Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification

François Bry and Sebastian Schaffert

Institute for Computer Science, University of Munich
http://www.pms.informatik.uni-muenchen.de

Abstract. The growing importance of XML as a data interchange standard demands languages for data querying and transformation. Since the mid 90es, several such languages have been proposed that are inspired from functional languages (such as XSLT [1]) and/or database query languages (such as XQuery [2]). This paper addresses applying logic programming concepts and techniques to designing a declarative, rule-based query and transformation language for XML and semistructured data.
The paper first introduces issues specific to XML and semistructured data such as the necessity of flexible "query terms" and of "construct terms". Then, it is argued that logic programming concepts are particularly appropriate for a declarative query and transformation language for XML and semistructured data. Finally, a new form of unification, called "simulation unification", is proposed for answering "query terms", and it is illustrated on examples.

1 Introduction

This article addresses applying logic programming to XML and semistructured data querying and transformation. This issue is of growing importance in both, practice and research. XML now is the data interchange standard of choice in application areas such as e-commerce, molecular biology, and astronomy. Furthermore, with the XML application XHTML, XML is becoming the preferred standard for textual web contents. Also recent languages for static and/or animated graphics such as SVG (Scalable Vector Graphics), X3D (an XML-based redefinition of the Virtual Reality Modelling Language VRML), and SMIL (Synchronised Multimedia Integration Language) are XML applications. "Native XML" database management systems are already marketed.

As a consequence, the design and implementation of selector and query languages for XML such as CSS selectors and XPath are premier concerns of the World Wide Web Consortium (W3C). Selector languages such as CSS selectors and XPath have been developed initially for style-sheet and/or transformation languages such as CSS and XSLT. Selector languages are "path-oriented", i.e. a node (i.e. a subterm) in a tree (i.e. a term) is specified in terms of a root-to-node path in the manner of the file selection formalisms of operating systems. Constructs from regular expressions languages such as $*$, $+$, $|$, ? and "wildcards" give rise to expressing node repetitions, options, and nodes with unspecified labels. These constructs are essential in selecting and/or querying XML and semistructured data, for a basic principle of XML and semistructured data is that data items, called "documents", do not have to conform to a predefined schema (expressed in XML in the DTD or XML Schema formalisms). This principle aims at facilitating the interchange of data in unnormed contexts like the World Wide Web. XML-like tree structured data items that do not necessarily conform to a schema are called "semistructured" in database research [3].

Query languages [4] inspired from SQL and OQL [5] have been developed for XML and semistructured data. Queries in these languages for XML and semistructured data in general consist of two parts: a query proper and a construct part. The query part expresses a selection of nodes (i.e. subterms) from a data item (i.e. term). Node selection is in general expressed in a path-oriented formalism à la XPath extended with "and" and "or" connectives. The construct parts serve to re-assemble into new terms (some of) the nodes (i.e. terms) selected in the query part. The construct part is also called restructuring or transformation part. For example, the following query from [6] (there with `books-with-prices` instead of `book-price-comparison`) expresses in XML Query [2], also called XQuery, the request to list for each book found as `entry` element at `amazon.com` and as `book` element at `bn.com` the book's title and the prices at both sources:

```
<book-price-comparison>
   { for $a in document("www.amazon.com/reviews.xml")//entry,
         $b in document("www.bn.com/bib.xml")//book
     where $b/title = $a/title
     return
        <book-with-prices>
           { $b/title }
           <price-amazon>{  $a/price/text() } </price-amazon>
           <price-bn>{  $b/price/text() } </price-bn>
        </book-with-prices>
    }
</book-price-comparison>
```

The query part is contained between the first `{` and `return`. The construct part is specified in the `book-with-prices` element. Note the node selection (expressed with the XPath expressions `$a/price/text()` and `$b/price/text()`) contained in the construct part.

The work reported about in this paper is based upon the conviction that logic programming provides with concepts giving rise to query and transformation languages more declarative than those based on a path-oriented node selection. A query term inspired from a Prolog goal atom would give rise to a "context-conscious" selection of several nodes (i.e. subterms) within a same term (the "context") at a time, a term inspired from a Prolog head atom would be a convenient construct expression, rules relating (conjunctions or disjunctions of) query terms to construct terms would define views (in the database sense) and give rise to a deduction-like (backward or forward) chaining of term constructions – a feature often needed in XML and semistructured data processing.

In such a language inspired from logic programming, the previous query example can be expressed as follows (symbols beginning with upper case latters denote variables):

```
construct
   <book-price-comparison>
      all <book-with-prices>
            <title> T</title>
            <price-amazon> Pa</price-amazon>
            <price-bn> Pb</price-bn>
          </book-with-prices>
   </book-price-comparison>
where
   in amazon.com:
   <entry>
      <title>T<title>
      <price>Pa</price>
   </entry>
   and
   in bn.com:
   <book>
      <title>T</title>
      <price>Pb</price>
   </book>
```

In a more conventional syntax, this query can be expressed as follows (an element name is shortened to the first letters of its constituting words and the locations `amazon.com` and `bn.com` are omitted):

`bpc{ all bwp[t[T],pa[Pa],pb[Pb]] }` $\leftarrow$ `e[t[T],p[Pa]] and b[t[T],p[Pb]]`

An advantage of such a rule is to clearly separate node selection, expressed only in the query terms i.e. in the rule body, from construction, expressed in the construct term i.e. in the rule head. This is beneficial for both, the programmer and query evaluation. Another advantage of the approach is to avoid the rather procedural navigation through data item imposed by a path-oriented node selection. In the rule given above, the contents of both elements `t` (i.e. `title`) and `pa` (i.e. `price-amazon`) are selected in a single query term `e[t[T], p[Pa]]`. In contrast, the XQuery expression needs two paths for the same selection, `$a/title` and `$a/price/`. The query term `e[t[T], p[Pa]]` stresses the common context and the relative position of the selected nodes (i.e. subterms) `T` and `Pa`. In contrast, the paths `$a/title` and `$a/price/` specify two independent navigations through a term. Arguably, a term-oriented (or context-conscious or positional) node selection is more declarative than a path-oriented (or navigational) node selection.

