# Reasoning about Keys for XML

Peter Buneman[‡]
University of Pennsylvania
peter@cis.upenn.edu

Susan Davidson[*]
University of Pennsylvania
susan@cis.upenn.edu

Wenfei Fan[†]
Bell Labs and Temple University
wenfei@research.bell-labs.com

Carmem Hara
Universidade Federal do Parana, Brazil
carmem@inf.ufpr.br

Wang-Chiew Tan[‡]
University of Pennsylvania
wctan@saul.cis.upenn.edu

**Abstract**

We study absolute and relative keys for XML, and investigate their associated decision problems. We argue that these keys are important to many forms of hierarchically structured data including XML documents. In contrast to other proposals of keys for XML, these keys can be reasoned about efficiently. We show that the (finite) satisfiability problem for these keys is trivial, and their (finite) implication problem is finitely axiomatizable and decidable in PTIME in the size of keys.

## 1 Introduction

Keys are of fundamental importance in databases. They provide a means of locating a specific object within the database and of referencing an object from another object (e.g. relationships); they are also an important class of constraints on the validity of data. In particular, value-based keys (as used in relational databases) provide an invariant connection from an object in the real world to its representation in the database. This connection is crucial for modifying the database as the world that it models changes.

As XML is increasingly used to model real world data, it is natural to require a value-based method of locating an element in an XML document. Key specifications for XML have been proposed in the XML standard [10], XML Data [28], and XML Schema [33]. However existing proposals cannot handle one or more of the following situations. First, one may want to define keys with complex structure. For example, the `name` subelement of a `person` element could be a natural key, but may itself have `first-name` and `last-name` subelements. Keys should not be constrained to be character strings (attribute values.) Second, in hierarchically structured data, one may want to identify elements within the scope of a sub-document. For example, the `number` subelement of a `chapter` element may be a key for chapters of a specific book, but would not be unique among chapters of different books. The idea of keys having a scope is not new. In relational databases, scoped keys exists in the form of weak entities. Using the same example, `chapter` is a weak entity of `book`. A chapter number would only make sense in the context of some book name (assuming that the keys for `book` and `chapter` are `name` and `number` respectively). Currently, there is no counterpart of weak entities in key specification proposals. Third, since XML data is not required to conform to a DTD or schema definition, it is useful to have a definition of keys that is independent of any specification (such as a DTD or XML Schema) of the type of an XML document.

To overcome these limitations, the authors recently [12] proposed a key structure for XML which has the following benefits:

1. Keys are defined in terms of one or more path expressions, i.e. they may involve one or more attributes, subelements or more general structures. Equality is defined on tree structures instead of on simple text, referred to as *value equality*.
2. Keys, in their general form, are defined relative to a set of context nodes, referred to as *relative keys*. Such keys can be concatenated to form a hierarchical key structure, common in scientific data sets. An *absolute key* is a special case of a relative key, in which the set of context nodes consists of the root.
3. The specification of keys does not depend on any typing specification of the document (e.g. DTD or XML Schema).

In developing our notion of keys for XML, we start with a tree model of data as used in DOM [6], XSL [18, 36], XQL [31] and XML Schema [33]. An example of this representation for some XML data is shown in Fig. 1 in which nodes are annotated by their type: $E$ for element, $A$ for attribute, and $S$ for string (or PCDATA). Some value-based keys for this data might include: 1) A `book` node is identified by `@isbn`; 2) An `author` node is identified by `name`, no matter where the `author` node appears; and 3) Within any subtree rooted at `book`, a `chapter` node is identified by `@number`. These keys are defined independently of any type specification. The first two are examples of absolute keys since they must hold globally throughout the tree. Observe that `name` has a complex structure. As a consequence, to test whether two authors violate this constraint involves testing value-equality on the subtrees rooted at their `name` nodes. The last one is an example of a relative key since it holds locally within each subtree rooted at a `book`. It

[*]Supported by NSF DBI99-75206
[†]Supported by NSF CAREER IIS
[‡]Supported by NSF IIS 99-77408 and NSF DL-2 IIS 98-17444

```
⟨db⟩
        ⟨book  isbn=123⟩
           ⟨title⟩ HTML ⟨/title⟩
           ⟨author⟩
              ⟨name⟩ ⟨first-name⟩ Tim ⟨/first-name⟩ ⟨last-name⟩ Bray ⟨/last-name⟩ ⟨/name⟩
           ⟨/author⟩
           ⟨chapter  number=1⟩ text ⟨/chapter⟩
           .  .  .
           ⟨chapter  number=10⟩ text ⟨/chapter⟩
        ⟨/book⟩
        ⟨book  isbn=234⟩
           ⟨title⟩ XML ⟨/title⟩
           ⟨author⟩
              ⟨name⟩ ⟨first-name⟩ Tim ⟨/first-name⟩ ⟨last-name⟩ Bray ⟨/last-name⟩ ⟨/name⟩
           ⟨/author⟩
           ⟨author⟩
              ⟨name⟩ ⟨first-name⟩ Jean ⟨/first-name⟩ ⟨last-name⟩ Paoli ⟨/last-name⟩ ⟨/name⟩
           ⟨/author⟩
           ⟨chapter  number=1⟩ text ⟨/chapter⟩
           .  .  .
           ⟨chapter  number=12⟩ text ⟨/chapter⟩
        ⟨/book⟩
⟨/db⟩
```

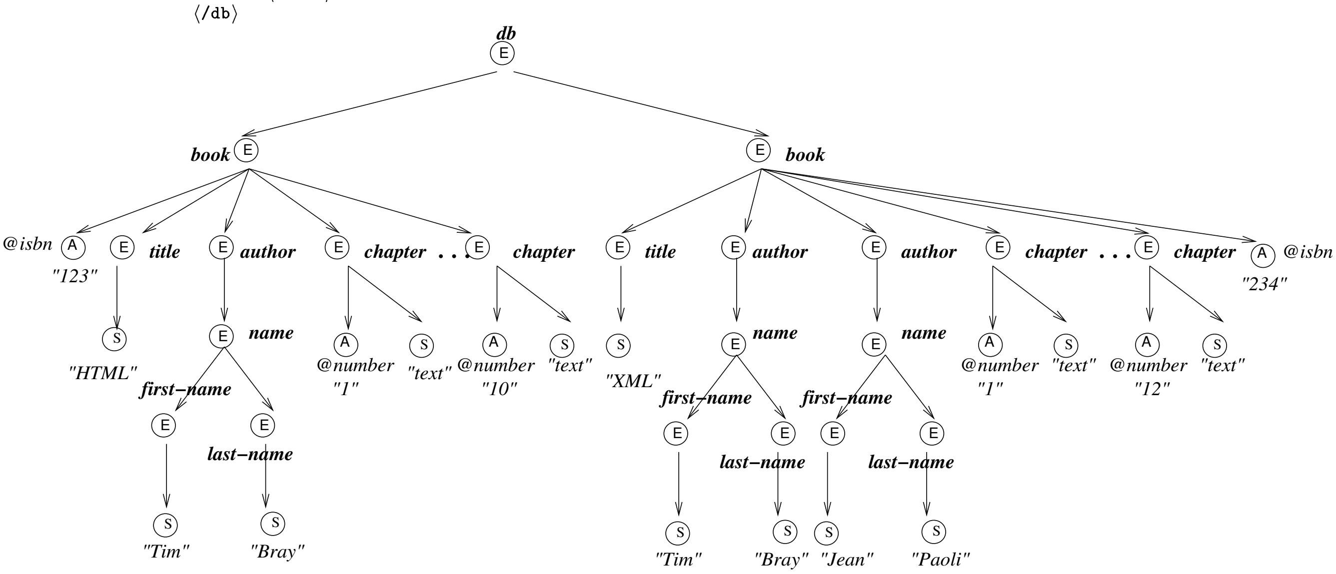


**Figure 1**: Example of some XML data and its representation as a tree

should be noted that a chapter `@number` is not a key for the set of all `chapter` nodes in the document since two different books have chapters with `@number= 1`. It is worth remarking that proposals prior to [12] were not capable of expressing the second and third constraints.

The notion of relative keys is particularly natural for hierarchically structured data, and is motivated in part by our experience with scientific data formats. Many scientific databases transmit their data in one of a variety of data formats. Some of these data formats are general purpose (e.g. ASN.1, which is used in GenBank [9]; AceDB [32]; and EMBL, which is used in SwissProt [8]).[1] XML itself is also emerging as a standard for data exchange, especially with micro-array data (see for example the DTDs GEML [1] and MAML [2]). All of these specifications have a hierarchical structure. As a typical example, SwissProt [7] at the top level consists of a large set of entries, each of which is identified by an accession number. Within each entry there is a sequence of citations, each of which is identified by a number 1,2,3... within the entry. Thus to identify a citation fully, we need to provide both an accession number for the entry and the number of the citation within the entry. Note that the same number for a citation (e.g. 3) may occur within many different entries, thus the citation number is a relative key within each entry. In relational database design we also find the notion of a hierarchical key structure in *weak entities*. Here the key of the weak entity consists of a key of the parent entity and some additional identification of the dependent entity [30].

One of the most interesting questions involving keys is that of logical implication, i.e., deciding if a new key holds given a set of existing keys. This is important for minimizing the expense of checking that a document satisfies a set of key constraints, and may also provide the basis for reasoning about how constraints can be propagated through view definitions. Thus a central task for the study of XML keys is to develop an algorithm for determining logical implication. It is also desirable to develop a sound and complete set of inference rules for generating symbolic proofs of logical implication. The existence of such inference rules, referred to as *axiomatizability*, is a stronger property than the existence of an algorithm, because the former implies the

[1] All these data formats have easy translations to XML.

latter but not the other way around [4]. Another interesting question is whether a set of keys is "reasonable" in the sense that there exists some (finite) document that satisfies the key specification (*finite satisfiability*).

