A Unified Constraint Model for XML

Wenfei Fan
Temple University
fan@joda.cis.temple.edu

Gabriel M. Kuper
Bell Laboratories
kuper@research.bell-labs.com

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

ABSTRACT

Integrity constraints are an essential part of modern schema definition languages. They are useful for semantic specification, update consistency control, query optimization, information preservation, etc. In this paper, we propose UCM, a model of integrity constraints for XML that is both simple and expressive.

Because it relies on a single notion of keys and foreign keys, the UCM model is easy to use and makes formal reasoning possible. Because it relies on a powerful type system, the UCM model is expressive, capturing in a single framework the constraints found in relational databases, object-oriented schemas and XML DTDs. We study the problem of consistency of UCM constraints, the interaction between constraints and subtyping, and algorithms for implementing these constraints.

1. INTRODUCTION

XML has become the universal format for the representation and exchange of information over the Internet. In many applications, XML data is generated from legacy repositories (relational or object databases, proprietary file formats, etc.), or exported to a target application (Java applets, document management systems, etc.). In this context, integrity constraints play an essential role in preserving the original information and semantics of data. The choice of a constraint language is a sensitive one, where the main challenge is to find an optimal trade-off between expressive power (How many different kinds of constraints can be expressed?) and simplicity (Can one reason about these constraints and their properties? Can they be implemented efficiently?). The ID/IDREF mechanism of XML DTDs [3] (Document Type Definitions) is too weak in terms of expressive power. On the other hand, XML Schema [17] features a very powerful mechanism with three different forms of constraints, using full XPath expressions, and therefore the reasoning and implementation of XML Schema constraints has a high complexity.

WWW10, May 1-5, 2001, Hong Kong.
ACM 1-58113-348-0/01/0005.

In this paper, we introduce UCM, a model of integrity constraints for XML. UCM relies on a single notion of keys and foreign keys, using a limited form of XPath expressions. The main idea behind UCM is a tight coupling of the integrity constraints with the schema language. This results in a model which is both simple and expressive enough to support the classes of constraints that are most common in practice. UCM constraints are easy to manipulate in theory: we study the consistency of UCM schemas and how their constraints interact with subtyping. UCM constraints are easy to manipulate in practice: we illustrate their use with a number of examples and give simple algorithms for their implementation. In particular, we make the following technical contributions:

- We extend the type system of [13], along with the subtyping mechanism of [15], with a notion of keys and foreign keys. This constitutes UCM, a schema language for XML with integrity constraints.
- We show that UCM schemas can capture relational constraints, object-oriented models (with object identity and scoped references), and the ID/IDREF mechanism of DTDs.
- We show that, as for XML Schema, deciding consistency over full UCM schemas is a hard problem. We then propose a practical restriction over UCM schemas that guarantees consistency. This restriction is general enough to cover both the relational and object-oriented cases.
- We propose an algorithm for propagating constraints through subtyping. This mechanism is the basis for supporting the notion of object-identity of object models within UCM schemas.
- We present algorithms for schema validation in the presence of UCM constraints

The paper is organized as follows. Section 2 illustrates the issues involved through examples from relational, OO and XML data sources. Section 3 presents the UCM constraint model by means of examples, showing how to represent DTDs, relational and ODMG schemas. Section 4 gives the complete syntax and semantics of UCM schemas. Section 5 considers the formal properties of the model, notably the consistency of UCM schemas with constraints and the interaction between subtyping and constraints. Section 6 describes some basic algorithms for using UCM constraints, illustrating the feasibility of the approach. Section 7 concludes the paper and outlines future research directions.

2. INTEGRITY CONSTRAINTS IN EXISTING MODELS

The need to handle integrity constraints originating in a variety of different application domains imposes strong requirements on the expressive power of the model. To give a flavor of the various types of constraints that one needs to capture, we give examples of integrity constraints in some of the most popular data models, namely relational, object-oriented schemas and DTDs.

Capturing constraints from legacy sources

EXAMPLE 2.1. Our first example is a relational database with two tables, one for companies and one for the departments in the companies, whose schema is defined with the following SQL statements.

```
CREATE TABLE Company ( co      CHAR(20),
                       stock   REAL,
                       PRIMARY KEY (co) )
CREATE TABLE Dept ( dname   CHAR(20),
                    co      CHAR(20),
                    topic   CHAR(100),
                    PRIMARY KEY (dname,co),
                    FOREIGN KEY (co)
                      REFERENCES Company(co) )

INSERT INTO Company ( co, stock )
       VALUES( "Locent", 25;
               "IT&T", 25;
               "Maxihard", 25 )
INSERT INTO Dept ( dname, co, topic )
       VALUES( "11358", "Locent", "Databases";
               "11279", "Locent", "Languages";
               "11358", "IT&T", "Visualization" )
```

Note that each table comes with a structural specification, as well as with integrity constraints. The specification of keys and foreign keys is an essential part of a relational schema: they prevent erroneous updates, and are used for the choice of indices and for query optimization [16]. In the above schema, the name of the company (attribute `co`) is a key for the table `Company`, i.e., each row must have a distinct value for attribute `co`. Hence, the name of the company can be used to *identify* the company. A foreign key imposes the requirement that values of a particular (sequence of) attribute(s) in one relation must match the values of some (sequence of) attribute(s) in another relation. For instance, the `co` attribute of the table `Dept` must be a valid company name in the table `Company`. Foreign keys provide the means to represent *references* within the relational model. □

EXAMPLE 2.2. The same information can be represented in an object database using the following schema, written using the ODMG data definition language [6]:

```
class Company
(key  co)
{ attribute String co;
  attribute Float stock; }

class Dept
(key  (dname,co))
{ attribute String dname;
  attribute Company co;
  attribute String topic; }
```

□

In the ODMG model, every object has an identifier (Oid), which is unique across the whole database. This is a significant departure from the relational model, where keys are local to a table: in the above example, objects of class `Dept` and `Company` must all have distinct Oids. Oids can be used as a reference to the object. For instance, attribute `co` of class `Dept` is a *reference* to an object of class `Company`. The ODMG model also supports a notion of a local key (e.g., attribute `co` for the class `Company`).

It is important to note that in the context of information integration, both of these models, along with XML documents exported from other sources, may occur in a single XML database. For instance, some tables in a relational form may coexist with objects. This means that the constraint model must deal with several different sorts of constraints in the same framework, and therefore that the results presented in [12] are not directly applicable.

Reasoning with XML constraints

Integrity constraints have been extensively studied in the relational database context [1, 16], which is a much simpler model than XML. Despite this, the experience with the relational mode shows that reasoning about constraints is a non-trivial task. Simple constraint languages can have high complexity, the best-known example of this being the undecidability of implication for functional and inclusion dependencies (the most widely used relational constraints).

