

REASONING ABOUT KEYS FOR XML

PETER BUNEMAN¹, SUSAN DAVIDSON², WENFEI FAN³,
CARMEM HARA⁴ and WANG-CHIEW TAN⁵¹ University of Edinburgh² University of Pennsylvania³ Bell Laboratories⁴ Universidade Federal do Parana⁵ University of California, Santa Cruz

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, we show that these keys are always (finitely) satisfiable, and their (finite) implication problem is finitely axiomatizable. Furthermore, we provide a polynomial time algorithm for determining (finite) implication in the size of keys. Our results also demonstrate, among other things, that the analysis of XML keys is far more intricate than its relational counterpart.

Key words: XML keys, axiomatization, logical implication, satisfiability

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. They are also an important class of constraints on the validity of data. In particular, keys – as opposed to addresses or internal object identifiers – 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 [11], XML Data [28], and XML Schema [38]. However, existing proposals cannot handle one or more of the following situations. First, as in databases, one may want to define keys with compound 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. We will call a key such as **number** a *relative key* of **chapter** since it is a key in the context of a given **book** element, i.e., it is to hold on the sub-document rooted at the book element. If the context element is the root of the document, the key is to hold on the entire document, and is referred to as an *absolute key*. The idea of keys having a scope is not new. In relational databases, scoped keys exist 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 a certain book. 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 have recently [13] provided a definition of keys for XML which overcomes the limitations above and which appears to be applicable – among other things – to a wide variety of hierarchical scientific data sets. However, reasoning about keys, which is trivial in a relational setting, becomes significantly more complicated in a hierarchical setting. In [12] we described a set of rules for key implication and outlined soundness and completeness

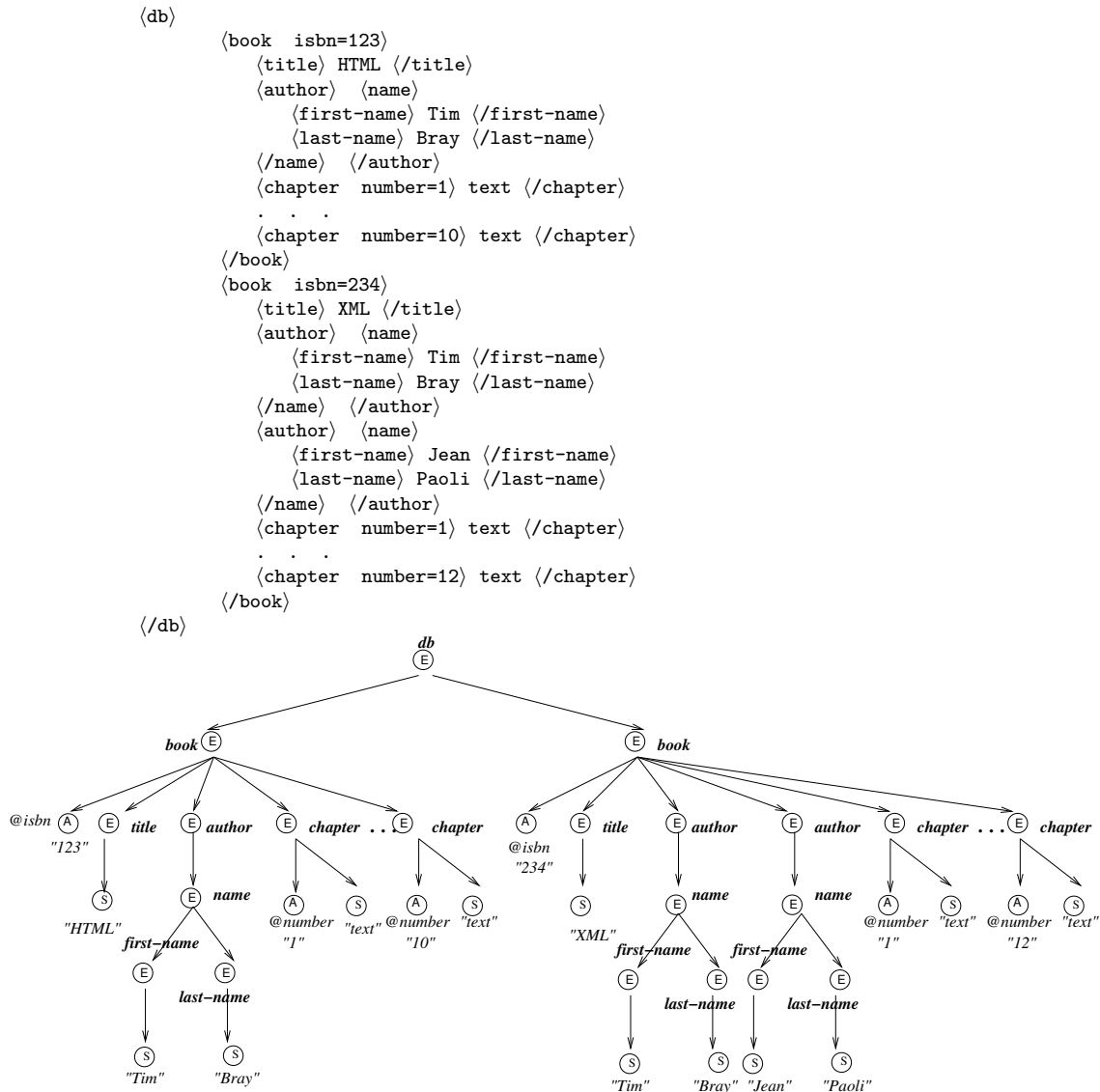


Fig. 1: Example of some XML data and its representation as a tree

results. This paper is a full version of that conference presentation and includes motivation and full proofs. The authors understand [37] that these definitions and results have significantly influenced the design of keys in the current version of XML Schema [38].

In developing our notion of keys for XML, we start with a tree model of data as used in DOM [4], XSL [17, 41], XQL [35] and XML Schema [38]. An example of this representation for some XML data is shown in Figure 1. Nodes are annotated by their type: *E* for element, *A* for attribute, and *S* for string (or PCDATA). Some 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, checking 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 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 [13] 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 represent and 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 [10] and AceDB [36]; Others, such as EMBL, which is used in SwissProt [8], are domain-specific. 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.

All the non-XML data formats mentioned above have an easy conversion to XML. We have also found that the data sets in these formats have a natural key structure. However, it is *not* the case that it is easy to find a natural non-trivial DTD or XML Schema description for the converted data. Finding an appropriate DTD for a given data format may be problematic, and even if one exists, the conversion to XML conforming to that DTD may be a more complex task. Indeed, in many applications XML data does not come with a DTD or schema. This observation supports our claim that, in some applications, keys should be treated independently of any other type constraints. It is worth mentioning that there has been recent work [5, 22] on the analyses of XML keys in the presence of DTDs.

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 cost 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 a finite set of such inference rules, referred to as *finite axiomatizability*, is a stronger property than the existence of an algorithm, because the former implies the latter but not the other way around [2]. Another interesting question is whether a set of keys is “reasonable”, that is, if there exists some (finite) document that satisfies the key specification. This is referred to as the (*finite*) *satisfiability* problem.

In relational databases, 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 (finite) implication problem for keys and, more generally, functional dependencies is also straightforward. It has been shown [2, 34] that the problem is decidable in linear time, and there are exactly two inference rules that 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:

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

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 [18] 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, 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 [18], which is a limited yet complicated language. Recently, it has been shown [29, 32] that it is rather expensive to determine containment of XPath expressions, which is important in the implication analysis of XML keys. Indeed, the containment problem is undecidable in the presence of disjunction, DTDs, and variables [32], and it is coNP-complete even for a small fragment of XPath in the absence of DTDs [29]. For the (finite) axiomatizability of equivalence of XPath expressions, which is important in studying the (finite) axiomatizability of XML key implication, the analysis is even more intriguing [9]. Thus, not surprisingly, reasoning about keys defined in XML Schema is prohibitively expensive: Even for unary keys, i.e., keys defined in terms of a single subelement, the finite satisfiability problem is NP-hard and the implication problem is coNP-hard [6]. For the entire class of keys of XML Schema, to the best of our knowledge, both the implication and axiomatizability problems are still open. The reason that the implication problem in XML Schema is different from that addressed in this paper is twofold. First, there are bad interactions between DTDs and keys, as observed in [5, 22]. Second, XML Schema imposes the additional constraint on keys that they exist and be unique. These are the “strong keys” described in [13].

In contrast, we show in this paper that the basic keys of [13] 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 [5, 22, 21]. For a class of keys and foreign keys, the decision problems were studied in the absence [21] and presence [5, 22] of DTDs. The keys considered there are defined in terms of XML attributes and are not as expressive as keys studied in this paper. Integrity constraints defined in terms of navigation paths have been studied for semistructured [1] and XML data in [3, 15, 16]. These constraints are generalizations of inclusion dependencies commonly found in relational databases, and are not capable of expressing keys. Generalizations of functional dependencies (FDs) have also been studied. In the context of object-oriented data models, these generalizations include a definition of FDs in terms of navigation paths [27, 39]; in the context of nested relational models, FDs have been defined in terms of equality on set types [33] and within the scope of nested sets [24]. However, these constraints 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, 40].

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.

2. KEYS

Our notion of keys is based on a tree model of XML data, as illustrated in Figure 1. 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 $\{\mathbf{S}\}$ denoting

text (PCDATA).

Definition 1 An XML 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 $V \rightarrow \mathbf{E} \cup \mathbf{A} \cup \{\mathbf{S}\}$ 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 $lab(v) = \mathbf{S}$; (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 *list* 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' ;
- 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, Figure 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 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. Formally:

Definition 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) if $att(n_1) = \{a_1, \dots, a_m\}$ then $att(n_2) = \{a'_1, \dots, a'_m\}$ and for all a_i there exists a'_j , $i, j \in [1, m]$, such that $a_i =_v a'_j$; and 2) if $ele(n_1) = [v_1, \dots, v_k]$, then $ele(n_2) = [v'_1, \dots, v'_k]$ and for all $i \in [1, k]$, $v_i =_v v'_i$. Here, $[v_1, \dots, v_k]$ denotes a list of nodes v_1, \dots, v_k .

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 Figure 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 XPath. However, to develop inference rules for keys, we need to be able to reason about inclusion of path expressions, the so called *containment* problem. It is well known that for regular languages, the containment problem is not finitely axiomatizable [26]; and for XPath, preliminary work [9] has shown that it is not much easier. We therefore restrict our attention to the path language PL , which is expressive enough to capture most practical cases, yet simple enough to be reasoned about efficiently. We will also use a simpler language (PL_s) in defining keys, and therefore show both 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) list of node labels. Here, ϵ represents the empty path, node label $l \in \mathbf{E} \cup \mathbf{A} \cup \{\mathbf{S}\}$, and “.” is a binary operator that concatenates two path expressions. The

language PL_s describes the class of finite lists 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 list 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 or in PL is used to describe a set of paths in an XML tree T . Recall that an attribute node or a text node is a leaf in T and it does not have any child. Thus, a path ρ in PL_s is said to be *valid* if for any label l in ρ , if $l \in \mathbf{A}$ or $l = \mathbf{S}$, then l is the last symbol in ρ . 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 contains 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 . Note that although we have presented the language PL using the syntax of regular expressions, there is an easy conversion from a PL expression to an XPath expression: We replace “_*” of a PL expression with “/”, and “.” with “/”. In addition, if a PL path is meant to start from the root, the converted path is preceded with the symbol “/”.