In relational databases, the (finite) implication problems for keys (and more generally, functional dependencies) have been well studied (see, e.g., [4, 30]). The finite satisfiability problem is trivial: given any finite set of keys over a relational schema, one can always find a finite instance of the schema that satisfies the keys. The implication problem for keys is also easy, and is decidable in linear time. In fact, there are exactly two inference rules, and these are sound and complete for the implication analysis. Let $R$ be a relation schema and $Att(R)$ denote the set of attributes of $R$. We use $X \rightarrow R$ to denote that $X$ is a key of $R$, where $X \subseteq Att(R)$. Then the rules can be given as:

$$\frac{}{Att(R) \rightarrow R} \quad \text{(schema)}$$

$$\frac{X \rightarrow R \quad X \subseteq Y}{Y \rightarrow R} \quad \text{(superkey)}$$

The first rule says that for any relation schema $R$, the set of all the attributes of $R$ is a key of $R$. The second asserts that if $X$ is a key of $R$ then so is any superset of $X$.

For XML the story is more complicated since the hierarchical structure of data is far more complex than the 1NF structure of relational data. In some proposals keys are not even finitely satisfiable. For example, consider a key of XML Schema (in a simplified syntax): $(//*, [id])$, where "//*" (in XPath [19] syntax) traverses to any descendant of the root of an XML document tree. This key asserts that any node in an XML tree must have a unique `id` subelement (of text value) and its `id` uniquely identifies the node in the entire document. However, it is clear that no finite XML tree satisfies this key because any `id` node must have an `id` itself, and this yields an infinite chain of `id` nodes. For implication of XML keys, the analysis is even more intriguing. Keys of XML Schema are defined in terms of XPath [19], which is a powerful yet complicated language. A number of technical questions in connection with XPath are still open, including the containment of XPath expressions which is important in the interpretation of XML keys. Therefore, to the best of our knowledge, the implication problem for keys defined in XML Schema is still open, as is its axiomatizability.

In contrast, we show in this paper that the keys of [12] can be reasoned about efficiently. More specifically, we show that they are finitely satisfiable and their implication is decidable in PTIME. Better still, their (finite) implication is *finitely axiomatizable*, i.e., there is a finite set of inference rules that is sound and complete for implication of these keys. In developing these results, we also investigate value-equality on XML subtrees and containment of path expressions, which are not only interesting in their own right but also important in the study of decision problems for XML keys.

Despite the importance of reasoning about keys for XML, little previous work has investigated this issue. The only work closely related to this paper is [21, 22]. For a class of keys and foreign keys, the decision problems were studied in the absence [22] and presence [21] of DTDs. The keys considered there are defined in terms of XML attributes and are not as expressive as keys studied in this paper.[2] Integrity constraints defined in terms of navigation paths have been studied for semistructured [3] and XML data in [5, 15, 16, 17]. These constraints are generalizations of inclusion dependencies commonly found in relational databases, and are not capable of expressing keys. Generalizations of functional dependencies have also been studied [24, 27, 34]. However these generalizations were investigated in database settings, which are quite different from the tree model for XML data considered in this paper. Surveys on XML constraints can be found in [14, 35].

The remainder of the paper is organized as follows. Section 2 formally defines XML trees, value equality, and (absolute and relative) keys for XML. Section 3 establishes the finite axiomatizability and complexity results: First, we give a quadratic time algorithm for determining inclusion of path expressions. The ability to determine inclusion of path expressions is then used in developing inference rules for keys, for which a PTIME algorithm is given. Finally, Section 4 identifies directions for further research. All the proofs are given in the full version of the paper [13].

## 2 Keys

As illustrated in Fig. 1, our notion of keys is based on a tree model of XML data. Although the model is quite simple, we need to do two things prior to defining keys: the first is to give a precise definition of value equality for XML keys; the second is to describe a path language that will be used to locate sets of nodes in an XML document. We therefore introduce a class of regular path expressions, and define keys in terms of this path language.

### 2.1 A tree model and value equality

An XML document is typically modeled as a node-labeled tree. We assume three pairwise disjoint sets of labels: $\mathbf{E}$ of element tags, $\mathbf{A}$ of attribute names, and a singleton set $\{\mathtt{S}\}$ denoting text (PCDATA).

**Definition 2.1:** An *XML (document) tree* is defined to be $T = (V, lab, ele, att, val, r)$, where (1) $V$ is a set of nodes; (2) $lab$ is a mapping $V \rightarrow \mathbf{E} \cup \mathbf{A} \cup \{\mathtt{S}\}$ which assigns a label to each node in $V$; a node $v$ in $V$ is called an *element* (*E node*) if $lab(v) \in \mathbf{E}$, an *attribute* (*A node*) if $lab(v) \in \mathbf{A}$, and a text node (*S node*) if $lab(v) = \mathtt{S}$; (3) $ele$ and $att$ are partial mappings that define the edge relation of $T$: for any node $v$ in $V$,

- if $v$ is an element then $ele(v)$ is a *sequence* of elements and text nodes in $V$ and $att(v)$ is a *set* of attributes in $V$; for each $v'$ in $ele(v)$ or $att(v)$, $v'$ is called a *child* of $v$ and we say that there is a (directed) edge from $v$ to $v'$;

[2] We do not consider foreign keys and DTDs in the current paper.

- if $v$ is an attribute or a text node then $ele(v)$ and $att(v)$ are undefined;

(4) $val$ is a partial mapping that assigns a string to each attribute and text node: for any node $v$ in $V$, if $v$ is an $A$ or $S$ node then $val(v)$ is a string, and $val(v)$ is undefined otherwise; (5) $r$ is the unique and distinguished root node. An XML tree has a tree structure, i.e., for each $v \in V$, there is a unique path of edges from root $r$ to $v$. An XML tree is said to be *finite* if $V$ is finite. ■

For example, Fig. 1 depicts an XML tree that represents an XML document.

With this, we are ready to define value equality on XML trees. Let $T = (V, lab, ele, att, val, r)$ be an XML tree, and $n_1, n_2$ be two nodes in $V$. Informally, $n_1, n_2$ are value equal if they have the same tag (label) and in addition, either they have the same (string) value (when they are `S` or `A` nodes) or their children are pairwise value equal (when they are `E` nodes). More formally:

**Definition 2.2:** Two nodes $n_1$ and $n_2$ are *value equal*, denoted by $n_1 =_v n_2$, iff the following conditions are satisfied:

- $lab(n_1) = lab(n_2)$;
- if $n_1, n_2$ are $A$ or $S$ nodes then $val(n_1) = val(n_2)$;
- if $n_1, n_2$ are $E$ nodes, then 1) for any $a_1 \in att(n_1)$, there exists $a_2 \in att(n_2)$ such that $a_1 =_v a_2$, and vice versa; and 2) if $ele(n_1) = [v_1, \ldots, v_k]$, then $ele(n_2) = [v'_1, \ldots, v'_k]$ and for all $i \in [1, k]$, $v_i =_v v'_i$.

That is, $n_1 =_v n_2$ iff their subtrees are isomorphic by an isomorphism that is the identity on string values. ■

As an example, in Fig. 1, the `author` subelement of the first `book` and the first `author` subelement of the second `book` are value equal.

### 2.2 Path Languages

There are many options for a path language, ranging from very simple ones involving just labels to more expressive ones such as regular languages or even XPath. However, to develop inference rules for keys we need to be able to reason about inclusion of path expressions (the *containment* problem). It is well known that for regular languages, the containment problem is not finitely axiomatizable; and for XPath, although nothing is known at this point we strongly suspect that it is not much easier. We therefore restrict our attention to the path language $PL$, which is expressive enough to be interesting yet simple enough to be reasoned about efficiently. We will also use a simpler language ($PL_s$) in defining keys, and therefore show both these languages in the table below.

| Path Language | Syntax |
|---|---|
| $PL_s$ | $\rho \ ::= \ \epsilon \mid l.\rho$ |
| $PL$ | $q \ ::= \ \epsilon \mid l \mid q.q \mid \_*$ |

In $PL_s$, a path is a (possibly empty) sequence of node labels. Here $\epsilon$ represents the empty path, node label $l \in \mathbf{E} \cup \mathbf{A} \cup \{\mathtt{S}\}$, and "." is a binary operator that concatenates two path expressions. The language $PL_s$ describes the class of finite sequences of node labels. The language $PL$ is a generalization of $PL_s$ that allows the symbol "_*", a combination of wildcard and Kleene closure. This symbol represents any (possibly empty) finite sequence of node labels. It should be noted that for any path expression $p$ in any of the path languages, the following equality holds: $p.\epsilon = \epsilon.p = p$. These path languages are fragments of regular expressions [25], with $PL_s$ contained in $PL$.

A path in $PL_s$ is used to describe a path in an XML tree $T$, and a path expression in $PL$ describes a set of such paths. Recall that an attribute node or a text node is a leaf in $T$ and it does not have any child. Thus a path $\rho$ in $PL_s$ is said to be *valid* if for any label $l$ in $\rho$, if $l \in \mathbf{A}$ or $l = \mathtt{S}$, then $l$ is the last symbol in $\rho$. Similarly, we define *valid* path expressions of $PL$. In what follows we only consider valid paths and we assume that the regular language defined by a path expression of $PL$ containing only valid paths. For example, *book.author.name* is a valid path in $PL_s$ and $PL$, while $\_*.author$ is a valid path expression in $PL$ but it is not in $PL_s$.

We now give some notation that will be used throughout the rest of the paper. Let $\rho$ be a path in $PL_s$, $P$ a path expression in $PL$ and $T$ an XML tree.

**Length.** The *length* of path $\rho$, denoted by $|\rho|$, is the number of labels in $\rho$ (the empty path has length 0). By treating "_*" as a special label, we also define the length of $PL$ expression $P$, denoted by $|P|$, to be the number of labels in $P$.

**Membership.** We use $\rho \in P$ to denote that path $\rho$ is in the regular language defined by path expression $P$. For example, $book.author.name \in book.author.name$ and $book.author.name \in \_*.name$.

**Reachability.** Let $n_1, n_2$ be nodes in $T$. We say that $n_2$ is *reachable* from $n_1$ by following path $\rho$, denoted by $T \models \rho(n_1, n_2)$, iff $n_1 = n_2$ if $\rho = \epsilon$, and if $\rho = \rho'.l$, then there exists node $n$ in $T$ such that $T \models \rho'(n_1, n)$ and $n_2$ is a child of $n$ with label $l$.

We say that node $n_2$ is *reachable* from $n_1$ by following path expression $P$, denoted by $T \models P(n_1, n_2)$, iff there is a path $\rho \in P$ such that $T \models \rho(n_1, n_2)$.

For example, if $T$ is the XML tree in Fig. 1, then all the `name` nodes are reachable from the root by following *book.author.name*; they are also reachable by following $\_*$.