For DTDs, determining whether a specification is consistent or not requires a complex analysis of the interaction between structural constraints, keys, and foreign key constraints [11]. A number of restricted cases with good complexity properties are proposed in [12], but none of the corresponding languages can capture all of the above uses of constraints in the same framework.

Following ideas in [11], one can easily define non-consistent XML Schema specifications. For instance, consider the following schema describing people and their parents.

```
<element name="root">
  <complexType>
     <xsd:group ref="Person" minOccurs="0"
                             maxOccurs="unbounded"/>
  </complexType>

  <key name="personName">
     <selector xpath="//person"/>
     <field xpath="@name"/>
  </key>
  <key name="parentName">
     <selector xpath="//parent"/>
     <field xpath="@name"/>
  </key>
  <keyref name="parent_isa_person" refer="personName">
     <selector xpath="//parent"/>
     <field xpath="@name"/>
  </keyref>
</element>

<group name="Person">
  <element name="person">
    <complexType><attribute name="name" type="string">
                 <element name="parent" ref="Parent">
                 <element name="parent" ref="Parent">
    </complexType>
```

```
  </element>
<group>
<group name="Parent">
  <element name="person">
    <complexType><attribute name="name" type="string">
    </complexType>
  </element>
<group>
```

Each person has two parents and an attribute name, and each parent has a pointer back to itself as a `Person`, which is described as a foreign key. This seemingly simple schema is actually inconsistent: there are no (non empty) documents that comply with it. The reason is that the key on `Parent` requires that each such object be uniquely identified by its name, and point to a `Person` with a different name. This means that the number of objects of type `Parent` is less that the number of objects of type `Person`, whereas the definition of `Person` implies that the number of objects of type `Parent` is at least twice the number of type `Person`.

The problem is made even harder by the fact that XML Schema [17] provides three different constraint mechanisms: ID/IDREF, unique constraints, and keys/foreign keys. Furthermore, it allows specifications using full XPath expressions, which include upward navigation as well as some form of recursion and function calls, each of these mechanisms having been introduced to simulate some of the constraints found in traditional models. As a result of this, even reasoning about consistency for these constraints is very hard.

3. UCM BY EXAMPLES

3.1 XML algebra support

The UCM model relies on the XML algebra of [13] for the structural part of the schema language and for the semantics of integrity constraints. This algebra uses a type system that is similar to other proposals [2, 8, 14] and that captures the structural aspects of XML schema [17]. In this section, we review those features of the algebra that we use, and extend the algebra to support ID values.

Documents and types

The XML algebra uses a "square brackets" notation for types and documents. For instance, the following XML document and DTD:

```
<companies>
  <company co="Locent">
     <stock>25</stock>
  </company>
  <company co="IT&T">
     <stock>25</stock>
  </company>
</companies>

<!ELEMENT companies company*>
<!ELEMENT company stock>
<!ATTLIST company co #PCDATA #required>
<!ELEMENT stock #PCDATA>
```

are represented in the algebra as:

```
type Companies = companies [ Company* ]
type Company   = company [ @co [ String ],
                           stock [ String ] ]
```

```
let doc0 : Companies =
      companies [ company [ @co [ "Locent" ],
                            stock [ "25" ] ],
                  company [ @co [ "IT&T" ],
                            stock [ "25" ] ] ]
```

The notation `doc0:Companies` indicates that document `doc0` is of type `Companies`. A tag prefixed by `@` corresponds to an attribute. The type system uses *regular expressions*, as in DTDs, with a `*` to indicate a collection of elements. `~` is a wildcard, meaning that any element name is allowed. Similarly, `@~` means that any attribute name is allowed.

Path expressions and for loops

The XML algebra uses path expressions for navigating in documents and `for` loops for iterating over them. The following expression accesses the content of the `co` attribute of each company:

```
query doc0/company/@co/data()

==> [ "Locent", "IT&T" ]
:   String*
```

The algebra supports a type inference algorithm which computes the type of each expression. In the examples, '`==>`' indicates the value of the expression and '`:`' its type (here a collection of `String` values). The `./data()` notation is used to access the atomic value of an element, playing a role similar to that of `./text()` in XPath.

The following `for` loop iterates over each company in the document, thus constructing a collection of elements, each of which has a tag `k` and contains the name of a company. Note that `for` loops can be used to express joins.

```
query for v in children(doc0) do
        k [ v/@co/data() ]

==> k [ "Locent" ], k [ "IT&T" ]
:   (k [ String ])*
```

Representing and accessing ID types

In order to support DTDs, we need to represent the type of an object ID, a notion that is not in the XML algebra. To do this, we simply add a new data type, with name `ID`, and provide a syntax for values of this type. The following example adds an attribute `compid` of type `ID` to the previous schema and document:

```
type Companies' = companies [ Company'* ]
type Company'   = company [ @compid [ ID ],
                            @co [ String ],
                            stock [ String ] ]

let doc0' : Companies' =
  companies [ company [ @compid [ ^c1 ],
                        @co [ "Locent" ],
                        stock [ "25" ] ],
              company [ @compid [ ^c2 ],
                        @co [ "IT&T" ],
                        stock [ "25" ] ] ]
```

We indicate values of type `ID` by using identifiers prefixed by `^`. In a similar way to the use of `./data()` for other atomic values, `ID` values can be accessed using `./ID()`, which selects all children with type `ID`. For instance, the following expression retrieves all ID values from companies, from within any attribute.

```
query doc0/company/@~/ID()

==> ^c1, ^c2
:   ID*
```

An important difference between UCM and XML schema is that the semantics of the ID type in UCM is no different from the semantics of any other data type: we shall see later how the uniqueness of ID values is enforced by an appropriate key constraint, and how referential integrity is enforced by an appropriate foreign key constraint.

3.2 Keys and foreign keys in UCM

We are now ready to write our first UCM constraints. The following description captures the structural part of the relational database we saw in the introduction:

```
schema rel =
  root Companies,Depts

  type Companies = companies [ Company* ]
  type Company   = company [ co [ String ],
                             stock [ Decimal ] ]
  type Depts     = depts [ Dept* ]
  type Dept      = dept [ dname [ String ],
                          co [ String ],
                          topic [ String ] ]
```

Note that each UCM schema has a root described by a type expression, in this example a sequence composed of the two tables. In order to represent the corresponding integrity constraints, we just have to declare appropriate keys and foreign keys:

```
  key Company  [| ./co/data() |]
  key Dept     [| ./dname/data(), ./co/data() |]

  foreign key  Dept [| ./co/data() |]
   references  Company [| ./co/data() |]
end
```

The first declaration corresponds to table Company's primary key. UCM constraints are similar in syntax and spirit to relational constraints. They are composed of a type name and a sequence of path expressions starting at the current node (.). Here, the key constraint states that for any two distinct objects of type Company their co sub-elements must have two different values. The foreign key states that any value of the co element in an object of type Dept is also the value of the co element in some object of type Company.

