Keys with Upward Wildcards for XML

Wenfei Fan[1], Peter Schwenzer[2], and Kun Wu[3]

[1] Bell Laboratories and Temple University
wenfei@acm.org
[2] Temple University Health System
schwenp@tuhis.temple.edu
[3] Motorola Inc.
akw037@motorola.com

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 to hierarchically structured data. In addition, keys defined in the language can be reasoned about efficiently. The paper provides a sound and complete set of inference rules and a cubic time algorithm for determining implication of the keys.

1 Introduction

As XML [5] is increasingly used in data exchange, data integration and modeling Web data [1], the importance of keys for XML is being recognized: for specifying the semantics of the data, preserving information in data exchange and integration, preventing update anomalies, indexing and archiving the data, and for formulating and optimizing XML queries. Several key specifications have been proposed for XML (e.g., [20]). These proposals, however, have a number of limitations. This highlights the need to provide a key specification language for XML that is capable of expressing important semantics of XML data and is simple enough to be reasoned about efficiently.

In a relational database, we define a key for a relation (a set of tuples) by providing a set of attributes 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*.

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 [7], two forms of keys are particularly important to hierarchically structured data, such as XML documents and scientific databases. The first is *absolute key*, which identifies a node x in $[\![Q]\!]$ with the values of some nodes in the subtree rooted at x. That is, S consists of paths that lead to nodes in the subtree of x. The second is *relative key*, which, to identify a node y in $[\![Q]\!]$, needs to refer to ancestors of y, as opposed to an absolute key specification.

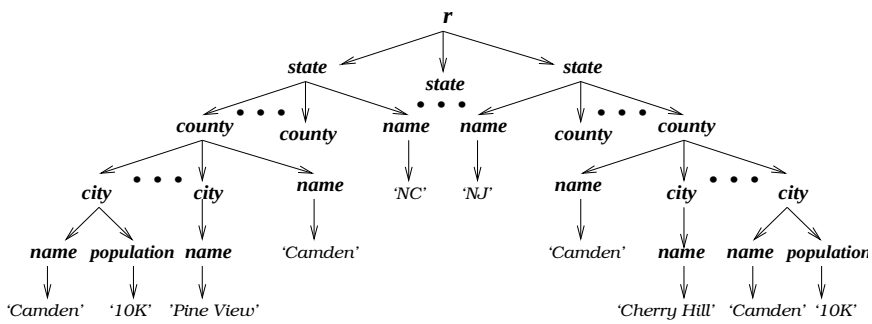

Fig. 1. An XML tree

In other words, S consists of not only paths leading to nodes in the subtree of y, but also paths that refer to nodes in the subtrees of some ancestors of y. This is analogous to a key for a weak entity in a relational database, which is composed of a key of its "parent" entity and some additional identification [18].

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. 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 it does not need to refer to any ancestor of `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 to 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 a `state`. 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 the `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. This is a relative key because it needs `name` of the `state`, i.e., it is relative to an ancestor of `county`.

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 the `city`, `name` of the `county` in which the `city` is, and `name` of the `state` where the `county` is located. That is:

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

These are keys 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 to XML data. For example, in data integration, it is increasingly common to use XML as a uniform data format for mediators [4]. 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 [15].

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 [16, 17, 19] for recent work). As observed by [8, 7], 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 [9, 12].

Key specifications for XML have been proposed in the XML standard (DTD) [5], XML Schema [20] and in a recent proposal [7]. 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 a DTD. Second, IDs are unique within the entire document rather than among a "class" of elements. Because of this, IDs cannot capture keys in relational databases. Third, "keys" can only be defined with attributes. In practice, one may want to define keys with complex structures. Fourth, with ID attributes one can only specify unary and primary keys, i.e., at most one ID attribute can be specified for each element type. Our key specification overcomes these limitations.

XML Schema specifies keys with XPath [11] expressions. XPath provides a (downward) wildcard and a "parent" function that is similar to our upward

wildcard. Thus our $\mathcal{K}$ constraints can be expressed in XPath syntax. However, as observed by [7], XPath is too complicated to be reasoned about efficiently. It is still open whether the decision problems are decidable or not for keys defined with XPath. 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 of [7]. In [7], a general notion of value equality and the notions of absolute and relative keys were introduced. However, [7] 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. As upward wildcards are supported by XPath and XSL [10], one may also want to use them to express XML keys and to develop a better understanding of XML key analysis in their presence. This paper investigates the impact of upward wildcards on the decision problems associated with absolute and relative keys for XML. These problems for absolute keys were studied in [6] (in the absence of upward wildcards).

Implication problems associated with a class of simple XML keys (and foreign keys) were investigated in [14, 12]. The keys considered in those papers are defined with XML attributes and are not as expressive as keys studied here.