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

Length. The *length* of path ρ , denoted by $|\rho|$, is the number of labels in ρ , where 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 ρ 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 ρ , denoted by $T \models \rho(n_1, n_2)$, iff (1) $n_1 = n_2$ if $\rho = \epsilon$, and (2) 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 Figure 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 are reachable from n by following path expression P . That is, $n[[P]] = \{n' \mid T \models P(n, n')\}$. We shall use $[[P]]$ as an abbreviation for $r[[P]]$, where r is the root node of T . For example, let n be the first **book** element in Figure 1. 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 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:

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

Thus $n_1[[P]] \cap_v n_2[[P]]$ consists of node pairs that are value equal and are reachable by following path expression 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 Figure 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 4 A *key constraint* φ for XML is an expression of the form

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

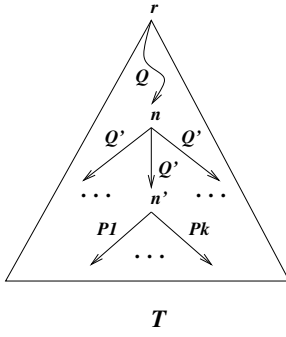


Fig. 2: Illustration of a key $(Q, (Q', \{P_1, \dots, P_k\}))$

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, \dots, P_k are called the *key paths* of φ .

When $Q = \epsilon$, we call φ an *absolute key*, and abbreviate the key to $(Q', \{P_1, \dots, P_k\})$; otherwise φ 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} .

As illustrated in Figure 2, a key $\varphi = (Q, (Q', \{P_1, \dots, P_k\}))$ specifies the following:

- the context path Q , starting from the root of an XML tree T , identifies a set of nodes $\llbracket Q \rrbracket$;
- for each node $n \in \llbracket Q \rrbracket$, φ defines an absolute key $(Q', \{P_1, \dots, P_k\})$ 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, \dots, 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.

For example, the keys on Figure 1 mentioned in Section 1 can be written as follows:

1. `@isbn` is a key of `book` nodes: $(book, \{\text{@isbn}\})$;
2. `name` is a key of `author` nodes no matter where they are: $(-* .author, \{name\})$;
3. within each subtree rooted at a `book`, `@number` is a key of `chapter`: $(book, (chapter, \{\text{@number}\}))$.

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

Definition 5 Let $\varphi = (Q, (Q', \{P_1, \dots, P_k\}))$ be a key of \mathcal{K} . An XML tree T *satisfies* φ , denoted by $T \models \varphi$, iff for any n in $\llbracket Q \rrbracket$ and any n_1, n_2 in $n[Q']$, if for all $i \in [1, k]$ there exist nodes $x \in n_1[P_i]$, $y \in n_2[P_i]$ such that $x =_v y$, then $n_1 = n_2$. That is,

$$\forall n \in \llbracket Q \rrbracket \quad \forall n_1 n_2 \in n[Q'] \quad \left(\left(\bigwedge_{1 \leq i \leq k} n_1[P_i] \cap_v n_2[P_i] \neq \emptyset \right) \rightarrow n_1 = n_2 \right).$$

As mentioned earlier, the key φ defines an absolute key on the subtree rooted at each node n in $\llbracket Q \rrbracket$. 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 $n \in \llbracket Q \rrbracket$ and for any distinct nodes n_1, n_2 in $n[Q']$, there must exist some P_i , $1 \leq i \leq k$, 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 φ is an absolute key, the set $\llbracket Q \rrbracket$ consists of a unique node, namely, the root of the tree. In this case $T \models \varphi$ iff

$$\forall n_1 n_2 \in \llbracket Q' \rrbracket \left(\left(\bigwedge_{1 \leq i \leq k} n_1 \llbracket P_i \rrbracket \cap_v n_2 \llbracket P_i \rrbracket \neq \emptyset \right) \rightarrow n_1 = n_2 \right).$$

As an example, let us consider the keys defined earlier on the XML tree T in Figure 1.

1) $T \models (\text{book}, \{\text{@isbn}\})$ because the `@isbn` attributes of the two `book` nodes in T have different string values. For the same reason, $T \models (\text{book}, \{\text{@isbn}, \text{author}\})$. However, $T \not\models (\text{book}, \{\text{author}\})$ 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) $T \not\models (-*.author, \{\text{name}\})$ 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, $T \models (\text{book}, (\text{author}, \{\text{name}\}))$ because under each `book` node, the same author does not appear twice.

3) $T \models (\text{book}, (\text{chapter}, \{\text{@number}\}))$ because in the subtree rooted at each `book` node, the `@number` attribute of each `chapter` has a distinct value. However, observe that $T \not\models (\text{book.chapter}, \{\text{@number}\})$ 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 $\psi = (\text{book}, \{\text{@isbn}, \text{author}\})$ on the XML tree T in Figure 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 $n \in \llbracket Q \rrbracket$ and n_1, n_2 in $n \llbracket Q' \rrbracket$, if P_i is missing at either n_1 or n_2 then $n_1 \llbracket P_i \rrbracket$ and $n_2 \llbracket P_i \rrbracket$ 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.

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 Section 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 Σ 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* Σ . That is, for any $\psi \in \Sigma$, $T \models \psi$.

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

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

Observation. Any finite set Σ of keys in \mathcal{K} is finitely satisfiable.

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 Σ *implies* φ , 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 Σ *finitely implies* φ , that is, whether 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 and as a result:

Proposition 1 *The implication and finite implication problems for keys coincide.*

Proof. Observe that given any finite set $\Sigma \cup \{\varphi\}$ of \mathcal{K} constraints, $\Sigma \models \varphi$ iff no XML tree T exists such that $T \models \bigwedge \Sigma \wedge \neg\varphi$. Thus it suffices to show that if there exists 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, the complement of the implication problem for \mathcal{K} has the finite model property [20]. This can be verified as follows. Let $\varphi = (Q, (Q', \{P_1, \dots, P_k\}))$. Since $T \not\models \varphi$, there are nodes $n \in \llbracket Q \rrbracket$, $n_1, n_2 \in n \llbracket Q' \rrbracket$, $x_i \in n_1 \llbracket P_i \rrbracket$ and $y_i \in n_2 \llbracket P_i \rrbracket$ for all $i \in [1, k]$ such that $x_i =_v y_i$ but $n_1 \neq n_2$. Let T' be the finite subtree of T that consists solely of all the nodes in the paths from root to x_i, y_i for all $i \in [1, k]$. It is easy to verify that $T' \models \Sigma$ and $T' \models \neg\varphi$. Clearly, T' is a finite XML tree. \square

In light of Proposition 1, we can also use $\Sigma \models \varphi$ to denote that Σ finitely implies φ . 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 2 *The finite implication problem for \mathcal{K} is finitely axiomatizable and decidable in polynomial time in the size of keys.*

We prove this theorem by showing that there is a finite axiomatization (see Lemma 6) and an algorithm for determining finite implication of \mathcal{K} constraints (see Lemma 7). A road map for the proof of this theorem is as follows. Since our axioms for finite implication for \mathcal{K} rely on path containment, we shall first study the containment of path expressions for the language PL in Section 3.1. We then provide a finite set of inference rules and show that it is sound and complete for finite implication of \mathcal{K} constraints in Section 3.3. Based on the inference rules, we also develop a polynomial time algorithm for determining finite implication. We shall also present complexity results in connection with finite implication of absolute keys of \mathcal{K}_{abs} in Section 3.2.

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 , $n \llbracket P \rrbracket \subseteq n \llbracket Q \rrbracket$. That is, the nodes reached from n by following P are contained in the set of nodes reached by following Q from n . We write $P = Q$ if $P \subseteq Q$ and $Q \subseteq P$.

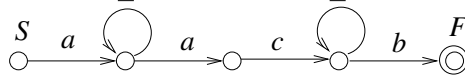
In the absence of DTDs, $P \subseteq Q$ is equivalent to the containment of the regular language defined by P in the regular language defined by Q . Indeed, if there exists a path ρ such that $\rho \in P$ but $\rho \notin Q$, then one can construct an XML tree T with a path ρ from the root. It is obvious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

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 2, and it is decidable with low complexity.

We provide in Table 1 a set of inference rules, denoted by \mathcal{I}_p , and to develop a quadratic time algorithm for testing inclusion of PL expressions.

Theorem 3 *Given two PL expressions P and Q , \mathcal{I}_p is a set of sound and complete inference rules for determining whether $P \subseteq Q$. Moreover, there is a quadratic time algorithm in the size of P and Q for determining whether $P \subseteq Q$.*

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

Table 1: \mathcal{I}_p : rules for PL expression inclusionFig. 3: NFA for the PL expression $a._*.a.c._*.b$

Proof. The soundness of \mathcal{I}_p is easily verified by induction on the lengths of \mathcal{I}_p -proofs. The proof of completeness of \mathcal{I}_p is more involved. Our goal is to establish that if $P \subseteq Q$, then this can be proved by applying rules of \mathcal{I}_p . A road map for the proof is as follows: First, nondeterministic finite state automata $M(P)$ and $M(Q)$ are defined for P and Q , respectively. Then, we show that there exists a *simulation relation* (see Definition 6) between the start states of $M(Q)$ and $M(Q)$ if and only if $P \subseteq Q$ (see Lemma 1). Finally, we show that the existence of such relation can be established using the rules of \mathcal{I}_p (see Lemma 2), thus completing the proof.

A quadratic time algorithm that determines whether a PL expression P is contained in a PL expression Q is given in Figure 1. The correctness and the complexity analysis of the algorithm is shown in Lemma 3. \square

Next, we develop the lemmas used in establishing Theorem 3. To simplify discussion, we assume that a PL expression P is in *normal form*. A PL expression P is in *normal form* if it does not contain consecutive $_*$'s and it does not contain ϵ unless $P = \epsilon$. It is easy to see that given any PL expression P , P can be rewritten into its normal form in linear time as stated by the following proposition.

Proposition 4 *Given any PL expression P , it takes linear time to transform P to an equivalent PL expression in the normal form.*

Proof. It is easy to see that $_*. _*$ can be reduced to $_*$ using the *star* and *composition* rules of \mathcal{I}_p . Moreover, by the *empty-path* rule, we can also assume that P does not contain ϵ unless $P = \epsilon$. Obviously, a single pass over P can transform P into its normal form. \square

The proofs of Lemmas 1, 2, and 3 rely on an underlying construction called a *simulation relation*. Given two PL expressions P and Q , a simulation relation is defined on the transition diagrams of the nondeterministic finite state automata (NFAs) [25] associated with P and Q . We first describe the NFA associated with a PL expression and then define the simulation.

Let P and Q be path expressions in PL , and 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, δ_1, δ_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 . Observe also that the transition diagram of a PL expression is always a NFA that has a “linear” structure as depicted in Figure 3. More specifically, $M(P)$ has the following properties (similarly for $M(Q)$):

- There is a single final state F_1 .
- 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' in 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 $\delta_1(n, l) = \{n'\}$.
- 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 Figure 3, the only cycles in the transition diagram of the NFA are introduced by “ $_$ ”, which go from a state back to itself.