As a convenience, we allow keys to be named. For instance, the previous constraints could also be written as:

```
  key compk = Company  [| ./co/data() |]
  key deptk = Dept     [| ./dname/data(),
                          ./co/data() |]

  foreign key Dept [| ./co/data() |]
   references compk
```

But as opposed to XML Schema, UCM foreign keys do not have to refer to an explicitly declared key. We will see later on examples where it is useful to define foreign keys whose right-hand side is not declared, but can be inferred by the system.

As opposed to other approaches [4, 5], and especially XML Schema [17], UCM keys and foreign keys are defined over *type names*. The first argument for this choice is a logical one: type names play a role similar to table names in the relational model or to class names in object models. This makes them natural entities on which to add additional semantics by means of integrity constraints. The second argument is technical: (1) this approach takes advantage of the expressive power of the type system to define the set of elements on which a constraint applies, and (2) a minimal subset of XPath is then sufficient for the definition of components for keys and foreign keys.

3.3 Interaction between types and constraints

XML has a much more flexible type system than the relational model. Very often, XML documents have optional components, alternative structures, or allow repetition over certain sub-elements. Assume for instance, that companies and departments may have several alternative names (in attribute @co and @dname, as well as element co):

```
schema
  root companies [ Company* ], dept [ Dept* ]

  type Company = company [ @co [ String* ],
                           stock [ String ] ]

  type Dept      = dept [ @dname [ String* ],
                          co [ String* ] ]

  let doc0 : Companies =
    companies [ company [ @co [ "Locent",
                                "Locent Corp.",
                                "Lo. Corp." ],
                          stock [ "25" ] ],
                company [ @co [ "IT&T",
                                "IT&T Corp." ],
                          stock [ "25" ] ] ]
                dept [ @dname [ "Databases",
                                "BL1135" ],
                       co [ "Locent" ] ]

  key Company [| ./@co/data() |]
  key Dept    [| ./@dname/data(), ./co/data() |]

  foreign key Dept [| ./co/data() |]
   references Company [| ./@co/data() |]
end
```

We still want to be able to identify specific companies or departments, even though each of them may declare several variations of their name. The semantics of UCM constraints is such that any one of the values of attribute @co is considered to be a key for the company, and any pair of values (@dname,co) is a key for the department. For example, "Locent" and "Locent Corp." are both keys for Company, while ("Databases","Locent") and ("BL1135","Locent") are both keys for Dept. In the latter case, the foreign key then says that "Locent" must be *one* of the keys for some element of type Company.

One can exploit the typing of the XML algebra to better understand the structure allowed in elements of a key. In this example, the key for Company has a unique element defined by the path ./@co/data(). The type of information reached from Company through this path is String*, indicating several values, each of which will identify the company. The static typing of the XML Algebra can be used to enforce that components of a key are unique and not empty, although this restriction is not required for UCM constraints to work.

One must be more careful with foreign keys, for which it is important to check that the types of their components are type-compatible. in this example the type of each co element of Dept and of each @co attribute of Company are both String, so the key is valid. If, on the other hand, Dept had been declared as

```
type Dept      = dept [ @dname [ String ],
                        co [ Integer ] ]
```

then

```
foreign key Dept [| ./co/data() |]
 references Company [| ./@co/data() |]
```

would not be valid due to an incompatibility between the types of the corresponding keys.

3.4 The UrSchema with ID types

It is not surprising that one can capture relational constraints with a notion of keys and foreign keys. More surprising is the fact that UCM schemas can capture the semantics of the ID/IDREF mechanism. Once again, this is possible by exploiting the expressive power of the schema language, using a *generic* schema and imposing the appropriate constraints on values of type ID. This schema, called the UrSchema, describes all possible documents, enforcing uniqueness of ID, and referential integrity.

```
schema UrSchema =
    (* atomic types *)
  type UrScalar = String|Integer|Boolean

    (* standard attributes and elements *)
  type UrTree = ~[ UrAttForest, UrForest ]
  type UrAtt = @~[UrScalar*]

    (* collections of elements and attributes *)
  type UrForest = (UrScalar|UrTree|UrTreeID|UrRef)*
  type UrAttForest = (UrAtt|UrAttRef)*

    (* element identified with an ID *)
  type UrAttID = @~[ID]
  type UrTreeID = ~[ UrAttID, UrAttForest, UrForest ]

    (* references *)
  type UrRef = &[ID]
  type UrAttRef = @~[UrRef]

    (* root documents *)
  root UrTree*

   (* key constraint for uniqueness of ID *)
  key UrTreeID [| ./@~/ID() |]

   (* foreign key for referential integrity *)
  foreign key UrRef [| ./ID() |]
   references UrTreeID [| ./@~/ID() |]
end
```

The first part is similar to the UrTree type in the XML algebra, as used to captures XML Schema wildcards: trees are either leaves with atomic values (UrScalar), or elements with a name (~), any attributes (UrAttForest) and an arbitrary number of children (UrForest).

We extend the notion of UrForest to allow two other types of objects: trees with an ID, and references. A tree with an ID (UrTreeID) is basically an UrTree with a special attribute at the beginning corresponding to the ID of the object[1]. A reference is simply an ID with a special tag '&'. This syntactic separation between ID values that *identify* elements, and ID values that *references* them is necessary to avoid ambiguity in the schema, i.e.,to ensure that a given document cannot be typed in multiple ways.

The key toward the end of the definition of UrSchema ensures that no two distinct objects have the same ID value. Note that an attribute wildcard (@~) is used to access the ID value of each tree without requiring one to know the corresponding attribute name. The foreign key ensures that every reference points to an existing ID value in the document.

3.5 Subsumption between UCM Schemas

All well-formed documents are instances of the UrSchema and all UCM schemas are required to be *smaller* (in terms of subtyping) than the UrSchema. We model subtyping using the notion of subsumption introduced in [15]. Subsumption is a relation between two schemas, that relies on a mapping between their type names, and on inclusion between regular expressions over type names.