This paper reports about first achievements in designing a term-oriented, "context-conscious", or "positional" query and transformation language for XML and semistructured data. In order to conform to the semistructured data paradigm, a novel form of unification is needed. This paper is mostly devoted to motivating and specifying a nonstandard unification, called "simulation unification" convenient for a positional querying and transformation of XML and semistructured data.

This article is organised as follows. Section 1 is this introduction. Section 2 describes those aspects of the query and transformation language under development that are relevant to this paper. Simulation unification is addressed in Section 3. Section 4 is devoted to related work and a conclusion.

2 Elements of a Query and Transformation Language

This section introduces into those aspects of an experimental query and transformation language for XML and semistructured data, called Xcerpt, that are relevant to this paper. Aspects of XML, such as attributes and namespaces, that are irrelevant to this paper, are not explicitly addressed in the following. Two disjoint sets of symbols, the set $\mathcal{L}$ of labels (or tags) and the set $\mathcal{V}$ of variables are considered. Labels (variables, resp.) are denoted by words starting with a lower (upper, resp.) case letter. The following meta-variables (with or without indices and/or superscripts) are used:

- l denotes a label,
- X denotes a variable,
- t denotes a term (as defined below).

2.1 Database Terms

Database terms are an abstraction of XML documents. Following a common practice in XML query language and semistructured data research [3], a database is a set (or multiset) of database terms and the children of a document node may be either ordered (as in SGML and in standard XML), or unordered (as in the semistructured data model). In the following, a term whose root is labelled l and has *ordered* children $t_1, \dots, t_n$ is denoted $l[t_1, \dots, t_n]$; a term whose root is labelled l and has *unordered* children $t_1, \dots, t_n$ is denoted $l\{t_1, \dots, t_n\}$.

Definition 1 (Database Terms). *Database terms are inductively defined as follows:*

1. *A label is a (atomic) database term.*
2. *If l is a label and $t_1, \dots, t_n$ are $n \geq 1$ database terms, then $l[t_1, \dots, t_n]$ and $l\{t_1, \dots, t_n\}$ are database terms.*

Database terms are similar to classical logic ground terms except that, (1) the arity of a function symbol, called here "label", is not fixed (as in Prolog), and (2) the arguments of a function symbol may be unordered.

Whatever storage is used, a database term $t_0 = l\{t_1, \dots, t_n\}$ with unordered subterms $t_1, \dots, t_n$ will always be stored in a manner inducing an order on $t_1, \dots, t_n$. The notion of unordered subterms $t_1, \dots, t_n$ means that (1) the storage ordering of $t_1, \dots, t_n$ is left at the discretion of the storage system (giving rise e.g. to clustering as many t_i as possible on a secondary memory page), and (2) no given ordering is to be returned when t_0 is accessed.

In the following, $\mathcal{T}_{db}$ denotes the set of all database terms.

2.2 Query Terms

A query term is a "pattern" that specifies a selection of database terms very much like Prolog goal atoms and SQL selections. However, answers to query terms (cf. below Definition 13) differ from answers to Prolog goal atoms and SQL selections as follows:

– Database terms with additional subterms to those explicitly mentioned in a query term might be answers to this query term.
– Database terms with a different subterm ordering from that of the query term might be answers to this query term.
– A query term might specify subterms at an arbitrary depth.

In query terms, the single square and curly brackets, [ ] and { }, denote "exact subterm patterns", i.e. single (square or curly) brackets are used in a query term to be answered by database terms with no more subterms than those given in the query term. Double square and curly brackets, [[ ]] and {{ }}, on the other hand, denote "partial subterm patterns" as described above.

[ ] and [[ ]] are used if the subterm order in the answers is to be that of the query term, { } and {{ }} are used otherwise. Thus, possible answers to the query term $t_1 = a[b, c\{\{d, e\}\}, f]$ are the database terms $a[b, c\{d, e, g\}, f]$ and $a[b, c\{d, e, g\}, f\{g, h\}]$ and $a[b, c\{d, e\{g, h\}, g\}, f\{g, h\}]$ and $a[b, c[d, e], f]$. In contrast, $a[b, c\{d, e\}, f, g]$ and $a\{b, c\{d, e\}, f\}$ are no answers to t_1. The only answers to $f\{\ \}$ are f-labelled database terms with no children.

The construct *descendant*, short *desc*, introduces a subterm at an unspecified depth. Thus, possible answers to the query term $t_2 = a[\mathit{desc}\ f[c, d], b]$ are $a[f[c, d], b]$ and $a[g[f[c, d]], b]$ and $a[g[f[c, d], h], b]$ and $a[g[g[f[c, d]]], b]$ and $a[g[g[f[c, d], h], i], b]$.

In a query term, a variable X can be restricted to some query terms using the construct $\leadsto$, read "as". Thus, the query term $t_3 = a[X_1 \leadsto b[[c, d]], X_2, e]$ constrains the variable X_1 to such database terms that are possible answers to the query term $b[[c, d]]$. Note that the variable X_2 is unconstrained in t_3. Possible answers to t_3 are e.g. $a[b[c, d], f, e]$ which binds X_1 to $b[c, d]$ and X_2 to f, $a[b[c, d], f[g, h], e]$ which binds X_1 to $b[c, d]$ and X_2 to $f[g, h]$, $a[b[c, d, e], f, e]$ which binds X_1 to $b[c, d, e]$ and X_2 to f, and $a[b[c, e, d], f, e]$ which binds X_1 to $b[c, e, d]$ and X_2 to f.

