Constraints for Semistructured Data and XML*

Peter Buneman
University of Pennsylvania
peter@cis.upenn.edu

Wenfei Fan
Temple University
fan@cis.temple.edu

Jérôme Siméon
Bell Laboratories
simeon@research.bell-labs.com

Scott Weinstein
University of Pennsylvania
weinstein@cis.upenn.edu

Abstract

Integrity constraints play a fundamental role in database design. We review initial work on the expression of integrity constraints for semistructured data and XML.

1 Introduction

Semistructured data is often described as "schema-less" or "self-describing". What these terms mean is that no pre-imposed schema or type system is needed for the interpretation of semistructured data, which is usually understood as some form of labeled graph. In XML terminology we start only with the notion that a document is "well-formed". This is a very weak condition on the syntax of XML, which only guarantees that the data can be represented by a labelled tree. This view of data immediately invites the question: what does it mean to impose structure on semistructured data? In fact, much of the literature on semistructured data and XML is directly concerned with this question.

1.1 Constraints and types

The use of a graph model of data blurs the distinction between *types* and *constraints* that is commonly made in traditional database systems. In the world of semistructured data these are both constraints which restrict the structure of the graph interpretation of the data. One of the goals of this paper is to express both (traditional) types and (traditional) constraints as constraints on semistructured data. It is therefore worth examining the distinction in the context of conventional database systems. As an example, consider the following ODL [22] schema:

```
class student
   ( extent students
     key SSN)
   { attribute string SSN;
     attribute string name;
      relationship set<course> taking
                inverse course::taken_by;}
class course
   ( extent courses
     key cno)
   { attribute string cno;
     attribute string title;
      relationship set<student> taken_by
                inverse student::taking;}
```

If we remove the **extent** and **key** assertions and replace the **relationship** assertions with type declarations such as attribute set<course> taking we obtain a class declaration for a conventional object-oriented language. These classes or types are an essential part of any program that constructs or queries data. Without them the program is meaningless. Contrast this with the situation in most XML query languages where no types are needed ("type" errors such as mis-spelled tag names show up not as static errors but as empty answers.) In addition, this schema also defines integrity constraints: the **extent** and **inverse** declarations specify the following: (a) *Inclusion constraints.* For any student s, $s.taking$ is included in the extent *courses*. Similarly, for any course c, $c.taken\_by$ is a subset of the extent *students*. (b) *Inverse constraints.* For any student s and any course c, if s is taking c then c is taken by s, and vice versa. (c) *Key constraints.* Any two distinct student objects (they resulted from separate calls to a constructor) have different SSN's. Although such constraints cannot be expressed in any object-oriented system, they can be expressed in most schema definition languages, and – just as the static analysis of types is important for program correctness and efficiency – the static analysis of such constraints is important for query optimization [16, 13, 21].

It appears, therefore, that the distinction between types and constraints is dictated largely by what conventional programming languages treat as types. That there is a non-trivial interaction between types and constraints is evident from consideration of the following SQL [20] specification:

```
create table students
   ( SSN char(9),
```

*Database Principles Column.
Column editor: Leonid Libkin, Department of Computer Science, University of Toronto, Toronto, Ontario M5S 3H5, Canada. E-mail: libkin@cs.toronto.edu.

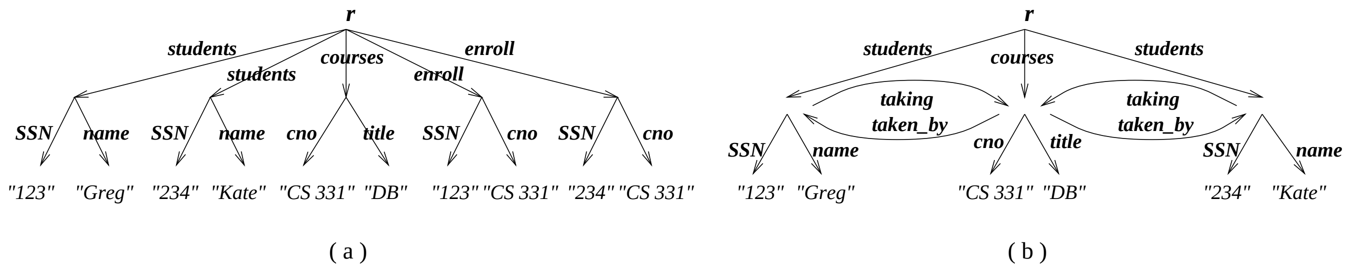

Figure 1: Graph representation of structured databases

```
      name char(20),
      primary key (SSN) )
  create table courses
    ( cno char(7),
      title char(20),
      primary key (cno) )
  create table enroll
    ( SSN char(9),
      cno char(7),
      primary key (SSN, cno),
      foreign key SSN references students,
      foreign key cno references courses)
```

The types associated with these ODL and SQL schemas are quite different. However the two schemas appear to specify equivalent information (to within details of the base types), and this equivalence depends on the presence of the constraints as well as the types.

