The XML Query Language Xcerpt: Design Principles, Examples, and Semantics

François Bry and Sebastian Schaffert

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

Abstract. Most query and transformation languages for XML and semistructured data – e.g. XQuery [12], the precursors of XQuery [5], and XSLT [10] – build upon a path-oriented node selection: A node in a data item is specified in terms of a root-to-node path in the manner of the file selection languages of operating systems. Constructs inspired from the regular expression constructs $*$, $+$, ?, and "wildcards" give rise to a flexible node retrieval from incompletely specified data items.
This paper introduces into Xcerpt, a query and transformation language further developing an alternative approach to querying XML and semistructured data first introduced with the language UnQL [3]. A metaphor for this approach views queries as patterns, answers as data items matching the queries. Formally, an answer to a query is defined as a simulation [6] of an instance of the query in a data item.

1 Introduction

Essential to semistructured data is the selection of data from incompletely specified data items. For such a data selection, a regular path expression language such as XPath is convenient because it gives rise to a flexible node retrieval. For example, the XPath expression `/descendant::a/descendant::b[following-sibling::c]` selects all elements of type `b` followed by a sibling element of type `c` that occur at any depth within an element of type `a`, itself at any depth in the document.

Most query and transformation languages developed since the mid 90es for XML and semistructured data – e.g. XQuery, the precursors of XQuery, and XSLT – rely upon such a path-oriented selection. They use patterns (also called templates) for expressing how the data (selected using paths) are re-arranged (or re-constructed) into new data items. Such languages intertwine construct parts, i.e. the construction patterns, and query parts, i.e. path selectors.

Example 1. The following Xquery expression creates a list of book titles for each author in a bibliography database, like that of Example 2.

```
<results>
  { for $a in distinct-values(document("http://www.bn.com")//author)
    return
        <result>
            { $a }
            {   for $b in document("http://www.bn.com")/bib/book
                where some $ba in $b/author satisfies deep-equal($ba,$a)
                return $b/title
            }
        </result>
  }
</results>
```

The XQuery expression is a construct pattern specifying the structure of the data to return. The query parts, i.e. the definition of the values for the variables `$a` and `$b`, are included in the construct pattern. Note that the (path-oriented) definitions of the variables `$a` and `$b` refer to a common subpath `document("http://www.bn.com")`. Note also the rather complicated condition relating values of `$a` and `$b`: `some $ba in $b/author satisfies deep-equal($ba,$a)`. The same query can be expressed in Xcerpt as shown in Example 9. □

The intertwining of construct and query parts à la XQuery has some drawbacks: (1) Query-construct requests involving a complex data retrieval might be confusing, (2) unnecessarily complex path selections, e.g. XPath expressions involving both forward and reverse axes, are possible [8], (3) in case of several path selections, the overall structure of the retrieved data items might be difficult to grasp, as in Example 1.

Among the query and transformation languages, UnQL [3] is a noticeable exception. This language first considered using patterns instead of paths for querying semistructured data. UnQL query patterns may contain variables. Applying a kind of pattern matching algorithm, reminding of those pattern matching algorithms used in functional programming and in automated reasoning, to a UnQL query pattern and a (variable-free) data item binds the variables of the query pattern to parts of the data item. This paper further investigates this approach proposing the following ideas and a query language called Xcerpt: (1) Instead of pattern matching, a (non-standard form of) unification is considered using which two query patterns, both containing variables, can be made identical through bindings of their variables. (2) Within a query pattern, a variable might be constrained through a (sub-)pattern to be bound only to data conforming to this (sub-)pattern. (3) Instead of building upon the functional paradigm, as UnQL does, the paradigm of SQL and of

logic programming is retained. Thus, a query might have several answers and the choice of some or all of the answers specified by a query can be expressed with language constructs reminding of the well-known set operators of elementary mathematics. (4) A chaining of queries, the answers to which are not necessarily sought for, makes it possible to rather naturally split complex queries into intuitive parts.

A metaphor for Xcerpt is to see queries as forms, answers as form fillings yielding database items. With Xcerpt, patterns are used not only in construct expressions, but also for data selection.