Definition 2 (Query Terms). *Query terms are inductively defined as follows:*

1. *If l is a label, then l and $l\{\}$ are (atomic) query terms.*
2. *A variable X is a query term.*
3. *If X is a variable and t a query term, then $X \leadsto t$ is a query term.*
4. *If X is a variable and t is a query term, then $X \leadsto \mathit{desc}\ t$ is a query term.*
5. *If l is a label and $t_1, \ldots, t_n$ are $n \geq 1$ query terms, then $l[t_1, \ldots, t_n]$, $l\{t_1, \ldots, t_n\}$, $l[[t_1, \ldots, t_n]]$, and $l\{\{t_1, \ldots, t_n\}\}$ are query terms.*

Multiple variable constraints are not precluded. A possible answer to e.g. $a\{\{X \leadsto b\{\{c\}\}, X \leadsto b\{\{d\}\}\ \}\}$ is $a\{b\{c, d\}\}$. The query term $a[[X \leadsto b\{\{c\}\}, X \leadsto f\{\{d\}\}]]$, however, has no answers, as the labels b and f are distinct.

Subterms (of query terms) are defined as usual (e.g. a and X and $Y \leadsto \mathit{desc}\ b\{X\}$ and $h\{a, X \leadsto k\{c\}\}$ and $X \leadsto k\{c\}$ and t itself are subterms of $t = f\{a, g\{Y \leadsto \mathit{desc}\ b\{X\}, h\{a, X \leadsto k\{c\}\}\}\}$). In the following, query terms are assumed to be variable well-formed, a notion defined as follows.

Definition 3 (Variable Well-Formed Query Terms). *A term variable X* depends *on a term variable Y in a query term t if $X \rightsquigarrow t_1$ is a subterm of t and Y is a subterm of t_1. A query term t is* variable well-formed *if t contains no term variables $X_0, \ldots, X_n$ ($n \geq 1$) such that 1. $X_0 = X_n$ and 2. for all $i = 1, \ldots, n$, X_i depends on X_{i-1} in t.*

E.g. $f\{X \rightsquigarrow g\{X\}\}$ and $f\{X \rightsquigarrow g\{Y\}, Y \rightsquigarrow h\{X\}\}$ are not variable well-formed. Variable well-formedness precludes queries specifying infinite answers. Usually terms that are not variable well-formed are called cyclic. However, Xcerpt also allows for arbitrary graph structures (which are not discussed in this paper, cf. [7]) which might by cyclic in another sense.

In the following, query terms are implicitly assumed to be variable well-formed and the set $\mathcal{T}_q$ is defined as the set of all (variable well-formed) query terms.

2.3 Construct Terms

Construct terms serve to re-assemble variables, the "values" of which are specified in query terms, so as to form new database terms. Thus, construct terms may contain both constructs [ ] and { } (like database terms) as well as variables. However, the construct $\rightsquigarrow$ is not allowed in construct terms, as variables should be constrained where they are defined, (i.e. in query terms), not in construct terms where they are used to specify new terms.

Definition 4 (Construct Terms). *Construct terms are inductively defined as follows:*

1. *A label l is a (atomic) construct term.*
2. *A variable X is a construct term.*
3. *If l is a label and $t_1, \ldots, t_n$ are $n \geq 1$ construct terms, then $l[t_1, \ldots, t_n]$ and $l\{t_1, \ldots, t_n\}$ are construct terms.*

The set of construct terms will be denoted with $\mathcal{T}_c$ in the rest of this paper. Note that $\mathcal{T}_{db} \subseteq \mathcal{T}_c \subseteq \mathcal{T}_q$.

2.4 Construct-Query Rules

Construct-query rules, short rules, relate queries, consisting of a conjunction of query terms, and construct terms. It is assumed (cf. below Point 3 of Definition 5) that each variable occurring in the construct term of a construct-query rule also occurs in at least one of the query terms of the rule, i.e. variables in construct-query rules are assumed to be "range-restricted" or "allowed". A relaxation of this condition like in Prolog does not seem to be desirable.

Definition 5 (Construct-Query Rule). *A construct-query rule is an expression of the form $t^c \leftarrow t_1^q \wedge \ldots \wedge t_n^q$ such that:*

1. *$n \geq 1$ and for all $i = 1, \ldots n$, t_i^q is a query term,*
2. *t^c is a construct term, and*

3. every variable occurring in t^c also occurs in at least one of the t_i^q.

The left hand-side, i.e. the construct term, of a (construct-query) rule will be referred to as the rule "head". The right hand-side of a (construct-query) rule will be referred to as the rule "body". Note that, in contrast to the body of a Prolog clause, the body of a (construct-query) rule cannot be empty, for empty rule bodies do not seem to be needed for the applications considered.

An Xcerpt program consists of a finite set of (construct-query) rules with a (conjunction of) query term(s). The scope of an occurrence of a variable in an Xcerpt program is, like in Prolog, restricted to the rule it occurs in.

2.5 Further Features

The full version of this paper [8] describes in more details further features of the experimental language Xcerpt, among others the construct *all* mentioned in the introduction.

3 Simulation Unification

The rule-based language Xcerpt, the main elements of which have been introduced above in Section 2, can be processed by both forward and backward chaining. Techniques similar to those used in implementations of Prolog (e.g. the use of the run-time stack for implementing a depth-first search) or of Datalog (e.g. a database storage of goal atoms) can be used for Xcerpt as well. However, Xcerpt cannot rely on standard unification because of the requirements on query terms listed in Section 2.2: A query term of the form $l[[t_1, \dots, t_n]]$ or $l\{\{t_1, \dots, t_n\}\}$ should "unify" with l-labelled terms with more subterms than those matching $t_1, \dots,$ and t_n; also unordered subterms (like in $l\{\{t_1, \dots, t_n\}\}$), the descendant construct *desc* and the *as* construct $\leadsto$ have to be dealt with. This section is devoted to introducing a nonstandard unification called "simulation unification" fulfilling these requirements.

For space reasons, simulation unification is defined in this paper under the assumptions that $\{\{\ \}\}$ and $\{\ \}$ are the only kinds of braces, and that braces are only allowed immediately on the right of a label (like in $f\{\{a, g\{b, c\}, d\}\}$) and not directly within other braces (like in $f\{\{a, \{b, c\}, d\}\}$). The full article [8] explains how to skip these restrictions.