Organization. The rest of the paper is organized as follows. Section 2 defines 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. The proofs are given in the full version of the paper [13].

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 [3]), query languages (XSL [10]) and specifications (XML Schema [20], XPath [11]), we define an XML document tree as follows. Assume a countable set $\mathbf{E}$ of element labels (tags), a countable set $\mathbf{A}$ of attribute names, and a symbol $\mathtt{S}$ indicating text (e.g., PCDATA in XML [5]). Assume that $\mathbf{E}$, $\mathbf{A}$ and $\{\mathtt{S}\}$ are pairwise disjoint.

An *XML tree* T is defined by $(V, lab, ele, att, val, r)$. Here V is the set of *nodes* of T and lab is a function that labels each node with a symbol in $\mathbf{E}$, $\mathbf{A}$ or with $\mathtt{S}$. A node v is called an *element* (resp. *attribute* or *text* node) if $lab(v)$ is in $\mathbf{E}$ (resp. $lab(v) \in \mathbf{A}$ or $lab(v) = \mathtt{S}$). Elements may have children, whereas attributes and text nodes are leaves of T. More specifically, if v is an element, then the mappings ele and att define the *children* of v: $ele(v)$ is a list of elements of V called the *subelements* of v, and $att(v)$ is a set of attributes with distinct labels, called the *attributes* of v. There is a parent-child *edge* from v to each of its children. Observe that the subelements are ordered (a list), while the attributes are not (a set). Attributes and text nodes carry string values. The mapping val assigns a string value to each attribute and text node x, i.e., $val(x)$ is a string. Finally, r is a distinguished node in V and is called the *root* of T.

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

For example, Fig. 1 depicts an XML tree.

Value equality. This notion is central to a definition of keys. To check keys in relational databases, one needs only to compare atomic values, e.g., integer, real and string values. 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 the following conditions are satisfied: (1) $lab(v) = lab(v')$, i.e., they have the same label; (2) if they are text or attribute nodes then $val(v) = val(v')$, i.e., they have the same string value; (3) if they are element nodes, then their children are pairwise value equal, i.e., for any $v_1 \in att(v)$, there is $v_2 \in att(v')$ such that $v_1 =_v v_2$ and vice versa, and moreover, 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 label (`city`) and all their children (`name`, `population`) are pairwise value equal.

2.2 Extended Regular Path Expressions

A key φ of $\mathcal{K}$ is specified by (Q, S), where Q is a path expression called the *target path*, 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 the nodes in $[\![Q]\!]$. As pointed out by [7], the language for specifying key paths could be simpler than the one for describing target paths. As mentioned earlier, XPath is too complex for one to reason about keys. Here we present two regular path languages, denoted by TL and KL, for specifying target paths and key paths, respectively. These languages are both powerful enough to specify important keys and simple enough to yield low complexity for 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 downward 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 is a TL expression 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 (`S`) nodes are leaves and do not have outgoing edges. Hence we assume that if 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 language 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 KL expressions.

In an XML tree T, $\widehat{up}$ 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 label v_1 has. In general, given any KL expression $P = \varrho.\rho$ 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 there exists at most one v reachable by moving up k levels from v_2. Note that nonempty (simple) paths are also KL expressions, i.e., when their upward prefixes are the empty path.

Let n be a node in T and P be a KL expression. We use $n[\![P]\!]$ to denote the set of nodes reachable from n by following P: $n[\![P]\!] = \{v \mid T \models P(n, v)\}$.

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 $\widehat{up}$ symbols in ϱ. Thus for any KL expression P, $|P^u|$ denotes the number of the $\widehat{up}$ symbols in P. For example, let $P = \widehat{up}.\widehat{up}.name$ then $|P^u| = 2$.

Let Q_1, Q_2 be 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 Q in TL (resp. KL) such that $Q_2 = Q_1.Q$. Similarly, Q_1 is called a *suffix* of Q_2, denoted by $Q_1 \preceq_s Q_2$, if there is Q such that $Q_2 = Q.Q_1$. For example, $_*.state \preceq_p _*.state.county$ 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 $\mathcal{K}$ as follows.

Definition 2.1: A *key constraint* φ of $\mathcal{K}$ is an expression of the form

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

where Q is a TL expression, called the *target path* of φ; P_1, ..., P_k are KL expressions, 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, $\forall\ n_1\, n_2 \in [\![Q]\!]\ (\bigwedge_{1 \leq i \leq k} \exists\ x \in n_1[\![P_i]\!]\ \exists\ y \in n_2[\![P_i]\!](x =_v y) \rightarrow n_1 = n_2)$.