1.2 Graph models of data

We start by adopting an edge-labeled graph as a model of semistructured data. Figure 1 (a) and (b) depict relational and object-oriented databases for student/course data, respectively. We first consider how the traditional type of these databases constrains the structure of the graph. For this we adopt a simple object-oriented type system, which also serves to describe the type of relational databases. Let $\mathcal{C}$ be some finite set of *classes*, then the set of *types over* $\mathcal{C}$, $Types^{\mathcal{C}}$, can be defined by:

$$\tau \ ::= \ b \ | \ C \ | \ \{\tau\} \ | \ [l_1 : \tau_1, \ldots, l_n : \tau_n]$$

where b denotes some atomic type such as integer or string, C is a class in $\mathcal{C}$, and the notations $\{.\}$ and $[.]$ represent *set type* and *record type*, respectively. In addition, there is a mapping ν from $\mathcal{C}$ to $Types^{\mathcal{C}}$ that defines the types of classes. For example, the type of our object-oriented schema is expressed by:

$$\begin{aligned} \nu(\mathsf{student}) &= [\ \mathsf{name\colon string,\ taking\colon \{course\}}\ ] \\ \nu(\mathsf{course}) &= [\ \mathsf{cno\colon string,\ cname\colon string,} \\ &\qquad \mathsf{taken\_by\colon \{student\}}\ ] \end{aligned}$$

In the graph representation, a value of type b is represented by a node carrying the value. An object of class C has a structure defined by $\nu(C)$. A set of type $\{\tau\}$ is modeled by a node with outgoing edges that are labeled with τ and lead to nodes with a τ structure. A record of type $[l_1 : \tau_1, \ldots, l_n : \tau_n]$ is represented by a node that has n outgoing edges. Each of these edges is labeled with a unique l_i and leads to a node with a τ_i structure, for any $i \in [1, n]$.

Inclusion and inverse constraints are both examples of *path constraints*, i.e., constraints defined in terms of navigation paths. As an example, consider the following (implicit) path constraints expressed on the ODL graph in Figure 1 (b):

$$\begin{aligned} students.taking &\subseteq courses, \\ courses.taken\_by &\subseteq students, \\ students.taking &\rightleftharpoons courses.taken\_by. \end{aligned}$$

We shall define these constraints more rigorously in Section 2. The first constraint says that the set of nodes reached by following a *students.taking* path (from the root) is contained in the set of nodes reached by following a *courses* path. The third says that if we follow a *students* edge to a node x and x has a *taking* edge to a node y, then y must have a *taken_by* edge to x; similarly for *courses* edges.

We defer the discussion of key constraints to Section 3. It is worth remarking that both the type constraints and path constraints so far described have a simple first-order interpretation over data graphs if we treat edges as binary relations [11].

1.3 XML

When it comes to XML (eXtensible Markup Language [5]), the story is more interesting. XML data also has a graph representation, but it is typically modeled as a node-labeled *tree*. Like semistructured data, XML data does not require a type system or schema, but it may have a DTD (Document Type Definition). In addition, a number of proposals have been developed for XML that correspond to a data definition language, e.g., XML Schema [24], XML Data [19]. DTDs and these forms of specification can also be viewed as constraints. For example, a DTD for our student/course data is as follows:

```
<!ELEMENT r        (student*, course*)>
```

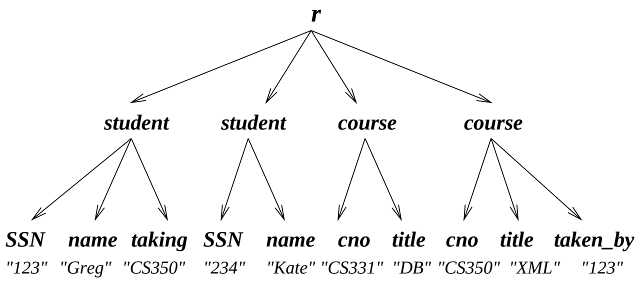

Figure 2: An XML document tree

```
<!ELEMENT student (SSN, name, taking*)
<!ELEMENT course  (cno, title, taken_by*)
```

Here we omit the descriptions of elements whose type is string (PCDATA). This DTD constrains the structures of XML document trees that conform to the DTD. For example, it specifies, among other things, that the root of such a tree has `student` children followed by `course` children, and in addition, a `student` node in the tree must have a unique `SSN`, a unique `name` and a possibly empty sequence of `taking` subelements. An XML document tree conforming to the DTD is depicted in Figure 2. It should be noted that element types in a DTD are defined in terms of regular expressions, which are more complex than relation and class specifications found in traditional databases.