3.1 Simulation

Intuitively, a simulation of a graph G_1 in a graph G_2 is a mapping of the nodes of G_1 in the nodes of G_2 preserving the edges. In other words, there exists a simulation of G_1 in G_2, if the node/edge structure of G_1 can be found as a subgraph of G_2. Efficient algorithms for computing simulation (bisimulation, resp.) are given e.g. in [9]. In [3,10], simulation is used for verifying the conformity of semistructured data to a schema. The language UnQL [11] introduces (bi)simulation for query answering, but the usage is restricted to pattern matching.

Definition 6 (Graph Simulation). *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs and let $\sim$ be an equivalence relation on $V_1 \cup V_2$. A relation $\mathcal{S} \subseteq V_1 \times V_2$ is a simulation with respect to $\sim$ of G_1 in G_2 if:*

1. *If $v_1\ \mathcal{S}\ v_2$, then $v_1 \sim v_2$.*
2. *If $v_1\ \mathcal{S}\ v_2$ and $(v_1, v_1') \in E_1$, then there exists $v_2' \in V_2$ such that $v_1'\ \mathcal{S}\ v_2'$ and $(v_2, v_2') \in E_2$.*

A simulation $\mathcal{S}$ of a tree T_1 with root r_1 in a tree T_2 with root r_2 is a rooted simulation of T_1 in T_2 if $r_1\ \mathcal{S}\ r_2$.

Note that the definition of a simulation $\mathcal{S}$ of G_1 in G_2 does not preclude that two distinct vertices v_1 and v_1' of G_1 are simulated by the same vertice v_2 of G_2, i.e. $(v_1, v_2) \in \mathcal{S}$ and $(v_1', v_2) \in \mathcal{S}$. Figure 1 gives examples of simulations (represented by the dashed edges) with respect to vertice label equality.

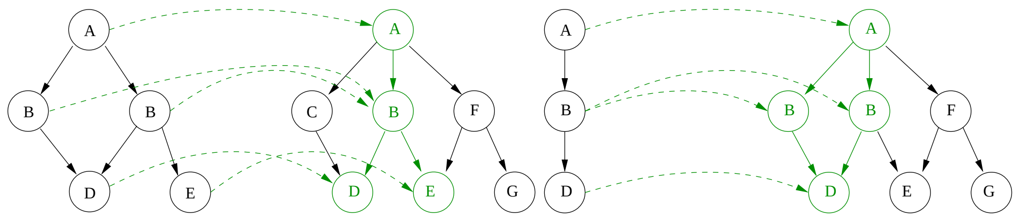

Fig. 1. Rooted Simulations (with respect to label equality)

Simulation with respect to label equality is a first notion towards a formalisation of answers to query terms: If a database term t^{db} is to be an answer to a query term t^q (both terms being considered as trees), then there must exist a rooted simulation with respect to label equality of (the term/tree with no $\rightsquigarrow$ and *desc* constructs subjacent to) t^q in t^{db}.

3.2 Term Lattice

Definition 7 (Ground Query Term). *A query term is* ground *if it contains no variables, no $\rightsquigarrow$ and no desc.*

In the following, the set of all ground query terms, extended by the two special terms $\bot$ (the "empty" term) and $\top$ (the "full" term) will be denoted by $\mathcal{T}_{\text{ground}}$. Note that $\mathcal{T}_{\text{ground}} \neq \mathcal{T}_{db}$, since in contrast to database terms ground query terms may contain both constructs { } and {{ }}.

Definition 8 (Ground Query Term Simulation). *Let $t_1 \in \mathcal{T}_{ground}$ and $t_2 \in \mathcal{T}_{ground}$. Let $S_i \subseteq \mathcal{T}_{ground}$ denote the set of subtrees of t_i ($i \in \{1, 2\}$). A relation $\mathcal{S} \subseteq S_1 \times S_2$ is a simulation of t_1 in t_2 if:*

1. *$t_1\ \mathcal{S}\ t_2$*
2. *If $l_1\ \mathcal{S}\ l_2$ then $l_1 = l_2$.*

3. *If $l_1\{\{t^1_1, \dots, t^1_n\}\}\ \mathcal{S}\ l_2\{\{t^2_1, \dots, t^2_m\}\})$, then $l_1 = l_2$ and for all $i \in \{1, \dots, n\}$ there exists $j \in \{1, \dots, m\}$ such that $t^1_i\ \mathcal{S}\ t^2_j$*
4. *If $l_1\{\{t^1_1, \dots, t^1_n\}\}\ \mathcal{S}\ l_2\{t^2_1, \dots, t^2_m\})$, then $l_1 = l_2$ and for all $i \in \{1, \dots, n\}$ there exists $j \in \{1, \dots, m\}$ such that $t^1_i\ \mathcal{S}\ t^2_j)$*
5. *If $l_1\{t^1_1, \dots, t^1_n\}\ \mathcal{S}\ l_2\{\{t^2_1, \dots, t^2_m\}\})$, then $l_1 = l_2$ and for all $i \in \{1, \dots, n\}$ there exists $j \in \{1, \dots, m\}$ such that $t^1_i\ \mathcal{S}\ t^2_j)$, and for all $j \in \{1, \dots, m\}$ there exists $i\{1, \dots, n\}$ such that $t^1_i\ \mathcal{S}\ t^2_j$*
6. *If $l_1\{t^1_1, \dots, t^1_n\}\ \mathcal{S}\ l_2\{t^2_1, \dots, t^2_m\})$, then $l_1 = l_2$ and for all $i \in \{1, \dots, n\}$ there exists $j \in \{1, \dots, m\}$ such that $t^1_i\ \mathcal{S}\ t^2_j$, and for all $j \in \{1, \dots, m\}$ there exists $i\{1, \dots, n\}$ such that $t^1_i\ \mathcal{S}\ t^2_j$*

Definition 9 (Simulation Preorder). *$\preceq$ is the preorder on $\mathcal{T}_{ground} \setminus \{\bot, \top\}$ defined by $t_1 \preceq t_2$ if there exists a ground query term simulation of t_1 in t_2.*

The preorder $\preceq$ is not an order, for although $t_1 = f\{a\} \preceq t_2 = f\{a, a\}$ and $t_2 = f\{a, a\} \preceq t_1 = f\{a\}$ (both a of t_2 can be simulated by the same a of t_1), $t_1 = f\{a\} \neq t_2 = f\{a, a\}$.

