$. Suppose that for all $i \in [1, k]$ there exist $x_i \in n_1[\![P_i]\!]$ and $y_i \in n_2[\![P_i]\!]$ such that $x_i =_v y_i$. Then since ρ is a prefix of P_l, there must be distinct nodes $n'_1, n'_2 \in [\![Q.\rho]\!]$ such that $n'_1 \in n_1[\![\rho]\!]$ and $n'_2 \in n_2[\![\rho]\!]$. Observe that $x_l \in n'_1[\![P'_l]\!]$ and $y_l \in n'_2[\![P'_l]\!]$, and moreover, for any $i \in [1, k]$, if $i \neq l$ then $x_i \in n'_1[\![\varrho.P_i]\!]$ and $y_i \in n'_2[\![\varrho.P_i]\!]$. Since $(Q.\rho,\ \{\varrho.P_1, ..., P'_l, ..., \varrho.P_k\})$ is a key, we must have $n'_1 = n'_2$. Hence $n_1 = n_2$ since n_1, n_2 are ancestors of n'_1, n'_2, respectively, and because XML trees do not allow sharing of nodes. Thus $(Q,\, \{P_1, ..., P_l, ..., P_k\})$ is also a key. Note that if $\rho = \epsilon$, then the key in the precondition and the one in the consequence of the rule are the same.
(3) *containment*. This rule holds because $[\![Q']\!] \subseteq [\![Q]\!]$ given $Q' \subseteq Q$ and moreover, a key for $[\![Q]\!]$ is also a key for any subset of $[\![Q]\!]$. Note that containment of path expressions is needed here.
(4) *empty-path*. It is sound because $[\![\epsilon]\!]$ consists of a single node, i.e., the root.

Given a finite set $\Sigma \cup \{\varphi\}$ of $\mathcal{K}$ constraints, we use $\Sigma \vdash \varphi$ to denote that φ is provable from Σ using $\mathcal{I}$. That is, there is an $\mathcal{I}$-proof of φ from Σ.

For example, recall Σ_0 and φ_0 given earlier. We show $\Sigma_0 \vdash \varphi_0$ as follows. By $_*.person \subseteq _*$ and the containment rule, $\Sigma_0 \vdash (_*.person, \{ID\})$. By the superkey rule, $\{(_*.person, \{ID\})\} \vdash (_*.person, \{ID, name\})$. Thus $\Sigma_0 \vdash \varphi_0$.

We show that $\mathcal{I}$ is indeed a finite axiomatization. That is, given any finite set $\Sigma \cup \{\varphi\}$ of $\mathcal{K}$ constraints, $\Sigma \models \varphi$ iff $\Sigma \vdash \varphi$. See the appendix for a proof.

Lemma 3.3: The set $\mathcal{I}$ of inference rules is sound and complete for implication of $\mathcal{K}$ constraints. ∎

A similar result was shown in [7]. However, keys considered in [7] do not contain the upward wildcard $\widehat{up}$. The presence of $\widehat{up}$ complicates the proof.

For example, we have shown $\Sigma_0 \vdash \varphi_0$. By Lemma 3.3, we have $\Sigma_0 \models \varphi_0$.

Complexity. Finally, we show that keys of $\mathcal{K}$ can be reasoned about efficiently. A cubic-time algorithm for determining implication of $\mathcal{K}$ constraints is given in the appendix. The algorithm is developed based on the inference rules of $\mathcal{I}$.

Lemma 3.4: The implication problem for $\mathcal{K}$ is decidable in cubic time. ∎

Lemmas 3.3 and 3.4 suffice to prove Theorem 3.1.

4 Conclusion

We have proposed a key constraint language for XML defined in terms of regular path expressions extended with downward and upward wildcards. Despite the simple syntax of the language, we have demonstrated that it is capable of expressing both absolute and relative keys, which are important for hierarchically structured data such as XML documents and scientific databases. We have shown that any keys expressed in the language can be satisfied by an XML document. We have also shown that the implication problem for the key language is finitely axiomatizable and cubic-time decidable.