2 Language Principles

The following principles have prevailed to the definition of Xcerpt:
Positional instead of navigational queries. The relative positions of variables in a query should be easily recognizable. Within a query, it should be possible to constrain variables.
Referential transparency. The meaning of an expression, especially of a variable, should be the same wherever it appears.
Compositional semantics. A (structurally) recursive definition of the semantics of a query in terms of the semantics of its parts, i.e. a Tarski-style model theory, is sought for.
Multiple variable bindings. Like with SQL and logic programming languages, queries might have several answers, each answer binding the query variables differently.
Strict separation of construct and query proper. Query expressions should not occur in construct patterns. In construct expressions only variables should occur, but no conditions on the variables. Such conditions should occur in query expressions.
Symmetry. Queries should allow similar forms of incomplete specifications in breadth, i.e. concerning siblings, and in depth, i.e. concerning children.
Circularity. Queries and answers should be a queryable data items. This is more stringent than requiring an XML representation of queries.

The requirements of [7] are fulfilled by or compatible with Xcerpt.

3 Xcerpt Basic Constructs

Aspects of XML, such as attributes and namespaces, that are irrelevant to this paper, are not explicitly addresses in the following. The following pairwise disjoint sets of symbols are referred to: A set $\mathcal{I}$ of identifiers, a set $\mathcal{L}$ of labels (or tags or strings), a set $\mathcal{V}_l$ of label variables, a set $\mathcal{V}_t$ of term

(or data item) variables. Identifiers are denoted by id, labels (variables, resp.) by lower (upper, resp.) case letters with or without indices. The following meta-variables (with or without indices and/or superscripts) are used: *id* denotes an identifier, l denotes a label, L a label variable, X a term variable, t a term (as defined below), v a label or a term, and V a label or term variable.

3.1 Database Terms

A database is a set (or multiset) of database terms. The children of a document node may be either ordered (as in standard XML), or unordered. In the following, a term whose root is labelled l and has *ordered* (*unordered*, resp.) children $t_1, \ldots, t_n$ is denoted $l[t_1, \ldots, t_n]$ ($l\{t_1, \ldots, t_n\}$, resp.).

Definition 1 (Database Terms). Xcerpt Database Terms *are expressions inductively defined as follows and satisfying Conditions 1 and 2 given below:*

1. *If l is a label, then l is a (atomic) database term.*
2. *If id is an identifier and t is a database term neither of the form id_0: t_0 nor of the form $\uparrow id_0$, then id: t is a database term.*
3. *If id is an identifier, then $\uparrow$id is a database term.*
4. *If l is a label and $t_1, \ldots, t_n$ are $n \geq 1$ database terms, then $l[t_1, \ldots, t_n]$ and $l\{t_1, \ldots, t_n\}$ are database terms.*

Condition 1: *For a given identifier id an* identifier definition *id*: t_0 *occurs at most once in a term.*
Condition 2: *For every* identifier reference $\uparrow$*id occurring in a term t an identifier definition id*: t_0 *occurs in t.*

Example 2. The following database terms (inspired from the XQuery Use-Cases [4]) describe the book offers of two online book stores `bn.com` and `amazon.com` . Note that the bookstores use different formats.
`bn.com`:

```
bib { book { title { "TCP/IP Illustrated" },
             author { last { "Stevens" }, first { "W." } },
             publisher { "Addison-Wesley" },
             price { "65.95" } },
      book { title { "Advanced Programming in the Unix environment" },
             author { "Stevens" },
             publisher { "Addison-Wesley" },
             price { "65.95" } },
      book { title { "Data on the Web" },
```

```
            author { last { "Abiteboul" }, first { "Serge" } },
            author { last { "Buneman" }, first { "Peter" } },
            author { last { "Suciu" }, first { "Dan" } },
            publisher { "Morgan Kaufmann Publishers" },
            price { "39.95" } },
      book { title { "The Economics of Technology and Content for
                      Digital TV" },
            editor { last { "Gerbarg" },
                      first { "Darcy" },
                      affiliation { "CITI" } },
            publisher { "Kluwer Academic Publishers" },
            price { "129.95" } } }
```

amazon.com:

```
reviews { entry { title { "Data on the Web" },
                  price { "34.95" },
                  review { "A good discussion of semi-structured
                            database systems and XML." } },
          entry { title { "Advanced Programming in the Unix environment" },
                  price { "65.95" },
                  review { "A clear and detailed discussion of UNIX
                            programming." } },
          entry { title { "TCP/IP Illustrated" },
                  price { "65.95" },
                  review { "One of the best books on TCP/IP." } } }
```