However, $\preceq$ induces as follows a (partial) order on $\mathcal{T}_{\text{ground}}$. First, consider the equivalence relation $\equiv$ on $\mathcal{T}_{\text{ground}}$ defined by the bisimulation $t_1 \equiv t_2$ if both, $t_1 \preceq t_2$ and $t_2 \preceq t_1$ hold. Since $\preceq$ is reflexive and transitive, $\equiv$ is also reflexive and transitive. $\equiv$ is by definition symmetric.

It is natural to chose as representative of an equivalence class of $\mathcal{T}_{\text{ground}} / \equiv$ the class member with the minimal number of repeated subterms, e.g. $f\{a\}$ is chosen as representative of class $\{f\{a\}, f\{a, a\}, f\{a, a, a\}, f\{a, a, a, a\}, \dots\} \in \mathcal{T}_{\text{ground}} / \equiv$.

In the following, referring to this representative will always be meant as a reference to the whole equivalence class and the (partial) order induced by $\preceq$ on $\mathcal{T}_{\text{ground}} / \equiv$ will be noted $\preceq$, too. In other words, answers to query terms will be defined up to $\equiv$ as representatives of elements of $\mathcal{T}_{\text{ground}} / \equiv$. Intuitively, $t_1 \preceq t_2$ means that it is possible to remove from t_2 subterms at arbitrary depth, until the remaining term is either t_1 or some $\preceq$-smaller term from the same $\equiv$-class as t_1.

Definition 10 (Ground Query Term Lattice). *$\preceq$ is extended to $\bot$ and $\top$ as follows: For all $t \in \mathcal{T}_{ground}$, $\bot \preceq t$ and $t \preceq \top$. $(\mathcal{T}_{ground} / \equiv, \preceq)$ is the ground query term lattice.*

3.3 Answers

An answer in a database $D \subseteq \mathcal{T}_{db}$ to a query term t^q is characterised by a set of values for the variables in t^q such that the ground query term t^q_g resulting from substituting these values for the variables in t^q is simulated by an element t of D (i.e. $t^q_g \preceq t$).

Consider for example the query $t^q = f\{\{X \rightsquigarrow g\{\{b\}\}, X \rightsquigarrow g\{\{c\}\}\ \}\}$ against the database $D = \{f\{g\{a, b, c\}, g\{a, b, c\}, h\},\ f\{g\{b\}, g\{c\}\}\}$. The $\rightsquigarrow$ constructs in t^q yield the constraint $g\{\{b\}\} \preceq X \wedge g\{\{c\}\} \preceq X$. The first database term in D yields the constraint $X \preceq g\{a, b, c\}$. The second database term in D yields the constraint $X \preceq g\{b\} \wedge X \preceq g\{c\}$. The constraint $g\{\{b\}\} \preceq X \wedge g\{\{c\}\} \preceq X$ is incompatible with $X \preceq g\{b\} \wedge X \preceq g\{c\}$. Thus, the only possible value for X is $g\{a,b,c\}$ and the only possible answer to t^q in D is $t^q_a = f\{g\{a,b,c\}, g\{a,b,c\}, h\}$.

Note that, in contrast to Prolog and SQL, the binding $X = g\{a,b,c\}$ does not suffice to characterise the answer t^q_a, for t^q does not have any "handle" for the subterm h of t^q_a. If not only the bindings for X but the complete answers to t^q are sought for, then the query term $Y \rightsquigarrow f\{\{X \rightsquigarrow g\{\{b\}\}, X \rightsquigarrow g\{\{c\}\}\}\}$ is to be used instead of t^q.

Definition 11 (Substitutions and Instances). *Let t^q be a query term and let $X_1, \ldots, X_n$ be the variables occurring (left or right of $\rightsquigarrow$ or elsewhere) in t^q.*

A substitution is a function which assigns a construct term to each variable of a finite set of variables. A substitution σ is a grounding substitution for a query term t^q if σ assigns a ground query term to each variable in t^q.

If σ is a substitution (grounding substitution, resp.) for t^q assigning t_i to X_i $(1 \leq i \leq n)$, then the instances (ground instances, resp.) of t^q with respect to σ are those construct terms (ground query terms, resp.) that can be constructed from t^q as follows:

1. *Replace each subterm $X \rightsquigarrow t$ by X.*
2. *Replace each occurrence of X_i by t_i $(1 \leq i \leq n)$.*

Requiring in Definition 2 *desc* to occur to the right of $\rightsquigarrow$ makes it possible to characterise ground instances of query terms by substitutions. This is helpful for formalising answers but not necessary for language implementions.

Not all ground instances of a query term are acceptable answers, for some instances might violate the conditions expressed by the $\rightsquigarrow$ and *desc* constructs.