A number of questions remain open. First, the decision problems are investigated in the absence of XML DTDs. Keys may interact with DTDs and the interaction may complicate reasoning about keys [15, 11]. This issue needs further investigation. Second, keys also interact with foreign keys. As shown by [16], the implication problem for simple keys and foreign keys is undecidable even in the context of relational databases. It is important to find practical restrictions under which the decision problems for XML keys and foreign keys are decidable. Finally, one may want to define keys with more general path expressions. The decision problems for more expressive keys need to be studied.

Acknowledgments. We thank Peter Buneman for helpful discussions.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufman, 2000.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

3. S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 122–133, 1997.
4. V. Apparao et al. *Document Object Model (DOM) Level 1 Specification.* W3C Recommendation, Oct. 1998. `http://www.w3.org/TR/REC-DOM-Level-1`.
5. C. Baru et al. XML-based information mediation with MIX. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 597–599, 1999.
6. T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C Recommendation, Feb 1998. `http://www.w3.org/TR/REC-xml`.
7. P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about keys for XML. Technical Report TUCIS-TR-2000-005, Temple University, 2000.
8. P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *Proc. 10th Int'l World Wide Web Conf. (WWW'10)*, 2001.
9. P. Buneman, W. Fan, J. Siméon, and S. Weinstein. Integrity constraints for semistructured data and XML. *SIGMOD Record*, 30(1), Mar. 2001.
10. P. Buneman, W. Fan, and S. Weinstein. Path constraints on semistructured and structured data. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 129–138, 1998.
11. P. Buneman, W. Fan, and S. Weinstein. Interaction between path and type constraints. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 56–67, 1999.
12. P. Buneman, W. Fan, and S. Weinstein. Path constraints in semistructured databases. *J. Computer and System Sciences (JCSS)*, 61:146–193, 2000.
13. J. Clark. *XSL Transformations (XSLT).* W3C Recommendation, Nov. 1999. `http://www.w3.org/TR/xslt`.
14. J. Clark and S. DeRose. *XML Path Language (XPath).* W3C Recommendation, Nov. 1999. `http://www.w3.org/TR/xpath`.
15. W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, 2001.
16. W. Fan and J. Siméon. Integrity constraints for XML. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 23–34, 2000.
17. D. Florescu, L. Raschid, and P. Valduriez. A methodology for query reformulation in CIS using semantic knowledge. *Int'l J. Cooperative Information Systems (IJCIS)*, 5(4):431–468, 1996.
18. C. Hara and S. Davidson. Reasoning about nested functional dependencies. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 91–100, 1999.
19. H. Hunt, D. Rosenkrantz, and T. Szymanski. On the equivalence, containment, and covering problems for the regular and context-free languages. *J. Computer and System Sciences (JCSS)*, 12:222–268, 1976.
20. M. Ito and G. E. Weddell. Implication problems for functional constraints on databases supporting complex objects. *J. Computer and System Sciences (JCSS)*, 50(1):165–187, 1995.
21. R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw-Hill Higher Education, 2000.
22. Z. Tari, J. Stokes, and S. Spaccapietra. Dependency constraints and normal forms for object-oriented schemata. *ACM Transaction on Database Systems (TODS)*, 22(4), 1997.
23. H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures.* W3C, Apr. 2000. `http://www.w3.org/TR/xmlschema-1/`.
24. P. Wadler. A formal semantics for patterns in XSL. Technical report, Computing Sciences Research Center, Bell Labs, Lucent Technologies, 2000.
25. S. Yu. Regular languages. In G. Rosenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, pages 41–110. Springer, 1996.