In both database terms the element order is of no importance. This is expressed in the Xcerpt syntax using the single curly brackets { }. □

3.2 Query Terms

A query term is a pattern that specifies a selection of database terms very much like logical atoms and SQL selections do. The evaluation of query terms (cf. below Definition 12 for a formalisation) differs from the evaluation of logical atoms and SQL selections as follows: (1) Answers might have additional subterms to those mentioned in the query term. (2) Answers might have another subterm ordering than the query. (3) A query term might specify subterms at an unspecified 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". [ ] 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.

In a query term, a term variable X can be constrained to some query terms using the construct $\rightsquigarrow$, read "as". In query terms, the construct *desc*, read "descendant", specifies a subterm at an unspecified depth. Thus, the query term $t_2 = a[X_1 \rightsquigarrow b[c, d], X_2, e]$ constrains the term variable X_1 to such database terms that are possible answers to the query term $b[c, d]$. Note that the term variable X_2 is unconstrained in t_2. Possible answers to t_2 are $a[b[c, d], f, e]$ 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.

In query terms, the construct *desc*, read "descendant", specifies a subterm at an unspecified depth. Thus, possible answers to the query term $t_3 = a[X \rightsquigarrow \mathit{desc}\ f[c, d], b]$ 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 $a[g[g[f[c, d], h], i], b]$.

Definition 2 (Query Terms). Xcerpt Query terms *are expressions inductively defined as follows and satisfying Conditions 1 and 2 of Definition 1:*

1. *If l is a label and L is a label variable, then l, L, $l\{\{\}\}$, and $L\{\{\}\}$ are (atomic) query terms.*
2. *A term variable is a query term.*
3. *If id is an identifier and t is a query term neither of the form $id_0\colon t_0$ nor of the form $\uparrow id_0$, then $id\colon t$ is a query term.*
4. *If id is an identifier, then $\uparrow id$ is a query term.*
5. *If X is a variable and t a query term, then $X \rightsquigarrow t$ is a query term.*
6. *If X is a variable and t is a query term, then $X \rightsquigarrow \mathit{desc}\ t$ is a query term.*
7. *If l is a label, L a label variable and $t_1, \ldots, t_n$ are $n \geq 1$ query terms, then $l[t_1, \ldots, t_n]$, $L[t_1, \ldots, t_n]$, $l\{t_1, \ldots, t_n\}$, $L\{t_1, \ldots, t_n\}$, $l[[t_1, \ldots, t_n]]$, $L[[t_1, \ldots, t_n]]$, $l\{\{t_1, \ldots, t_n\}\}$, and $L\{\{t_1, \ldots, t_n\}\}$ are query terms.*

Query terms in which no variables occur are ground. *Query terms that are not of the form $\uparrow id$, are* strict. *The* leftmost label *of strict and ground query terms of the form l, $l\{\{\}\}$, $l\{t_1, \ldots, t_n\}$, and $l[t_1, \ldots, t_n]$ is l; the*

leftmost label *of a strict and ground query term of the form* $id : t$ *is the leftmost label of* t.

Example 3. Consider Example 2. The following query term queries the `amazon.com`database for titles and authors and bind the variables `TITLE` and `AUTHOR` to the corresponding values in the `bn.com`database:

```
bib {{ book {{ title { TITLE }, author { AUTHOR } }} }}
```

The evaluation strategy of Xcerpt is based on *simulation unification* [2]. The two variables `TITLE` and `AUTHOR` will have several possible bindings as a result of the simulation unification, representing all valid combinations of a title with an author that can be found in the database, e.g. `AUTHOR="Dan Suciu" and TITLE="Data on the Web"` or `AUTHOR="Serge Abiteboul" and TITLE="Data on the Web"`. □

Example 3 binds the variables to the "leafs" of the database terms. Xcerpt also allows variables at a "higher position" in a query term:

Example 4. The following Xcerpt query binds the variable `TITLE` to the compound element `title { ... }` thus retrieving `title` elements (instead of their content):

```
bib {{ book {{ TITLE ⇝ title, author { AUTHOR } }} }}
```

□

Thanks to simulation unification [2], the "leaf" of a `title` element does not have to be explicitly mentioned in the query of Example 4 for being included in the answers. The descendant construct serves to express indefiniteness:

Example 5. The following Xcerpt query retrieves the titles of books with an author "Stevens" at any depth:

```
bib {{ book {{ TITLE ⇝ title, author {{ X ⇝ desc "Stevens" }} }} }}
```

□

Definition 2 requires a *desc* expression to be preceded by $X \rightsquigarrow$ for some variable X. This is convenient for simplifying the formalisation of Xcerpt's declarative semantics (cf. below Definition 10). This is dispensible in practice at the cost of a more complicated counterpart to Definition 10.