Given $M(P)$ and $M(Q)$, we can define a *simulation relation*, \triangleleft , on $N_1 \times N_2$. Similar to simulations used in the context of semistructured data [1], the relation \triangleleft defines a correspondence between the nodes (or edges) in $M(P)$ and $M(Q)$. Intuitively, the relation \triangleleft is defined in such a way that given an input string, every step taken by $M(P)$ in accepting this string has a corresponding step in $M(Q)$ according to the simulation relation.

Definition 6 Let $M(P)$ and $M(Q)$ be NFAs defined for path expressions P and Q , respectively. Then, for any $n_1 \in N_1$ and $n_2 \in N_2$, there is a *simulation relation* $n_1 \triangleleft n_2$ if all of 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 $n'_1 \triangleleft n'_2$, or
 - $\delta_2(n_2, _) = n_2$ and $n'_1 \triangleleft n_2$.

As Lemma 1 will show, given two PL expressions P and Q , showing that $P \subseteq Q$ is equivalent to showing that there is a simulation relation $S_1 \triangleleft S_2$. Observe that by the definition of the simulation relation, the final state F_1 of $M(P)$ can only correspond to the final state F_2 of $M(Q)$. Therefore, if $S_1 \triangleleft S_2$ and every transition in $M(P)$ corresponds to a transition in $M(Q)$, whenever $M(P)$ accepts an input string, $M(Q)$ also does, and thus $P \subseteq Q$.

Lemma 1 *Let P and Q be two PL expressions and $M(P) = \{N_1, C \cup \{_\}, \delta_1, S_1, F_1\}$ and $M(Q) = \{N_2, C \cup \{_\}, \delta_2, S_2, F_2\}$ be their respective NFAs. Then, $P \subseteq Q$ if and only if $S_1 \triangleleft S_2$.*

Proof. The proof makes use of the closure of a transition function δ as defined in [25]:

$$\begin{aligned} \hat{\delta}(n, \epsilon) &= \{n\} \\ \hat{\delta}(n, w.l) &= \{p \mid \exists x \in \hat{\delta}(n, w), p \in \delta(x, l)\} \end{aligned}$$

Let $\hat{\delta}_1$ and $\hat{\delta}_2$ be the closure functions of δ_1 and δ_2 , respectively. Observe that $P \subseteq Q$ if and only if for any $\rho \in P$, if $F_1 \in \hat{\delta}_1(S_1, \rho)$ then $F_2 \in \hat{\delta}_2(S_2, \rho)$. Using this observation, we show the lemma as follows. Assume $S_1 \triangleleft S_2$. We first show that if $n_1 \in \hat{\delta}_1(S_1, \rho)$ where ρ is a path in P then there exists $n_2 \in \hat{\delta}_2(S_2, \rho)$ such that $n_1 \triangleleft n_2$. This can be shown by induction on the length of ρ , denoted by $|\rho|$. For the base case, if $\rho = \epsilon$ then by the definition of $\hat{\delta}$, $\hat{\delta}_1(S_1, \epsilon) = \{S_1\}$, and $\hat{\delta}_2(S_2, \epsilon) = \{S_2\}$, and $S_1 \triangleleft S_2$ by assumption. We now assume that the statement is true when $|\rho| < k$ and we shall show that the statement is also true when $|\rho| = k$. Assume $\rho \in P$, $|\rho| > 0$ and let $\rho = \rho'.l$ where $l \in C$. Let $n'_1 \in \hat{\delta}_1(S_1, \rho')$ and by induction hypothesis, there exists $n'_2 \in \hat{\delta}_2(S_2, \rho')$ such that $n'_1 \triangleleft n'_2$. Since $\rho \in P$, ρ is accepted by $M(P)$. Therefore, the last transition taken by $M(P)$ on l from n'_1 to the final state can be one of the following cases:

- l is consumed by a “ $_$ ” transition from n'_1 . More precisely, $\delta_1(n'_1, _) = n'_1$ and by the definition of \triangleleft , it must be that $\delta_2(n'_2, _) = n'_2$. Hence $n'_1 = F_1$ which implies that $n'_2 = F_2$.
- l is consumed by a “ l ” transition from n'_1 . More precisely, $\delta_1(n'_1, l) = F_1$ and by the definition of \triangleleft , either
 - for some state $n''_2 \in N_2$, $\delta_2(n'_2, l) = n''_2$ and $F_1 \triangleleft n''_2$ which implies that $n''_2 = F_2$ or
 - $\delta_2(n'_2, _) = n'_2$ and $F_1 \triangleleft n'_2$ which implies that $n'_2 = F_2$.

Thus, if $F_1 \in \hat{\delta}_1(S_1, \rho)$ then we have $F_2 \in \hat{\delta}_2(S_2, \rho)$. That is, $P \subseteq Q$.

For the other direction, we assume $P \subseteq Q$. Our goal is to show that for any path ρ , if $n_1 \in \hat{\delta}_1(S_1, \rho)$ then there exists $n_2 \in \hat{\delta}_2(S_2, \rho)$ such that $n_1 \triangleleft n_2$. To see this, note that for any $\rho \in P$, we have $F_1 \in \hat{\delta}_1(S_1, \rho)$, and since $P \subseteq Q$, $F_2 \in \hat{\delta}_2(S_2, \rho)$. Thus we can define $F_1 \triangleleft F_2$. In addition, for any path ρ , if $\hat{\delta}_1(S_1, \rho) \subseteq N_1$, then there exists path ρ' such that $F_1 \in \hat{\delta}_1(S_1, \rho, \rho')$. Thus the statement can be easily verified by contradiction. Observe that $\hat{\delta}_1(S_1, \epsilon) = \{S_1\}$ and $\hat{\delta}_2(S_2, \epsilon) = \{S_2\}$. Thus $S_1 \triangleleft S_2$ and the lemma follows. \square

The next lemma establishes the relationship between the existence of a simulation relation such that $S_1 \triangleleft S_2$ and the rules of \mathcal{I}_p .

Lemma 2 *Let P and Q be two PL expressions and $M(P) = \{N_1, C \cup \{_\}, \delta_1, S_1, F_1\}$ and $M(Q) = \{N_2, C \cup \{_\}, \delta_2, S_2, F_2\}$ be their respective NFAs. If $S_1 \triangleleft S_2$, then $P \subseteq Q$ can be proven using the inferences rules of \mathcal{I}_p .*

Proof. We prove the lemma by first assuming that there exists a simulation relation \triangleleft such that $S_1 \triangleleft S_2$. By the definition of \triangleleft and the properties of $M(P)$, there exists a total mapping $\theta : N_1 \rightarrow N_2$ such that $\theta(S_1) = S_2$, $\theta(F_1) = F_2$, and for any state $n_1 \in N_1$, $n_1 \triangleleft \theta(n_1)$. Let the sequence of states in $M(P)$ be $\vec{v}_1 = p_1, \dots, p_k$, where $p_1 = S_1$ and $p_k = F_1$, and similarly, let the sequence of states in $M(Q)$ be $\vec{v}_2 = q_1, \dots, q_l$, where $q_1 = S_2$ and $q_l = F_2$. It is easy to verify that for any $i, j \in [1, k]$, if $i < j$, $\theta(p_i) = q_{i'}$ and $\theta(p_j) = q_{j'}$, then $i' \leq j'$. We define an equivalence relation \sim on N_1 as follows:

$$p_i \sim p_j \quad \text{iff} \quad \theta(p_i) = \theta(p_j).$$

Let $[p]_{\sim}$ denote the equivalence classes of p with respect to \sim . An equivalence class is *non-trivial* if it contains more than one state. For any equivalence class $[p]$, let p_i and p_j be the smallest and largest states in $[p]$ respectively. That is, for any $p_s \in [p]$, $i \leq s \leq j$. By treating p_i as the start state, and p_j as the final state, we have a NFA that recognizes a regular expression, denoted by $P_{i,j}$. Similarly, we can define $P_{1,i}$ and $P_{j,k}$ such that $P = P_{1,i}.P_{i,j}.P_{j,k}$. It is easy to verify that if $[p]$ is a non-trivial equivalence class, then there must be $\delta_2(\theta(p_i), _) = \theta(p_i)$. In other words, $\theta(p_i)$ indicates an occurrence of “ $_*$ ” in Q . Observe that $P_{1,i}.P_{i,j}.P_{j,k} \subseteq P_{1,i._*.}P_{j,k}$. This can be proved by using the *star* and *composition* rules of \mathcal{I}_p . By an induction on the number of non-trivial equivalence classes, one can show that $P \subseteq Q$ can always be proved using the *star*, *composition*, *transitivity* and *reflexivity* rules in \mathcal{I}_p as illustrated above. Thus \mathcal{I}_p is complete for inclusion of PL expressions. \square

Based on the previous lemmas, we provide a recursive function $Incl(n_1, n_2)$ (see Algorithm 1) for testing inclusion of PL expressions.

Lemma 3 *Given two PL expressions P and Q , there exists a quadratic time algorithm for determining whether $P \subseteq Q$.*

Proof. The function $Incl(n_1, n_2)$ is an implementation of Definition 6, and assumes the existence of $M(P')$ and $M(Q')$, where P' and Q' are the normal forms of PL expressions P and Q , respectively. To test whether $P \subseteq Q$, the function $Incl(n_1, n_2)$ is invoked with arguments $Incl(S_1, S_2)$, where S_1 and S_2 are the start states of $M(P')$ and $M(Q')$ respectively. By Lemma 1, $P \subseteq Q$ if and only if $S_1 \triangleleft S_2$. Since the function with inputs S_1 and S_2 determines whether $S_1 \triangleleft S_2$, it in fact determines whether $P \subseteq Q$.

Algorithm 1 $Incl(n_1, n_2)$: Inclusion of PL expressions

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 (a): 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 (b): 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 (c): 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

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 $n_2 \in N_2$.

We now show that the algorithm runs in quadratic time. By Proposition 4, P and Q can be rewritten into their normal forms in $O(|P|)$ and $O(|Q|)$ time respectively, where $|P|$ and $|Q|$ are the lengths of P and Q . The construction of $M(P)$ can also be done in $O(|P|)$ time and the same argument applies for Q . The initialization statement can be executed in $O(|P| |Q|)$ time. Since each condition of the cases (a)-(c) 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. \square

3.2. Axiomatization for Absolute Key Implication

Recall that an absolute key (Q', S) is a special case of a \mathcal{K} constraint $(Q, (Q', S))$ when $Q = \epsilon$. Absolute keys are constraints imposed on the entire XML tree T rather than on certain subtrees of T . Not surprisingly, the problem of determining (finite) implication of absolute keys is simpler than that for relative keys. We therefore start by giving a discussion on the rules for absolute key implication. The set of rules, denoted as \mathcal{I}_{abs} , is shown in Table 2 and is subsequently extended as rules for relative key implication.