Appendix

Proof sketch of Lemma 3.2. Given two TL expressions Q_1, Q_2, we consider their NFAs (nondeterministic finite state automata), denoted by $M(Q_1), M(Q_2)$, respectively. We treat these NFAs as graphs and define a certain simulation relation from $M(Q_1)$ to $M(Q_2)$ based on the rewriting rules (see, e.g., [1] for discussions of simulation). It can be shown that $Q_1 \subseteq Q_2$ if and only if there exists a simulation relation from $M(Q_1)$ to $M(Q_2)$, and moreover, there is a quadratic time algorithm for determining whether such a simulation exists or not. See [7] for details. ∎

Proof of Lemma 3.3: Soundness of $\mathcal{I}$ can be verified by induction on the lengths of $\mathcal{I}$-proofs. To prove its completeness, it suffices to show that given any finite set $\Sigma \cup \{\varphi\}$ of $\mathcal{K}$ constraints, if $\Sigma \not\vdash \varphi$, then $\Sigma \not\models \varphi$. That is, if $\Sigma \not\vdash \varphi$, then there exists an XML tree G such that $G \models \Sigma$ but $G \models \neg\varphi$.

To construct G, we introduce a notation of abstract trees. An *abstract tree* T is an extension of XML tree by allowing '$_*$' as a node label. As a result, a parent-child path Q in T is a path expression of TL, which may contain '$_*$'. Let x, y be two nodes in T and Q' be a path expression of TL. We say $T \models Q'(x, y)$ iff there exists a parent-child path Q in T such that $T \models Q(x, y)$ and $Q \subseteq Q'$. However, we do not allow the upward wildcard $\widehat{up}$ to move up from a node labeled $_*$, i.e., $T \not\models \widehat{up}(y, x)$ for any node y labeled $_*$. The connection between abstract trees and XML trees is revealed by the following claim.

Claim: If there is an abstract tree T such that $T \models \Sigma$ and $T \models \neg\varphi$, then there exists an XML tree G such that $G \models \Sigma$ if $G \models \neg\varphi$.

Proof of Claim: Given an abstract tree T such that $T \models \Sigma$ and $T \models \neg\varphi$, we construct an XML tree G from T as follows. Let ζ be a fresh element tag that does not appear in $\Sigma \cup \{\varphi\}$. We replace every occurrence of '$_*$' in T with ζ, and refer to the XML tree obtained in this way as G. We show $G \models \Sigma$ and $G \models \neg\varphi$.

We first show $G \models \Sigma$. Suppose, by contradiction, that there is $\phi \in \Sigma$ such that $G \models \neg\phi$. Assume that $\phi = (Q, \{P_1, \ldots, P_k\})$. Then by Definition 2.2, there must exist $n_1, n_2 \in [\![Q]\!]$, $x_i \in n_1[\![P_i]\!]$ and $y_i \in n_2[\![P_i]\!]$ in G for all $i \in [1, k]$ such that $x_i =_v y_i$, but however, $n_1 \neq n_2$. Observe that the set of nodes in G is the same as that in T. By the choice of ζ, we must have $n_1, n_2 \in [\![Q]\!]$ in T. Note that P_i's are in KL and thus do not contain '$_*$'. Thus by the definition of abstract trees, in T we also have $x_i \in n_1[\![P_i]\!]$ and $y_i \in n_2[\![P_i]\!]$ for all $i \in [1, k]$. Therefore, $T \models \neg\phi$. This contradicts the assumption that $T \models \Sigma$. Thus $G \models \Sigma$.

We next show $G \models \neg\varphi$. Let $\varphi = (Q, \{P_1, \ldots, P_k\})$. By $T \models \neg\varphi$, there exist $n_1, n_2 \in [\![Q]\!]$, $x_i \in n_1[\![P_i]\!]$ and $y_i \in n_2[\![P_i]\!]$ in T for all $i \in [1, k]$ such that $x_i =_v y_i$ but $n_1 \neq n_2$. By the construction of G and the choice of ζ, we must have $n_1, n_2 \in [\![Q]\!]$ in G. Recall that P_i's do not contain '$_*$' by the definition of KL expressions. Therefore, we have $x_i \in n_1[\![P_i]\!]$ and $y_i \in n_2[\![P_i]\!]$ in G for all $i \in [1, k]$. As a result, $G \models \neg\varphi$.