In writing down constraints for XML we need to make two changes. The first is the straightforward transition from edge-labeled graphs to node-labeled trees. The second is more interesting. DTD specifications are *global*. For example, the DTD constrains all `student` nodes no matter where they occur (a more complicated DTD could allow such nodes to occur in various places). To this end we need to introduce path constraints that also operate globally. For example, consider:

$$\begin{aligned} \_*.student.taking &\subseteq \_*.course.cno, \\ \_*.course.taken\_by &\subseteq \_*.student.SSN. \end{aligned}$$

Here '$\_*$' is a combination of wild card and the Kleene star, which matches any navigation path. These are path inclusion constraints. Referring to an XML document tree representing our student/course data, the first constraint asserts that for each node reached by following the path $\_*.student.taking$ from the root of the tree, its *value* must match that of a node reachable by following $\_*.course.cno$ from the root. Intuitively, it states that for any students, no matter where they occur in the tree, their `taking` subelements stand for courses; similarly for any courses.

1.4 Decision problems

There are two classical decision problems associated with integrity constraints. One concerns *consistency:* given any constraints, does there exist a database that satisfies those constraints? The other is *implication:* given that certain constraints are known to hold, does it follow that some other constraint is necessarily satisfied? Implication is important in, among other things, data integration. For example, one may want to know whether a constraint φ holds in a mediator interface, which may use XML as a uniform data format [4]. This cannot be verified directly since the mediator interface does not contain data. One way to verify φ is to show that it is implied by constraints that are known to hold [16]. Other important applications of implication include query optimization [13, 21] and database normalization [23]. The consistency and implication problems have been well studied in the relational dependency theory (see, e.g., [2] for a survey).

The analyses of consistency and implication in the context of semistructured and XML data are more intriguing than their counterparts in relational databases. First, integrity constraints on semistructured and XML data are typically defined in terms of path expressions and thus are more complex than relational constraints. Second, type constraints in this context, when present, are also more complex than those found in a relational schema. Type constraints may interact with integrity constraints. This is certainly the case for XML: DTDs are defined with regular expressions and they may interact with integrity constraints in a highly intricate way. As XML data may or may not come with a DTD, the analyses need to be conducted in the presence and absence of DTDs. For semistructured data, one wants to know whether complexity results established in the context of typed data still hold, and vice versa. This highlights the need for investigating the interaction between type constraints and path constraints. An interesting observation is that types do not necessarily complicate the analyses of consistency and implication.

In the rest of the paper, we shall present an informal overview of path constraints for semistructured data in Section 2, and keys, foreign keys for XML in Section 3. In Section 4 we suggest directions for further research.

2 Constraints on semistructured data

Path inclusion constraints have been developed for semistructured data. A first class of path constraints was introduced in [3], denoted by $\mathcal{P}_{inc}$. The general form of a $\mathcal{P}_{inc}$ constraint is:

$$\alpha \subseteq \beta,$$

where α and β are regular path expressions. Referring to a rooted edge-labeled directed graph representing semistructured data, the constraint asserts that for any node in the graph, if it is reached by following α from the root, then it must also be reachable from the root by following β. If we use r to denote the root and treat α, β as logic formulas with two free variables indicating the tail and head of a path, respectively, then the semantics of the constraint can be described by

$$\forall x\,(\alpha(r, x) \rightarrow \beta(r, x)).$$

Regular path expressions are defined by

$$\alpha \quad ::= \quad \epsilon \;\mid\; l \;\mid\; \alpha.\alpha \;\mid\; \alpha|\alpha \;\mid\; \alpha*$$

where ϵ is the empty sequence, l is an edge label, and '|', '.' and '$*$' denote union, concatenation, and the Kleene closure, respectively. A path expression is called a *simple path* if it is a sequence of edge labels, i.e., it contains neither '|' nor '$*$'. One can express in $\mathcal{P}_{inc}$:

$$\begin{array}{ll} CS.course.taken\_by & \subseteq CS.student, \\ CS.course.prerequisites* & \subseteq (CS|Math).course. \end{array}$$

That is, referring to a university, courses of CS department are taken by CS students only, and the prerequisites of these courses are offered by CS or Math department. Constraints of $\mathcal{P}_{inc}$ describe inclusion relations.