- *superkey*. If S is a key for the set of nodes in $\llbracket Q \rrbracket$ then so is any superset of S . This is the only rule of \mathcal{I}_{abs} that has a counterpart in relational key inference.
- *subnodes*. Observe that any node $v \in \llbracket Q.Q' \rrbracket$ must be in the subtree rooted at some node v' in $\llbracket Q \rrbracket$ 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 $\llbracket Q.Q' \rrbracket$ then $Q'.P$ uniquely identifies a node in $\llbracket Q \rrbracket$.
- *containment-reduce*. If $S \cup \{P_i, P_j\}$ is the set of key paths that uniquely identifies nodes in $\llbracket Q \rrbracket$ and $P_i \subseteq P_j$ then we can leave out P_j from the set of key paths. This is because for any nodes n_1, n_2 in $\llbracket Q \rrbracket$, if $n_1 \llbracket P_i \rrbracket \cap_v n_2 \llbracket P_i \rrbracket \neq \emptyset$, then we must have $n_1 \llbracket P_j \rrbracket \cap_v n_2 \llbracket P_j \rrbracket \neq \emptyset$ since $P_i \subseteq P_j$. Thus, by the definition of keys, $S \cup \{P_i\}$ is also a key for $\llbracket Q \rrbracket$.
- *target-path-containment*. A key for the set $\llbracket Q \rrbracket$ is also a key for any subset of $\llbracket Q \rrbracket$. Observe that $\llbracket Q' \rrbracket \subseteq \llbracket Q \rrbracket$ if $Q' \subseteq Q$.

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

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

- *prefix-epsilon*. If a set $S \cup \{\epsilon, P\}$ is a key of $\llbracket Q \rrbracket$, then we can extend the key path P by appending to it another path P' , and the modified set is also a key of $\llbracket Q \rrbracket$. This is because for any nodes $n_1, n_2 \in \llbracket Q \rrbracket$, 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 $\llbracket Q \rrbracket$. 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 $\llbracket Q \rrbracket$ 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, $\llbracket \epsilon \rrbracket$ is exactly the root node and therefore any set of path expressions forms a key for the root.

Observe that these rules are far more complex than the rules for relational key inference (given in Section 1). Moreover, observe that some rules rely on the ability to reason about path inclusion.

As our next theorem shall show, the set of inference rules \mathcal{I}_{abs} is sound and complete for determining the (finite) implication of absolute keys. Moreover, there is an $O(n^5)$ algorithm for determining the (finite) implication of absolute keys, where n is the size of keys.

Theorem 5 *The finite implication problem for \mathcal{K}_{abs} is finitely axiomatizable and decidable in $O(n^5)$ time, where n is the size of keys.*

Proof. We omit the proof of soundness and completeness of \mathcal{I}_{abs} because most of the proof can be verified along the same lines as the proof of Lemma 6 discussed in Section 3.3.

A function for determining finite implication of absolute keys is given in Algorithm 2. The correctness of the algorithm follows from the axioms for finite implication of absolute keys. Step 1 of the algorithm is a simple application of the *epsilon* rule. Step 2 applies *containment-reduce* to transform keys to the key normal form. A key $\phi = (Q, (Q', S))$ of \mathcal{K} is in the *key normal form* if for every pair of paths P_i and P_j in S , $P_i \not\subseteq P_j$. In Step 3, the algorithm checks whether a key ϕ in Σ can prove φ by verifying the applicability of rules of \mathcal{I}_{abs} in three cases: when ϕ has many key paths (Step 3(i)), when ϕ has only one key path (Step 3(ii)), and when ϕ has no key paths (Step 3(iii)).

In Step 3(i), we apply *target-path-containment* rule to infer $(Q, \{P'_1, \dots, P'_m\})$ from ϕ , if possible. If successful, our remaining goal is to transform the set of key paths $\{P'_1, \dots, P'_m\}$ of ϕ to $\{P_1, \dots, P_k\}$ of φ . Since Step 2 has been applied, the set $\{P'_1, \dots, P'_m\}$ cannot be reduced further. Furthermore, observe that at this point, only *superkey*, *containment-reduce*, and *prefix-epsilon* rules are relevant rules for key paths. Our goal is thus to transform every key path P'_i into some P_j using these rules.

Algorithm 2 Finite implication of absolute keys

Input: a finite set $\Sigma \cup \{\varphi\}$ of absolute keys, where $\varphi = (Q, \{P_1, \dots, P_k\})$

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

1. if $Q = \epsilon$ then output true and terminate
2. for each $(Q_i, S_i) \in (\Sigma \cup \{\varphi\})$ 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. for each $\phi \in \Sigma$ do
 - (i) if $\phi = (Q', \{P'_1, \dots, P'_m\})$, $Q \subseteq Q'$ and for all $i \in [1..m]$ there exists $j \in [1..k]$ such that either
 - (a) $P_j \subseteq P'_i$ or
 - (b) $P_j = R_1.R_2$, $R_1 \subseteq P'_i$ and there exists $l \in [1, k]$ such that $P_l = \epsilon$
 then output true and terminate
 - (ii) if $\phi = (Q'.Q'', \{P\})$, $Q \subseteq Q'$ and for some $j \in [1..k]$, either
 - (a) $P_j \subseteq Q''.P$ or
 - (b) $P_j = R_1.R_2$, $R_1 \subseteq Q''.P$, and there exists $l \in [1, k]$ such that $P_l = \epsilon$
 then output true and terminate
 - (iii) if $\phi = (Q'.Q'', \{\epsilon\})$, $Q \subseteq Q'$ and for some $j \in [1..k]$, either
 - (a) $P_j \subseteq Q''$ or
 - (b) $P_j = R_1.R_2$, $R_1 \subseteq Q''$, and there exists $l \in [1, k]$ such that $P_l = \epsilon$
 then output true and terminate
4. output false

The resulting set of key paths can be augmented, through the use of *superkey* rule, so that the final set of key paths is $\{P_1, \dots, P_k\}$, as desired. Obviously, a key path P'_i can be replaced with P_j for some $j \in [1, k]$ if $P_j \subseteq P'_i$ (this corresponds to Step 3(i)(a)). The replacement can be done through the use of *superkey* rule, to add the path P_j , and then *containment-reduce*, to remove the path P'_i . Otherwise, if a proper prefix of P_j is contained in P'_i (see Step 3(i)(b)), then the replacement can occur through the use of *superkey* rule to add the key path R_1 . Then through *containment-reduce*, we remove the path P'_i . The path R_1 in the current set of key paths can then be extended to $R_1.R_2$: We first add another key path ϵ (if it does not already exist) through *superkey* rule. Then, through *prefix-epsilon* rule, $R_1.R_2$ can be obtained. This also explains the requirement that the key path set of φ must contain ϵ . Observe that these are the only possible ways to obtain the desired set of key paths. No inference rules can be applied if P'_i is contained in P_j or a proper prefix of P_j . Thus Step 3(i) is correct in the case when there are many key paths in ϕ .

In Step 3(ii), the *subnode* rule is applied to first obtain $(Q', \{Q''.P\})$ from ϕ and the rest of the argument is similar to the preceding discussion. In Step 3(iii), the *superkey* rule must be applied to obtain $(Q'.Q'', \{\epsilon\})$ before the *subnode* rule can be applied.

Observe that in Step 3, we test whether φ can be proven from Σ by going through each $\phi \in \Sigma$ at most once. This is sufficient because if indeed $\Sigma \models \varphi$, then φ must be the consequence of a sequence of applications of rules of \mathcal{I}_{abs} on a single key in Σ , as illustrated in the previous discussion. That is, since the applicability of *epsilon* rule has already been checked in Step 1 and every other rule of \mathcal{I}_{abs} has a premise that consists of only one key, the first rule applied in the sequence of rule applications must have a premise that makes use of only one of the keys in Σ .

We next show that the algorithm runs in $O(n^5)$ where n is the size of keys. Let $\Sigma \cup \{\varphi\}$ be a finite set of keys in \mathcal{K}_{abs} . Without loss of generality, we assume that all path expressions in the set are in the normal form. If not, by Proposition 4, it takes linear time to transform a *PL* expression to an equivalent *PL* expression in the normal form. It only takes constant time to execute Step 1 of the algorithm. From Theorem 3, Step 2 can be done in cubic time. To see this, consider a key $\phi = (Q_i, S_i)$ in $\Sigma \cup \{\varphi\}$. It takes $|S_i| * |S_i|$ units of time to check containment of path expressions in S_i . Since there are at most $|\Sigma| + |\varphi|$ keys, Step 2 is $O(n^3)$ in the size of Σ and φ . Case 3(i) of the algorithm requires one to test for containment of path expressions P_j in P'_i , which can be done in $O(|P_j| * |P'_i|)$ time, and, in case (b), partition P_j in $|P_j|$ possible ways and test for containment

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

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

in P'_i . This requires $O(|P_j| * |P_j| * |P'_i|)$ time. Therefore, for a key $\phi \in \Sigma$, the cost of Case 3(i) is at most $(|P_1| + \dots + |P_k|)(|P'_1| + \dots + |P'_m|) + (|P_1|^2 + \dots + |P_k|^2)(|P'_1| + \dots + |P'_m|)$, which is $O(n^3)$. The cost of Cases 3(ii) and 3(iii) of the algorithm is $O(n^4)$ because they require one to execute the same containment test as Case 3(i) for $|Q'.Q''|$ possible ways to partition $Q'.Q''$. Since each constraint ϕ in Σ is examined at most once, the algorithm is $O(n^5)$, where n is the size of Σ and φ . It is possible that this algorithm can be improved further to achieve a lower complexity but this is beyond the scope of this paper. \square

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 generalizations of rules shown in Table 2 except for rules that deal with the context path in the setting of relative keys: context-path-containment, context-target and interaction. We briefly illustrate these rules below.

- *context-path-containment*. Note that $\llbracket Q_1 \rrbracket \subseteq \llbracket Q \rrbracket$ if $Q_1 \subseteq Q$. If (Q', S) holds on all subtrees rooted at nodes in $\llbracket Q \rrbracket$, then it must also hold on all subtrees rooted at nodes in any subset of $\llbracket Q \rrbracket$.
- *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 $\llbracket Q \rrbracket$, S is a key for $n\llbracket Q_1.Q_2 \rrbracket$, then in any subtree of T rooted at n' in $n\llbracket Q_1 \rrbracket$, S is a key for $n'\llbracket Q_2 \rrbracket$. Note that $n'\llbracket Q_2 \rrbracket$ consists of nodes that are in both $n\llbracket Q_1.Q_2 \rrbracket$ and the subtree rooted at n' . In particular, when $Q = \epsilon$ this rule 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 $\llbracket Q_1 \rrbracket$, $Q'.P_1, \dots, Q'.P_k$

uniquely identify a node in $n[[Q_2]]$. The second key in the precondition prevents the existence of more than one Q' node under Q_2 that coincide in their P_1, \dots, P_k nodes. Therefore, P_1, \dots, 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, $(Q_1, (Q_2.Q', \{P_1, \dots, P_k\}))$ holds.

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

To illustrate how \mathcal{I} is used in an 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) $\phi \models (A.B, (C, \{._*.D, E\}))$ 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 $(Q, (Q', S \cup \{P'\}))$. By *containment-reduce* we have that $(Q, (Q', S \cup \{P'\}))$ holds.

We now show that \mathcal{I} is indeed a finite axiomatization for \mathcal{K} constraint implication. The proof for soundness and completeness is given in Lemma 6 and relies on the notion of abstract trees. An *abstract tree* is an extension of an XML tree by allowing “ $._*$ ” as a node label. Abstract trees have the following property, given by Lemma 5: whenever a finite abstract tree can be constructed such that it satisfies a set of keys Σ but not a key φ , then an XML tree can be derived with the same property – it satisfies Σ but not φ . Thus this XML tree is a proof witnessing $\Sigma \not\models \varphi$. Given this, to prove that \mathcal{I} is complete for determining (finite) implication of keys, it suffices to show that whenever $\Sigma \not\vdash_{\mathcal{I}} \varphi$, there exists an abstract tree T such that T satisfies Σ but not φ .