This completes the proof of the claim. ∎

Equipped with the claim, we proceed to prove Lemma 3.3. Assume $\Sigma \not\vdash \varphi$. We construct an XML tree G such that $G \models \Sigma$ and $G \models \neg\varphi$. Assume

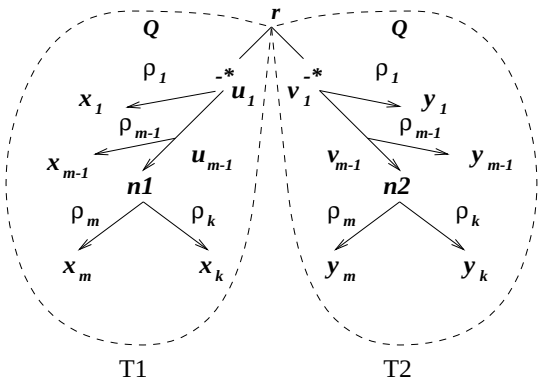

Fig. 2. An abstract tree

$$\varphi = (Q,\ \{P_1, \ldots, P_k\}), \quad \Sigma = \{\phi_i \mid i \in [1,n]\}, \quad \phi_i = (Q_i,\ \{P_{i1}, \ldots, P_{ik_i}\}).$$

Consider the key φ. By the definition of KL expressions, we assume that for each $j \in [1,k]$, $P_j = \varrho_j.\rho_j$, where ϱ_j is the upward prefix of P_j and ρ_j is a nonempty path of node labels. Without loss of generality, also assume that for all $j, l \in [1,k]$, $|\varrho_j| \geq |\varrho_l|$ if $j < l$ and that there is $m \in [1,k]$ such that for all $j > m$, we have $P_j = \rho_j$, i.e., $|\varrho_j| = 0$. Given these, we construct G in two steps. We first build an abstract tree T using φ, and then construct G from T. Observe that if $Q = \epsilon$, then $\Sigma \vdash \varphi$ by the empty-path rule in $\mathcal{I}$, which contradicts the assumption. Thus we assume $Q \neq \epsilon$ in the sequel. The abstract tree T is depicted in Fig. 2. The root r of T has two disjoint branches, referred to as T_1 and T_2, which consist of a Q path each. Let n_1, n_2 be the two distinct nodes at the end of the Q paths, respectively. For each $j \in [1,k]$, let u_j, v_j be the two nodes that are $|\varrho_j|$-level up from n_1 and n_2, respectively. From u_j and v_j we add two disjoint ρ_j paths. Let x_j and y_j refer to the nodes at the ends of the ρ_j paths, respectively. We assume that for $j \in [1,k]$, $x_j =_v y_j$, but for all other pair of nodes x, y in T, $x \neq_v y$. This can be achieved by assuming that each node has a text subelement (if it is not already an attribute or a text node) and assigning appropriate string values to these text (attribute) nodes. Observe the following: (1) $T \models \neg\varphi$. Indeed, n_1 and n_2 agree on all the key paths but they are distinct nodes. (2) For all $j \in [1,k]$, ρ_j contains neither '$_*$' nor '$\widehat{up}$'. The path from root r to u_1 (v_1) is a prefix of Q and may contain "$_*$". In contrast, the paths below u_1 and v_1 do not contain '$_*$' by Definition 2.2.

We next modify T such that the modified tree satisfies Σ. For each $i \in [1,n]$, if $T \not\models \phi_i$, then there must be $x, y \in [\![Q_i]\!]$, $x'_j \in x[\![P_{ij}]\!]$, $y'_j \in y[\![P_{ij}]\!]$ for all $j \in [1,k_i]$ such that $x'_j =_v y'_j$, but $x \neq y$ (recall $\phi_i = (Q_i,\ \{P_{i1}, \ldots, P_{ik_i}\})$). By the construction of T, it is easy to see that x, x'_j and y, y'_j are in different branches. Without loss of generality, assume that x, x'_j are in T_1 and y, y'_j are in T_2. By the definition of the tree T, if $x'_j =_v y'_j$, then there must be $s \in [1,k]$ such that $x'_j = x_s$ and $y'_j = y_s$. Moreover, $k_i \leq k$. Without loss of generality, assume that for $j \in [1,k_i]$, $x'_j = x_j$ and $y'_j = y_j$. To ensure the satisfaction of ϕ_i, we merge x and y to be the same node, and also merge their ancestor such that they are pairwise identical. There are two cases to consider here.