**Node set.** Let $n$ be a node in $T$. We use the notation $n[\![P]\!]$ to denote the set of nodes in $T$ that can be reached by following the path expression $P$ from node $n$. That is, $n[\![P]\!] = \{n' \mid T \models P(n, n')\}$. We shall use $[\![P]\!]$ as abbreviation for $r[\![P]\!]$, when $r$ is the root node of $T$. For example, referring to Fig. 1 and let $n$ be the first `book` element, then $n[\![chapter]\!]$ is the set of all `chapter` elements of the first book and $[\![\_*.chapter]\!]$ is the set of all `chapter` elements in the entire document.

**Definition 2.3:** The *value intersection* of $n_1[\![P]\!]$ and $n_2[\![P]\!]$, denoted by $n_1[\![P]\!] \cap_v n_2[\![P]\!]$, is defined by:

$$n_1[\![P]\!] \cap_v n_2[\![P]\!] = \{(z, z') \mid \exists \rho \in P,\ z \in n_1[\![\rho]\!],\ z' \in n_2[\![\rho]\!],\ z =_v z'\}$$

■

Thus $n_1[\![P]\!] \cap_v n_2[\![P]\!]$ consists of node pairs that are value equal and are reachable by following the same simple path in the language defined by $P$ starting from $n_1$ and $n_2$, respectively. For example, let $n_1$ and $n_2$ be the first and second `book` elements in Fig. 1, respectively. Then $n_1[\![author]\!] \cap_v n_2[\![author]\!]$ is a set consisting of a single pair $(x, y)$, where $x$ is the `author` subelement of the first book and $y$ is the first `author` subelement of the second book.

### 2.3 A Key constraint language for XML

We are now in a position to define keys for XML and what it means for an XML document to satisfy a key constraint.

**Definition 2.4:** A *key constraint* $\varphi$ for XML is an expression of the form

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

where $Q$, $Q'$ and $P_i$ are $PL$ expressions such that for all $i \in [1, k]$, $Q.Q'.P_i$ is a valid path expression. The path $Q$ is called the *context path*, $Q'$ is called the *target path*, and $P_1$, ..., $P_k$ are called the *key paths* of $\varphi$.

When $Q = \epsilon$, we call $\varphi$ an *absolute key*, and abbreviate the key to $(Q', \{P_1, \ldots, P_k\})$, otherwise $\varphi$ is called a *relative* key. We use $\mathcal{K}$ to denote the language of keys, and $\mathcal{K}_{abs}$ to denote the set of absolute keys in $\mathcal{K}$. ■

A key $\varphi = (Q, (Q', \{P_1, \ldots, P_k\}))$ specifies the following:

- the context path $Q$, starting from the root of an XML tree $T$, identifies a set of nodes $[\![Q]\!]$;
- for each node $n \in [\![Q]\!]$, $\varphi$ defines an absolute key $(Q', \{P_1, \ldots, P_k\})$ that is to hold on the subtree rooted at $n$; specifically,
  - the target path $Q'$ identifies a set of nodes $n[\![Q']\!]$ in the subtree, referred to as the *target set*,
  - the key paths $P_1, \ldots, P_k$ identify nodes in the target set. That is, for each $n' \in n[\![Q']\!]$ the values of the nodes reached by following the key paths from $n'$ uniquely identify $n'$ in the target set.

As illustrated in Fig. 2, the context path $Q$ starts at the root of $T$, the target path $Q'$ starts at a node $n$ in $[\![Q]\!]$ and the key paths start at a node $n'$ in $n[\![Q']\!]$. That is why we require $Q.Q'.P_i$ to be valid in Definition 2.4.

For example, the keys on Fig. 1 mentioned in Sec. 1 can be written as follows:

1. `@isbn` is a key of `book` nodes: $(book,\ \{@isbn\})$;
2. `name` is a key of `author` nodes no matter where they are: $(\_*.author,\ \{name\})$;

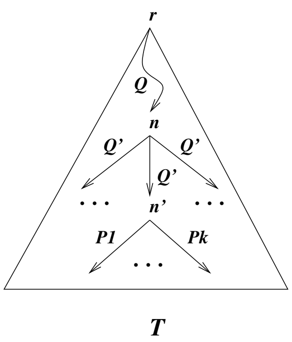


**Figure 2**: Illustration of a key $(Q, (Q', \{P_1, \ldots, P_k\}))$

3. within each subtree rooted at a `book`, `@number` is a key of `chapter`: $(book,\ (chapter,\ \{@number\}))$.

The first two are absolute keys of $\mathcal{K}_{abs}$ and the last one is a relative key of $\mathcal{K}$.

**Definition 2.5:** Let $\varphi = (Q, (Q', \{P_1, \ldots, P_k\}))$ be a key of $\mathcal{K}$. An XML tree $T$ *satisfies* $\varphi$, denoted by $T \models \varphi$, iff for any $n$ in $[\![Q]\!]$ and any $n_1, n_2$ in $n[\![Q']\!]$, if for all $i \in [1, k]$ there exist a path $\rho \in P_i$ and nodes $x \in n_1[\![\rho]\!]$, $y \in n_2[\![\rho]\!]$ such that $x =_v y$, then $n_1 = n_2$. That is,

$$\forall n \in [\![Q]\!]\ \ \forall n_1\, n_2 \in n[\![Q']\!] \\ ((\bigwedge_{1 \le i \le k} n_1[\![P_i]\!] \cap_v n_2[\![P_i]\!] \neq \emptyset) \to n_1 = n_2).$$

■

As mentioned earlier, the key $\varphi$ defines an absolute key that is to hold on the subtree rooted at each node $n$ in $[\![Q]\!]$. That is, if two nodes in $n[\![Q']\!]$ are distinct, then the two sets of nodes reached on some $P_i$ must be disjoint (by value equality.) More specifically, for any $n \in [\![Q]\!]$ and for any distinct nodes $n_1, n_2$ in $n[\![Q']\!]$, there must exist some $P_i$, $1 \le i \le k$, and path $\rho \in P_i$ such that for all $x$ in $n_1[\![P_i]\!]$ and $y$ in $n_2[\![P_i]\!]$, $x \neq_v y$.

Observe that when $Q = \epsilon$, i.e., when $\varphi$ is an absolute key, the set $[\![Q]\!]$ consists of a unique node, namely, the root of the tree. In this case $T \models \varphi$ iff

$$\forall n_1\, n_2 \in [\![Q']\!]\ ((\bigwedge_{1 \le i \le k} n_1[\![P_i]\!] \cap_v n_2[\![P_i]\!] \neq \emptyset) \to n_1 = n_2).$$

As an example, let us consider $\mathcal{K}$ constraints on the XML tree $T$ in Fig. 1.

1) $T \models (book,\ \{@isbn\})$ because the `@isbn` attributes of the two `book` nodes in $T$ have different string values. For the same reason $T \models (book,\ \{@isbn, author\})$. However, $T \not\models (book,\ \{author\})$ because the two books agree on the values of their first author. Observe that the second `book` node has two `author` subelements, and the key requires that none of these `author` nodes is value equal to the `author` of the first `book`.

2) $T \not\models (\_*.author, \{name\})$ because the `author` of the first `book` and the first `author` of the second `book` agree on their names but they are distinct nodes. Note that all `author` nodes are reachable from the root by following $\_*.author$. However, $T \models (book,\ (author, \{name\}))$ because under each `book` node, the same author does not appear twice.

3) $T \models (book,\ (chapter, \{@number\}))$ because in the subtree rooted at each `book` node, the `@number` attribute of each `chapter` has a distinct value. However, observe that $T \not\models (book.chapter, \{@number\})$ since both `book` nodes have a `chapter` with `@number` = 1 but the two `chapter`'s are distinct.

Several subtleties are worth pointing out before we move on to the associated decision problems. First, observe that each key path can specify a *set* of values. For example, consider again $\psi = (book,\ \{@isbn, author\})$ on the XML tree $T$ in Fig. 1, and note that the key path `author` reaches two `author` subelements from the second `book` node. In contrast, this is not allowed in most proposals for XML keys, e.g., XML Schema. The reason that we allow a key path to reach multiple nodes is to cope with the semistructured nature of XML data. Second, the key has no impact on those nodes at which some key path is *missing*. Observe that for any $n \in [\![Q]\!]$ and $n_1, n_2$ in $n[\![Q']\!]$, if $P_i$ is missing at either $n_1$ or $n_2$ then $n_1[\![P_i]\!]$ and $n_2[\![P_i]\!]$ are by definition disjoint. This is similar to *unique constraints* introduced in XML Schema. In contrast to unique constraints, however, our notion of keys is capable of comparing nodes at which a key path may have multiple values. Third, it should be noted that two notions of equality are used to define keys: value equality ($=_v$) when comparing nodes reached by following key paths, and node identity ($=$) when comparing two nodes in the target set. This is a departure from keys in relational databases, in which only value equality is considered.

Our definition of a key says that if the key paths from two nodes are the same and the values at the ends of those paths are the same then those two nodes are identical. This is in some sense a generalization of the relational definition; it allows key values to be "scoped" by their paths. As an example, the following XML data satisfies the absolute key `(part, {_*.@id})`.

```
⟨part⟩
    ⟨widget id=1⟩⟨/widget⟩
⟨/part⟩
⟨part⟩
    ⟨widget id=2⟩⟨/widget⟩
⟨/part⟩
⟨part⟩
    ⟨gadget id=1⟩⟨/gadget⟩
⟨/part⟩
```

However, there is a case to be made for not requiring equality on paths in the definition of a key[3]. One might want, for example, `isbn` to be a key for `book` no matter where it occurs in a book. E,g., we would want the following example to violate a key constraint:

```
⟨book isbn=123⟩
    ...
⟨/book⟩
⟨book⟩
    ⟨identifier isbn=123/⟩
    ...
⟨/book⟩
```

The constraint language we propose will express the first but not the second of these. It is possible to reformulate the definitions to express both by modifying the definition of value intersection (Definition 2.3,) but we do not yet know whether the proofs in this paper can be extended to a more general definition.

[3]We are grateful to one of the referees for pointing this out and for providing the example

## 2.4 Decision problems

In a relational database one can specify arbitrary keys without worrying about their satisfiability. The analysis of implication of relational keys is also trivial. However, as mentioned in Sec. 1, the satisfiability and implication analyses of XML keys are far more intriguing.