For instance, assume a schema with type Companies', as defined above, as a root. This schema is subsumed by the UrSchema, under the following *subsumption mapping*:[2]

```
Companies' <: UrTree
Company <: UrTreeID
...
```

For each two mapped types (here those in Companies and in UrTree), containment must hold between the respective element names (e.g., companies in ~), and their corresponding regular expressions must be contained under the given mapping (e.g., here UrTreeID* in UrAttForest,UrForest).

The reason for declaring a subsumption mapping is that it has an impact on the constraints that hold on the new schema. In our example, the fact that Company is subsumed by UrTreeID implies that all ID values in the company elements must be distinct. This constraint is derived from the key constraint that holds over elements of type UrTreeID. Propagation of constraints through subsumption in fact provides a mechanism that captures the nature of ID/IDREFs in DTDs (resp. object ids in object models): i.e., uniqueness across the whole document (resp. the whole database).

Finally, consider the ODMG schema of Example 2.2. Once again, it is straightforward to convert the structural part of an object schema into an UCM schema. We use one type name for each class, and map each data structure to a simple XML equivalent. We also add a constraint for each key and a foreign key constraint that restrict the scope of ID references. This allows us to capture *typed* object references, and results in the following schema:

[1] A more precise representation would require the use of unordered collections for attributes. To simplify the presentation, we deal only with ordered collections in this paper.

[2] Note that the subsumption mapping is limited to type names associated to elements. See [15] for more details on the properties of subsumption and its relationship to XML Schema.

```
schema COMPANY <: UrSchema =
  root Company*,Dept*

  type Company = tuple [ @oid [ ID ],
                         name [ String ],
                         stock [ Float ] ]

  type Dept = tuple [ @oid [ ID ],
                      name [ String ],
                      co [ &[ID] ],
                      topic [ String ] ]

  key Company [| ./name/data() |]
  key Dept [| ./name/data(), ./co/data() |]

  foreign key Dept [| ./co/&/ID() |]
   references Company [| ./@oid/ID() |]
end
```

Note the declaration of the subsuming schema (`UrSchema`) for the new schema (`COMPANY`). Once again, propagation of constraints from `UrSchema` makes sure that the `@oid[ID]` attributes behave like object ids, and that `&[ID]` elements behave like object references. The process of constraint propagation through subsumption is described in Section 5. Together with structured types, subsumption, and integrity constraints, UCM covers almost every aspect of the ODMG model, with the notable exception of multiple inheritance which can involve attribute renaming: this cannot be handled due to the structural nature of subsumption.

4. SYNTAX AND SEMANTICS OF UCM

4.1 Syntax of UCM schemas

The first part of the syntax is the type specification, which is summarized in Figure 1. This is similar to the syntax of types in the XML Algebra [13], with the addition of attributes, of the type ID, and of the special tag `&`. The syntax of types is based on attribute and element names, scalar types, and the ID type. One can give names to types, construct elements, attributes and references, and build *regular expressions* over types using sequence, choice, and repetition (Kleene star).

The second important part of the schema language is the subset of path expressions that are used to define the components of keys and foreign keys. Paths used in UCM are given in Figure 2. These are only very simple paths, that perform navigation by selecting children, elements, attributes, or values of a given node. Note the use of `./` to denote navigation from the current node. Remember that one can use wild cards for navigation, selecting all the elements or attributes, while disregarding their names. `./ID()` accesses all the nodes whose value is of type ID. This is indeed a very small subset of XPath [7], notably we do not allow: navigation among ancestors or siblings, predicates, recursive navigation (i.e., `//`), and function calls.

Finally, Figure 3 gives the syntax of keys, foreign keys, and top level schema declarations. As we have seen in the introduction, the definition of keys and foreign keys is composed of a type name and a sequence of path expressions. A schema is composed of a root, plus a number of type, key and foreign key declarations.

Note that the syntax allows one to define schemas that would be out of the scope of the XML model (e.g., a complex attribute definition like `@a[@b[ID]*]`. This allows us to keep the grammar as simple as possible, while avoiding such erroneous schemas by forcing UCM schemas to be subsumed by the `UrSchema`. By default, if no subsuming schema is declared, it is assumed to be the `UrSchema`.

We will call *element type names* of a schema S, the subset of type names X of S whose definition is of the form `type` X `=` $l[t]$. We will use `name()` and `regexp()` for the operations that access, for an element type name, the tag and the regular expression over its children.

4.2 Semantics of UCM schemas

We now describe the formal semantics of UCM schemas, in terms of the set of documents that they validate.

4.2.1 Databases.

Integrity constraints are used to identify nodes in XML documents. Therefore, we need to extend the data model of the XML algebra by a notion of node identity. In the following, we assume o to range over an infinite set of *OIDs* O. XML data is represented in the following simple data model.

DEFINITION 4.1. [database] A *database* consists of a sequence of documents, as written in the syntax given in Figure 4. Each document has a tree structure, in which each node (value, reference, attribute or element) has an associated OID o.

Note again that the grammar of Figure 4 allows invalid XML documents. In the following, we will assume all documents are *instances* – see definition below – of the `UrSchema`.

4.2.2 Path expressions.

In the XML Algebra, path expressions are defined using the more basic operations `children()` (that returns the list of children of a node), `for` loops, and `match` expressions. We will use the same static (typing) semantics for path expressions as the one given in [13], but we extend the evaluation semantics so it takes the notion of OID and the `ID` type into account.

DEFINITION 4.2. [value, tag, and children]
Let o be an OID. We write `val(`o`)` for the value associated with the node o, `name(`o`)` for the tag of the node o, and `children(`o`)` for the list of OIDs that are children of o (including attributes which appear at the begining, ordered alphabetically by name), respectively.

DEFINITION 4.3. [path expressions]
Now that `children()` is defined over OIDs, we can reuse the same definitions for path navigation as in the XML algebra. For lack of space, we only give the corresponding rule that deals with navigation among `ID` values. See again [13] for more details about the semantics of the `match` expression.

$$
\begin{array}{lcl}
e\,/\,\texttt{ID()} & = & \texttt{for}\ v_1\ \texttt{in}\ e\ \texttt{do} \\
 & & \quad \texttt{for}\ v_2\ \texttt{in children(}v_1\texttt{) do} \\
 & & \quad\quad \texttt{match}\ v_2 \\
 & & \quad\quad\quad \texttt{case}\ v_3\ :\ \ \mathrm{ID}\ \texttt{do}\ v_3 \\
 & & \quad\quad\quad \texttt{else ()}
\end{array}
$$