If the semistructured database contains information about more than one university, one cannot use $\mathcal{P}_{inc}$ constraints to describe that CS courses offered by a university can only be taken by students of the *same* university. In particular, the constraint

$$univ.CS.course.taken\_by \subseteq univ.CS.student$$

fails to capture the semantics precisely. Another limitation of $\mathcal{P}_{inc}$ is that it is not capable of expressing inverse constraints found in object-oriented databases.

These considerations motivated the formulation of another class of path constraints, denoted by $\mathcal{P}^p_{inc}$ [9, 11]. A $\mathcal{P}^p_{inc}$ constraint has one of the following forms:

$$\begin{array}{ll} \text{(form 1)} & (\alpha, \;\; \beta \subseteq \gamma), \\ \text{(form 2)} & (\alpha, \;\; l.\beta \rightleftharpoons l'.\gamma), \end{array}$$

where α, β, γ are regular path expressions, and l, l' are edge labels. A constraint of form 1 is interpreted by:

$$\forall x\,(\alpha(r, x) \rightarrow \forall y\,(\beta(x, y) \rightarrow \gamma(x, y))).$$

It asserts that for any nodes x and y, if x is reached by following α from root r and y is reached by following β from x, then y must also be reachable from x by following γ. A constraint of form 2 is interpreted by a pair of logic sentences:

$$\begin{array}{l} \forall x\,(\alpha.l(r, x) \rightarrow \forall y\,(\beta(x, y) \rightarrow \gamma(y, x))), \\ \forall x\,(\alpha.l'(r, x) \rightarrow \forall y\,(\gamma(x, y) \rightarrow \beta(y, x))). \end{array}$$

That is, for any x and y, if x is reached by following $\alpha.l$ (resp. $\alpha.l'$) from r and y is reached by following β (resp. γ) from x, then one can go back to x from y by following γ (resp. β). These capture the semantics of inverse relationship.

In $\mathcal{P}^p_{inc}$ we can express the following:

$$\begin{array}{l} (univ, \;\; CS.course.taken\_by \subseteq CS.student), \\ (univ.\_, \;\; course.taken\_by \rightleftharpoons student.taking). \end{array}$$

Here '_' is a wild card that matches any edge label. The first constraint states that courses offered by the CS department of a university are taken by CS students in the same university. The second describes an inverse relationship, i.e., in *any* department at a university, if a course c is taken by a student s, then s is taking c, and vice versa. It should be noted that $\mathcal{P}_{inc}$ constraints are a special case of $\mathcal{P}^p_{inc}$ constraints, namely, when α is the empty path ϵ. We shall omit α if it is ϵ.

Path constraints are important not only because they provide some form of semantic integrity. They are also useful in query optimization. This applies to both semistructured and structured data. As an example, consider the following queries that are to find students who are taking a course with Kate.

```
Q1: select  S2
    from    r.student S1, r.student S2, S1.taking C
    where   "Kate" in S1.name and C in S2.taking
Q2: select  S'
    from    r.student S, S.taking C, C.taken_by S'
    where   "Kate" in S.name
```

The query plan implicit in the query Q1 requires two iterations over students – with S1 and S2 – whereas the query Q2 requests only one iteration over this potentially very large set – with S. Given the simple path constraints on students and courses presented in Section 1, one can verify that Q1 and Q2 are equivalent. Thus, if Q1 is requested, we may use a query plan that iterates only once over the set of students. We should emphasize that this is just a glance at the optimization issue. We should also remark that to show the equivalence of Q1 and Q2, one needs to show that the following is implied by the given constraints, i.e., to consider constraint implication in query optimization:

$$student.taking.taken\_by \subseteq student.$$

Constraints of $\mathcal{P}_{inc}$ and $\mathcal{P}^p_{inc}$ are consistent. That is, given any $\mathcal{P}_{inc}$ or $\mathcal{P}^p_{inc}$ constraints, one can always find a (finite) semistructured data graph that satisfies them. When it comes to implication, the analysis is no longer simple. The implication problem for $\mathcal{P}_{inc}$ has been shown to be decidable in exponential space in terms of the size of constraints [3]. Better still, when the path expressions in constraints are restricted to simple paths, i.e., sequences of edge labels, the problem is decidable in polynomial time. However, the implication problem is undecidable for $\mathcal{P}^p_{inc}$, the mild generalization of $\mathcal{P}_{inc}$. Worse still, the problem remains undecidable even when the path expressions in constraints are simple paths [9]. Several practical and decidable cases of the implication problem were identified in [9, 11].