A 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:

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

The first one is an absolute key, which says that `name` is a key for all the `book` nodes in a document. The second one is a relative key, which 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`.

In a key φ, the target path Q starts at the root of an XML tree T. We require $|Q_s| \geq |P_i^u|$ to ensure that the sequence P_i^u of $\widehat{up}$ symbols does not seek the "parent" of the root of T. Intuitively, the (nonempty) set of key paths of φ provides identification for the nodes in $[\![Q]\!]$. If two nodes n_1, n_2 of $[\![Q]\!]$ have all the key paths and agree on them, then they must be the same node. 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. 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.

Observe 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. Also note that our key specification is independent of DTDs and other schema specifications.

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 a finite XML tree that satisfies the keys. Some key specifications do not have this finite model property. For example, keys proposed in XML Schema [20] and the strong keys defined in [7] may not have any finite XML tree that satisfies them (see [6] for examples).

In relational databases, given any 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. ∎

A proof sketch is given as follows. It involves containment of path expressions, a finite set of inference rules and an algorithm for determining key 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.

For TL expressions, we have shown the following in [6].

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

A sound and complete set of rewriting rules has been developed for determining containment of TL expressions. In particular, for any TL expressions Q_1, Q_2 and Q, the following holds: $Q_1.Q.Q_2 \subseteq Q_1._*.Q_2$. See [6] for the proof.

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: $$\frac{(Q.\rho,\ \{\varrho.P_1, ..., P'_l, ..., \varrho.P_k\}), \quad P'_l \text{ is either } \rho' \text{ or } \varrho.P_l \quad \rho, \rho' \text{ are paths}, \quad \varrho \text{ is a sequence of } \widehat{up}\text{'s and } |\varrho| = |\rho|, \quad P_l = \rho.\rho'}{(Q,\ \{P_1, ..., P_l, ..., P_k\})}$$
- 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 $[\![Q]\!]$.
(4) *empty-path*. It is sound because $[\![\epsilon]\!]$ consists of a single node, i.e., the root.

Note that containment of path expressions is used in the inference rules.

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 $\Sigma_0 \vdash \varphi_0$.

Lemma 3.3 asserts 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 [13] for the proof.

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

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

A similar result was shown in [6]. However, keys considered in [6] do not contain the upward wildcard $\widehat{up}$. The presence of $\widehat{up}$ complicates the proof.

Complexity. Finally, we show that keys of $\mathcal{K}$ can be reasoned about efficiently.

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

A cubic-time algorithm for determining implication of $\mathcal{K}$ constraints is given in [13]. The algorithm is developed based on the inference rules of $\mathcal{I}$.

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, it is capable of expressing both absolute and relative keys, which are important to hierarchically structured data including XML documents. 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 DTDs. Keys may interact with DTDs and the interaction may complicate reasoning about keys [12, 9]. This issue needs further investigation. Second, keys help information preservation in data exchange and integration. An important project is to study how to achieve this in practice. Third, keys may lead to efficient storage and access methods for XML data. This practical issue deserves further research.

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. V. Apparao et al. *Document Object Model (DOM) Level 1 Specification.* W3C Recommendation, Oct. 1998. `http://www.w3.org/TR/REC-DOM-Level-1`.
4. C. Baru et al. XML-based information mediation with MIX. In *Proc. ACM SIGMOD Conf. on Management of Data*, 1999.
5. 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`.
6. 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.
7. 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.
8. P. Buneman, W. Fan, J. Siméon, and S. Weinstein. Integrity constraints for semistructured data and XML. *SIGMOD Record*, 30(1), Mar. 2001.
9. P. Buneman, W. Fan, and S. Weinstein. Interaction between path and type constraints. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, 1999.
10. J. Clark. *XSL Transormations (XSLT).* W3C Recommendation, Nov. 1999. `http://www.w3.org/TR/xslt`.
11. J. Clark and S. DeRose. *XML Path Language (XPath).* W3C Recommendation, Nov. 1999. `http://www.w3.org/TR/xpath`.
12. 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.
13. W. Fan, P. Schwenzer, and K. Wu. Keys with upward wildcards for XML. Technical Report TUCIS-TR-2000-007, Temple University, 2000. `http://www.cis.temple.edu/~fan/papers/xml/upward.ps.gz`.
14. W. Fan and J. Siméon. Integrity constraints for XML. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, 2000.
15. 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.
16. C. Hara and S. Davidson. Reasoning about nested functional dependencies. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, 1999.
17. 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.
18. R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw-Hill Higher Education, 2000.
19. Z. Tari, J. Stokes, and S. Spaccapietra. Dependency constraints and normal forms for object-oriented schemata. *ACM Trans. on Database Systems*, 22(4), 1997.
20. 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/`.