DEFINITION 4.4. [path sequences]
Last, we need to define the semantics of values accessed by the sequence of paths which compose keys and foreign

name	a		`a1` | `a2` | $\cdots$	
attributes, element name	l	::=	a	element name
		|	`@`a	attribute name
		|	`~`	element name wildcard
		|	`@~`	attribute name wildcard
		|	`&`	reference tag
type name	X		`X1` | `X2` | $\cdots$	
scalar type	s	::=	`Integer`	
		|	`String`	
		|	`Boolean`	
ID type	i	::=	`ID`	
type	t	::=	X	type name
		|	s	scalar type
		|	i	id type
		|	l`[`t`]`	element and attributes
		|	t `,` t	sequence
		|	t `|` t	choice
		|	t`*`	repetition
		|	`()`	empty sequence
		|	$\emptyset$	empty choice

Figure 1: Types

path	p	::=	l	name selection
		|	`data()`	scalar selection
		|	`ID()`	ID selection
		|	l `/` p	nested path
path sequence	ps	::=	`./`p	single path
		|	ps `,` `./`p	path sequence

Figure 2: Path expressions

key name	k		`k1` | `k2` | $\cdots$	
schema name	S		`S1` | `S2` | $\cdots$	
schema item	i	::=	`key` X `[|` ps `|]`	key
		|	`key` k `=` X `[|` ps `|]`	named key
		|	`foreign key` X `[|` ps `|]` `references` X `[|` ps `|]`	foreign key
		|	`foreign key` X `[|` ps `|]` `references` k	named foreign key
		|	`type` X `=` t	type declaration
root	r	::=	`root` t	root declaration
schema	U	::=	`schema` S `=` r $i \ldots i$ `end`	schema
		|	`schema` S `<:` S `=` r $i \ldots i$ `end`	subsumed schema

Figure 3: Keys, foreign keys, and UCM Schemas

integer	c_{int}	::=	$\cdots \mid -1 \mid 0 \mid 1 \mid \cdots$	
string	c_{str}	::=	`""` | `"a"` | `"b"` | $\cdots$ | `"aa"` | $\cdots$	
boolean	c_{bool}	::=	`false` | `true`	
ID	c_{ID}	::=	`^c1` | `^c2`	
constant	c	::=	$c_{\text{int}} \mid c_{\text{str}} \mid c_{\text{bool}} \mid c_{\text{ID}}$	
data	d	::=	c	scalar constant
		|	a`[`d`]`	element
		|	`@`a`[`d`]`	attribute
		|	`&[`d`]`	reference
		|	d `,` d	sequence
		|	`()`	empty sequence
database	db	::=	d `,` d `,` $\cdots$	document sequence

Figure 4: XML data

keys. Recall from Section 3.3, that key components are actually compared through a cross product semantics. This is captured using a series of nested `for` loops that iterate over each key component.

$$
\begin{array}{rcl}
e\texttt{/[|}\ p_1\ \texttt{,}\ \ldots\ \texttt{,}\ p_n\ \texttt{|]} & = & e\texttt{[|}\ \texttt{./}p_1\ \texttt{,}\ \ldots\ \texttt{,}\ \texttt{./}p_n\ \texttt{|]} \\
& = & \texttt{for}\ v_1\ \texttt{in}\ e\ \texttt{/}\ p_1\ \texttt{do} \\
& & \quad \ldots \\
& & \quad \texttt{for}\ v_n\ \texttt{in}\ e\ \texttt{/}\ p_n\ \texttt{do} \\
& & \quad\quad \texttt{k[}v_1\ \texttt{,}\ \ldots\ \texttt{,}\ v_n\texttt{]}
\end{array}
$$

This results in a sequence of *key elements* `k`, each containing a sequence of values that participate in the definition of a key or foreign key.

4.2.3 Equality.

Finally, our constraints rely on two different notions of equality: *node equality*, which is used to identify nodes in the document, and *value equality*, which is used to compare values of keys. Node equality is defined to be equality on OIDs. We assume that value equality is defined over atomic values in the straightfoward way.

DEFINITION 4.5. [value equality]
Let o_1 and o_2 be OIDs. $o_1 =_v o_2$ iff o_1 and o_2 contain two atomic values that are equal, or (1) o_1 and o_2 have the same tag, (2) $\texttt{attributes}(o_1) = \texttt{attributes}(o_2)$, and (3) if $\texttt{children}(o_1) = (o_1^1\ \texttt{,}\ \ldots\ \texttt{,}\ o_1^k)$ and $\texttt{children}(o_2) = (o_2^1\ \texttt{,}\ \ldots\ \texttt{,}\ o_2^l)$, then $k = l$ and $o_1^i =_v o_2^i$ for all $1 \leq i \leq k$.

4.2.4 Typing.

Typing corresponds to the structural part of schema validation. Following the approach of [2, 15], typing consists of finding a mapping, or *type assignment* from OIDs to type names for which names match, and for which the children verify the regular expression defining the type of the parent.

DEFINITION 4.6. [Typing]
Let `D` be a database and `S` a schema. We say `D` is of type `S` under the type assignment θ, and write $\texttt{D} :_\theta \texttt{S}$, iff θ is a function from the set of OIDs in `D` to the set of element type names `X1`, ..., `Xn` in `S` such that for each OID o

1. $\texttt{name}(o)$ satisfies the label (wildcard) of type $\theta(o)$, and
2. if $\texttt{children}(o) = (o_1\ \texttt{,}\ \ldots\ \texttt{,}\ o_m)$, then the word $\theta(o_1)$, ... , $\theta(o_m)$ is in the language defined by the regular expression of $\theta(o)$ over its element type names components.

Note that all the types involved in that definition must be *element type names* (i.e., describing elements), and require regular expressions to be over each *element type names*. The user syntax however, allows the use of *anonymous types*, by nesting sub-elements, and therefore typing also requires type names to be generated. But as type assignment is only an internal structure, this can be done by the system, transparently for the user. Whenever we need to talk about such system-generated type names, we use strings preceded by '_'. For example, the definition of the type `Company` could be mapped to:

```
type Company = company[_t1, _t2, _t3]
type _t1     = @compid[ID]
type _t2     = @co[String]
type _t3     = stock[String]
```

We assume that each schema is *unambiguous*, i.e., if θ exists, it is unique. This is a necessary assumption for the semantics of constraints, reasoning, and any practical implementation.

We write Models(`S`) for the set of databases of type `S`, i.e., $\{\texttt{D} \mid \exists\theta, \texttt{D} :_\theta \texttt{S}\}$.

We write $ext_{\texttt{D}}(\texttt{X})$ for the extension of type `X` (with respect to schema S), i.e., the set of objects of `D` of type `X`, or $ext_{\texttt{D}}(\texttt{X}) = \{o \mid o \in \texttt{D}, \theta(o) = \texttt{X}\}$.