Path functional constraints are generalizations of functional dependencies. They were studied for nested relational and object models [17, 18, 25], i.e., in the presence of type constraints. One would be tempted to think that the complexity results for constraint implication established in the typed context would also hold for semistructured data, i.e., in the absence of types. In the typed context one considers graphs whose structures are constrained by types, whereas for semistructured data one considers arbitrary graphs free of type constraints. This leads to the question about the impact of types on the implication analysis of path constraints. If the presence of types would make the analysis harder, then undecidability results developed in the typed context could carry over to the untyped context. If it would simplify the analysis, then decidability results in the typed context would still hold for semistructured data. However, the interaction between type and path constraints is far more intriguing.

It has been shown that the presence of types may in some cases simplify the implication analysis of path constraints and in other cases make it harder [8]. To illustrate that types may simplify reasoning about path constraints, let us consider a restriction of the object-oriented type system given in Section 1, defined by:

$$\begin{array}{rcl} t & ::= & b \mid C \\ \tau & ::= & t \mid [l_1 : t_1, \ldots, l_n : t_n] \end{array}$$

where b and C are atomic and class types, respectively. This type system supports the record construct only. When a type defined in this system is imposed on the data, the graph representing the data has a rather simple regular structure. For constraint implication in this typed context, we only consider graphs of these simple structures, which yields a simpler analysis. Indeed, let us consider the class of $\mathcal{P}^p_{inc}$ constraints in which path expressions are restricted to be simple paths, referred to as $\mathcal{P}^s_{inc}$. In this typed context, the implication problem for $\mathcal{P}^s_{inc}$ is decidable in $O(n^3)$ time [8], whereas it is undecidable in the context of semistructured data [9].

On the other hand, there is a path constraint implication problem that is decidable in the untyped context but it becomes undecidable in a typed context. In particular, let us consider a set Σ of $\mathcal{P}^s_{inc}$ constraints of the form: $(\alpha_0.\rho,\ \beta \subseteq \gamma)$, where α_0 is a fixed path, and ρ is either a fixed edge label l_0, or a path that does not contain l_0. Let φ be a constraint of this form with $\rho = l_0$. We are interested in the implication problem to determine whether for any graph satisfying Σ, it must also satisfy φ. This is a problem that originates in practical applications [8]. In the untyped context, this implication problem is decidable in polynomial time. In contrast, if we consider only those graphs constrained by the object-oriented type system given in Section 1, then the problem becomes undecidable [8]. In other words, in this case the presence of types complicates the analysis of path constraint implication.

Another practical restriction on semistructured data graphs is a deterministic edge relation. For data found in many applications, the graph representing the data is *deterministic*, i.e., the edges emanating from each node in the graph have distinct labels. For example, when modeling Web pages as a graph, a node stands for an HTML document and an edge represents a link with an HTML label from one document (source) to another (target). It is reasonable to assume that the HTML label uniquely identifies the target document. Even if this is not literally the case, one can achieve this by including the URL (Universal Resource Locator) of the target document in the edge label. This yields a deterministic graph. This restriction simplifies the analysis of path constraint implication. Indeed, if we consider deterministic graphs only, then the implication problem for $\mathcal{P}^s_{inc}$ becomes decidable, even if we allow wild cards in path expressions. However, this deterministic restriction does not reduce the analysis of path constraint implication to a trivial problem. In particular, in the context of deterministic graphs the implication problem for $\mathcal{P}^p_{inc}$ remains undecidable [10].

3 Keys and foreign keys for XML

The first and simplest form of constraints we encounter in relational database is that of a *key* [2]. Keys are used to provide a "canonical identifier" for a data element; they are important in query optimization; and they are used in *foreign key* constraints – one of the most widely used forms of integrity constraints. If XML documents are to be considered as databases, we shall need keys for them, yet only recently has there been any systematic study of keys for XML or semistructured data.

Both the XML specification itself [5] and XML Schema [24] contain some form of key specification. In the XML standard, ID attributes in DTD provide a rather simple notion of keys, which are global and unary. In XML Schema, key specification depends on XPath [12], a rather complex language that makes reasoning about path inclusion, and hence key implication, difficult. This problem is compounded by other technical details in the XML Schema specification. Moreover in neither specification can one express *relative keys*, which we shall briefly describe below. See [6] for a more detailed discussion of these issues.

A simple form of key constraints is proposed in [15] using ordinary XML attributes, i.e., attributes with a string value referred to as *single-valued attributes*. A key of [15] has the general form:

$$\tau(X) \to \tau,$$

where τ is an element type and X is a *set* of attributes of τ. An XML document tree T satisfies the key if and only if for any two τ nodes in T, if they agree on the values of their X attributes, then they must be the same node. A foreign key is specified with an inclusion constraint and a key:

$$\tau_1[X] \subseteq \tau_2[Y], \quad \tau_2(Y) \to \tau_2,$$

where τ_1, τ_2 are element types, and X, Y are *lists* of single-valued attributes of τ_1, τ_2, respectively. An XML tree T satisfies the foreign key if and only if it satisfies the key and moreover, for any τ_1 node x there exists a τ_2 node y in T such that the values of the X attributes of x match the values of the Y attributes of y. This asserts that the set of attributes X of τ_1 elements is a foreign key referencing the key attributes Y of τ_2 elements.