Example 6. Example 5 can be expressed using the following query term (although not conforming to Definition 2):

```
bib {{ book {{ TITLE ⇝ title, author {{ desc "Stevens" }} }} }}
```

□

Child subterms and *subterms* of query terms are defined such that if $t = f[a, g\{Y \rightsquigarrow desc\ b\{X\}, h\{a, X \rightsquigarrow k\{c\}\}]$, then a and $g\{Y \rightsquigarrow desc\ b\{X\}, h\{a, X \rightsquigarrow k\{c\}\}$ are the only child subterms of t and e.g. a and X and $Y \rightsquigarrow desc\ b\{X\}$ and $h\{a, X \rightsquigarrow k\{c\}\}$ and $X \rightsquigarrow k\{c\}$ and t itself are subterms of t. Note that f is not a subterm of t. The $\rightsquigarrow$ construct makes it possible to express (undesirable) "cyclic" query terms. Definition 3 avoids such "cyclic" query terms.

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.*

$f\{X \rightsquigarrow g\{X\}\}$ and $f\{X \rightsquigarrow g\{Y\}, Y \rightsquigarrow h\{X\}\}$ are not variable well-formed. Variable well-formedness precludes queries specifying infinite answers. Below, query terms are assumed to be variable well-formed.

3.3 Construct Terms

Xcerpt Construct terms serve to re-assemble variables, the "values" of which are specified in query terms, so as to form new database terms. Thus, like in database terms both constructs [ ] and { } can occur in construct terms. Variables as references to subterms specified in a query can also occur in construct terms. However, the construct $\rightsquigarrow$ is not allowed in construct terms. The rationale for forbidding $\rightsquigarrow$ in construct terms is that 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.

Since querying a database may yield multiple alternative bindings for the same variables, it might be desirable to collect all such bindings in the construction of a result. The construct *all* serves this purpose. *all* t denotes the collection of all instances of t (binding the variables free in term t in all possible ways and recursively evaluating nested *all* constructs). A variable X is *free* in t, if X is not already contained within the argument of an *all* construct.

Definition 4 (Construct Terms). *Xcerpt Construct terms are expressions inductively defined as follows satisfying Conditions 1 and 2 of Definition 1:*

1. *Labels and label variables are (atomic) construct term.*
2. *If id is an identifier and t is a construct term, then id: t is a construct term.*
3. *If id is an identifier, then ↑id is a construct term.*
4. *A term variable is a construct term.*
5. *If t is a construct term, then all t is a construct term.*
6. *If l is a label, L is a label variable and* $t_1, \ldots, t_n$ *are* $n \geq 1$ *construct 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 construct terms.*

Construct terms that are not of the form *all t* are (simple kinds of) query terms and database terms are (simple kinds of) construct terms.

Example 7. Consider the database `bn.com` of Example 2. Assume that some query term (e.g. that of Example 3) binds the variables `TITLE` and `AUTHOR`. The construct term

```
results { result { TITLE, AUTHOR } }
```

assembles the bindings of `TITLE` and `AUTHOR` into new database terms, like e.g.

```
results { result { title { "TCP/IP Illustrated" },
                   author { last { "Stevens" }, first { "W." } } } }
```

In general, the variables `TITLE` and `AUTHOR` will have several bindings. The construct *all* may be used to collect all bindings:

```
results { all result { TITLE, AUTHOR } }
```

This yields as a result an unordered collection of `result` elements, collecting all possible combinations for `TITLE` and `AUTHOR`, e.g. like in

```
results { result { title { "TCP/IP Illustrated" },
                   author { last { "Stevens" }, first { "W." } } },
          result { title { "Advanced Programming in the Unix environment" },
                   author { "W. Stevens" } },
          ... }
```