We start by giving a discussion on abstract trees. In an abstract tree, “ $._*$ ” is treated as an ordinary label. Therefore, the sequence of labels in an abstract tree is a PL expression that may contain occurrences of “ $._*$ ”. Let R be the sequence of labels in the path from node a to b in an abstract tree T , and let P be a path expression in PL . We say that $T \models P(a, b)$ if $R \subseteq P$. Given this, the definitions of node sets can be easily generalized for abstract trees. Given a node n in an abstract tree T and a PL expression P , the *node set* $n[[P]]$ in T consists of all nodes x such that $T \models P(n, x)$. The satisfaction of a \mathcal{K} constraint for abstract trees uses this definition of node set and is very similar to Definition 5. An abstract tree T *satisfies* a key $(Q, (Q', \{P_1, \dots, P_k\}))$ if for every node $n \in [[Q]]$, n satisfies the key $(Q', \{P_1, \dots, P_k\})$. A node n *satisfies* a key $(Q', \{P_1, \dots, P_k\})$ if for any n_1, n_2 in $n[[Q']]$, if for all $i \in [1, k]$ there exist nodes x_i and y_i in T such that $T \models P_i(n_1, x_i)$, $T \models P_i(n_2, y_i)$, and $x_i =_v y_i$, then $n_1 = n_2$.

The following definition describes the construction of an XML tree from an abstract tree.

Definition 7 Given an abstract tree T , and an element tag η , we say that G is the XML tree defined from T using η if G is obtained by substituting every occurrence of “ $._*$ ” in T by η .

Observe that G and T have the same set of nodes. In addition, for any nodes a, b in G , there is a path ρ such that $G \models \rho(a, b)$ iff there is a path R in T such that $T \models R(a, b)$, where R is the same as ρ except that for each occurrence of “ $._*$ ” in R , the label η appears at the corresponding position in ρ . Let us refer to R as the *path expression w.r.t.* ρ and conversely, ρ as the *path w.r.t.* R .

Our goal is to show that, given a set of keys $\Sigma \cup \{\varphi\}$, if T satisfies Σ but not φ , then the XML tree G defined from T using some label η also satisfies Σ and not φ . To do so, we first establish that there is a correspondence between nodes in T and in G reached by following a path expression P in PL . This result is then used to prove the desired property.

Lemma 4 *Let T be an abstract tree, η be an element tag that does not occur anywhere in T , and G be the XML tree defined from T using η . Let P be a path expression in PL , and a, b be nodes in G . Then, there exists a path $\rho \in P$ such that $G \models \rho(a, b)$ if and only if $T \models P(a, b)$, i.e., $T \models R(a, b)$ and $R \subseteq P$ where R is the path expression w.r.t. ρ .*

Proof. (1) Assume that $T \models P(a, b)$, i.e., there is a path R from a to b in T such that $R \subseteq P$. By the definition of G , we must have $G \models \rho(a, b)$, where ρ is the path w.r.t. R . Recall that ρ is obtained by substituting η for occurrences of “ $_*$ ”. Since $R \subseteq P$, we have $\rho \in P$.

(2) Conversely, assume that there exists a path $\rho \in P$ such that $G \models \rho(a, b)$. By the definition of G , we have $T \models R(a, b)$, where R is the path expression w.r.t. ρ . Thus, it suffices to show that $R \subseteq P$. To do so, we consider the NFAs of R , P and ρ as defined in Section 3.1:

$$\begin{aligned} M(R) &= (N_R, A \cup \{_ \}, \delta_R, S_R, F_R), \\ M(P) &= (N_P, A \cup \{_ \}, \delta_P, S_P, F_P), \\ M(\rho) &= (N_\rho, A \cup \{\eta\}, \delta_\rho, S_\rho, F_\rho), \end{aligned}$$

where A is an alphabet that contains neither “ $_$ ” nor η . Recall that NFAs for PL expressions have a “linear” structure as shown in Figure 3. In particular, since ρ does not contain “ $_*$ ”, $M(\rho)$ has a strict linear structure. More specifically, let the sequence of states in N_ρ be s_1, \dots, s_m , where $s_1 = S_\rho$ and $s_m = F_\rho$. Then for any $i \in [1, m-1]$, there is exactly one $l \in A \cup \{\eta\}$ such that $\delta_\rho(s_i, l) \neq \emptyset$. More precisely, $\delta_\rho(s_i, l) = s_{i+1}$, and for any $l \in A \cup \{\eta\}$, $\delta_\rho(F_\rho, l) = \emptyset$. Let the sequence of states in N_R be n_1, \dots, n_k , where $n_1 = S_R$ and $n_k = F_R$. Then we can define a function f from N_ρ to N_R with the following properties:

- $f(S_\rho) = S_R$ and $f(F_\rho) = F_R$.
- For any $i, j \in [1, m]$, if $f(s_i) = n_{i'}$, $f(s_j) = n_{j'}$ and $i < j$, then $i' \leq j'$.
- For any $i \in [1, m]$ and $l \in A$, $\delta_\rho(s_i, l) = s_{i+1}$ iff $\delta_R(f(s_i), l) = f(s_{i+1})$ and $f(s_i) \neq f(s_{i+1})$.
- For the special letters “ $_$ ” and “ η ”, for any $i \in [1, m]$, we let $\delta_\rho(s_i, _) = s_{i+1}$ iff $\delta_R(f(s_i), _) = f(s_{i+1})$ and $f(s_i) = f(s_{i+1})$. In particular, if it is the case that $\delta_R(F_R, _) = F_R$ then we have $\delta_\rho(s_{m-1}, _) = F_\rho$ and $f(s_{m-1}) = f(F_\rho) = F_R$.

We define an equivalence relation \sim on N_ρ such that

$$s \sim s' \quad \text{iff} \quad f(s) = f(s').$$

Let us use $[s]$ to denote the equivalence class of s w.r.t. \sim . Without loss of generality, assume that R is in the normal form, i.e., it does not contain two consecutive $_*$'s and it does not contain ϵ unless it is ϵ . Then it is easy to verify that $[s]$ consists of at most two states. More precisely, if $[s] = \{s\}$, then either s is a final state or there is $l \in A$ such that $\delta_\rho(s, l) = s'$, and if $[s] = \{s, s'\}$ then there is some $i \in [1, m-1]$ such that $s = s_i$, $s' = s_{i+1}$, $\delta_\rho(s, \eta) = s'$ and $f(s) = f(s')$. Given these, we define a function g from N_R to the equivalence classes such that for all $n \in N_R$,

$$g(n) = [s] \quad \text{iff} \quad f(s) = n.$$

Recall that in Lemma 1, we have shown that given two PL expressions Q and Q' with their respective NFAs $M(Q)$ and $M(Q')$ then, $Q \subseteq Q'$ if and only if $S_Q \triangleleft S_{Q'}$. The symbols S_Q and $S_{Q'}$ are the start states of $M(Q)$ and $M(Q')$ respectively and \triangleleft is a simulation as defined in Definition 6. Furthermore, there is a function θ from N_Q to $N_{Q'}$ such that $\theta(S_Q) = S_{Q'}$, $\theta(F_Q) = F_{Q'}$, and for any state $s \in N_Q$, $s \triangleleft \theta(s)$. The symbols, N_Q and $N_{Q'}$, denote the sets of states in $M(Q)$ and

$M(Q')$ respectively. Since $\rho \in P$, the language defined by ρ (which consists of a single string ρ) is contained in the language defined by P , i.e., $\rho \subseteq P$. Thus, there exists a function θ from N_ρ to N_P and a simulation relation \triangleleft such that $\theta(S_\rho) = S_P$, $\theta(F_\rho) = F_P$, and for any $s \in N_\rho$, $s \triangleleft \theta(s)$. It is easy to verify the following claim:

Claim: For all $s, s' \in [s]$, $\theta(s) = \theta(s')$.

Indeed, as observed earlier, if $s, s' \in [s]$, then there is some $i \in [1, m-1]$ such that $s = s_i$, $s' = s_{i+1}$ and $\delta_\rho(s, \eta) = s'$. Since η does not appear in P , if $\theta(s) = n'$ and $\theta(s') = n''$, then there must be $\delta_P(n', _) = n''$ and $n' = n''$, by the definition of simulation relations. As a result, we can define $\theta([s])$ to be $\theta(s)$. Given these, to show $R \subseteq P$, it suffices to show that for any $n \in N_R$,

$$n \triangleleft \theta(g(n)).$$

For if it holds, then $S_R \triangleleft \theta(g(S_R)) = \theta(S_\rho) = S_P$. We next show that this holds. Assume, by contradiction, that there is $n \in N_R$ such that it is not the case that $n \triangleleft \theta(g(n))$. Let n be such a state with the largest index in the sequence of states in N_R starting from S_R . Then by the definition of simulation relations given in Section 3.1, we must have one of the following cases.

(i) $n = F_R$ and either

1. $\theta(g(F_R)) \neq F_P$, or
2. $\theta(g(F_R)) = F_P$ but $\delta_R(F_R, _) = F_R$, $\delta_P(F_P, _) = \emptyset$.

The first case contradicts the assumption that $g(F_R) = [F_\rho]$ and $\theta([F_\rho]) = \theta(F_\rho) = F_P$. If it were the second case, then by $\delta_R(F_R, _) = F_R$, we would have $g(F_R) = \{F_\rho, s_{m-1}\}$ and $\delta_\rho(s_{m-1}, \eta) = F_\rho$. By the above claim, there must be $\theta(s_{m-1}) = \theta(F_\rho) = F_P$ and $\delta_P(F_P, _) = F_P$. Again this contradicts the assumption.

(ii) $n \neq F_R$ and either

1. $\delta_R(n, _) = n$ but $\delta_P(\theta(g(n)), _) \neq \theta(g(n))$, or
2. there is some label $l \in A$ such that $\delta_R(n, l) = n'$, but we have neither $\delta_P(\theta(g(n)), l) \neq \theta(g(n'))$ nor $\delta_P(\theta(g(n)), _) = \theta(g(n))$.