Let us refer to the class of constraints proposed in [15] as $\mathcal{K}_{att}$. Keys and foreign keys defined in terms of a single attribute are referred to as *unary* keys and foreign keys. We write $\tau[l]$ and $\tau(l)$ as $\tau.l$ for any attribute l.

In $\mathcal{K}_{att}$ one can express keys and foreign keys found in relational databases. For example, assuming that `SSN` and `cno` are attributes of element type `enroll`, and `SSN` is an attribute of `student`, we can write

$$\begin{array}{ll} enroll(SSN, cno) & \to enroll, \\ student.SSN & \to student, \\ enroll.SSN & \subseteq student.SSN. \end{array}$$

That is, (`SSN`, `cno`) is a key of `enroll`, `SSN` is a key of `student` and also a foreign key of `enroll` referencing `student`. The value of an `SSN` attribute of `student` is unique among `student` elements instead of in the entire document. An `SSN` attribute of `enroll` references `student` elements only. Thus, unlike ID and IDREF attributes in DTDs, keys and foreign keys of $\mathcal{K}_{att}$ are scoped within a class of elements.

To capture the semantics of IDREFS attributes in DTDs, $\mathcal{K}_{att}$ includes *set-valued* foreign keys:

$$\tau_1.l_1 \subseteq_S \tau_2.l_2, \quad \tau_2.l_2 \to \tau_2,$$

where τ_1, τ_2 are element types, l_2 is an ordinary (single-valued) attribute of τ_2, while l_1 is a *set-valued* attribute of τ_1, i.e., an attribute whose value is (like IDREFS attributes) interpreted as a set of strings. It asserts that for any τ_1 node x and any string s in the l_1 attribute of x, there exists a τ_2 node y such that s matches the value of the l_2 attribute of y. To express object identifiers (oids) and inverse constraints in object-oriented databases, $\mathcal{K}_{att}$ also includes *ID* and *inverse constraints*. In contrast to oids, ID attributes are value-based, user-specified and mutable. To convert an object-oriented database to XML, one needs *ID* constraints to capture the semantics of oids. In short, $\mathcal{K}_{att}$ aims at capturing important database constraints with a minimum extension to XML DTDs.

Roughly following the notion of a key in XML Schema, key specifications based on path expressions have also been studied in [6], where two notions of keys were introduced, namely absolute and relative keys. An *absolute key* has the general form:

$$(Q, \quad \{P_1, \ldots, P_l\}),$$

where Q is a path expression called the *target path* and $\{P_1, \ldots, P_l\}$ is a set of path expressions called the *key paths*. In an XML tree, Q identifies a set of nodes on which the key is defined, i.e., the set of nodes reachable from the root by following Q, denoted by $[\![Q]\!]$. The key paths emanate from nodes of $[\![Q]\!]$ and they provide an identification for nodes in $[\![Q]\!]$, along the same lines as key attributes in relational database keys. The tree satisfies the key if and only if for any two nodes n_1, n_2 in $[\![Q]\!]$, if they have all the key paths and agree on them, then they must be the same node. By agreeing on a path P we mean that there exist two nodes n'_1, n'_2 reachable by following P from n_1, n_2, respectively, such that n'_1 and n'_2 have the same *value*. Note that two notions of equality are needed here because of the tree semantics of XML data: *value equality* when comparing nodes at the ends of key paths, and *node identity* when comparing nodes in $[\![Q]\!]$. Value equality is elaborated in [6]. We should remark that it is not required that all nodes in $[\![Q]\!]$ must have all the key paths. The key has no impact on those nodes at which some key path is missing. It is also possible that a key path may lead to multiple nodes. This key specification captures the semistructured nature of XML data [1].