□

Example 8. The XQuery expression of Example 1 returning for each author a list of all his titles can be expressed in Xcerpt as follows:

```
results { all result { AUTHOR, all TITLE } }
```

The symmetric query listing for each title all its authors is expressed in Xcerpt as follows:

```
results { all result { all AUTHOR, TITLE } }
```

The only difference between the two Xcerpt construct terms is the position of *all*. In constrast, XQuery requires two completely different queries (cf. queries Q3 and Q4 of use case "XMP" in [4]). □

3.4 Construct-Query Rules

Xcerpt Construct-query 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 term variable occurring (left or right of $\rightsquigarrow$ or elsewhere) 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".

Definition 5 (Construct-Query Rule). *A construct-query rule is an expression of the form* $t^c \leftarrow t^q_1 \wedge \ldots \wedge t^q_n$ *such that:*

1. $n \geq 1$ *and for all* $i = 1, \ldots n$, t^q_i *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^q_i.

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". In contrast to Prolog, the body of an Xcerpt rule cannot be empty.

Example 9. The following construct-query rule combines the query and construct terms used in the previous Examples 3 and 8:

```
rule { cons { results { all result { TITLE, all AUTHOR } }  },
       eval { in { "bn.com" } ,
              bib {{ book {{ TITLE ⇝ title, AUTHOR ⇝ author }} }} } }
```

□

The advantage of the clear separation between construct and query parts in Xcerpt is obvious, if you recall Example 1. The construct *in* in an *eval* par of a rule allows to specify a different resource for each query term:

Example 10. The following rule creates a list of books with their prices in both stores `bn.com` and `amazon.com`:

```
rule { cons { books { all book { title { TITLE },
                                 price-a { PRICEA },
                                 price-b { PRICEB } } } },
  and { eval { in { "bn.com" },
               bib {{ book {{ title { TITLE }, price { PRICEA } }} }} },
       eval { in { "amazon.com" },
              reviews {{ entry {{ title { TITLE },
                                  price { PRICEB } }} }} } } }
```

□

The conjunction of query terms in example 10 expresses an equijoin.

Xcerpt allows to "chain" rules, i.e. to evaluate one rule against the result of another rule. This allows for very complex queries and transformations, encapsulating subqueries and calculations in separate rules.

Example 11. Consider Example 10. Assume the data constructed is to be further transformed into two different formats, HTML [11] (suitable for PC screens) and WML (suitable for small screens). In Xcerpt, this could be expressed using additional rules that query the "result" of the first rule. A transformation into an HTML table and WML card could look like this:

```
rule { cons { table { tr { td { "Booktitle" },
                           td { "Price at A" },
                           td { "Price at B" } },
      all tr { td { TITLE }, td { PRICEA }, td { PRICEB } } } },
      eval { books {{ book { title { TITLE },
                             price-a { PRICEA },
                             price-b { PRICEB } } }} } }

rule { cons { all card { "Title: ", TITLE, br{},
                        "Price at A", PRICEA, br{},
                        "Price at B", PRICEB, br{} } },
       eval { books {{ book { title { TITLE },
                              price-a { PRICEA },
                              price-b { PRICEB } } }} } }
```

□

Both forward chaining (as in deductive databases) and backward chaining (as in Prolog) are possible and reasonable for processing Xcerpt rules. Backward chaining is often more efficient but requires a "unification" of query and construct terms. Xcerpt relies on Simluation Unification [2].

An Xcerpt program consists of one or several construct-query rules and of a "main query". Modules makes it possible to re-use parts of Xcerpt programs.

3.5 Further Language Constructs

The previous sections describe the basic constructs of the language Xcerpt. While these are sufficient for basic queries and transformations, a query language also needs to provide higher-level constructs. Xcerpt has e.g. different basic data types (with elementary text processing primitives), aggregation, user defined constraints, system and user-defined functions, a polymorphic type system, declarations and shadowing, modules.

4 Query Semantics

Xcerpt's query semantics is based on graph simulation. Informally, 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. The graphs considered are directed, ordered and rooted and their nodes are labelled.