If it were the first case, then by the definition of the function g , we would have that $g(n) = \{s_i, s_{i+1}\}$ and $\delta_\rho(s_i, \eta) = s_{i+1}$. Thus by the above claim, there must be $\theta(s_i) = \theta(s_{i+1})$, $\delta_P(\theta(s_i), _) = \theta(s_i)$ and, in addition, $\theta(g(n)) = \theta(s_i)$. Hence $\delta_P(\theta(g(n)), _) = \theta(g(n))$, which contradicts the assumption. If it were the second case, then given $\delta_R(n, l) = n'$, we would have either $\delta_P(\theta(g(n)), l) = \theta(g(n'))$ or $\delta_P(\theta(g(n)), _) = \theta(g(n))$, by the definition of simulation relations and $g(n) \triangleleft \theta(g(n))$. Again this contradicts the assumption. Thus $n \triangleleft \theta(g(n))$ for all $n \in N_R$. \square

We are now on position to show that abstract trees have the following property:

Lemma 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 $G \models \neg\varphi$.*

Proof. Let $\Sigma \cup \{\varphi\}$ be a finite set of keys in \mathcal{K} , and T be a finite abstract tree such that $T \models \Sigma$ and $T \not\models \varphi$. Let η be an element tag that does not occur in any key of $\Sigma \cup \{\varphi\}$, and G be the XML tree defined from T using η . We shall prove that $G \models \Sigma$ and $G \models \neg\varphi$. From Lemma 4, it follows immediately that for any path expression P in PL , $\llbracket P \rrbracket$ consists of the same nodes in T and G . For if $T \models P(r, a)$, where r is the root, then there is a path R in T such that $T \models R(r, a)$ and $R \subseteq P$. By Lemma 4, we have $G \models \rho(r, a)$, where ρ is the path w.r.t. R and $\rho \in P$. That is, a is in $\llbracket P \rrbracket$ in the tree G . Conversely, if a is in $\llbracket P \rrbracket$ in the tree G , then there is a path $\rho \in P$ such that $G \models \rho(r, a)$. Again by Lemma 4, $T \models R(r, a)$ and $R \subseteq P$, where R is the path expression w.r.t. ρ . Thus, a is in $\llbracket P \rrbracket$ in the abstract tree T .

We are now ready to show that $G \models \Sigma$ and $G \models \neg\varphi$. Suppose, by contradiction, that there exists a key $\phi = (Q, (Q', \{P_1, \dots, P_k\}))$ in Σ such that $G \models \neg\phi$. Then there exist a node $n \in \llbracket Q \rrbracket$, two distinct nodes $n_1, n_2 \in n \llbracket Q' \rrbracket$ and, in addition, for all $i \in [1, k]$, there exist nodes $x_i \in n_1 \llbracket P_i \rrbracket$,

$y_i \in n_2 \llbracket P_i \rrbracket$ such that $x_i =_v y_i$. But by Lemma 4, we would have $T \models P_i(n_1, x_i) \wedge P_i(n_2, y_i)$ for all $i \in [1, k]$. Therefore, $T \not\models \phi$, which contradicts our assumption. We next show $G \models \neg\varphi$. Let $\varphi = (Q, (Q', \{P_1, \dots, P_k\}))$. Since $T \models \neg\varphi$, there must exist a node $n \in \llbracket Q \rrbracket$, two distinct nodes $n_1, n_2 \in n \llbracket Q' \rrbracket$, and for all $i \in [1, k]$, there exist nodes x_i, y_i such that $x_i =_v y_i$ and, in addition, there exists a path R_i in T such that $T \models R_i(n_1, x_i) \wedge R_i(n_2, y_i)$, where $R_i \subseteq P_i$. Thus, by Lemma 4, there is path $\rho_i \in P_i$ such that $x_i \in n_1 \llbracket \rho_i \rrbracket, y_i \in n_2 \llbracket \rho_i \rrbracket$. Hence $G \models \neg\varphi$. \square

This property of abstract trees is now used to show that \mathcal{I} is a finite axiomatization for \mathcal{K} constraint implication.

Lemma 6 *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, $\Sigma \models \varphi$ if and only if $\Sigma \vdash_{\mathcal{I}} \varphi$.*

Proof. To simplify the discussion, we assume that all keys are in key normal form and all path expressions are in normal form. In general, given constraints ϕ and ϕ' in \mathcal{K} , where ϕ' is the key normal form of ϕ , ϕ and ϕ' are equivalent. That is, for any XML tree T , $T \models \phi$ iff $T \models \phi'$. Thus, the assumption does not lose generality.

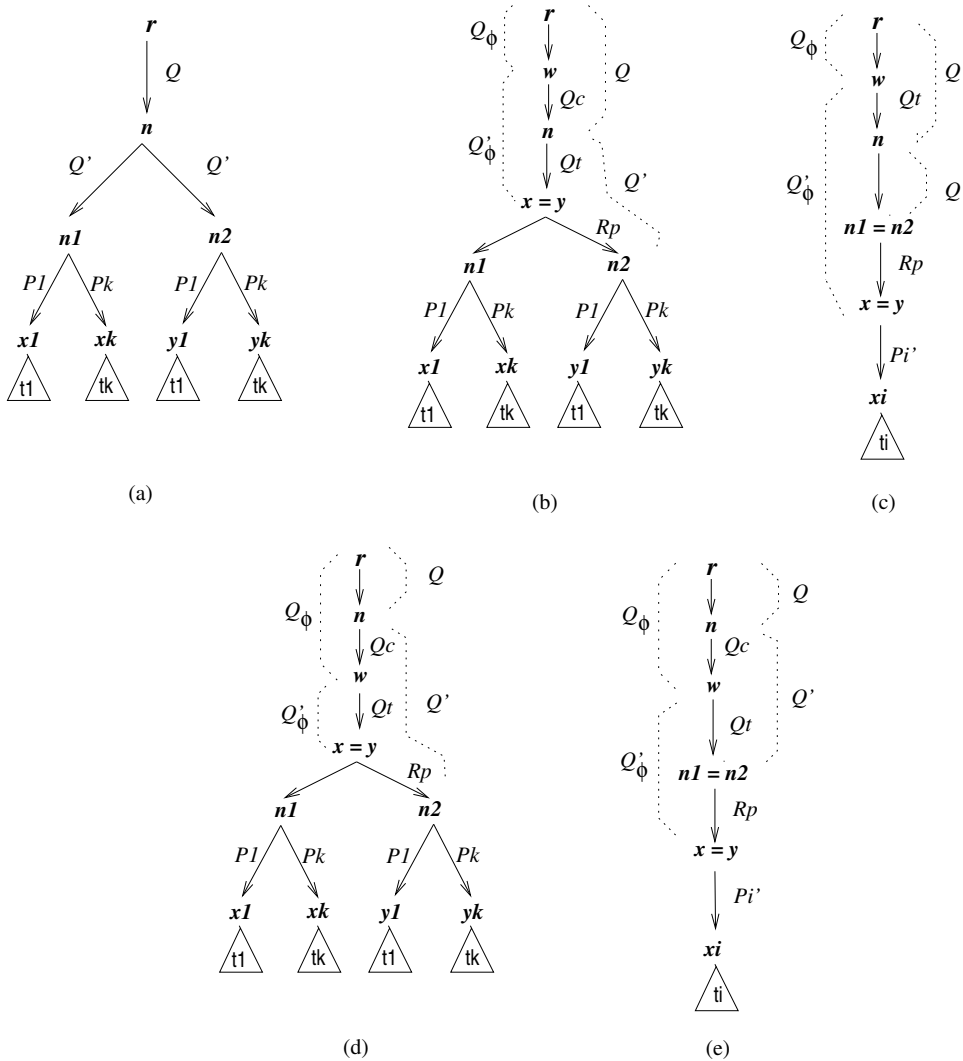


Fig. 4: Abstract trees constructed in the proof of Lemma 6

Soundness of \mathcal{I} can be verified by induction on the lengths of \mathcal{I} -proofs. For the proof of completeness, let $\Sigma \cup \{\varphi\}$ be a finite set of keys in \mathcal{K} , where $\varphi = (Q, (Q', \{P_1, \dots, P_k\}))$. A

roadmap of the proof is as follows. Suppose $\Sigma \not\vdash_{\mathcal{I}} \varphi$. Assume $Q' \neq \epsilon$, since otherwise we have $\Sigma \vdash_{\mathcal{I}} \varphi$ by the *epsilon* rule in \mathcal{I} . We show $\Sigma \not\models \varphi$ by constructing a finite XML tree G such that $G \models \Sigma$ but $G \not\models \varphi$. In other words, if $\Sigma \models \varphi$ then $\Sigma \vdash_{\mathcal{I}} \varphi$. The construction of G involves the following steps: First, we define a finite abstract tree T such that $T \not\models \varphi$. Then, T is modified in a way that the resulting tree T_f satisfies Σ . That is, for each key $\phi \in \Sigma$, we check whether T satisfies ϕ . If not, certain nodes in T are merged such that the modified tree satisfies ϕ . At the end of the merging process, $T_f \models \Sigma$ and either: (1) $T_f \not\models \varphi$, and by Lemma 5, we can construct an XML tree G from T_f that satisfies Σ but not φ ; or (2) $T_f \models \varphi$. In this case, we show that our assumption that $\Sigma \not\vdash_{\mathcal{I}} \varphi$ does not hold. That is, we show that each step of the merging operations corresponds to applications of certain rules in \mathcal{I} . Therefore, if $T_f \models \varphi$ then $\Sigma \vdash_{\mathcal{I}} \varphi$, which contradicts the assumption.

We start by giving the construction of a finite abstract T that does not satisfy φ . 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, \dots, P_k , as depicted in Figure 4 (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 .

Assume that for each $i \in [1, k]$, $x_i =_v y_i$, but for any other pair x, y in T , $x \neq_v y$. This can be achieved as follows: for each element in T we add a new text subelement. For any x, y in T , if they are x_i, y_i then we let them have the same value when they are A or S nodes, and let their text subelements have the same value when they are E nodes (in this case the text subelements are their only subelements). If they are not x_i, y_i then we let them have different values if they are A or S nodes, and let their text subelements have different values if they are E nodes. 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 $T \models \Sigma$. Using the following algorithm and starting with T constructed above, we examine each ϕ in Σ . If the abstract tree does not satisfy ϕ , then we merge certain nodes in the tree such that the modified tree satisfies ϕ . Assume that for each ϕ in Σ , $\phi = (Q_\phi, (Q'_\phi, \{P'_1, \dots, P'_m\}))$.

repeat until no further change in T

if there exist key $\phi \in \Sigma$ and nodes x, x'_1, \dots, x'_m in T_1 , y, y'_1, \dots, y'_m in T_2 ,
and node w in T such that

$$T \models Q_\phi(w, w) \wedge Q'_\phi(w, x) \wedge Q'_\phi(w, y) \wedge P'_1(x, x'_1) \wedge \dots \wedge P'_m(x, x'_m) \wedge P'_1(y, y'_1) \wedge \dots \wedge P'_m(y, y'_m) \wedge x'_1 =_v y'_1 \wedge \dots \wedge x'_m =_v y'_m \wedge x \neq y$$

then merge x, y and their ancestors in T as follows:

Case 1: if x, y are on Q' paths from n to n_1, n_2 respectively, and they are not n_1, n_2 then merge nodes as shown in Figure 4 (b)

Case 2: if x, y are on some P_i in T_1, T_2 , respectively, or if they are n_1, n_2 then (i) merge nodes as shown in Figure 4 (c)
(ii) terminate the algorithm

By the construction of T , $x'_i =_v y'_i$ iff they are corresponding nodes in T_1 and T_2 , respectively. Moreover, the node w can only be either on path Q or on path Q' . In Case 1, the subtree under x and the subtree under y will both be under the same node $x = y$, as shown in Figure 4 (b). In Case 2, under the node n_1 (which is merged with n_2) only a single copy of the P_i path is retained and we discard the rest of the key paths in $\{P_1, \dots, P_k\}$. If x and y are n_1 and n_2 , respectively, a single copy of each of the P_i paths are retained under node n_1 .