4.2.5 Key and foreign key.

We now give the notion of satisfaction for keys and foreign keys.

DEFINITION 4.7. [Key satisfaction]
Let `S` be a schema, `X` a type of `S`, and k a key of `S` defined over type `X` with key component $\texttt{[|}\ \texttt{./}p_1\ \texttt{,}\ \ldots\ \texttt{,}\ \texttt{./}p_n\ \texttt{|]}$.

A database `D` satisfies the key k iff, for all OIDs o_1 and o_2 in $ext_{\texttt{D}}(\texttt{X})$, if there exist ke_1 in $o_1\texttt{/[|}\ p1\ \texttt{,}\ \ldots p_n\ \texttt{|]}$ and ke_2 in $o_2\texttt{/[|}\ p1\ \texttt{,}\ \ldots p_n\ \texttt{|]}$, such that $ke_1 =_v ke_2$, then $o_1 = o_2$.

DEFINITION 4.8. [Foreign key satisfaction]
Let `S` be a schema, `X` and `X'` types of `S`, and *fk* a foreign key of `S` from type `X` with component $\texttt{[|}\ \texttt{./}p_1\ \texttt{,}\ \ldots\ \texttt{,}\ \texttt{./}p_n\ \texttt{|]}$ to `X'` with component $\texttt{[|}\ \texttt{./}p'_1\ \texttt{,}\ \ldots\ \texttt{,}\ \texttt{./}p'_n\ \texttt{|]}$.

A database `D` satisfies *fk* iff, for all OIDs o in $ext_{\texttt{D}}(\texttt{X})$, and all ke in $o\texttt{/[|}\ p1\ \texttt{,}\ \ldots p_n\ \texttt{|]}$, then there exists o' in $ext_{\texttt{D}}(\texttt{X'})$ and ke' in $o'\texttt{/[|}\ p1\ \texttt{,}\ \ldots p_n\ \texttt{|]}$, such that $ke =_v ke'$.

4.2.6 Subsumption.

Recall from the object-oriented examples in the previous sections that a complete definition of UCM requires constraint propagation through subsumption. We borrow the definition of subsumption from [15]. Subsumption is a relationship between types that is strictly more expressive than subtyping in XML schema, while still being easy to manipulate. Subsumption relies on an idea similar to typing, i.e., it is defined through a mapping between type names, called a *subsumption mapping*.

DEFINITION 4.9. [Subsumption]
Let S and S' be two schemas. We say that schema S' *subsumes* S under the *subsumption mapping* θ, and write S $<:_\theta$ S', iff θ is a function from element type names in S to element type names in S', such that:

1. for all element type names X in S, name(X) is smaller[3] than name(θ(X)),
2. for all element type names X in S, $\theta(\mathcal{L}(\texttt{regexp(X)})) \subseteq \mathcal{L}(\texttt{regexp}(\theta(\texttt{X})))$, where $\mathcal{L}(r)$ is the language generated by regular expression r.
3. $\theta(\mathcal{L}(\texttt{regexp(root(S))})) \subseteq \mathcal{L}(\texttt{regexp(root(S'))})$.

We write S <: S' if there exists a θ such that S $<:_\theta$ S'.

4.2.7 Schema validation.

Finally, we define the notion of validation of documents by UCM schemas with constraints.

DEFINITION 4.10. [Validation of UCM schemas]
Let D be a database and S an UCM schema whose type is subsumed by S'. We say D validates S under the type assignment θ, and write D $::_\theta$ S, iff

1. D $:_\theta$ S,
2. D validates S',
3. for all key k in S, D satisfies k,
4. for all key *fk* in S, D satisfies *fk*.

The second condition is not as expensive as it may seem: we know already that S' subsumes S, and as a consequence we can deduce the extensions of the types in S' from the extensions of the types in S and from the subsumption mapping θ[4]. Therefore, we only need to check that D satisfies the *constraints* in the subsuming schema.

5. REASONING ABOUT CONSTRAINTS

In this section we study several forms of reasoning about constraints. As already pointed out, this is in general a difficult task. We therefore concentrate on finding practical solutions for two important problems in our context. The first problem is to tell whether a schema specified by the user makes sense or not, i.e., whether there exists at least one nonempty database that satisfies the schema. The second problem is the propagation of constraints through subsumption, which, as we have seen, is needed to capture object-oriented schemas.

[3] Where smaller is defined with the obvious meaning: a<a, a<~. etc.

[4] This is done by composition of subsumption and type mappings, see [15] for more details.

5.1 Consistency

The *consistency problem* for UCM schemas is to determine whether a given schema is consistent, i.e., whether there exists at least one nonempty database that satisfies the schema. This issue is important because one wants to know whether a schema specification makes sense.

Relational schemas (with keys and foreign keys) are always consistent if they are syntactically correct and do not have type mismatch. Under the same assumptions, object-oriented schemas are also consistent. But, as we have seen in Section 2, the situation is more complicated for XML schema specifications, as a schema can impose cardinality dependencies on elements, and these cardinality dependencies can interact in turn with the keys and foreign keys. As a result, the interaction between structural and integrity constraints makes the consistency analysis of XML schema languages much harder than it was for relational and object-oriented schemas.

PROPOSITION 5.1. The consistency problem is undecidable for UCM schemas, even when the paths in keys and foreign keys are restricted to be of length 1.

This undecidability result suggests that we look for restricted classes of UCM schemas for which the problem is decidable. As a first attempt, one might consider schemas with *unary* keys and foreign keys, i.e., ones that contain only one path. This is the case, for example, for keys and foreign keys specified with ID and IDREF in XML. Unary keys and foreign keys are commonly used and have been well studied for relational databases. In particular, Cosmadakis, Kanellakis and Vardi [9] have shown that the finite implication problem for unary keys and foreign keys is decidable in linear time in relational databases (in contrast to the undecidability of the problem for multi-attribute keys and foreign keys). Another restriction we might consider is the *primary key restriction*, which says that at most one key can be specified for any type in a schema. One might expect that the consistency problem would become much simpler under these restrictions, but that is not the case. Decidability itself is an open problem, but we can show that even consistency is decidable, the problem remains intractable:

PROPOSITION 5.2.

1. The consistency problem for UCM schemas is NP-hard when all keys and foreign keys are unary.
2. The consistency problem remains NP-hard for UCM schemas with unary keys and foreign keys, even when we allow at most one key on each type in the schema (the *primary key* assumption). □

Propositions 5.1 and 5.2 follow from similar results [11] for DTDs and (primary, unary) key and foreign key constraints. It should be mentioned that these results also hold for XML-Schema.