We first consider satisfiability of keys of our constraint language $\mathcal{K}$. Let $\Sigma$ be a finite set of keys in $\mathcal{K}$ and $T$ be an XML tree. Following [20], we use $T \models \Sigma$ to denote that $T$ *satisfies* $\Sigma$. That is: for any $\psi \in \Sigma$, $T \models \psi$.

The *satisfiability problem* for $\mathcal{K}$ is to determine, given any finite set $\Sigma$ of keys in $\mathcal{K}$, whether there exists an XML tree satisfying $\Sigma$. The *finite satisfiability problem* for $\mathcal{K}$ is to determine whether there exists a finite XML tree satisfying $\Sigma$.

As observed in Sec. 1, keys defined in some proposals (e.g., XML Schema) may not be finitely satisfiable at all. In contrast, any key constraints of $\mathcal{K}$ can always be satisfied by a finite XML tree, including the single node tree. That is,

**Observation.** For any finite set $\Sigma$ of keys in $\mathcal{K}$, one can always find a finite XML tree that satisfies $\Sigma$.

Next, we consider implication of $\mathcal{K}$ constraints. Let $\Sigma \cup \{\varphi\}$ be a finite set of keys of $\mathcal{K}$. We use $\Sigma \models \varphi$ to denote $\Sigma$ *implies* $\varphi$, that is, for any XML tree $T$, if $T \models \Sigma$, then $T \models \varphi$.

There are two implication problems associated with keys: The *implication problem* is to determine, given any finite set of keys $\Sigma \cup \{\varphi\}$, whether $\Sigma \models \varphi$. The *finite implication problem* is to determine whether $\Sigma$ *finitely implies* $\varphi$, that is, whether it is the case that for any finite XML tree $T$, if $T \models \Sigma$, then $T \models \varphi$.

Given any finite set $\Sigma \cup \{\varphi\}$ of keys in $\mathcal{K}$, if there is an XML tree $T$ such that $T \models \bigwedge \Sigma \wedge \neg\varphi$, then there must be a finite XML tree $T'$ such that $T' \models \bigwedge \Sigma \wedge \neg\varphi$. That is, key implication has the finite model property (see [13] for a proof) and as a result:

**Proposition 2.1:** The implication and finite implication problems for keys coincide. ∎

In light of Proposition 2.1, we can also use $\Sigma \models \varphi$ to denote that $\Sigma$ finitely implies $\varphi$. We investigate the finite implication problems for keys in the next section.

# 3 Key implication

In this section, we study the finite implication problem for keys. Our main result is the following:

**Theorem 3.1:** The finite implication problem for $\mathcal{K}$ is finitely axiomatizable and decidable in PTIME in the size of keys. ∎

We provide a finite axiomatization and an algorithm for determining finite implication of $\mathcal{K}$ constraints. In contrast to their relational database counterparts, the axiomatization and algorithm are not trivial. A road map for the proof of the theorem is as follows. We first study containment of path expressions in the language $PL$ defined in the last section, since the axioms rely on path inclusion. We then provide a finite set of inference rules and show that it is sound and complete for finite implication of $\mathcal{K}$ constraints. Finally, taking advantage of the inference rules, we develop a PTIME algorithm for determining finite implication. We shall also present complexity results in connection with finite implication of absolute keys in $\mathcal{K}_{abs}$.

### 3.1 Inclusion of $PL$ expressions

A path expression $P$ of $PL$ is said to be *included* (or *contained*) in another $PL$ expression $Q$, denoted by $P \subseteq Q$, if for any XML tree $T$ and any node $n$ in $T$, $n[\![P]\!] \subseteq n[\![Q]\!]$. That is, the nodes reached from $n$ by following $P$ are contained in the set of the nodes reached by following $Q$ from $n$. We write $P = Q$ if $P \subseteq Q$ and $Q \subseteq P$.

In the absence of DTDs, $P \subseteq Q$ is equivalent to the containment of the regular language defined by $P$ in the regular language defined by $Q$. Indeed, if there exists a path $\rho \in P$ but $\rho \notin Q$, then one can construct an XML tree $T$ with a path $\rho$ from the root. It is obvious that in $T$, $[\![P]\!] \not\subseteq [\![Q]\!]$. The other direction is immediate. Therefore, $P \subseteq Q$ iff for any path $\rho \in P$, $\rho \in Q$.

We investigate inclusion (containment) of path expressions in $PL$: given any $PL$ expressions $P$ and $Q$, is it the case that $P \subseteq Q$? As will become clear shortly, this is important to the proof of Theorem 3.1, among other things, and is decidable with low complexity:

**Theorem 3.2:** For determining inclusion of $PL$ expressions,

1. there is a sound and complete finite set of inference rules; and
2. there is a quadratic time algorithm.

■

It should be mentioned that $PL$ is a star-free regular language (see, e.g., [37] for the definition of star-free regular languages). In general, the inclusion problem for star-free languages is co-NP complete [26]. Another related result appears in [29] for a slightly more expressive language that allows single wildcards. In contrast to their PTIME complexity result for testing inclusion of path expressions, we are able to provide a set of inference rules, denoted by $\mathcal{I}_p$, in Table 1, and to develop a quadratic time algorithm for testing inclusion of $PL$ expressions.

*Proof sketch:* The soundness of $\mathcal{I}^p$ can be verified by induction on the lengths of $\mathcal{I}^p$-proofs. The proof of completeness is a little involved. Central to the completeness proof and the algorithm is a simulation relation on the transition diagrams of nondeterministic finite state automata (NFAs) [25] that characterize $PL$ expressions. Thus we first describe NFAs and define the simulation.

| | |
|---|---|
| $\dfrac{P \in PL}{\epsilon.P \subseteq P \quad P \subseteq \epsilon.P \quad P.\epsilon \subseteq P \quad P \subseteq P.\epsilon}$ | (empty-path) |
| $\dfrac{P \in PL}{P \subseteq P}$ | (reflexivity) |
| $\dfrac{P \in PL}{P \subseteq \_*}$ | (star) |
| $\dfrac{P \subseteq P' \quad Q \subseteq Q'}{P.Q \subseteq P'.Q'}$ | (composition) |
| $\dfrac{P \subseteq Q \quad Q \subseteq R}{P \subseteq R}$ | (transitivity) |

Table 1: $\mathcal{I}_p$**: rules for** $PL$ **expression inclusion**

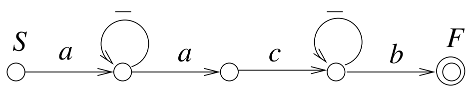


**Figure 3**: NFA for the $PL$ expression $a.\_*.a.c.\_*.b$

To simplify the discussion, we assume from here onwards that a $PL$ expression $P$ is in normal form. A $PL$ expression $P$ is in *normal form* iff it does not contain consecutive $\_*$'s and it does not contain $\epsilon$ unless $P = \epsilon$. Reducing $\_*.\_*$ to $\_*$ can be done using the *star* and *composition* rules of $\mathcal{I}_p$. By the *empty-path* rule, we can also assume that $P$ does not contain $\epsilon$ unless $P = \epsilon$. It takes linear time to rewrite $P$ to an equivalent normal form expression. Thus the assumption does not lose generality.

Let $P$ and $Q$ be path expressions in $PL$. Let the NFAs for $P$ and $Q$ be $M(P)$ and $M(Q)$ respectively, defined as follows:

$$\begin{aligned} M(P) &= (N_1, C \cup \{\_\}, \delta_1, S_1, F_1), \\ M(Q) &= (N_2, C \cup \{\_\}, \delta_2, S_2, F_2), \end{aligned}$$

where $N_1, N_2$ are sets of states, $C$ is the alphabet, $\delta_1, \delta_2$ are transition functions, $S_1, S_2$ are start states, and $F_1, F_2$ are final states of $M(P)$ and $M(Q)$, respectively. Observe that the alphabets of the NFAs have been extended with the special character "_" which can match any character in $C$. By the definition of $PL$ expressions, the transition diagram of such a NFA has a "linear" structure as depicted in Fig. 3. More specifically, $M(P)$ has the following properties (similarly for $M(Q)$):

1) There is a single final state $F_1$.

2) For any state $n \in N_1$ except the final state $F_1$, there exists exactly one letter $l \in C$ such that the NFA can make a move from $n$ on input $l$ to a single different state $n'$ of $N_1$. In other words, $\delta_1(n, l) = \{n'\}$, $n \neq n'$, and $\delta_1(n, l') = \emptyset$ for all $l' \in C$ if $l' \neq l$. For the final state, $\delta_1(F_1, l) = \emptyset$ for all $l \in C$.

We shall simply write $\delta_1(n, l) = n'$ if $\delta_1(n, l) = \{n'\}$.

3) At any state $n \in N_1$, given the special letter "_", the NFA either does not move at all, or goes back to $n$. That is, either $\delta_1(n, \_) = \emptyset$ or $\delta_1(n, \_) = n$.

As shown in Fig. 3, the only cycles in the transition diagram of the NFA are introduced by "_", which go from a state back to itself.

The transition diagrams of $M(P)$ and $M(Q)$ can be treated as graphs in which each edge is labeled with a letter in $C \cup \{\_\}$. Given the edge-labeled graphs, we define a *simulation relation*, $\lhd$, on $N_1 \times N_2$. Similar to simulations exploited in the context of semistructured data [3], the relation $\lhd$ defines a correspondence between the nodes (or edges) in $M(P)$ and $M(Q)$ such that any string that is accepted by $M(P)$ is also accepted by $M(Q)$ according to the simulation relation. More specifically, for any $n_1 \in N_1$ and $n_2 \in N_2$, $n_1 \lhd n_2$ iff the following conditions are satisfied:

- If $n_1 = F_1$ then $n_2 = F_2$.
- If $\delta_1(n_1, \_) = n_1$ then $\delta_2(n_2, \_) = n_2$.
- For any $l \in C$, if $\delta_1(n_1, l) = n_1'$ for some $n_1' \in N_1$, then
  - either there exists a state $n_2' \in N_2$ such that $\delta_2(n_2, l) = n_2'$ and $n_1' \lhd n_2'$, or
  - $\delta_2(n_2, \_) = n_2$ and $n_1' \lhd n_2$.