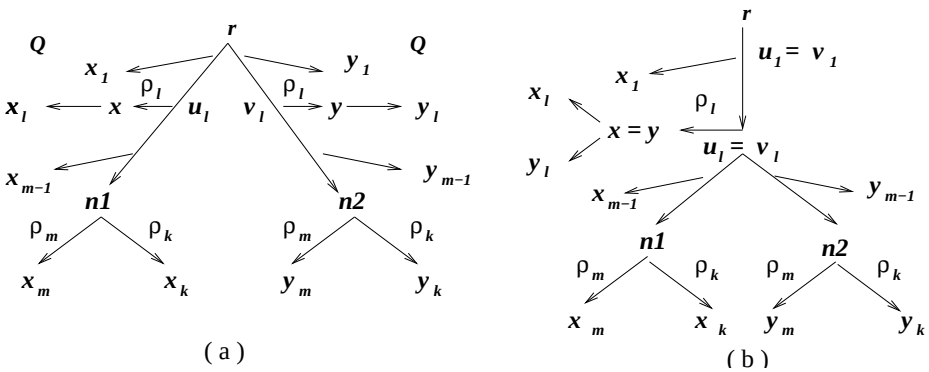

Fig. 3. The abstract trees in case 1

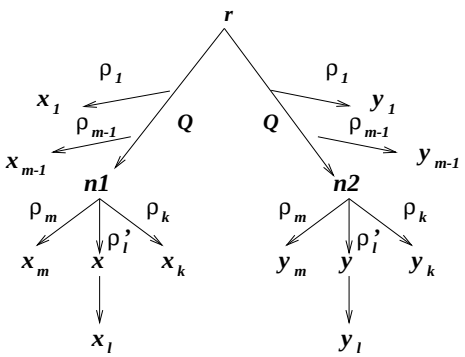

Fig. 4. The abstract tree in case 2

(1) x and y are above n_1 and n_2 (i.e., they are not in the subtrees rooted at n_1, n_2. See Fig. 3 (a)). By the definition of abstract trees, there must be TL expressions Q', Q'', number $l < m$ and path ρ'_l such that $Q = Q'.Q''$, $Q'.\rho'_l \subseteq Q_i$ and $\rho'_l \preceq_p \rho_l$. That is, the concatenation of a prefix of Q and a prefix of ρ_l is contained in Q_i. Note that it is possible that $\rho'_l = \epsilon$ and in that case, a prefix of Q is contained in Q_i. We merge x and y to be the same node and moreover, for all the corresponding nodes x', y' in the two paths from root r to x, y in the two branches, we let $x' = y'$. In particular, if $x' = u_s$ and $y' = v_s$ for some $s < l$, then we discard the ρ_s path in the T_2 branch (see Fig. 3 (b)). Let T' be the tree obtained after the process. It can be verified that $T' \models \phi_i$. In addition, for any $\phi \in \Sigma$, if ϕ is satisfied by the tree before the processing of ϕ_i, then T' also satisfies ϕ. Moreover, $T' \models \neg\varphi$ since n_1 and n_2 remain distinct in T' but they agree on all the key paths in φ.