Let us refer to the class of absolute keys as $\mathcal{K}_{path}$. For example, the following are keys in $\mathcal{K}_{path}$:

$$\begin{array}{l} (\_* .university.student, \quad \{firstname, \ lastname\}), \\ (\_*, \quad \{id\}). \end{array}$$

Again '$\_*$' is a combination of wild card and the Kleene closure that matches any path. The first key asserts that `firstname` and `lastname` are a key for university students, no matter where they occur in an XML document tree. That is, if two students have `firstname` and `lastname` subelements and the values of these subelements are pairwise equal, then they must be the same student. In other words, the subelements uniquely identify a university student in the document. The second key asserts that any element that has `id` subelements is uniquely identified by the values of the `id`'s. That is, any two nodes are disjoint on their `id` fields up to value-equality. Note that an `id` element does not have to have an `id` itself. This key captures the semantics of an ID attribute in DTDs. We should remark that keys expressed in $\mathcal{K}_{path}$ are scoped within a class of elements, and moreover, this specification of keys is orthogonal to any typing specification (e.g. DTDs).

Path expressions in keys of $\mathcal{K}_{path}$ are defined by:

$$\rho \ ::= \ \epsilon \ | \ l \ | \ \rho.\rho \ | \ \_*$$

Here l is a node label. This path language not only allows us to express important keys for XML, but also yields a low complexity ($O(n^2)$ time [7]) for determining equivalence and containment of its path expressions.

There are many situations in which a key provides only a relative specification. For example, in relational database design, the key of a weak entity is made up of the key of the "parent" entity and some additional identification [23]. Similarly, in many scientific data formats there is a hierarchical key structure in which subelements are located relative to some parent node. To describe this, a notion of *relative keys* was introduced in [6], with a general form:

$$(R, \ (Q, \ S)),$$

where R is a path expression that identifies a set of nodes $[\![R]\!]$ comparable to "parent" entities, and $(Q, \ S)$ is a key for every "sub-document" rooted at a node in $[\![R]\!]$. For example, consider

$(book, \ \{name\})$,
$(book, \ (chapter, \ \{number\}))$,
$(book.chapter, \ (section, \ \{number\}))$.

The first constraint is an absolute key in $\mathcal{K}_{path}$, which asserts that a book `name` uniquely identifies a `book` in a document. The second constraint states that chapter `number` uniquely identifies a `chapter`, but only within a book. In other words, chapter `number` is a key of `chapter` relative to `book`. To identify a `chapter` in the entire document, one also needs a key of book, e.g., book `name`. The third constraint is also a relative key, which asserts that section `number` uniquely identifies a `section` within a chapter. It should be noted that absolute keys are a special case of relative keys, namely, when the path expression R is the empty path. Relative keys are important for hierarchically structured data, including but not limited to XML documents.

Functional, inclusion and inverse constraints have been defined for XML in terms of simple paths [15], i.e., sequences of node labels. A simple path is interpreted with respect to a given set Σ of $\mathcal{K}_{att}$ constraints. In particular, an attribute in the path is treated as a reference to τ elements if it is a foreign key in Σ that references τ elements. In other words, a path may navigate across different XML subtrees. For example, given that `taking` is a foreign key of `student` referencing `course`, `taken_by` of `course` references `student`, `teaching` of `prof` references `course` and `taught_by` of `course` references `prof`, one can write:

$univ.student.SSN \rightarrow univ.student.taking$,
$univ.course.taken\_by \subseteq univ.student$,
$student.taking.taught\_by \rightleftharpoons prof.teaching.taken\_by$.

The first constraint is a *path functional constraint* asserting that `SSN` of a university student determines the courses that the student is taking. That is, if two students have the same SSN, then they take the same courses. The second constraint is a *path inclusion constraint* stating that courses offered by a university are taken by university students. The third constraint is a *path inverse constraint* asserting that if a student s is taking a course taught by professor p, then p is teaching a course taken by s, and vice versa.

Consistency and implication of XML constraints have been studied in the absence of DTDs and other typing constraints. In this context the consistency analysis is simple: for any $\mathcal{K}_{att}$ or $\mathcal{K}_{path}$ constraints there always exist an XML document satisfying them. For implication, it was shown [7] that there is a sound and complete set of inference rules for $\mathcal{K}_{path}$ constraints, which includes the following rules among others:

(superkey): If $(Q, \ S)$ and $S \subseteq S'$, then $(Q, \ S')$.

(containment): If $(Q, \ S)$ and $Q' \subseteq Q$, then $(Q', \ S)$.

The first rule states that if a set of paths S is a key for $[\![Q]\!]$, then so is any superset of S. This also holds for keys in relational databases. The second rule says that if path Q' is contained in Q, i.e., any nodes reached by following Q' are also reachable by following Q, then any key for $[\![Q]\!]$ is also a key for $[\![Q']\!]$. This rules cannot find a counterpart in relational databases. The implication problem for $\mathcal{K}_{path}$ constraints is decidable in $O(n^3)$ time in terms of the size of constraints [7]. For $\mathcal{K}_{att}$, it has been shown [15] that the implication problem is undecidable in general, but it becomes decidable in $O(n)$ time if only unary constraints are considered. In addition, implication of path functional, inclusion and inverse constraints by unary $\mathcal{K}_{att}$ constraints is decidable in $O(n^2)$ time. That is the problem to determine whether a path constraint is necessarily satisfied when a given set of unary $\mathcal{K}_{att}$ constraints is known to hold.