The simulation is defined in such a way that showing $P \subseteq Q$ is equivalent to showing $S_1 \lhd S_2$. Intuitively, this means that starting with the start states of $M(P)$ and $M(Q)$ and a given input string, every step taken by $M(P)$ in accepting this string has a corresponding step in $M(Q)$ according to the simulation relation in accepting this string.

Having defined the simulation, we proceed to prove Theorem 3.2. For the completeness of $\mathcal{I}^p$, it suffices to show the following (see [13] for a proof):

1. $P \subseteq Q$ iff $S_1 \lhd S_2$.
2. If $S_1 \lhd S_2$, then $P \subseteq Q$ can be proved using the inference rules of $\mathcal{I}^p$.

Given $\mathcal{I}^p$ and the claims, we provide a recursive function $Incl(n_1, n_2)$ in Fig. 4 for testing inclusion of $PL$ expressions. The function assumes the existence of $M(P), M(Q)$ as described above for any $P, Q \in PL$. In addition, assume that $P$ and $Q$ are in normal form. We use $visited(n_1, n_2)$ to keep track of whether $Incl(n_1, n_2)$ has been evaluated before. Initially, $visited(n_1, n_2)$ is false for all $n_1 \in N_1$ and $n_2 \in N_2$. The function $Incl(n_1, n_2)$ returns true iff $n_1 \lhd n_2$. Since $P \subseteq Q$ iff $S_1 \lhd S_2$, $P \subseteq Q$ iff $Incl(S_1, S_2)$. Therefore, the algorithm can be used to determine whether $P \subseteq Q$.

The correctness of the algorithm immediately follows from the claims given above. It is easy to see that transforming $P$ into its normal form can be done in $O(|P|)$ time where $|P|$ is the length of $P$. The construction of $M(P)$ can also be done in $O(|P|)$ time. The same argument applies for $Q$. The initialization statement can be executed in $O(|P|\ |Q|)$ time. Since each condition of the cases 1-3 can be tested in constant time and the first statement of the algorithm ensures that any pair of states $(n_1, n_2)$ from $N_1 \times N_2$ is never processed twice, it is easy to see that $Incl(S_1, S_2)$ runs in $O(|P|\ |Q|)$ time. We can therefore conclude that the algorithm is in quadratic time. ∎

**Algorithm 3.1:** $Incl(n_1,\ n_2)$

1. if $visited(n_1, n_2)$
   then return false
   else mark $visited(n_1, n_2)$ as true;
2. process $n_1$, $n_2$ as follows:
   Case 1: if $n_1 = F_1$
   then if $n_2 = F_2$ and $(\delta_1(F_1, \_) = \emptyset$ or $\delta_2(F_2, \_) = F_2)$
   then return true;
   else return false;
   Case 2: if $\delta_1(n_1, a) = n_1'$ and $\delta_2(n_2, a) = n_2'$ for letter $a$
   and $\delta_1(n_1, \_) = \emptyset$ and $\delta_2(n_2, \_) = \emptyset$
   then return $Incl(n_1',\ n_2')$;
   Case 3: if $\delta_1(n_1, a) = n_1'$ and $\delta_2(n_2, \_) = n_2$ and
   $\delta_2(n_2, a) = n_2'$ for letter $a$
   then return $(Incl(n_1',\ n_2)$ or $Incl(n_1',\ n_2'))$
   else if $\delta_1(n_1, a) = n_1'$ and $\delta_2(n_2, \_) = n_2$
   and $\delta_2(n_2, a) = \emptyset$
   then return $Incl(n_1',\ n_2)$;
3. return false

**Figure 4**: Algorithm for testing $PL$ inclusion

| | |
|---|---|
| $\dfrac{(Q,\ S) \quad P \in PL}{(Q,\ S \cup \{P\})}$ | (superkey) |
| $\dfrac{(Q.Q',\ \{P\})}{(Q,\ \{Q'.P\})}$ | (subnodes) |
| $\dfrac{(Q,\ S \cup \{P_i, P_j\}) \quad P_i \subseteq P_j}{(Q,\ S \cup \{P_i\})}$ | (containment-reduce) |
| $\dfrac{(Q,\ S) \quad Q' \subseteq Q}{(Q',\ S)}$ | (target-path-containment) |
| $\dfrac{(Q,\ S \cup \{\epsilon, P\}) \quad P' \in PL}{(Q,\ S \cup \{\epsilon, P.P'\})}$ | (prefix-epsilon) |
| $\dfrac{S \text{ is a set of } PL \text{ expressions}}{(\epsilon,\ S)}$ | (epsilon) |

Table 2: $\mathcal{I}_{abs}$: **Rules for absolute key implication**

### 3.2 Axiomatization for absolute key implication

Recall that an absolute key $(Q',\ S)$ is a special case of a $\mathcal{K}$ constraint $(Q,\ (Q',\ S))$, i.e., when $Q = \epsilon$. As opposed to relative keys, absolute keys are constraints imposed on the entire XML tree $T$ rather than on certain subtrees of $T$. The problem of determining (finite) implication of absolute keys is simpler than that for relative keys. Since most of the rules for relative key implication is an obvious generalization of that for absolute key implication, we start by giving a discussion on the rules for absolute key implication. The set of rules, denote as $\mathcal{I}_{abs}$ is shown in Table 2.

- *superkey*. If $S$ is a key for the set of nodes in $[\![Q]\!]$ then so is any superset of $S$. This is a generalization of the superkey rule for relational database keys. It is in fact the only rule of $\mathcal{I}_{abs}$ that has a counterpart in key inference in relational databases.
- *subnodes*. Observe that any node $v \in [\![Q.Q']\!]$ must be in the subtree rooted at some node $v'$ in $[\![Q]\!]$ and

since we have a tree model, there is no sharing of nodes. Hence $v$ uniquely identifies $v'$. Therefore, if a key path $P$ uniquely identifies a node in $[\![Q.Q']\!]$ then $Q'.P$ uniquely identifies nodes in $[\![Q]\!]$.

- *containment-reduce.* If $S \cup \{P_i, P_j\}$ is the set of the keys paths that uniquely identify nodes in $[\![Q]\!]$ and $P_i \subseteq P_j$ then we can leave out $P_j$ from the set of keys paths. This is because for any nodes $n_1, n_2$ in $[\![Q]\!]$, if $n_1[\![P_i]\!] \cap_v n_2[\![P_i]\!] \neq \emptyset$, then we must have $n_1[\![P_j]\!] \cap_v n_2[\![P_j]\!] \neq \emptyset$ since $P_i \subseteq P_j$. Thus by the definition of keys $S \cup \{P_i\}$ is also a key for $[\![Q]\!]$.
- *target-path-containment.* A key for the set $[\![Q]\!]$ is also a key for any subset of $[\![Q]\!]$. Observe that $[\![Q']\!] \subseteq [\![Q]\!]$ if $Q' \subseteq Q$.
- *prefix-epsilon.* If a set $S \cup \{\epsilon, P\}$ is a key of $[\![Q]\!]$, then we can extend a key path $P$ by appending to it another path $P'$, and the modified set is also a key of $[\![Q]\!]$. This is because for any nodes $n_1, n_2 \in [\![Q]\!]$, if $n_1[\![P.P']\!] \cap_v n_2[\![P.P']\!] \neq \emptyset$ and $n_1 =_v n_2$, then we have $n_1[\![P]\!] \cap_v n_2[\![P]\!] \neq \emptyset$. Note that $n_1 =_v n_2$ if $n_1[\![\epsilon]\!] \cap_v n_2[\![\epsilon]\!] \neq \emptyset$. Thus by the definition of keys, $S \cup \{\epsilon, P.P'\}$ is also a key for $[\![Q]\!]$. Observe however that the implication of $(Q, \{\epsilon\})$ from the premise is not sound. One can construct an XML tree with only two nodes $n_1$ and $n_2$ in $[\![Q]\!]$ that are value equal but do not have any paths in $P$. Since paths of $P$ are missing in the trees of $n_1$ and $n_2$, the XML tree satisfies the premise trivially. However, this tree clearly does not satisfy $(Q, \{\epsilon\})$ since $n_1 =_v n_2$.
- *epsilon.* This rule is sound because there is only one root. In other words, $[\![\epsilon]\!]$ is exactly the root node and therefore any set of path expressions is a valid set of its key paths.

We omit the proof of the following theorem. Most of the proof can be verified along the same lines as the proof of Lemma 3.4 discussed in the next section. Details can also be found in [11].

**Theorem 3.3:** For determining (finite) implication of absolute keys of $\mathcal{K}_{abs}$,

- the set $\mathcal{I}_{abs}$ is sound and complete; and
- there is an $O(n^4)$ time algorithm, where $n$ is the length of constraints involved.

■

### 3.3 Axiomatization for key implication

We now turn to the finite implication problem for $\mathcal{K}$, and start by giving in Table 3 a set of inference rules, denoted by $\mathcal{I}$. Most rules are simply a generalization of rules shown in Table 2. The only exceptions are rules that deal with the context path in relative keys: context-path containment, context-target and interaction. We briefly illustrate these rules below.