(2) x and y are below n_1 and n_2 (i.e., x and y are in the subtrees of n_1 and n_2. See Fig. 4). In this case, we show $\Sigma \vdash \varphi$. Indeed, if x and y are below n_1 and n_2, then there must be $l \geq m$ and paths ρ'_l, ρ''_l such that $Q.\rho'_l \subseteq Q_i$ and $P_l = \rho'_l.\rho''_l$. In addition, there is a sequence ϱ of $\widehat{up}$ symbols such that $|\varrho| = |\rho'_l|$ and for all $j \in [1, k_i]$ and $j \neq l$, we have $\varrho.P_j = P_{ij}$. When $j = l$, we have either $\varrho.P_l = P_{il}$ or

Algorithm

Input: a finite set $\Sigma \cup \{\varphi\}$ of $\mathcal{K}$ constraints, where $\varphi = (Q, \{P_1, ..., P_k\})$

Output: 'yes' iff $\Sigma \models \varphi$ and 'no' otherwise

1. if $Q = \epsilon$
   then output 'yes' and terminate;
2. for each $\phi \in \Sigma$ do
   if $\phi = (Q', \{P'_1, \ldots P'_m\})$ and all of the following conditions are satisfied:
   (i) there is a path ρ such that $Q.\rho \subseteq Q'$, and
   (ii) there exists a sequence ϱ of $\widehat{up}$ symbols such that $|\varrho| = |\rho|$, and
   (iii) there exist a path ρ' and a number $l \in [1, k]$ such that $P_l = \rho.\rho'$, and
   (iv) there exists $s \in [1, m]$ such that either $P'_s = \rho'$ or $P'_s = \varrho.\rho.\rho'$, and
   (v) for all $i \in [1, m]$ and $i \neq s$ there is $j \in [1, k]$ such that $P'_i = \varrho.P_j$
   then output 'yes' and terminate;
3. output 'no';

Fig. 5. An algorithm for testing $\mathcal{K}$ constraint implication

$\rho'' = P_{il}$. That is, ϕ_i is of the form: $(Q_i, \{\varrho.P_1, ..., P_{il}, ..., \varrho.P_{k_i}\})$. By ϕ_i and the containment rule in $\mathcal{I}$, we have $\Sigma \vdash (Q.\rho'_l, \{\varrho.P_1, ..., P_{il}, ..., \varrho.P_{k_i}\})$. By this and the subnode rule, $\Sigma \vdash (Q, \{P_1, ..., P_l, ..., P_{k_i}\})$. Finally, by the superkey rule, we have $\Sigma \vdash (Q, \{P_1, ..., P_l, ..., P_{k_i}, ..., P_k\})$. Observe that a special case is when $\rho'_l = \epsilon$, i.e., Q is contained in Q_i. In that case the containment and superkey rules suffice. Thus $\Sigma \vdash \varphi$. This contradicts our assumption that $\Sigma \nvdash \varphi$. Therefore, case (2) cannot happen if $\Sigma \nvdash \varphi$.

We repeat the procedure until all keys in Σ are processed. Let T' denote the abstract tree obtained at the end of the process. We show $T' \models \Sigma$ and $T' \models \neg\varphi$. By the assumption that $\Sigma \nvdash \varphi$, for any $\phi \in \Sigma$, either $T \models \phi$ or we have that $T \models \neg\phi$ and ϕ has the properties described in case 1. From the discussions given above follows that the processing of case 1 ensures that $T' \models \phi$ and $T' \models \neg\varphi$.

Now we have an abstract tree T' such that $T' \models \Sigma$ and $T' \models \neg\varphi$. By the claim that we have shown earlier, there must exist an XML tree G such that $G \models \Sigma$ and $G \models \neg\varphi$. Therefore, $\Sigma \not\models \varphi$.

This completes the proof of Lemma 3.3. ∎

Proof of Lemma 3.4: Using the inference rules of $\mathcal{I}$, we develop a cubic time algorithm for determining implication of $\mathcal{K}$ constraints (Fig. 5).

From the proof of Lemma 3.3 follows that the algorithm is correct. We next analyze its complexity. For each $\phi \in \Sigma \cup \{\varphi\}$ in the input, let us use $|\phi|$ to denote the size of ϕ, and $|\Sigma|$ to denote the size of Σ. Observe that each $\phi \in \Sigma$ is processed at most once and the condition test in the body of the loop takes at most $O(|\phi|^2 \cdot |\varphi|)$ time by Lemma 3.2. Hence the loop takes $O(|\Sigma|^2 \cdot |\varphi|)$ time. Thus the algorithm is in cubic time. ∎