Definition 12 (Allowed Instances). *The constraint induced by a query term t^q and a substitution σ is the conjunction of all inequations $t\sigma \preceq X\sigma$ such that $X \rightsquigarrow t$ with $t \neq desc\ t_1$ is a subterm of t^q, and of all expressions $X\sigma \lhd t\sigma$ (read "$X\sigma$ subterm of $t\sigma$") such that $X \rightsquigarrow desc\ t$ is a subterm of t^q, if t^q has such subterms. If t^q has no such subterms, the constraint induced t^q and σ is the formula true.*

Let σ be a grounding substitution of a query term t^q. The instance $t\sigma$ of t^q is allowed if:

- *$t\sigma \neq \bot$ and $t\sigma \neq \top$.*
- *Each inequality $t_1 \preceq t_2$ in the constraint induced by t^q and σ is satisfied in $(\mathcal{T}/\equiv, \preceq)$.*
- *If $t_1 \lhd t_2$ occurs in the constraint induced by t^q and σ, then there exists a subterm t_1' of t_1 such that $t_2 \preceq t_1'$*

Definition 13 (Answers). *Let t^q be a query term, D a database (i.e. $D \subseteq \mathcal{T}_{db}$). An answer to t^q in D is a database term $t^{db} \in D$ such that there exists an allowed instance t^q_a of t^q satisfying $t^q_a \preceq t^{db}$.*

3.4 Simulation Unification

Simulation unification is a non-deterministic method for solving inequations of the form $t^q \preceq t^c$, where t^q is a query term, t^c is a construct term (possibly a database term), and t^q and t^c are variable disjoint, in the database term lattice $(\mathcal{T}_{db}/\equiv, \preceq)$, i.e. to determine substitutions σ such that $t^q\sigma$ and $t^c\sigma$ have instances $t^q\sigma\tau$ and $t^c\sigma\tau$ such that $t^q\sigma\tau$ and $t^c\sigma\tau$ are database terms and $t^q\sigma\tau \preceq t^c\sigma\tau$ holds.

Such inequations may result from a forward chaining evaluation of a query-construct rule against database terms. In such a case, the right-hand side t^c of the inequation contains no variables, i.e. it is a database term. An inequation $t^q \preceq t^c$ may also result from a backward chaining evaluation of the query term t^q against a query-construct rule whose head is t^c. In such a case, variables may occur in the construct term t^c but t^q and t^c are variable disjoint. That t^q and t^c do not share variables follows from the variable scoping rule for Xcerpt programs postulated in Section 2.4 above (this is the so-called "standardisation apart" of deduction methods).

Simulation unification consists in repeated applications of Term Decomposition phases followed by a Consistency Verification phase to a formula C (for $\underline{c}$onstraint store) consisting in disjunctions of conjunctions of inequations of the form $t^q \preceq t^c$ (with t^q query term and t^c construct term) and/or equations of the form $t^c_1 = t^c_2$ (with t^c_1 and t^c_2 construct terms). At the beginning C consists in a single inequation $t^q \preceq t^c$. Both phases Term Decomposition and Consistency Verification consist in stepwise changes of the constraint store C. These changes are expressed in the following formalism inspired from [12]:

A "simplification" $L \Leftrightarrow R$ replaces L by R.

Trivially satisfied inequations or equations are replaced by the atomic formula *true*. Inconsistent conjunctions of inequations or equations are replaced by the atomic formula *false*.

Definition 14 (Term Decomposition Rules). *Let l (with or without indices) denote a label. Let t^1 and t^2 (with or without indices) denote query terms.*

- **Root Elimination:**

(1) $l \preceq l\{t^2_1, \ldots, t^2_m\} \Leftrightarrow \mathit{true}$ *if* $m \geq 1$
$l \preceq l\{\} \Leftrightarrow \mathit{true}$
$l\{\} \preceq l\{t^2_1, \ldots, t^2_m\} \Leftrightarrow \mathit{false}$ *if* $m \geq 1$
$l\{\} \preceq l \Leftrightarrow \mathit{true}$
$l\{\} \preceq l\{\} \Leftrightarrow \mathit{true}$

(2) $l\{\{t^1_1, \ldots, t^1_n\}\} \preceq l \Leftrightarrow \mathit{false}$ *if* $n \geq 1$
$l\{\{t^1_1, \ldots, t^1_n\}\} \preceq l\{\} \Leftrightarrow \mathit{false}$ *if* $n \geq 1$

$l\{t^1_1, \ldots, t^1_n\} \preceq l \Leftrightarrow \mathit{false}$ *if* $n \geq 1$
$l\{t^1_1, \ldots, t^1_n\} \preceq l\{\} \Leftrightarrow \mathit{false}$ *if* $n \geq 1$

(3) Let Π be the set of total functions $\{t^1_1,\ldots,t^1_n\} \rightarrow \{t^2_1,\ldots,t^2_m\}$*:*

$$l\{\{t^1_1,\ldots,t^1_n\}\} \preceq l\{t^2_1,\ldots,t^2_m\} \Leftrightarrow \bigvee_{\pi\in\Pi}\bigwedge_{1\leq i\leq n} t^1_i \preceq \pi(t^1_i) \quad \textit{if } n,m\geq 1$$

Let Π be the set of total, surjective functions $\{t^1_1,\ldots,t^1_n\} \rightarrow \{t^2_1,\ldots,t^2_m\}$*:*

$$l\{t^1_1,\ldots,t^1_n\} \preceq l\{t^2_1,\ldots,t^2_m\} \Leftrightarrow \bigvee_{\pi\in\Pi}\bigwedge_{1\leq i\leq n} t^1_i \preceq \pi(t^1_i) \quad \textit{if } n,m\geq 1$$

(4) $l_1\{\{t^1_1,\ldots,t^1_n\}\} \preceq l_2\{t^2_1,\ldots,t^2_m\} \Leftrightarrow \textit{false} \; \textit{if } l_1 \neq l_2 \; (n,m\geq 0)$

$l_1\{t^1_1,\ldots,t^1_n\} \preceq l_2\{t^2_1,\ldots,t^2_m\} \Leftrightarrow \textit{false} \quad \textit{if } l_1 \neq l_2 \; (n,m\geq 0)$

– $\rightsquigarrow$ **Elimination:**

$$X \rightsquigarrow t^1 \preceq t^2 \quad \Leftrightarrow \quad t^1 \preceq t^2 \;\wedge\; t^1 \preceq X \;\wedge\; X \preceq t^2$$

– **Descendant Elimination:**

$$\textit{desc } t^1 \preceq l_2\{t^2_1,\ldots,t^2_m\} \Leftrightarrow t^1 \preceq l_2\{t^2_1,\ldots,t^2_m\} \vee \bigvee_{1\leq i\leq m} \textit{desc } t^1 \preceq t^2_i \quad \textit{if } m\geq 0$$

Applying the $\rightsquigarrow$ and descendant elimination rules to a constraint store C in disjunctive normal form may yield a constraint store not in disjunctive normal form. Thus, the method has to restore from time to time the disjunctive normal form of C. In doing so, the formulas *true* and *false* are treated as usual: *true* is removed from conjunctions, conjunctions containing *false* are removed.