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:*

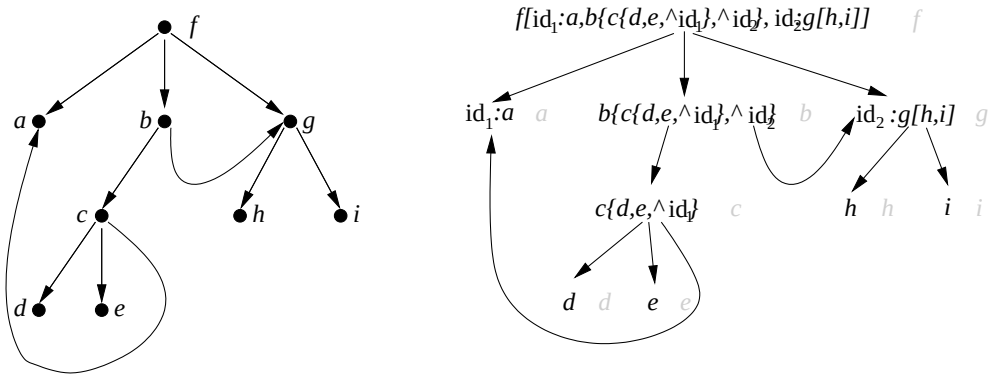
1. *If $v_1\ \mathcal{S}\ v_2$, then $v_1 \sim v_2$.*
2. *If $v_1\ \mathcal{S}\ v_2$ and $(v_1, v_1') \in E_1$, then there exists $v_2' \in V_2$ such that $v_1'\ \mathcal{S}\ v_2'$ and $(v_2, v_2') \in E_2$.*

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 $r_1\ \mathcal{S}\ r_2$.

Definition 7 (Graphs Induced by Strict and Ground Query Terms). *Let t be a strict and ground query term. The* graph $G_t = (N_t, V_t)$ induced by *t is defined by:*

1. *N_t is the set of strict subterms (cf. Definition 2) of t and each $t' \in N_t$ is labelled with the leftmost label (cf. Definition 2) of t'.*
2. *V_t is the set of pairs (t_1, t_2) such that either t_2 is a child subterm of t_1, or $\uparrow id$ is a child subterm of t_1 and the identifier definition id: t_2 occurs in t.*
3. *The children of a node are ordered in G_t like in t.*

Note that t is the root of G_t. Figure 1 illustrates Definition 7. The graph induced by a ground query term as defined in Definition 7 does not fully convey the term structure: [ ], { }, [[ ]] and {{ }} are not distinguished. Below, a database term is identified with the graph it induces.



(a) Abstract node representation

(b) Full node representation (node labels in gray)

Fig. 1. Graph induced by $t = f[\mathrm{id}_1 : a, b\{c\{d, e, \uparrow \mathrm{id}_1\}, \uparrow \mathrm{id}_2\}, \mathrm{id}_2 : g[h, i]]$.

Definition 8 (Ground Query Term Simulation). *Let t_1 and t_2 be ground query terms. Let S_i denote the set of subtrees of t_i ($i \in \{1, 2\}$). A relation $\mathcal{S} \subseteq S_1 \times S_2$ is a ground query term 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* $l_1\{\{t_1^1, \ldots, t_n^1\}\} \; \mathcal{S} \; l_2\{\{t_1^2, \ldots, t_m^2\}\})$, *then* $l_1 = l_2$ *and for all* $i \in \{1, \ldots, n\}$ *there exists* $j \in \{1, \ldots, m\}$ *such that* $t_i^1 \; \mathcal{S} \; t_j^2$.
4. *If* $l_1\{\{t_1^1, \ldots, t_n^1\}\} \; \mathcal{S} \; l_2\{t_1^2, \ldots, t_m^2\})$, *then* $l_1 = l_2$ *and for all* $i \in \{1, \ldots, n\}$ *there exists* $j \in \{1, \ldots, m\}$ *such that* $t_i^1 \; \mathcal{S} \; t_j^2$).
5. *If* $l_1\{t_1^1, \ldots, t_n^1\} \; \mathcal{S} \; l_2\{\{t_1^2, \ldots, t_m^2\}\})$, *then* $l_1 = l_2$ *and for all* $i \in \{1, \ldots, n\}$ *there exists* $j \in \{1, \ldots, m\}$ *such that* $t_i^1 \; \mathcal{S} \; t_j^2$), *and for all* $j \in \{1, \ldots, m\}$ *there exists* $i\{1, \ldots, n\}$ *such that* $t_i^1 \; \mathcal{S} \; t_j^2$.
6. *If* $l_1\{t_1^1, \ldots, t_n^1\} \; \mathcal{S} \; l_2\{t_1^2, \ldots, t_m^2\})$, *then* $l_1 = l_2$ *and for all* $i \in \{1, \ldots, n\}$ *there exists* $j \in \{1, \ldots, m\}$ *such that* $t_i^1 \; \mathcal{S} \; t_j^2$, *and for all* $j \in \{1, \ldots, m\}$ *there exists* $i\{1, \ldots, n\}$ *such that* $t_i^1 \; \mathcal{S} \; t_j^2$.