The algorithm terminates since T is finite and thus merging can be performed only finitely many times. Let T_f denote the tree obtained upon the termination of the algorithm. Note that $T_f \models \varphi$ iff $n_1 = n_2$, i.e., when the algorithm terminates in Case 2. If the algorithm does not terminate in Case 2, then $T_f \models \Sigma$ and $T_f \not\models \varphi$. By Lemma 5, there is a finite XML tree G such

that $G \models \Sigma$ and $G \not\models \varphi$. Thus what we need to do is to show that the algorithm does not terminate in Case 2, i.e., $T_f \not\models \varphi$.

We show $T_f \not\models \varphi$ by contradiction, that is, if $T_f \models \varphi$ then $\Sigma \vdash_{\mathcal{I}} \varphi$, which contradicts our assumption. Let us also use T to denote the tree obtained after executing z merging operations. We show by induction on z that each step of merging corresponds to applications of certain rules of \mathcal{I} , and thus if $T \models \varphi$ (i.e., the algorithm terminates in Case 2 after step z), then $\Sigma \vdash_{\mathcal{I}} \varphi$. When $z = 0$, the statement holds since $T_f \not\models \varphi$. Assume the statement for z and we show that it also holds for $z + 1$.

(1) First, consider the merging in Case 1 as shown in Figure 4 (b) and (d). This step generates \mathcal{I} -proofs for keys that will be used in establishing $\Sigma \vdash_{\mathcal{I}} \varphi$ if $T_f \models \varphi$. By the definition of abstract trees, Case 1 can only happen if there is a PL expression R_p such that $Q.Q' \subseteq Q_\phi.Q'_\phi.R_p$ and in addition, 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 is a PL expression R_j such that $R_p.P_s \subseteq P'_j.R_j$. If it is (ii) then there must exist some $l \in [1, k]$ such that $P_l = \epsilon$ in φ , by the definition of T . We consider the following cases.

(a) If the node w is on the path Q , i.e., it is above n in T , then there must be PL expression Q_t such that $Q' = Q_t.R_p$, and $x, y \in n[[Q_t]]$ as illustrated in Figure 4 (b). Moreover, from ϕ the following can be proved:

$$(Q, (Q_t, \{R_p.P_1, \dots, R_p.P_k\}))$$

by using *context-target*, three *containment* rules (i.e., *context-path-containment*, *target-path-containment* and *key-path-containment*) and *superkey*. If it is (ii) then *prefix-epsilon* is also needed.

(b) If the node w is on the path Q' , i.e., it is below n but above n_1, n_2 in T , then there must be PL expressions Q_c, Q_t such that $Q.Q_c \subseteq Q_\phi$, $Q' = Q_c.Q_t.R_p$, $w \in n[[Q_c]]$ and $x, y \in n[[Q_c.Q_t]]$ as illustrated in Figure 4 (d). This can only happen when some descendants x', y' of n on path Q' above x, y were merged in a previous step by the algorithm. More precisely, there are PL expressions Q_{t1}, Q_{t2} such that $Q_t = Q_{t1}.Q_{t2}$, $x', y' \in n[[Q_c.Q_{t1}]]$ and x', y' were merged in Case 1 of the algorithm. Thus by the induction hypothesis, we have that the following is provable from Σ by using \mathcal{I} :

$$(Q, (Q_c.Q_{t1}, \{Q_{t2}.R_p.P_1, \dots, Q_{t2}.R_p.P_k\})).$$

From ϕ the following can be proved

$$(Q.Q_c, (Q_{t1}.Q_{t2}, \{R_p.P_1, \dots, R_p.P_k\}))$$

by using the three *containment* rules and *superkey*. If it is (ii) then *prefix-epsilon* is also needed. Thus by *context-target* and *interaction* we have

$$(Q, (Q_c.Q_{t1}.Q_{t2}, \{R_p.P_1, \dots, R_p.P_k\})).$$

That is, $(Q, (Q_c.Q_t, \{R_p.P_1, \dots, R_p.P_k\}))$.

(2) Next, we consider the merging in Case 2 as shown in Figure 4 (c) and (e). If it is the case then we show $\Sigma \vdash_{\mathcal{I}} \varphi$. By the definition of abstract trees, Case 2 can only happen if there is a PL expression R_p such that $Q.Q'.R_p \subseteq Q_\phi.Q'_\phi$ and in addition, 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 is a PL expression R_j such that $P_s \subseteq R_p.P'_j.R_j$. If it is (ii) then there must exist some $l \in [1, k]$ such that $P_l = \epsilon$ in φ , by the definition of T . We consider the following cases.

(a) If the node w is on the path Q , i.e., it is above n in T , then there must be PL expression Q_t such that $Q_t.Q'.R_p \subseteq Q'_\phi$, $x \in n_1[[R_p]]$ and $y \in n_2[[R_p]]$ as illustrated in Figure 4 (c). If $R_p = \epsilon$ then φ can be proved from ϕ by using *context-target*, the three *containment* rules and *superkey*. Note that if it is (ii) then *prefix-epsilon* is also needed. If $R_p \neq \epsilon$ then by the construction of T , we must have $m = 1$. Thus, we can also prove φ from ϕ by using *subnodes*, *context-target*, the three *containment* rules and *superkey*. Thus, we have $\Sigma \vdash_{\mathcal{I}} \varphi$, which contradicts our assumption.

(b) If the node w is on the path Q' , i.e., it is below n but above n_1, n_2 in T , then there must be PL expressions Q_c, Q_t such that $Q.Q_c \subseteq Q_\phi$, $Q' = Q_c.Q_t$, $w \in n[[Q_c]]$, $x \in n_1[[R_p]]$ and $y \in n_2[[R_p]]$ as illustrated in Figure 4 (e). This can only happen when some descendants x', y' of n on path

Q' above n_1, n_2 were merged in a previous step by the algorithm. More precisely, there are PL expressions Q_{t1}, Q_{t2} such that $Q_t = Q_{t1}.Q_{t2}$, $x', y' \in n[[Q_c.Q_{t1}]]$ and x', y' were merged in Case 1 of the algorithm. Thus, by the induction hypothesis, we have that the following is provable from Σ by using \mathcal{I} :

$$(Q, (Q_c.Q_{t1}, \{Q_{t2}.P_1, \dots, Q_{t2}.P_k\})).$$

If $R_p = \epsilon$ then from ϕ the following can be proved

$$(Q.Q_c, (Q_{t1}.Q_{t2}, \{P_1, \dots, P_k\}))$$

by using the three *containment* rules and *superkey*. Observe that if it is (ii) then *prefix-epsilon* is also needed. If $R_p \neq \epsilon$ then by the construction of T , we must have $m = 1$. Thus, we can also prove it from ϕ by using *subnodes*, *context-target*, the three *containment* rules and *superkey*. Thus by *interaction* and *context-target* we have

$$(Q, (Q_c.Q_{t1}.Q_{t2}, \{P_1, \dots, P_k\})).$$

That is, $(Q, (Q', \{P_1, \dots, P_k\})) = \varphi$. Thus again we have $\Sigma \vdash_{\mathcal{I}} \varphi$, which contradicts our assumption.

This shows that \mathcal{I} is complete for \mathcal{K} constraint implication and thus completes the proof of Lemma 6. \square

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

Lemma 7 *There is an algorithm that, given any finite set $\Sigma \cup \{\varphi\}$ of \mathcal{K} constraints, determines whether $\Sigma \models \varphi$ in time $O(n^7)$, where n is the size of keys.*

Proof. A function for determining finite implication of \mathcal{K} constraints is given in Algorithm 3.

The correctness of the algorithm follows from Lemma 6 and its proof. Similar to the algorithm for implication of absolute keys (Algorithm 2), it applies \mathcal{I} rules to derive φ if $\Sigma \models \varphi$. Observe that Steps 1 and 2 are identical in both algorithms, and Step 4(a), when ignoring the context path, proves φ from ϕ by applying exactly the same inference rules as Algorithm 2. In fact, if we replace paths Q, Q_ϕ , and Q_t by ϵ in Step 4(a), it is identical to Algorithm 2.

The presence of a context path adds complexity to the algorithm for two reasons. First, it can be the case that either the context path of a key in Σ is contained in a prefix of Q , considered in Steps 4(a) and (c); or Q is contained in a prefix of the context path of a key in Σ , considered in Steps 4(b) and (d). Second, the application of the *interaction* rule depends on the existence of two distinct keys in Σ . As a consequence, we need to keep track of intermediate keys in the \mathcal{I} -proof. In the algorithm, these keys are produced by Steps 4(c) and (d), and they are kept in the set variable X .

Next, observe that each conditional statement in step 4 corresponds to applications of certain rules in \mathcal{I} . More specifically:

- Steps 4(a) and (c) use the 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.

For the interested reader, Step 4(a) corresponds to Figure 4(c). Since nodes n_1 and n_2 are merged as a result of this key, we can prove φ . Similarly, Step 4(b) corresponds to Figure 4(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 Figure 4(b) and (d) respectively. Here these keys do not prove φ 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 .

Algorithm 3 Finite implication of \mathcal{K} constraints

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

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

1. if $Q' = \epsilon$ then output true and terminate
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 Σ can be applied in cases (a)-(d).
 - for each $\phi = (Q_\phi, (Q'_\phi, \{P'_1, \dots, P'_m\})) \in \Sigma$ do

// See Figure 4(c) for an illustration of this case.

 - (a) if there is Q_t, R_p in PL such that $Q \subseteq Q_\phi \cdot Q_t$, $Q_t \cdot Q' \cdot R_p \subseteq Q'_\phi$, $R_p = \epsilon$ if $m > 1$ and for all $j \in [1, m]$ there is $s \in [1, k]$ such that either
 - (i) $P_s \subseteq R_p \cdot P'_j$ or
 - (ii) $P_s = R'_s \cdot R''_s$, $R'_s \subseteq R_p \cdot P'_j$, and there exists $l \in [1, k]$ such that $P_l = \epsilon$
 then output true and terminate
 - // See Figure 4(e) for an illustration of this case.
 - (b) if there are Q_c, Q_t, R_p in PL such that

$Q \cdot Q_c \subseteq Q_\phi$, $Q' \cdot R_p \subseteq Q_c \cdot Q'_\phi$, $Q' = Q_c \cdot Q_t$, $R_p = \epsilon$ if $m > 1$, and for all $j \in [1, m]$ there is $s \in [1, k]$ such that either