These negative results suggest that we consider restrictions on the type definitions instead. In particular, we would like to identify a class of UCM schemas that can express both relational and object-oriented schemas, but with consistency being decidable.

We identify a class of consistent UCM schemas as follows.

DEFINITION 5.3. A schema S is said to have the *database property* if it is of the form:

```
schema S =
  type root = X1*,..., Xn*
  type X1   = t1
  ...
  type Xn   = tn

  key  X [| p1,..., pn |]
  ...
  foreign key X [| p1,..., pn |]
   references Y [| p1',...,pn' |]
  ...
end
```

such that

- Xi does not appear in tj for any i, j;
- for any foreign key X [|p1,..., pn|] references Y [|p1',...,pn'|] in S, X/pi and Y/pi' have the same *unit type*, and the regular expression in the definition of this type does not use the union construct '|'. In addition, if *type*(X/pi) is ID, then pi has the form p'/&/ID() and pi does not appear in foreign key of X referencing Z for $Z \neq Y$.

Unit types are defined as either elements, scalar types or the ID type. These two restrictions are designed to avoid the complex interaction between typing and integrity constraints. By restricting the use of type names, the first condition also restricts the constraints that one can define in a schema. The second condition prevents having complex types in the key components.

PROPOSITION 5.4.
Let $\mathcal{C}$ denote the class of UCM schemas that have the database property. Then (1) All schemas in $\mathcal{C}$ are consistent, and (2) It is decidable in quadratic time whether a UCM schema is in $\mathcal{C}$. □

These restrictions might seem very strong, but they still cover a lot of practical cases:

PROPOSITION 5.5.
All relational and object-oriented database schemas can be expressed as schemas in $\mathcal{C}$. □

See the extended version of the paper [10] for sketches of the proofs of Propositions 5.4 and 5.5.

5.2 Constraints through subsumption

As explained above, the semantics of oids in an OO schema is captured in UCM by the constraints in UrSchema, i.e., by the fact that OIDs are unique, and that every reference is to an OID that is present in the database. The definition of a schema permits the user to reference keys without declaring them explicitly. For example, in Section 3.5, we described a schema where a Dept has a foreign key that references the value of @oid in Company, relying implicitly on the fact that UrSchema implies that the latter is a key.

In order to verify that the schema, as specified by the user, is indeed valid, we have to study the interaction between subsumption and integrity constraints. In order to do this, we first extend the notion of key to apply to unions of types, rather than just to single types. For example, we want the key on ID in UrSchema to imply that OIDs are unique over *all* objects in the user schema (more precisely, over those that have an ID), rather than just be unique over objects of a specific type. We write such keys with the syntax key (X1|...|Xj) [| ./p1, ..., ./pn |] The definition of satisfiability for multiple types is the same as the definition for single types, except that $ext_{\mathtt{D(X)}}$ is replaced by $ext_{\mathtt{D(X1)}} \cup \cdots \cup ext_{\mathtt{D(Xj)}}$.

Let S be a schema subsumed by schema S' (S<:S'). We have to check whether this declaration is valid. In order to check this, we need to verify that, for each foreign key foreign key Y [| ./p1, ..., ./pn |] references X [| ./q1, ..., ./qm |] in S, the right-hand side is indeed a key.

We do this by propagating keys from S' to S. This new set of keys, $K = K(\mathtt{S}, \mathtt{S}')$ is defined as follows. First, K contains all the keys of S. Then, for every key X' [| ./p1, ..., ./pn |] in S', let X1, ..., Xj be the set of types in S that are mapped by θ to the type X'. We then add the key (X1|...|Xj) [| ./p1, ..., ./pn |] to K.

PROPOSITION 5.6. Let S and S' be schemas, declared as S<:S' Then S is a valid schema declaration iff, for every foreign key

```
foreign key Y [| ./p1, ..., ./pn |]
 references X [| ./q1, ..., ./qm |]
```

in S, there is a key of the form

```
key (X1|...|Xj) [| ./q1, ..., ./qm |]
```

in $K(\mathtt{S}, \mathtt{S}')$, where X is in $\{\mathtt{X1}, \ldots, \mathtt{Xj}\}$. □

In the same way that we extended the definition of keys to include multiple types, we could also extend the definition of foreign keys. We would then obtain a nice correspondence between subsumption and keys. To show this, we define $FK = FK(\mathtt{S}, \mathtt{S}')$ to take foreign keys into account. Start with all the keys and foreign keys of S in FK, and add all the keys in $K(\mathtt{S}, \mathtt{S}')$ to FK. Then, for each

```
foreign key (Y1|...|Ym) [| ./p1, ..., ./pn |]
 references (X1|...|Xi) [| ./q1, ..., ./qm |]
```

in S', let X1, ..., Xn be the set of types in S which are mapped by θ to types in X1', ..., Xi', and let Y1, ..., Yj be the set of types in S which are mapped by θ to types in the set Y1', ..., Ym'. Add the foreign key

```
foreign key (Y1'|...|Yj') [| ./p1,...,./pn |]
 references (X1'|...|Xn') [| ./q1,...,./qm |]
```

to FK.

We can then show

PROPOSITION 5.7. Let D be a database and S an UCM schema of subsuming type S'. Then D validates S under the type assignment θ, iff $\mathtt{D} :_{\theta} \mathtt{S}$, and D satisfies all the keys in $FK(\mathtt{S}, \mathtt{S}')$.

EXAMPLE 5.8. We illustrate how OIDs are handled with the propagation mechanism. Consider the schema COMPANY from Section 3.5. With the system-defined types added, it becomes:

```
schema COMPANY <: UrSchema =
  root Company*, Dept*
```

```
  type Company = tuple [ _coid, _cname, _cstock ]
  type _coid   = @oid [ ID ]
  type _cname  = name [ String ]
  type _cstock = stock [ Float ]

  type Dept    = tuple [ _doid, _dname, _dco, _dtopic]
  type _doid   = @oid [ ID ]
  type _dname  = name [ String ]
  type _dco    = co [ _coref ]
  type _dtopic = topic [ String ]

  type _coref  = &[ID]

  key Company [| ./name/data() |]
  key Dept    [| ./name/data(), ./co/data() |]

  foreign key Dept [| ./co/&/ID() |]
   references Company [| ./@oid/ID() |]
end
```

We first observe that the declaration `COMPANY<:UrSchema` is valid, under the following subsumption mapping θ:

_coref	$\mapsto$	UrRef
_coid, _doid	$\mapsto$	UrAttID
_cname, _cstock, _dname, _dco, _dtopic	$\mapsto$	UrTree
Company, Dept	$\mapsto$	UrTreeID
Companies, Depts	$\mapsto$	UrForest

which then means that

```
key (Company | Dept) [| ./@~/ID() |]
```

is in K(COMPANY, UrSchema), and so the schema declaration is valid.