Definition 9 (Simulation Preorder). *$\preceq$ is the preorder on the set of ground query terms defined by $t_1 \preceq t_2$ if there exists a ground query term simulation of t_1 in t_2.*

Figure 2 illustrates Definition 8. The simulation of Figure 2 is minimal for $\subseteq$ in the sense that no strict subset of this simulation relation is a simulation of t^q in t^{db}.

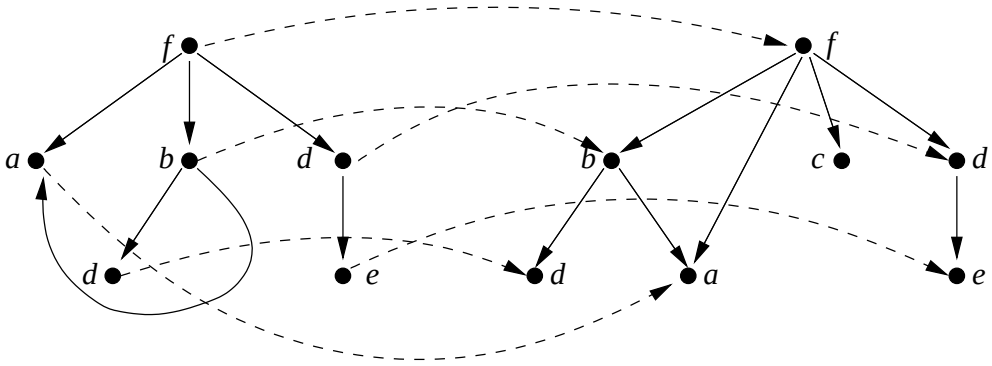


Fig. 2. A simulation of the (graph induced by the) ground query term $t^q = f\{\{\mathrm{id}_1 : a, b[d\{\}, \uparrow \mathrm{id}_1], desc\ e\}\}$ in the (graph induced by the) database term $t^{db} = f[b[d, \mathrm{id}_2 : a], \uparrow \mathrm{id}_2, c, d\{e\}]$.

By Definition 8, label identity is a rooted simulation of every ground query term in itself. By Definition 8, if $\mathcal{S}_1$ is a ground query term simulation of t_1 in t_2 and if $\mathcal{S}_2$ is a ground query term simulation of t_2 in t_3, then $\mathcal{S} = \{(l_1, l_3) \mid \exists l_2\ (l_1, l_2) \in \mathcal{S}_1 \wedge (l_2, l_3) \in \mathcal{S}_2\}$ is a ground query term simulation of t_1 in t_3. In other word, $\preceq$ is reflexive and transitive, i.e. it is a preorder on the set of database terms. $\preceq$ is not a partial 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), $t_1 = f\{a\} \neq t_2 = f\{a, a\}$.

Rooted simulation with respect to label equality is a first notion towards a formalisation of answers to query terms: If there exists a ground query term simulation of a ground query term t_1, in a database term t_2, then t_2 is an answer to t_1. An answer in a database D to a query term t^q is characterised by bindings for the variables in t^q such that the database term t resulting from applying these bindings to t^q is simulated in an element of D. Consider e.g. 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 yields the constraint $g\{\{b\}\} \preceq X \wedge g\{\{c\}\} \preceq X$. Matching t^q with the first database term in D yields the constraint $X \preceq g\{a, b, c\}$. Matching t^q with the second database term in D yields the constraint $X \preceq g\{b\} \wedge X \preceq g\{c\}$. $g\{b\} \preceq X \wedge g\{c\} \preceq X$ is not compatible with $X \preceq g\{b\} \wedge X \preceq g\{c\}$. Thus, the only possible value for X is $g\{a, b, c\}$, i.e. the only possible answer to t^q in D is $f\{g\{a, b, c\}, g\{a, b, c\}, h\}$.