- *context-path-containment.* Note that $[\![Q_1]\!] \subseteq [\![Q]\!]$ if $Q_1 \subseteq Q$. If $(Q', S)$ holds on all subtrees rooted at nodes in $[\![Q]\!]$, then it must also hold on all subtrees rooted at nodes in any subset of $[\![Q]\!]$.

| | |
|---|---|
| $\dfrac{(Q,\ (Q',\ S)) \quad P \in PL}{(Q,\ (Q',\ S \cup \{P\}))}$ | (superkey) |
| $\dfrac{(Q,\ (Q'.Q'',\ \{P\}))}{(Q,\ (Q',\ \{Q''.P\}))}$ | (subnodes) |
| $\dfrac{(Q,\ (Q',\ S \cup \{P_i, P_j\})) \quad P_i \subseteq P_j}{(Q,\ (Q',\ S \cup \{P_i\}))}$ | (containment-reduce) |
| $\dfrac{(Q,\ (Q',\ S)) \quad Q_1 \subseteq Q}{(Q_1,\ (Q',\ S))}$ | (context-path-containment) |
| $\dfrac{(Q,\ (Q',\ S)) \quad Q_2 \subseteq Q'}{(Q,\ (Q_2,\ S))}$ | (target-path-containment) |
| $\dfrac{(Q,\ (Q_1.Q_2,\ S))}{(Q.Q_1,\ (Q_2,\ S))}$ | (context-target) |
| $\dfrac{(Q,\ (Q',\ S \cup \{\epsilon, P\})) \quad P' \in PL}{(Q,\ (Q',\ S \cup \{\epsilon, P.P'\}))}$ | (prefix-epsilon) |
| $\dfrac{(Q_1,\ (Q_2,\ \{Q'.P_1, \ldots, Q'.P_k\})) \quad (Q_1.Q_2,\ (Q',\ \{P_1, \ldots, P_k\}))}{(Q_1,\ (Q_2.Q',\ \{P_1, \ldots, P_k\}))}$ | (interaction) |
| $\dfrac{Q \in PL,\ S \text{ is a set of } PL \text{ expressions}}{(Q,\ (\epsilon,\ S))}$ | (epsilon) |

Table 3: $\mathcal{I}$: **Inference rules for key implication**

- *context-target.* If a set $S$ of key paths can uniquely identify nodes of a set $X$ in the entire tree $T$, then it can also identify nodes of $X$ in any subtree of $T$. Along the same lines, if in a tree $T$ rooted at a node $n$ in $[\![Q]\!]$, $S$ is a key for $n[\![Q_1.Q_2]\!]$, then in any subtree of $T$ rooted at $n'$ in $n[\![Q_1]\!]$, $S$ is a key for $n'[\![Q_2]\!]$. Note that $n'[\![Q_2]\!]$ consists of nodes that are in both $n[\![Q_1.Q_2]\!]$ and the subtree rooted at $n'$. In particular, when $Q = \epsilon$ this rules says that if $(Q_1.Q_2,\ S)$ holds then so does $(Q_1,\ (Q_2,\ S))$. That is, if the (absolute) key holds on the entire document, then it must also hold on any sub-document.
- *interaction.* This is the only rule of $\mathcal{I}$ that has more than one key in its precondition. By the first key in the precondition, in each subtree rooted at a node $n$ in $[\![Q_1]\!]$, $Q'.P_1, \ldots, Q'.P_k$ uniquely identify a node in $n[\![Q_2]\!]$. The second key in the precondition prevents the existence of more than one $Q'$ nodes under $Q_2$ that coincide in their $P_1, \ldots, P_k$ nodes. Therefore, $P_1, \ldots, P_k$ uniquely identify a node in $n[\![Q_2.Q']\!]$ in each subtree rooted at $n$ in $[\![Q_1]\!]$. More formally, for any $n \in [\![Q_1]\!]$ and $n_1, n_2 \in n[\![Q_2.Q']\!]$, there must be $v_1, v_2$ in $n[\![Q_2]\!]$ such that $n_1 \in v_1[\![Q']\!]$, $n_2 \in v_2[\![Q']\!]$ and for all $i \in [1, k]$, we must have $n_1[\![P_i]\!] \subseteq v_1[\![Q'.P_i]\!]$ and $n_2[\![P_i]\!] \subseteq v_2[\![Q'.P_i]\!]$. If $n_1[\![P_i]\!] \cap_v n_2[\![P_i]\!] \neq \emptyset$, then $v_1[\![Q'.P_i]\!] \cap_v v_2[\![Q'.P_i]\!] \neq \emptyset$, for any $i \in [1, k]$. Thus by the first key in the precondition, $v_1 = v_2$. Hence $n_1, n_2 \in v_1[\![Q']\!]$ and as a result, $n_1 = n_2$ by the second key in the precondition. Therefore, $(Q_1,\ (Q_2.Q',\ \{P_1, \ldots, P_k\}))$ holds.

Observe that key inference in the XML setting relies heavily on path inclusion. That is why we need to develop inference rules for determining inclusion of $PL$ expressions.

Given a finite set $\Sigma \cup \{\varphi\}$ of $\mathcal{K}$ constraints, we use $\Sigma \vdash_{\mathcal{I}} \varphi$ to denote that $\varphi$ is provable from $\Sigma$ using $\mathcal{I}$ (and $\mathcal{I}_p$ for path inclusion).

To illustrate how $\mathcal{I}$ is used in implication proof, let us consider two $\mathcal{K}$ constraints:

$$\begin{aligned} \phi &= (A,\ (B.C.\_*,\ \{D,\ D.\_*\})), \\ \psi &= (A.B,\ (C,\ \{\_*.D,\ E\})). \end{aligned}$$

An $\mathcal{I}$-proof for $\phi \models \psi$ is given as follows.

1) $\phi \models (A,\ (B.C.\_*,\ \{D\}))$ by $D \subseteq D.\_*$ and the *containment-reduce* rule. Note that $D \subseteq D.\_*$ is proved by using *star*, *empty-path* and *composition* of $\mathcal{I}_p$.

2) $\phi \models (A,\ (B.C,\ \{\_*.D\}))$ by 1) and *subnodes*.

3) $\phi \models (A.B,\ (C,\ \{\_*.D\}))$ by 2) and *context-target*.

4) $\phi \models (A.B,\ (C,\ \{\_*.D,\ E\}))$, i.e., $\phi \models \psi$, by 3) and *superkey*.

As another example, observe that the following is provable from $\mathcal{I}$:

$$\frac{(Q,\ (Q',\ S \cup \{P\}))\quad P' \subseteq P}{(Q,\ (Q',\ S \cup \{P'\}))} \quad \text{(key-path-containment)}$$

Indeed, if $(Q,\ (Q',\ S \cup \{P\}))$ holds then by *superkey*, so does $(Q,\ (Q',\ S \cup \{P, P'\}))$. By *containment-reduce* we have that $(Q,\ (Q',\ S \cup \{P'\}))$ holds.

We next show that $\mathcal{I}$ is indeed a finite axiomatization for $\mathcal{K}$ constraint implication.

**Lemma 3.4:** The set $\mathcal{I}$ is sound and complete for finite implication of $\mathcal{K}$ constraints. That is, for any finite set $\Sigma \cup \{\varphi\}$ of $\mathcal{K}$ constraints, $\Sigma \models \varphi$ iff $\Sigma \vdash_{\mathcal{I}} \varphi$. ∎

*Proof sketch:* Soundness of $\mathcal{I}$ can be verified by induction on the lengths of $\mathcal{I}$-proofs. For the proof of completeness, we show that if $\Sigma \not\vdash_{\mathcal{I}} \varphi$, then there exists a finite XML tree $G$ such that $G \models \Sigma$ and $G \models \neg\varphi$, i.e., $\Sigma \not\models \varphi$. In other words, if $\Sigma \models \varphi$ then $\Sigma \vdash_{\mathcal{I}} \varphi$.

To do so, we introduce the notion of abstract trees. An *abstract tree* is an extension of an XML tree by allowing "$\_*$" as a node label. Observe that in an abstract tree $T$, the sequence of labels on a path is a $PL$ expression that possibly contains occurrences of "$\_*$". Let $R$ be the sequence of labels in the path from node $a$ to $b$ in $T$, and $P$ be any path expression in $PL$. We say that $T \models P(a, b)$ if $R \subseteq P$. Given this, the definitions of node sets and satisfaction of constraints in $\mathcal{K}$ can be easily generalized for abstract trees. Abstract trees have the following property (proof of the lemma can be found in [13]).

**Lemma 3.5:** Let $\Sigma \cup \{\varphi\}$ be a finite set of $\mathcal{K}$ constraints. If there is a finite abstract tree $T$ such that $T \models \Sigma$ and $T \models \neg\varphi$, then there is a finite XML tree $G$ such that $G \models \Sigma$ and $G \models \neg\varphi$. ∎

Given this, we prove completeness of $\mathcal{I}$ in two steps. Assume $\Sigma \vdash_{\mathcal{I}} \varphi$. We first define a finite abstract tree $T_f$ such that $T_f \models \Sigma$ but $T_f \not\models \varphi$. We then construct $G$ from $T_f$ following Lemma 3.5. To do this, we start with a finite abstract tree $T$ that does not satisfy $\varphi$. The abstract tree $T$ consists of a single path $Q$ from the root leading to a node $n$, which has two distinct subtrees $T_1$ and $T_2$. Each subtree has a $Q'$ path. These $Q'$ paths lead to nodes $n_1$ and $n_2$ from $n$ in $T_1$ and $T_2$, respectively. From each of $n_1$ and $n_2$ there are paths $P_1, \ldots, P_k$, as depicted in Fig. 6 (a). For each $i \in [1, k]$, let $x_i$ be the (single) node at the end of the $P_i$ path in $T_1$, and $y_i$ be the (single) node at the end of the $P_i$ path in $T_2$. We ensure that for each $i \in [1, k]$, $x_i =_v y_i$, but for any other pair $x, y$ in $T$, $x \neq_v y$. The only exception is when there is $i \in [1, k]$ such that $P_i = \epsilon$. In this case we have to assure $n_1 =_v n_2$. That is, for all $j \in [1, k]$ and for any $P'_j$ such that $P_j = P'_j.P''_j$ for some $P''_j \in PL$, we let $x'_j =_v y'_j$, where $x'_j, y'_j$ are the nodes in $n_1[\![P'_j]\!]$ and $n_2[\![P'_j]\!]$, respectively. For any other pair $x, y$ in $T$, we let $x \neq_v y$ as before. It is easy to see that $T \models \neg\varphi$.

We next modify $T$ such that $T \models \Sigma$. For each $\phi$ in $\Sigma$, if $T$ does not satisfy $\phi$, then we merge certain nodes in the tree such that the new tree satisfies $\phi$ as shown in Fig. 6 (b), (c), (d), and (e). Let $T'$ be the tree obtained after all keys in $\Sigma$ have been processed. Obviously, $T' \models \Sigma$. If $T' \models \varphi$ then we show that it is indeed the case that $\Sigma \vdash_{\mathcal{I}} \varphi$, which contradicts our assumption. Otherwise, by Lemma 3.5, there is a finite XML tree $G$ such that $G \models \Sigma$ but $G \not\models \varphi$. That is, $\Sigma \not\models \varphi$. The details of the proof can be found in [13]. ∎

Finally, we show that $\mathcal{K}$ constraint implication is decidable in PTIME.