Alternatively, FK(COMPANY, UrSchema) contains the following keys and foreign keys, as well as those in COMPANY.

```
key Company|Dept [| ./@~/ID() |]
foreign key _coref [| ./ID() |]
 references Company|Dept [| ./@~/ID() |]
```

To test whether a document validates a schema, it therefore suffices to test that it satisfies these constraints, i.e., that it satisfies the keys and foreign keys in COMPANY and (1) object IDs are unique across all elements of type Company and Dept, and (2) every reference points to an ID of some element in Company or Dept. □

6. UCM IN PRACTICE

In this section, we describe simple algorithms for validating UCM schemas in the presence of integrity constraints. The objective is only to demonstrate the practical feasibility of our approach, not to present optimized algorithms.

We try to take as much advantage as possible of the coupling between integrity constraints and type information, in order to reduce the number of passes over the document. In a nutshell, we try to perform both typing and constraint checking within the same algorithm.

Since the use of keys in UCM is quite close to their use in relational databases, we can exploit relational techniques. In particular, while processing the keys, we build an index which maps the values of keys to the internal node id of the element. This index has two uses: verifying whether a given key has already been used and checking validity of foreign keys. Still, there are several aspects in which UCM diverges the from the relational model.

Anonymous Keys

First, we must take into account that the right-hand side of foreign keys can contain keys that are propagated via subsumption from existing keys, as explained in 5.2, but are not declared themselves as keys, and we must build indices for such keys as well. Note that the set of these keys can be determined at compile-time.

Cross-product semantics of constraints and typing.

Remember that because a key component can reach more than one value, the definition of the semantics of UCM constraints uses a cross product. As a consequence, there may be more than one key (in the same index) for the same node, and the generation of the index must take this into account.

Equality.

The operations on the index are `get(Index,value)` and `insert(Index,value,node)`. These rely on value equality, not node equality.

This said, let us look at the algorithm itself. During validation, the system assigns a type to each node. At the same time, the system also considers all key constraints k that apply to this type. For each such key there is a corresponding (global) index $I_{k,T}$, and the system calls `index_insert` on these indices. A pseudo-code description of this function is:

```
index_insert ( n:Node, I_kt:Index, p:list(Path) )
  key_values := algebra_eval(n/p);
  for kv in key_values do
    n' := get(I_kt, kv);
    if (n' = Fail) then
      insert (I_kt, kv, n)
    else
      if n' ≠ n then Error;
  endfor;
```

Subsumption

The constraints which need to be checked are not just those that are declared explicitly in the schema, but also those that arise due to subsumption. We must keep track (in compile-time) of which types get mapped to which subsuming types, and verify the appropriate constraints on the subsuming schema as well. In the following pseudo-code this is represented as a recursive procedure of obtaining the `super_type` of the current type, and reiterating until we reach the `UrTree`.

The pre-processing needed in order to take subsumption into account is therefore first to make sure that the right-hand side of the foreign keys are key constraints, then to build the the subsumption mapping, and to hook each type in the subsuming schema, for which a constraint holds, to the appropriate indices.

This is summarized in the following pseudo-code, in which `key(t)` returns true if a key has been defined for type `t`, and `get_indices` returns the set of indices (and corresponding paths) for keys that apply to this type.

```
(* main procedure *)
check_node ( n:Node, t: Type )

  (* subsumption first *)
  t' := super_type(t);
  if (key(t') and t ≠ UrTree)
  then check_node(n, t');

  ixs := get_indices(t);
  for ix in ixs do
    p := get_paths(ix);
```

```
    index_insert(n, ix, p);
  forend;
```

Foreign keys

Foreign keys are easier to handle. They are validated during a second path, so that we already know the extension of each type, and have a full index for all the keys.

```
check_foreign_key ( n:Node, p:list(Path), ix:Index )
  fkey_values := algebra_eval(n/p);
  for kv in fkey_values do
    n' := get(ix, kv);
    if (n' = Fail) then
      Error
  endfor;
```

7. CONCLUSION

We have proposed UCM, a schema language that supports the specification of structures, subtyping and integrity constraints for XML. UCM is simple, relying on a single notion of keys and foreign keys. UCM allows one to capture, in a unified framework, constraints commonly found in different application domains, including XML DTDs, relational and object-oriented schemas. We have also described preliminary results for the analysis of specification consistency, constraint propagation through subtyping, and schema validation.

This work is a step toward an expressive yet simple schema language for XML. We are currently working on a first implementation of the algorithms presented in Section 6, based on our XML Algebra prototype. One of the objective is to obtain some more precise performance analysis. On the more theoretical side, we plan to work on the reasoning about UCM constraints, including but not limited to, questions in connection with consistency and implication.

8. REFERENCES

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

[2] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *Proceedings of International Conference on Database Theory (ICDT)*, Lecture Notes in Computer Science, Jerusalem, Israel, Jan. 1999.

[3] 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/`.

[4] P. Buneman, S. Davidson, W. Fan, C. Hara, and W.-C. Tan. Keys for XML. in these proceedings., 2000.

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

[6] R. G. G. Cattell and D. Barry, editors. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann, 2000.

[7] J. Clark and S. DeRose. XML path language (XPath). W3C Recommendation, Nov. 1999. `http://www.w3.org/TR/xpath/`.

[8] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! In *Proceedings of ACM Conference on Management of Data (SIGMOD)*, pages 177–188, Seattle, Washington, June 1998.

[9] S. S. Cosmadakis, P. C. Kanellakis, and M. Y. Vardi. Polynomial-time implication problems for unary inclusion dependencies. *Journal of the ACM*, 37(1):15–46, Jan. 1990.

[10] W. Fan, G. M. Kuper, and J. Siméon. A unified constraint model for XML (full version), Feb. 2001.

[11] W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, Santa Barbara, CA, May 2001. to appear.

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

[13] P. Fankhauser, M. Fernández, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. The XML query algebra. W3C Working Draft, Feb. 2001. `http://www.w3.org/TR/query-algebra/`.

[14] H. Hosoya and B. C. Pierce. XDuce: an XML processing language. In *International Workshop on the Web and Databases (WebDB'2000)*, Dallas, Texas, May 2000.

[15] G. M. Kuper and J. Siméon. Subsumption for XML types. In *Proceedings of International Conference on Database Theory (ICDT)*, London, UK, Jan. 2001.

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

[17] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML schema part 1: Structures. W3C Working Draft, Feb. 2000.