In the following, *mgcu*$(t_1,\ldots,t_n)$ (with $t_1,\ldots,t_n$ construct terms) returns a most general commutative-unifier of $t_1,\ldots,t_n$ (in the sense of [13]) expressed as either *false*, if t_1 and t_2 are not commutative-unifiable, or as *true* if t_1 and t_2 are commutative-unifiable and do not contain variables, or else as a conjunction of equations of the form $X = t$. Note that most general commutative-unifiers are only computed for construct terms (i.e. terms without $\rightsquigarrow$ and *desc* construct). Recall that commutative unification is decidable.

In the definition below, simulation unification is initialised with $X_0 \rightsquigarrow t^q \preceq t^c$, where X_0 is a variable occurring neither in t^q nor in t^c, instead of simply $t^q \preceq t^c$. The additional variable X_0 serves to a complete specification of the answers returned. This is useful in proving the correctness of simulation unification but can usually be dispensed of in practice.

Definition 15 (Simulation Unification).

1. **Initialisation:** $C := X_0 \rightsquigarrow t^q \preceq t^c$
(with t^q query term, t^c construct term and t^q, t^c and X_0 variable disjoint).
2. **Term Decomposition:**
Until C can no longer be modified, repeat performing one of:
   – *Apply a (applicable) Term Decomposition rule to C*
   – *Put C in disjunctive normal form*
3. **Variable Binding:**
Replace each $X \preceq t$ in C with $X = t$.

4. **Consistency Verification:**
For each disjunct D of C and for each variable X occurring in D do:
Replace in D the equations $X = t_1, \ldots, X = t_n$ by $mgcu(t_1, \ldots, t_n)$.

For efficiency reasons it is preferable to intertwine the Term Decomposition and Consistency Verification phases instead of performing them one after another. The sequential processing in Definition 15 simplifies the proofs.

Proposition 1 (Correctness and Completeness). *Let t^q be a query term, t^c a construct term, and X_0 a variable such that t^q, t^c and X_0 are variable disjoint. There exists a substitution τ such that $t^q\tau$ and $t^c\tau$ are database terms and $t^q\tau = t^c\tau$ if and only if a simulation unification initialised with $X_0 \rightsquigarrow t^q \preceq t^c$ returns a substitution σ such that*

- *For each variable X in t^q, $X\sigma$ is a subterm of $t^q\sigma$.*
- *$t^q\tau$ is an instance of $t^q\sigma$.*
- *$t^c\tau$ is an instance of $t^c\sigma$.*

The proof of Proposition 1 is given in the full version of this paper [8].

3.5 Examples

$f\{\{X \rightsquigarrow b, Y \rightsquigarrow b\{\{c,d\}\}\}\}$ and $f\{a, b\{c,d,e\}, b\{e\}\}$ "simulation unify" yielding the following constraints: $(X = b\{c,d,e\} \wedge Y = b\{c,d,e\}) \vee (X = b\{e\} \wedge Y = b\{c,d,e\})$.

Also, the terms $X \rightsquigarrow desc\ (Y \rightsquigarrow f\{\{a\}\})$ and $g\{f\{Z,b,c\}, h\{f\{a,b\}\}\}$ "simulation unify" yielding $((Y = f\{Y,b,c\} \wedge a \preceq Z) \vee Y = f\{a,b\}) \wedge X = g\{f\{Z,b,c\}, h\{f\{a,b\}\}\}$.

The steps of these simulation unifications are given in the full version [8] of this paper. Note that these simulation unifications constrain variables "on both sides", i.e. simulation unification is no matching but a full-fledged unification.

4 Related Work and Conclusion

The articles [14,15,16] have already pointed out the drawbacks of relying on a navigational node selection à la XPath [17] and XQuery [2] for query and transformation languages for XML and semistructured data.

The language UnQL [11] has introduced simulation as a means for query answering. UnQL, like Xcerpt, uses the notions of patterns and templates. UnQL and Xcerpt differ from each other as follows. First, a query in UnQL consists of a single "select-where" expression which can be processed with pattern matching. In contrast, a query in Xcerpt might "chain" several "construct-query rules" requiring a "unification" which is capable of binding variables from both of the terms to be "unified". Second, variables in UnQL can only occur as leaves of query patterns. Complex queries might require the use of several patterns in UnQL, where a single pattern suffices in Xcerpt.

In [14] a language for querying and transforming semistructured data is described. Like XPath and XQuery this language has variables for nodes, i.e. in the Xcerpt terminology labels.

[15] describes fxt, a language for querying and transforming semistructured data. fxt has variables for terms (or trees) and forests. fxt offers regular expressions similar to those of XPath for node selection. In contrast, the approach proposed in the present paper uses like Prolog variables for subterms. Arguably, languages with term variables makes data description less navigational than languages with node variables.

The language semantics in [14] is based upon a so-called component calculus and an algebra, very much in the style of XQuery's algebra which is inspired from functional languages. The language semantics given in [15] for fxt is in terms of tree automata. Arguably, Definition 13 is closer to a Tarski's style model theory and might therefore be seen as a more declarative semantics.