**Lemma 3.6:** There is an algorithm that, given any finite set $\Sigma \cup \{\varphi\}$ of $\mathcal{K}$ constraints, determines whether $\Sigma \models \varphi$ in PTIME. ∎

*Proof sketch:* An algorithm for determining finite implication of $\mathcal{K}$ constraints is given in Fig. 5. The correctness of the algorithm follows from Lemma 3.4 and its proof. It applies $\mathcal{I}$ rules to derive $\varphi$ if $\Sigma \models \varphi$. To keep track of intermediate keys in the $\mathcal{I}$-proof, it uses a set variable $X$. Step 1 of the algorithm is a simple application of the *epsilon* rule. Step 2 applies *containment-reduce* to normalize the keys. Steps 4 (a) and 4 (b) prove $\varphi$ from $\Sigma$, and steps 4 (c) and 4 (d) produce intermediate results of the $\mathcal{I}$-proof and save them in $X$. Steps 4 (a) and (c) consider the case when the context path of a key in $\Sigma$ contains a prefix of $Q$. Steps 4 (b) and (d) deal with the other case: when $Q$ is contained in a prefix of the context path of a key in $\Sigma$. Each conditional statement in step 4 corresponds to applications of certain rule in $\mathcal{I}$. More specifically:

- Steps 4 (a) and (c) use three *containment* rules (i.e., *context-path-containment*, *target-path-containment* and *key-path-containment*), *context-target*, *superkey*, and *subnodes*. If it is (ii) then *prefix-epsilon* is also used.
- Steps 4 (b) and (d) apply the three *containment* rules, *superkey*, *subnodes*, and *interaction*, which need intermediate results of the $\mathcal{I}$-proof stored in $X$. If it is (ii) then *prefix-epsilon* is also used.

**Algorithm 3.2: Finite implication of $\mathcal{K}$ constraints**

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

*Output:* true iff $\Sigma \models \varphi$

*// Epsilon rule.*
1. if $Q' = \epsilon$ then output true and terminate

*// Containment-reduce rule.*
2. for each $(Q_i, (Q'_i, S_i)) \in \Sigma \cup \{\varphi\}$ do
   repeat until no further change
   if $S_i = S \cup \{P', P''\}$ such that $P' \subseteq P''$
   then $S_i := S_i \setminus \{P''\}$
3. $X := \emptyset$;
4. repeat until no keys in $\Sigma$ can be applied in cases (a)-(d).
   for each $\phi = (Q_\phi, (Q'_\phi, \{P'_1, ..., P'_m\})) \in \Sigma$ do
   *// See Fig. 6(c) for an illustration of this case.*
   (a) if there is $Q_t, R_p$ in $PL$ such that $Q \subseteq Q_\phi.Q_t$,
   $Q_t.Q'.R_p \subseteq Q'_\phi$, $R_p = \epsilon$ if $m > 1$ and
   for all $j \in [1, m]$ there is $s \in [1, k]$ such that either
   (i) $P_s \subseteq R_p.P'_j$ or (ii) there exists $l \in [1, k]$ and
   $R_j$ in $PL$ such that $P_l = \epsilon$ and $P_s \subseteq R_p.P'_j.R_j$
   then output true and terminate
   *// See Fig. 6(e) for an illustration of this case.*
   (b) if there are $Q_c, Q_t, R_p$ in $PL$ such that $Q.Q_c \subseteq Q_\phi$,
   $Q'.R_p \subseteq Q_c.Q'_\phi$, $R_p = \epsilon$ if $m > 1$, $Q' = Q_c.Q_t$ and
   for all $j \in [1, m]$ there is there is $s \in [1, k]$
   such that either (i) $P_s \subseteq R_p.P'_j$ or
   (ii) there exists $l \in [1, k]$ and $R_j$ in $PL$
   such that $P_l = \epsilon$ and $P_s \subseteq R_p.P'_j.R_j$; and moreover,
   there is $(Q,\ (Q_c,\ \{Q_t.P_1, ..., Q_t.P_k\}))$ in $X$
   then output true and terminate
   *// See Fig. 6(b) for an illustration of this case.*
   (c) if there are $Q_c, Q_t, R_p$ in $PL$ such that $Q \subseteq Q_\phi.Q_c$,
   $Q_c.Q' \subseteq Q'_\phi.R_p$, $Q' = Q_t.R_p$ and for all $j \in [1, m]$
   there is $s \in [1, k]$ such that either (i) $R_p.P_s \subseteq P'_j$
   or (ii) there exists $l \in [1, k]$ and $R_j$ in $PL$ such that
   $P_l = \epsilon$ and $R_p.P_s \subseteq P'_j.R_j$
   then
   (1) if $m = 1$ then
   $X := X \cup \{(Q, (Q_1,\ \{Q_2.R_p.P_1, \ldots, Q_2.R_p.P_k\}))\}$
   where $Q_t = Q_1.Q_2$ for some $Q_1, Q_2 \in PL$;
   (2) if $m > 1$ then
   $X := X \cup \{(Q, (Q_t,\ \{R_p.P_1, \ldots, R_p.P_k\}))\}$;
   (3) $\Sigma := \Sigma \setminus \{\phi\}$;
   *// See Fig. 6(d) for an illustration of this case.*
   (d) if there are $Q_c, Q_t, R_p$ in $PL$ such that $Q.Q_c \subseteq Q_\phi$,
   $Q' \subseteq Q_c.Q'_\phi.R_p$, $Q' = Q_c.Q_t.R_p$ and for all $j \in [1, m]$
   there is $s \in [1, k]$ such that either (i) $R_p.P_s \subseteq P'_j$
   or (ii) there exists $l \in [1, k]$ and $R_j$ in $PL$ such that
   $P_l = \epsilon$ and $R_p.P_s \subseteq P'_j.R_j$; and moreover, there is
   $(Q,\ (Q_c,\ \{Q_t.R_p.P_1, ..., Q_t.R_p.P_k\}))$ in $X$
   then
   (1) if $m = 1$ then
   $X := X \cup \{(Q, (Q_1,\ \{Q_2.R_p.P_1, \ldots, Q_2.R_p.P_k\}))\}$
   where $Q_c.Q_t = Q_1.Q_2$ for some $Q_1, Q_2 \in PL$;
   (2) if $m > 1$ then
   $X := X \cup \{(Q, (Q_c.Q_t,\ \{R_p.P_1, \ldots, R_p.P_k\}))\}$;
   (3) $\Sigma := \Sigma \setminus \{\phi\}$;
5. output false

**Figure 5**: Algorithm for implication of $\mathcal{K}$ constraints

For the interested reader, step 4 (a) corresponds to Fig. 6(c). Since nodes $n_1$ and $n_2$ are merged as a result of this key, we can prove $\varphi$. Similarly, step 4 (b) corresponds to Fig. 6(e). The difference between steps 4 (a) and (b) is whether or not the context path of the key contains a prefix of $Q$. Steps 4 (c) and (d) correspond to Fig. 6 (b) and (d) respectively. Here these keys do not prove $\varphi$ directly, but they generate intermediate results, which are saved in $X$. Again the difference is whether the context path of the key contains a prefix of $Q$.

We next show that this algorithm is in PTIME. To see this, observe that step 1 takes constant time and step 2 takes at most $O(|\Sigma|^3)$ time. For step 4, the worst scenario can happen as follows: for each key in $\Sigma$, the conditions of (a) - (d) are tested and only the last key in $\Sigma$ is removed after testing all keys in $\Sigma$. Hence, the second time the for loop is performed, one less key is tested. Therefore if there are $s$ keys in $\Sigma$, a total of $O(s^2)$ keys will be tested. We next examine the complexity of each condition of steps (a) - (d). For step (a), we need to partition $Q$ to find $Q_t$. Also, for each such $Q_t$, we need to partition $Q'_\phi$ to find $R_p$. Since containment of path expressions is tested in quadratic time, the first two inclusion tests cost at most $|Q| * (|\varphi| * (|\phi| + |\varphi|) + |Q'_\phi| * ((|\varphi| + |\phi|) * |\phi|))$, which is $O(n^4)$ in total, where $n$ is the size of keys. Then for each key path $P'_j$ in $\phi$, we check if there is a key path $P_s$ in $\varphi$ and partition $P_s$ to get $R_j$ such that case (i) or (ii) is satisfied. This costs $|P_s| * (|R_p| + |P_s|) + |P_s| * |P_s| * (|R_p| + |P'_j| + |R_j|)$. Since there are $m$ key paths in $\phi$, for all $k$ key paths in $\varphi$ these tests cost $(|P_1| + \ldots + |P_k|) * (m * |R_p| + |P'_1| + \ldots + |P'_m|) + (|P_1| * |P_1| + \ldots |P_k| * |P_k|)(m * |R_p| + |P'_1| + \ldots + |P'_m| + m * |R_j|)$. There are $|Q| * |Q'_\phi|$ possible expressions for $Q_t$, and $R_p$. Therefore, the cost of step (a) is at most $|Q| * |Q'_\phi| * |\varphi| * ((m * |\phi| + \phi) + |\varphi|^2 * (m * |\phi| + |\phi| + m * |\varphi|))$, which is $O(n^6)$. It is easy to see that the steps (b), (c), and (d) involve at most the same cost. Hence the complexity is $O(n^6)$. Since these tests are performed $O(s^2)$ times, the overall cost of the algorithm is $O(n^8)$, and therefore we have a PTIME algorithm. ■

Theorem 3.1 follows from Lemmas 3.4 and 3.6.

## 4 Discussion

We have investigated a key constraint language introduced in [12] for XML data and studied the associated (finite) satisfiability and (finite) implication problems in the absence of DTDs. These keys are capable of expressing many important properties of XML data; moreover, in contrast to other proposals, this language can be reasoned about efficiently. More specifically, keys defined in this language are always finitely satisfiable, and their (finite) implication is finitely axiomatizable and decidable in PTIME in the size of keys. We believe that these key constraints are simple yet expressive enough to be adopted by XML designers and maintained by systems for XML applications.