 - (i) $P_s \subseteq R_p \cdot P'_j$ or
 - (ii) $P_s = R'_s \cdot R''_s$, $R'_s \subseteq R_p \cdot P'_j$, and there exists $l \in [1, k]$ such that $P_l = \epsilon$; and moreover, there is $(Q, (Q_c, \{Q_t \cdot P_1, \dots, Q_t \cdot P_k\}))$ in X
 then output true and terminate
 - // See Figure 4(b) for an illustration of this case.
 - (c) if there are Q_c, Q_t, R_p in PL such that $Q \subseteq Q_\phi \cdot Q_c$, $Q_c \cdot Q' \subseteq Q'_\phi \cdot R_p$, $Q' = Q_t \cdot R_p$ and for all $j \in [1, m]$ there is $s \in [1, k]$ such that either
 - (i) $R_p \cdot P_s \subseteq P'_j$ or
 - (ii) $P_s = R'_s \cdot R''_s$, $R_p \cdot R'_s \subseteq P'_j$, and there exists $l \in [1, k]$ such that $P_l = \epsilon$
 then
 - (1) if $m = 1$ then $X := X \cup \{(Q, (Q_1, \{Q_2 \cdot R_p \cdot P_1, \dots, Q_2 \cdot R_p \cdot P_k\}))\}$ where $Q_t = Q_1 \cdot Q_2$ for some $Q_1, Q_2 \in PL$;
 - (2) if $m > 1$ then $X := X \cup \{(Q, (Q_t, \{R_p \cdot P_1, \dots, R_p \cdot P_k\}))\}$;
 - (3) $\Sigma := \Sigma \setminus \{\phi\}$;
 - // See Figure 4(d) for an illustration of this case.
 - (d) if there are Q_c, Q_t, R_p in PL such that $Q \cdot Q_c \subseteq Q_\phi$, $Q' \subseteq Q_c \cdot Q'_\phi \cdot R_p$, $Q' = Q_c \cdot Q_t \cdot R_p$ and for all $j \in [1, m]$ there is $s \in [1, k]$ such that either
 - (i) $R_p \cdot P_s \subseteq P'_j$ or
 - (ii) $P_s = R'_s \cdot R''_s$, $R_p \cdot R'_s \subseteq P'_j$, and there exists $l \in [1, k]$ such that $P_l = \epsilon$; and moreover, there is $(Q, (Q_c, \{Q_t \cdot R_p \cdot P_1, \dots, Q_t \cdot R_p \cdot P_k\}))$ in X
 then
 - (1) if $m = 1$ then $X := X \cup \{(Q, (Q_1, \{Q_2 \cdot R_p \cdot P_1, \dots, Q_2 \cdot R_p \cdot P_k\}))\}$ where $Q_c \cdot Q_t = Q_1 \cdot Q_2$ for some $Q_1, Q_2 \in PL$;
 - (2) if $m > 1$ then $X := X \cup \{(Q, (Q_c \cdot Q_t, \{R_p \cdot P_1, \dots, R_p \cdot P_k\}))\}$;
 - (3) $\Sigma := \Sigma \setminus \{\phi\}$;
5. output false

We next show that this algorithm runs in polynomial time. To see this, observe that step 1 takes constant time and step 2 takes at most $O((|\Sigma| + |\varphi|)^3)$ time. For step 4, the worst scenario can happen as follows: for each key in Σ , the conditions of (a) - (d) are tested and only the last key in Σ is removed after testing all keys in Σ . Hence, the second time the for loop is performed, one less key is tested. Therefore if there are s keys in Σ , 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'_ϕ 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 ϕ , we check if there is a key path P_s in φ and partition P_s to get R'_s such that case (i) or (ii) is satisfied. This costs $|P_s| * (|R_p| + |P'_j|) + |P_s| * |P_s| * (|R_p| + |P'_j|)$. Since there are m key paths in ϕ , for all k key paths in φ these tests cost $(|P_1| + \dots + |P_k|) * (m * |R_p| + |P'_1| + \dots + |P'_m|) + (|P_1| * |P_1| + \dots + |P_k| * |P_k|)(m * |R_p| + |P'_1| + \dots + |P'_m|)$. Note that $R_p = \epsilon$ when $m > 1$, and there are $|Q| * |Q'_\phi|$ possible expressions for Q_t , and R_p . Therefore, the cost of step (a) is at most $|Q| * |Q'_\phi| * (|\varphi| * (|\phi| + |\varphi|) + |\varphi|^2 * (|\phi| + |\phi|))$, which is $O(n^5)$. It is easy to see that the steps (b), (c), and (d) involve at most the same cost. Since these tests are performed $O(s^2)$ times, the overall cost of the algorithm is $O(n^7)$, and therefore we have a polynomial algorithm. It is possible that this algorithm can be improved further to achieve a lower complexity but this is beyond the scope of this paper. \square

4. DISCUSSION

We have investigated a key constraint language introduced in [13] 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, keys defined in this language can be reasoned about efficiently. More specifically, keys expressed 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 D (see, e.g., Figure 5 (a)), and there exists a finite XML tree that satisfies the key φ (e.g., Figure 5 (b)). However, there is no XML tree that both conforms to D and satisfies φ . This is because D requires an XML tree to have two distinct X elements, whereas φ 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 [38], also interact with DTDs or other type systems for XML. This issue was recently investigated in [5, 22] 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. Among them are keys defined requiring *existence* and *uniqueness*, as happens in XML Schema. This means that a key path has to exist and be unique at every node in the target set. This is the *strong-key* definition mentioned in [13]. Another possibility is to require equal keys to imply value-equality on nodes rather than node identity [†]. This is useful in XML documents in which redundancy is tolerated. It is sometimes useful to put the same information in more than once place in an XML document in order to avoid having to do joins to recover this information.

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, they are not finitely axiomatizable. Another alternative is to adopt the language of [30], 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

[†]Thanks to David Maier for the observation.

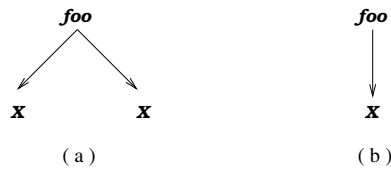


Fig. 5: Interaction between DTDs and XML keys

complexity comparable to the related result established in this paper. For XPath [18] expressions, questions in connection with (finite) satisfiability and (finite) implication of keys defined in terms of these complex path expressions are, to the best of our knowledge, still open.

Fourth, along the same lines as our XML key language, a language of foreign keys needs to be developed for XML. As shown by [21, 22, 5], the implication and finite implication problems for a class of keys and foreign keys defined in terms of XML attributes are undecidable, 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 validation. Native constraint validators are emerging for XML Schema keys [31, 42] as well as for the more general form of keys presented in this paper [44]. However, if the XML data is being stored using relational technology then it may also be possible to validate keys using triggered procedures or PRIMARY KEY techniques. Using the restricted form of keys found in XML Schema, a relational schema can be generated or augmented from the keys to generate relational primary keys [43]. A different strategy for generating a relational schema to store XML data based on keys is to compute a minimum cover of all functional dependencies propagated from XML keys on a universal relation and then refine the schema into a normal form using the dependencies computed [19].

Acknowledgements — Peter Buneman and Wang-Chiew Tan are supported by NSF IIS 99-77408 and NSF DL-2 IIS 98-17444. Susan Davidson is supported by NSF DBI99-75206. Wenfei Fan is supported in part by NSF Career Award IIS-0093168. Part of this work was done while Wang-Chiew Tan was a Ph.D. student at the University of Pennsylvania. The authors thank Michael Benedikt, Chris Brew, Dave Maier, and Henry Thompson for helpful discussions. The authors are grateful to the referees for valuable suggestions on improving the paper.

REFERENCES

- [1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web. From Relations to Semistructured Data and XML*. Morgan Kaufman (2000).
- [2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley (1995).
- [3] S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pp. 122–133, Tucson, Arizona (1997).
- [4] V. Apparao et al. *Document Object Model (DOM) Level 1 Specification*. W3C Recommendation, <http://www.w3.org/TR/REC-DOM-Level-1/> (1998).
- [5] M. Arenas, W. Fan, and L. Libkin. On verifying consistency of XML specifications. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)* (2002).
- [6] M. Arenas, W. Fan, and L. Libkin. What’s hard about XML Schema constraints? In *Proc. of 13th International Conference on Database and Expert Systems Applications (DEXA)*, pp. 269–278, Aix-en-Provence, France (2002).
- [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] M. Benedikt, W. Fan, and G. Kuper. Structural properties of XPath fragments. In *Proceedings of the International Conference on Database Theory*, Siena, Italy (2003).
- [10] D. Benson et al. GenBank. *Nucleic Acids Research*, **28**:15–18 (2000).
- [11] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium (W3C), <http://www.w3.org/TR/REC-xml> (1998).

- [12] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about keys for XML. In *Lecture Notes in Computer Science (DBPL'2001)*, volume 2397, pp. 133–148 (2001).
- [13] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. *Computer Networks*, **39**(5):473–487 (2002).
- [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. Interaction between path and type constraints. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pp. 56–67, Philadelphia, Pennsylvania. to appear in *ACM Transactions on Computational Logic (TOCL)* (1999).
- [16] P. Buneman, W. Fan, and S. Weinstein. Path constraints in semistructured databases. *Journal of Computer and System Sciences (JCSS)*, **61**:146–193 (2000).
- [17] J. Clark. *XSL Transformations (XSLT)*. W3C Recommendation, <http://www.w3.org/TR/xslt> (1999).
- [18] J. Clark and S. DeRose. *XML Path Language (XPath)*. W3C Working Draft, <http://www.w3.org/TR/xpath> (1999).
- [19] S. Davidson, W. Fan, C. Hara, and J. Qin. Propagating xml constraints to relations. In *Proceedings of the International International Conference on Data Engineering (ICDE'2003)* (2003).
- [20] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press (1972).
- [21] W. Fan and J. Siméon. Integrity constraints for XML. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pp. 23–34, Dallas, Texas. to appear in *Journal of Computer and System Sciences (JCSS)* (2000).
- [22] Wenfei Fan and Leonid Libkin. On XML integrity constraints in the presence of DTDs. *Journal of the ACM (JACM)*, **49**(3):368–406 (2002).
- [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)*, pp. 91–100, Philadelphia, Pennsylvania (1999).
- [25] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley (1979).
- [26] H. Hunt, D. Rosenkrantz, 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, <http://www.w3.org/TR/1998/NOTE-XML-data> (1998).
- [29] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pp. 65–76, Madison, Wisconsin (2002).
- [30] T. Milo and D. Suciu. Index structures for path expressions. *Lecture Notes in Computer Science (ICDT'99)*, **1540**:277–295 (1999).
- [31] Microsoft XML Parser 4.0(MSXML), Available at: <http://msdn.microsoft.com>.
- [32] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, dtlds, and variables. In *Proceedings of the International Conference on Database Theory, Siena, Italy* (2003).
- [33] Z.M. Ozsoyoglu and L.-Y. Yuan. A new normal form for nested relations. *ACM Transactions on Database Systems*, **12**(1):111–136 (1987).
- [34] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education (2000).
- [35] J. Robie, J. Lapp, and D. Schach. *XML Query Language (XQL)*. Workshop on XML Query Languages (1998).
- [36] J. Sulston *et al.* The *C. elegans* genome sequencing project: A beginning. *Nature*, **356**(6364):37–41 (1992).
- [37] H. Thompson, Personal Communication.
- [38] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures*. W3C Working Draft, <http://www.w3.org/TR/xmlschema-1/> (2001).
- [39] 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).
- [40] V. Vianu. A Web odyssey: From Codd to XML. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pp. 1–15, Santa Barbara, California (2001).
- [41] P. Wadler. A Formal Semantics for Patterns in XSL. Technical report, Computing Sciences Research Center, Bell Labs, Lucent Technologies (2000).
- [42] XML Schema Validator, Available at: <http://www.ltg.ed.ac.uk/ht/xsv-status.html>.
- [43] Y.Chen, S. Davidson, and Y. Zheng. Constraint preserving xml storage in relations. In *Proceedings of the International Workshop on the Web and Databases (WebDB 2002)*, Available at: <http://db.cis.upenn.edu/Publications>. (2002).
- [44] Y.Chen, S. Davidson, and Y. Zheng. Validating Constraints in XML. Technical report, University of Pennsylvania, Computer and Information Science Department (2002).