Several articles propose inference methods either rule-based or based upon consistency verification for XML data. [16] proposes a rule language very similar to Prolog called nowadays RuleML [18]. Several approaches that are too numerous for being explicitly mentioned here adapt techniques from feature logics to XML data. These approaches are usually named referring to "ontology" and/or "Semantic Web". Common to RuleML and the ontology or Semantic Web approaches is that the language they propose do not support a direct access to XML data. Instead, their languages require a translation into a specific syntax. In some cases, like the binary predicate language RDF, this syntax might seem too stringent. For the authors of this paper, a direct access to XML data is an essential feature of an inference language for Web-based databases and semantic reasoning with Web data.

Simulation is no new notion. It is commonly used in process algebra and graph theory. It has been applied to semistructured data e.g. in [10,19,3] for schema validation. Graph simulation in general has been studied extensively cf. [9,20] (simulation is called "path inclusion" in [20]).

Several unification methods have been proposed that, like simulation unification, process flexible terms or structures, notably feature unification [21,22] and associative-commutative-unification, short AC-unification, [23]. Simulation unification differs from feature unification in several aspects (discussed in [8]).

Simulation unification might remind of theory unification [24]. The significant difference between both is that simulation unification is based upon an order relation, while theory unification refers to a congruence relation.

There are interesting similarities between simulation unification and approaches to constraint solving over finite domains [25]. Simulation unification relies on a possibly disjunctive constraint store. This is rarely the case for constraint solvers. However, constraint programming approaches such as aggregation constraints [26] and constructive disjunction [27] seem interesting techniques for the future development of the language Xcerpt.

In this paper, a novel approach to querying and transforming XML and semistructured data based has been outlined. This approach is based on logic programming and a novel form of unification, simulation unification. A few aspects of a language under development, Xcerpt, have been presented. Many issues deserve further investigations. In particular, the complexity of simulation unification and its efficient implementation deserve further research.

Acknowledgements. The authors are thankful to Slim Abdennadher and Norbert Eisinger for useful suggestions.

References

1. W3C http://www.w3.org/Style/XSL/: Extensible Stylesheet Language (XSL). (2000)
2. W3C http://www.w3.org/TR/xquery/: XQuery: A Query Language for XML. (2001)
3. Abiteboul, S., Buneman, P., Suciu, D.: Data on the Web. From Relations to Semistructured Data and XML . Morgan Kaufmann Publishers, San Francisco, CA (2000)
4. Fernandez, M., Siméon, J., Wadler, P.: XML Query Languages: Experiences and Examplars. Communication to the XML Query W3C Working Group (1999)
5. Alashqur, A.M., Su, S.Y.W., Lam., H.: OQL: A Query Language for Manipulating Object-Oriented Databases. In: Proc. 15th Int. Conf. on Very Large Data Bases (VLDB). (1989)
6. Chamberlin, D., Fankhauser, P., Marchiori, M., Robie, J.: XML Query Use Cases. W3C Working Draft 20 (2001)
7. Bry, F., Schaffert, S.: Pattern Queries for XML and Semistructured Data. Technical Report PMS-FB-2002-5, Inst. for Computer Sciences, University of Munich, http://www.pms.informatik.uni-muenchen.de/publikationen/#PMS-FB-2002-5 (2002)
8. Bry, F., Schaffert, S.: Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. Technical Report PMS-FB-2002-2, http://www.pms.informatik.uni-muenchen.de/publikationen/#PMS-FB-2002-2 (2002)
9. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing Simulations on Finite and Infinite Graphs (1996)
10. Fernandez, M., Suciu, D.: Optimizing Regular Path Expressions Using Graph Schemas. In: Proceedings of the Int. Conf. on Data Engineering. (1988) 14–23
11. Buneman, P., Fernandez, M., Suciu, D.: UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. VLDB Journal **9** (2000) 76–110
12. Frühwirth, T.: Theory and Practice of Constraint Handling Rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming **37** (1998) 95–138
13. Baader, F.: Unification in Commutative Theories. In: Unification. Academic Press (1989) 417–435
14. Grahne, G., Lakshmanan, L.V.S.: On the Difference between Navigating Semi-structured Data and Querying It. In: Workshop on Database Programming Languages. (1999)
15. Berlea, A., Seidl, H.: fxt – A Transformation Language for XML Documents. Journal of CIT, Special Issue on Domain-Specific Languages (2001)
16. Boley, H.: Relationships Between Logic Programming and XML. In: Proc. 14th Workshop Logische Programmierung, Würzburg (2000)
17. W3 Consortium http://www.w3.org/TR/xpath: XML Path Language (XPath). (1999)
18. DFKI: RuleML – Rule Markup Language. http://www.dfki.uni-kl.de/ruleml/ (2002)
19. Buneman, P., Davidson, S.B., Fernandez, M.F., Suciu, D.: Adding Structure to Unstructured Data. In: Proceedings of ICDT'97. Volume 1186., Springer (1997) 336–350
20. Kilpeläinen, P.: Tree Matching Problems with Applications to Structured Text Databases. PhD thesis, Dept. of Computer Sciences, University of Helsinki (1992)
21. Aït-Kaci, H., Podelski, A., Goldstein, S.C.: Order-Sorted Theory Unification. Technical Report 32, digital – Paris Research Laboratory (1993)
22. Smolka, G.: Feature Constraint Logics for Unification Grammars. Journal of Logic Programming **12** (1992) 51–87
23. Fages, F.: Associative-Commutative Unification. In: Proc. 7th Int. Conf. on Automated Deduction (Napa, CA). Volume 170., Berlin, Springer (1984) 194–208
24. Baader, F., Snyder, W.: Unification Theory. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Elsevier Science Publishers (1999)
25. Montanari, U., Rossi, F.: Finite domain constraint solving and constraint logic programming. In Benhamou, F., Colmerauer, A., eds.: Constraint Logic Programming: Selected Research. MIT press (1993) 201–221
26. Ross, K.A., Srivastava, D., Stuckey, P.J., Sudarshan, S.: Foundations of aggregation constraints. Theoretical Computer Science B **190** (1994)
27. Würtz, J., Müller, T.: Constructive disjunction revisited. In: KI - Künstliche Intelligenz. (1996) 377–386