For further research, a number of issues deserve investigation. First, despite their simple syntax, there is an interaction between DTDs and our key constraints. To illustrate this, let us consider a simple DTD $D$:

```
<!ELEMENT  foo  (X, X)>
```

and a simple (absolute) key $\varphi = (X, \emptyset)$. Obviously, there exists a finite XML tree that conforms to the DTD

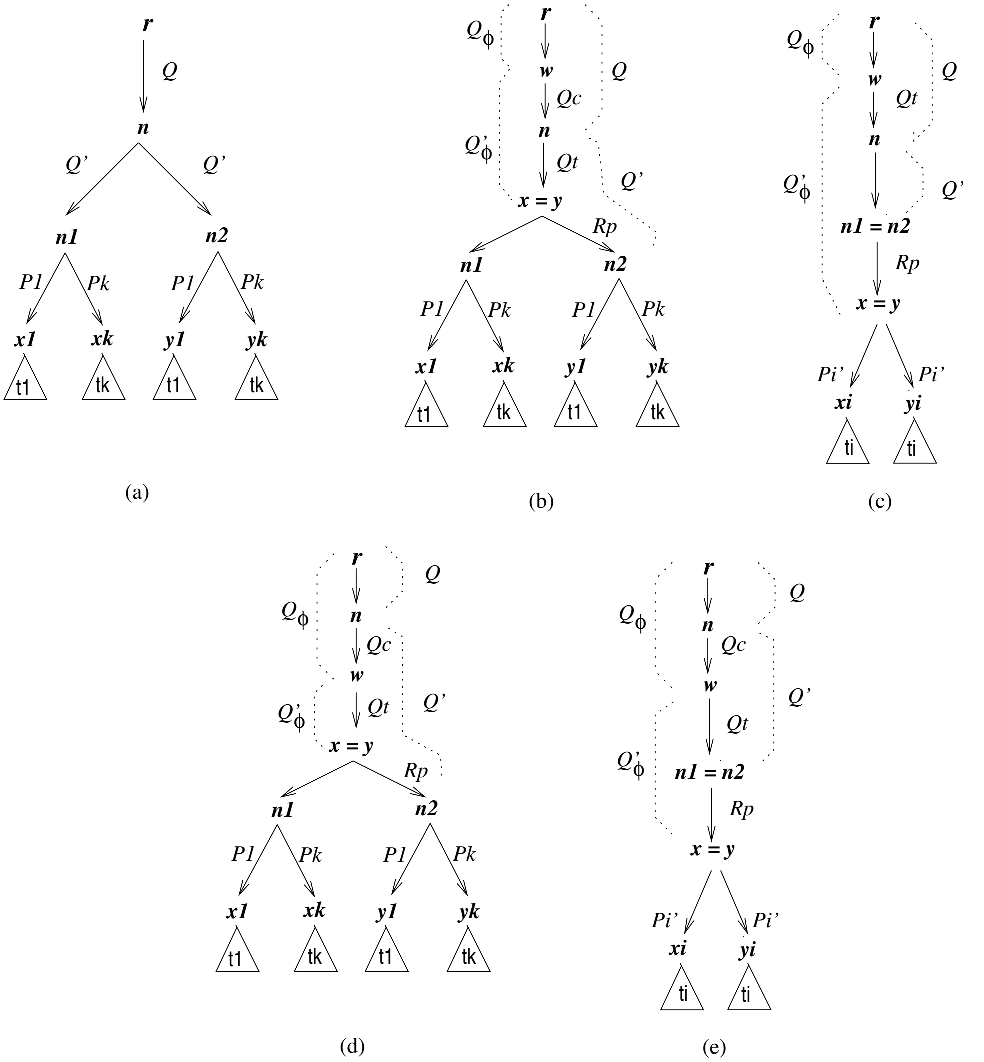


**Figure 6**: Abstract trees constructed in the proof of Lemma 3.4

$D$ (see, e.g., Fig. 7 (a)), and there exists a finite XML tree that satisfies the key $\varphi$ (e.g., Fig. 7 (b)). However, there is no XML tree that both conforms to $D$ and satisfies $\varphi$. This is because $D$ requires an XML tree to have two distinct $X$ elements, whereas $\varphi$ imposes the following restriction: the path $X$, if it exists, must be unique at the root. This shows that in the presence of DTDs, the analysis of key satisfiability and implication can be wildly different. It should be mentioned that keys defined in other proposals for XML, such as those introduced in XML Schema [33], also interact with DTDs or other type systems for XML. This issue was recently investigated in [21] for a class of keys and foreign keys defined in terms of XML attributes.

Second, as mentioned in Section 2 there are more general definitions of key constraints that should be considered.

Third, one might be interested in using different path languages to express keys. The containment and equivalence problems for the full regular language are PSPACE-complete [23], and as mentioned earlier, these are not finitely axiomatizable. Also, for star-free languages in general, this is co-NP complete. Another alternative is to adopt the language of [29], which simply adds a single wildcard to $PL$. Despite the seemingly trivial addition, containment of expressions in their language is only known to be in PTIME. It would be interesting to develop an algorithm for determining containment of expressions in this language with a complexity comparable to the related result established in this paper. For XPath [19] expressions, questions in connection with their containment and equivalence, as well as (finite) satisfiability and (finite) implication of keys defined in terms of these complex path expressions are, to the best of our knowledge, still open. Observe however that regardless of the choice of path language we use for expressing keys, the finite implication problem is decidable as long as the containment problem for our path language is decidable.

Fourth, along the same lines as our XML key language, a language of foreign keys needs to be developed for XML. As shown by [22, 21], the implication and finite implication problems for a class of keys and foreign keys defined in terms of XML attributes are undecid-

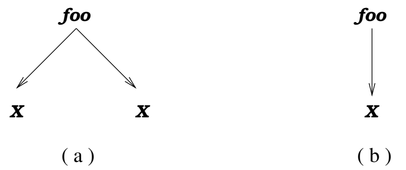


**Figure 7**: Interaction between DTDs and XML keys

able, in the presence or absence of DTDs. However, under certain practical restrictions, these problems are decidable in PTIME. Whether these decidability results still hold for more complex keys and foreign keys needs further investigation.

A final question is about key constraint checking. An efficient incremental checking algorithm for our keys is currently under development.

**Acknowledgments.** The authors thank Michael Benedikt, Chris Brew, Dave Maier, Keishi Tajima and Henry Thompson for helpful discussions. They would also like to thank one of the referees for pointing out the possible need for a more general definition of a key constraint (Section 2.)

## References

[1] http://www.geml.org.

[2] http://www.oasis-open.org/cover/maml.html.

[3] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web. From Relations to Semistructured Data and XML*. Morgan Kaufman, 2000.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[5] S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 122–133, Tucson, Arizona, May 1997.

[6] V. Apparao et al. *Document Object Model (DOM) Level 1 Specification*. W3C Recommendation, Oct. 1998.
`http://www.w3.org/TR/REC-DOM-Level-1/`.

[7] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL. *Nucleic Acids Research*, 28:45–48, 2000.

[8] W. Baker et al. The EMBL nucleotide sequence database. *Nucleic Acids Research*, 28:19–23, 2000.

[9] D. Benson et al. GenBank. *Nucleic Acids Research*, 28:15–18, 2000.

[10] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium (W3C), Feb 1998.
`http://www.w3.org/TR/REC-xml`.

[11] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about keys for XML. Technical Report MS-CIS-00-26, University of Pennsylvania, Sept. 2000.

[12] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *Proceedings of the 10th International World Wide Web Conference (WWW'10)*, 2001.

[13] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about absolute and relative keys for XML. Technical Report TUCIS-TR-2001-002, Temple University, 2001.

[14] P. Buneman, W. Fan, J. Siméon, and S. Weinstein. Constraints for semistructured data and XML. *SIGMOD Record*, 30(1), 2001.

[15] P. Buneman, W. Fan, and S. Weinstein. Path constraints on semistructured and structured data. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 129–138, Seattle, Washington, June 1998.

[16] P. Buneman, W. Fan, and S. Weinstein. Interaction between path and type constraints. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 56–67, Philadelphia, Pennsylvania, May 1999.

[17] P. Buneman, W. Fan, and S. Weinstein. Path constraints in semistructured databases. *Journal of Computer and System Sciences (JCSS)*, (61):146–193, 2000.

[18] J. Clark. *XSL Transformations (XSLT)*. W3C Recommendation, Nov. 1999.
`http://www.w3.org/TR/xslt`.

[19] J. Clark and S. DeRose. *XML Path Language (XPath)*. W3C Working Draft, Nov. 1999.
`http://www.w3.org/TR/xpath`.

[20] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[21] W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 114–125, Santa Barbara, California, May 2001.

[22] W. Fan and J. Siméon. Integrity constraints for XML. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 23–34, Dallas, Texas, May 2000.

[23] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[24] C. S. Hara and S. B. Davidson. Reasoning about nested functional dependencies. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 91–100, Philadelphia, Pennsylvania, May 1999.

[25] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addision Wesley, 1979.

[26] H. Hunt, D. Resenkrantz, and T. Szymanski. On the equivalence, containment, and covering problems for the regular and context-free languages. *Journal of Computer and System Sciences (JCSS)*, (12):222–268, 1976.

[27] M. Ito and G. E. Weddell. Implication problems for functional constraints on databases supporting complex objects. *Journal of Computer and System Sciences (JCSS)*, 50(1):165–187, 1995.

[28] A. Layman et al. *XML-Data*. W3C Note, Jan. 1998.
`http://www.w3.org/TR/1998/NOTE-XML-data`.

[29] T. Milo and D. Suciu. Index structures for path expressions. *Proceedings of the International Conference on Database Theory*, 1999.

[30] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.

[31] J. Robie, J. Lapp, and D. Schach. *XML Query Language (XQL)*. Workshop on XML Query Languages, Dec. 1998.

[32] J. Sulston et al. The C. elegans genome sequencing project: A beginning. *Nature*, 356(6364):37–41, 1992.

[33] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures*. W3C Working Draft, Apr. 2000.
`http://www.w3.org/TR/xmlschema-1/`.

[34] M. F. van Bommel and G. E. Weddell. Reasoning about equations and functional dependencies on complex objects. *IEEE Transactions on Data and Knowledge Engineering*, 6(3):455–469, 1994.

[35] V. Vianu. A Web odyssey: From Codd to XML. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 1–15, Santa Barbara, California, May 2001.

[36] P. Wadler. A Formal Semantics for Patterns in XSL. Technical report, Computing Sciences Research Center, Bell Labs, Lucent Technologies, 2000.

[37] S. Yu. Regular languages. In G. Rosenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, pages 41–110. Springer, 1996.