Definition 10 (Ground Instances of Query Terms). *A* grounding substitution *is a function which assigns a label to each label variable and a database term to each term variable of a finite set of (label or term) variables. Let t^q be a query term, $V_1, \ldots, V_n$ be the (label or term) variables occurring in t^q and σ be a grounding substitution assigning v_i to V_i. The* ground instance *$t^q\sigma$ of t^q with respect to σ is the ground query term that can be constructed from t^q as follows: (1) Replace each subterm $X \rightsquigarrow t$ by X. (2) Replace each occurrence of V_i by v_i $(1 \leq i \leq n)$.*

Requiring in Definition 2 *desc* to occur to the right of $\rightsquigarrow$ makes it possible to characterise a ground instance of a query term by a grounding substitution. 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 11 (Allowed Instances). *The* constraint induced by a query term *t^q and a substitution σ is the conjunction of all inequalities $t\sigma \preceq X\sigma$ such that $X \rightsquigarrow t$ is a subterm of t^q not of the form desc t_0, and of all expressions $X\sigma \lhd t\sigma$ (read "$t\sigma$ subterm of $X\sigma$") 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 and $t^q\sigma$ a ground instance of t^q. $t^q\sigma$ is* allowed *if:*

1. *Each inequality $t_1 \preceq t_2$ in the constraint induced by t^q and σ is satisfied.*
2. *For each $t_1 \lhd t_2$ in the constraint induced by t^q and σ, t_2 is simulated in a subterm of t_1.*

Definition 12 (Answers). *Let t^q be a query term and D a database. An answer to t^q in D is a database term $t^{db} \in D$ such that there exists an allowed ground instance t of t^q satisfying $t \preceq t^{db}$.*

5 Conclusion

This article introduces the rule-based XML query and transformation language Xcerpt. While the World Wide Web Consortium has proposed XQuery as a generic XML query language, rule-based querying may be advantageous in cases involving more complex queries. Rule-based querying arguably allows for programs that are easier to grasp because of a clearer separation of construction and query parts.

In [2], a more detailed presentation of simulation unification is given. A prototype currently being worked on is presented at [9]. Commented examples of Xcerpt programs are given in [1].

References

1. François Bry and Sebastian Schaffert. A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proceedings of the Int. Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, http://www.soi.city.ac.uk/ msch/conf/ruleml/, 2002. Invited paper.
2. François Bry and Sebastian Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proceedings of the Int. Conf. on Logic Programming (ICLP)*, Copenhagen, 2002. Springer-Verlag LNCS 2401.
3. Peter Buneman, Mary Fernandez, and Dan Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDB Journal*, 9(1):76–110, 2000.
4. Don Chamberlin, Peter Fankhauser, Massimo Marchiori, and Jonathan Robie. XML query use cases. W3C Working Draft 20, December 2001.
5. Mary Fernandez, Jérôme Siméon, and Philip Wadler. XML Query Languages: Experiences and Examplars. Communication to the XML Query W3C Working Group, September 1999.
6. Monika R. Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing Simulations on Finite and Infinite Graphs. Technical report, Computer Science Department, Cornell University, July 1996.
7. D. Maier. Database Desiderata for an XML Query Language. In *Proceedings of QL'98 - The Query Languages Workshop*, 1998. http://www.w3.org/TandS/QL/QL98/.
8. Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking Forward. In *Proceedings of Workshop on XML Data Management (XMLDM)*, http://www.pms.informatik.uni-muenchen.de/publikationen/#PMS-FB-2002-4, 2002. Springer-Verlag LNCS.
9. Sebastian Schaffert. *Xcerpt Prototype.* http://demo.xcerpt.org, 2002.
10. W3C, http://www.w3.org/Style/XSL/. *Extensible Stylesheet Language (XSL)*, 2000.
11. W3C, http://www.w3.org/TR/xhtml1/. *XHTML 1.0: The Extensible HyperText Markup Language*, 2000.
12. W3C, http://www.w3.org/TR/xquery/. *XQuery: A Query Language for XML*, Feb 2001.