In the presence of DTDs or other type systems, it becomes more difficult to reason about constraints. To illustrate this, let us consider a simple key $\varphi = (X, \{ \})$ in $\mathcal{K}_{path}$ and a simple DTD D:

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

There exist an XML tree that conforms to the DTD (Fig. 3 (a)), and another tree that satisfies the key (Fig. 3 (b)). However, no XML tree can both conform to D and satisfy φ, because D requires an XML tree to have two distinct X nodes immediately under the root, whereas φ allows at most one. This shows that DTDs may interact with integrity constraints. In contrast, the consistency analysis in relational databases is trivial: one can write arbitrary (`primary`) `key` and `foreign key` specifications in SQL, without worrying about consistency. The interaction between DTDs and $\mathcal{K}_{att}$ constraints has recently been studied. Preliminary results show that the interaction is highly intricate [14].

Keys as specified in XML Schema [24] also interact with types. To the best of our knowledge, consistency and implication of constraints of XML Schema have not been studied, either in the absence or presence of types.

4 Research directions

For further research, a host of issues deserve investigation. First, functional and inclusion constraints with

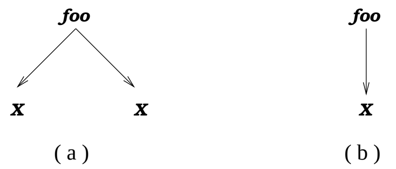

Figure 3: Impacts of DTDs and XML keys

regular path expressions need to be studied for XML. These are particularly important if one wants to develop a design theory for XML specifications as we do for normalization in relational database design. These constraints may not be a straightforward generalization of functional and inclusion constraints considered in the relational dependency theory, because of the tree semantics of XML data and its related equality issues.

A second issue concerns consistency and implication of XML constraints as well as their interaction with types. These questions are still open in connection with constraints defined in XML Schema.

Third, a practical project is to use integrity constraints to distinguish good XML design (specification) from bad design, along the lines of normal forms for relational schemas.

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 *PODS'97*, pages 122–133.

[4] C. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu. XML-based information mediation with MIX. In *SIGMOD'99*, pages 597–599.

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

[6] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. Draft manuscript, 2000.

[7] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about keys for XML. Draft manuscript, 2000.

[8] P. Buneman, W. Fan, and S. Weinstein. Interaction between path and type constraints. In *PODS'99*, pages 56–67.

[9] P. Buneman, W. Fan, and S. Weinstein. Path constraints on semistructured and structured data. In *PODS'98*, pages 129–138.

[10] P. Buneman, W. Fan, and S. Weinstein. Query optimization for semistructured data using path constraints in a deterministic data model. In *DBPL'99*.

[11] P. Buneman, W. Fan, and S. Weinstein. Path constraints in semistructured databases. *JCSS*, 61(2):146–193, 2000.

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

[13] A. Deutsch, L. Popa, and V. Tannen. Physical data independence, constraints, and optimization with universal plans. In *VLDB'99*, pages 459–470.

[14] W. Fan and L. Libkin. On integrity constraints for XML in the presence of DTDs. Draft manuscript, 2000.

[15] W. Fan and J. Siméon. Integrity constraints for XML. In *PODS'00*, pages 23–34.

[16] D. Florescu, L. Raschid, and P. Valduriez. A methodology for query reformulation in CIS using semantic knowledge. *Int'l J. Cooperative Information Systems (IJCIS)*, 5(4):431–468, 1996.

[17] C. S. Hara and S. B. Davidson. Reasoning about nested functional dependencies. In *PODS'99*, pages 91–100.

[18] M. Ito and G. Weddell. Implication problems for functional constraints on databases supporting complex objects. *JCSS*, 50(1):165–187, 1995.

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

[20] J. Melton and A. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufman, 1993.

[21] L. Popa, A. Deutsch, A. Sahuguet, and V. Tannen. A chase too far? In *SIGMOD'00*, pages 273–284.

[22] R. G. Cattell et al. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

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

[24] H. S. Thompson et al. XML Schema Part 1: Structures. W3C Working Draft, Apr. 2000. `http://www.w3.org/TR/xmlschema-1/`.

[25] M. van Bommel and G. Weddell. Reasoning about equations and functional dependencies on complex objects. *TKDE*, 6(3):455–469, 1994.