A Gentle Introduction to Xcerpt, a Rule-based Query and Transformation Language for XML

François Bry and Sebastian Schaffert

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

Abstract. This articles introduces into Xcerpt, a rule-based query and transformation language for XML. First, the design principles of Xcerpt are given. Then, the essential construct of Xcerpt are explained and illustrated on examples: "query terms", i.e. patterns using which Xcerpt queries are posed, "construct terms", i.e. pattern re-assembling the data selected in a query term into a new data item, and "construct-query rule" linking queries with construct terms. Then, Xcerpt and XQuery are compared on examples and the advantages of Xcerpt are discussed. Finally, an outlook into Xcerpt's declarative and procedural semantics as well as into Xcerpt's features currently developed are given.

1 Introduction

Essential to semistructured data [1] is the selection of data from incompletely specified data items. For such a data selection, a path language such as XPath [2] is convenient because it provides constructs similar to regular expressions such as $*$, $+$, ?, and "wildcards" that give 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.

Query and transformation languages developed since the mid 90es for XML [2] and semistructured data – e.g. XQuery [2], the precursors of XQuery [3], and XSLT [2] – 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. Thus, such languages intertwine construct parts, i.e. the construction patterns, and query parts, i.e. path selectors.

Example 1. An example for this intertwining of construct and query parts is the following XQuery query from the XQuery Use Cases [4]. This query creates a list of book titles for each author from a bibliography database like that of Example 2. It consists in a construction 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 construction pattern. Note that the (XPath) definitions of the variables `$a` and `$b` refer to a common sub-path `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)`.

```
<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 same query is expressed in Xcerpt in Example 19. □

The intertwining of construct and query parts as with XQuery can be found in dynamic HTML documents that contain (ECMAScript, Java Applets or Servlet) programs: The HTML page specifies a construction pattern, the programs corresponds to queries. The intertwining of construct and query parts has some advantages: For simple query-construct requests, the approach is rather natural and results in an easily understandable code. However, intertwining construct and query parts also has drawbacks:

1. Complex queries are often difficult to express and to understand,
2. unnecessarily complex path selections, e.g. XPath expressions involving both forward and reverse axes [5], are possible,
3. in presence of several path selections, as in Example 1, the overall structure of the retrieved data items might be difficult to grasp.

Among the query and transformation languages, UnQL [6] is a noticeable exception. This language first considered using patterns instead of paths for selecting semistructured data. Applying a kind of pattern matching algorithm (reminding of those pattern matching algorithms used in functional programming and in automated reasoning) to an UnQL query pattern and a (variable-free) database item binds the variables occurring in the query pattern to parts of the database item. This paper further investigates using query patterns. It describes Xcerpt, a query and transformation language for XML. Salient features of Xcerpt are the following:

1. Instead of (a form of) pattern matching, a (non-standard form of) unification, called simulation unification [7], is used giving rise to make two two query patterns both with variables identical.
2. Within an Xcerpt query pattern, a variable may be constrained to a (sub-)pattern.
3. The paradigm of Xcerpt is that of SQL and of logic programming. Thus, a query might have several answers. Some of (all, resp.) the answers to a (sub)query can be selected using the Xcerpt constructs `all` (`some`, resp.).
4. A chaining of queries expressed using rules makes it possible to split complex queries into simple parts.

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

2 Design Principles of Xcerpt

Query as form. An Xcerpt query corresponds to a form, an answer to a filling yielding a database item. It is possible to constrain a variable within an Xcerpt query to some pattern.
Referential transparency. The meaning of a subexpression is the same wherever it appears, i.e. destructive variable assignments are prohibited.
Compositional semantics. The semantics of an Xcerpt expression is defined recursively in terms of the semantics of its parts, i.e. Xcerpt has a Tarski-style model theory.
Multiple variable bindings. Like SQL queries and Logic Programming goals, Xcerpt queries might have several answers, each answer binding the query variables differently.
Separation of construction and query proper. In construct expressions variables may occur, but neither query expressions, nor conditions on variables.
Symmetry. Xcerpt queries allow similar forms of incomplete specifications in breadth, i.e. concerning siblings, and in depth, i.e. concerning children.
Circularity. Both, Xcerpt expressions and answers are query-able using Xcerpt, i.e. they are XML data. Note that this is more stringent than an XML format for Xcerpt queries.
Furthermore, the requirements of [8] are fulfilled by Xcerpt.

3 Basic Constructs of Xcerpt

3.1 Database Terms

Database terms are XML documents in a simplified syntax. Following a common practice in XML query language and semistructured data research [1], the children of a document node may be either ordered (as in SGML and in standard XML), or unordered (as in semistructured data). A database is an XML document or a set (or multiset) of XML documents.

Example 2. The following database term with root labeled `bib` describes the book offers of an online book stores called `bn.com` . This example is inspired from the XQuery Use-Cases [4].

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

In this database term, the square brackets [ ] around the `author` elements indicate that the list of authors is ordered. The curly brackets { } used in other elements indicate that their subelements are unordered. □

Example 3. The following database term with root labeled `reviews` gives the book offers of an online book stores called `amazon.com` . Like Example 2, this example is inspired from [4].

```
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." }
  }
}
```

□

Xcerpt database terms are "XML documents in disguise": Apart for the curly brackets ({ }) for unordered subelements that XML (still) does not supports, the conversion form Xcerpt syntax into standard XML syntax is straightforward, as the following example illustrates:

Example 4. Except for the element ordering which in (current) XML cannot be dispensed of, the database term of Example 3 denotes the following XML document:

```
<reviews>
  <entry>
    <title>Data on the Web</title>
    <price>34.95</price>
    <review>A good discussion of semi-structured database systems and XML.</review>
  </entry>
  <entry>
```

```
    <title>Advanced Programming in the Unix environment</title>
    <price>65.95</price>
    <review>A clear and detailed discussion of UNIX programming.</review>
  </entry>
  <entry>
    <title>TCP/IP Illustrated</title>
    <price>65.95</price>
    <review>One of the best books on TCP/IP.</review>
  </entry>
</reviews>
```

□

For expressing that the content of an element is not ordered, one might conveniently rely as follows on a Boolean attribute `ordered` (with default value `ordered = "yes"`).

Example 5.

```
<reviews ordered="no">
  <entry ordered="no">
    <title>Data on the Web</title>
    <price>34.95</price>
    <review>A good discussion of semi-structured database systems and XML.</review>
  </entry>
  ⋮
</reviews>
```

□

3.2 Query Terms

A query term is a pattern that specifies a selection of database terms very much like Prolog goals and SQL selections. The evaluation of query terms differs from the evaluation of logical atoms and SQL selections as follows:

1. Answers might have additional sub-terms to those mentioned in the query term.
2. Answers might have a different sub-term ordering than the query.
3. A query term might specify sub terms at an unspecified depth.

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

Example 6. The following Xcerpt query term has as an answer the database term of Example 2:

```
bib {{
  book {{
    title { "Data on the Web" },
  }}
}}
```

However, the database of Example 2 is not an answer to the following query term. Indeed, because of the single braces, answers to this query term cannot have more than one `book` element.

```
bib {
  book {{
    title { "Data on the Web" },
  }}
}
```

□

[ ] and [[ ]] are used if the sub-term 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-labeled database terms with no children.

Query terms may contain variables that are bound during query evaluation.

Example 7. The following query term queries the database of Example 2 for titles and authors. An answer to this query binds the variables `TITLE` and `AUTHOR` to the corresponding values:

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

The variables `TITLE` and `AUTHOR` are bound in several manners as follows: `AUTHOR = "Dan Suciu"` and `TITLE = "Data on the Web"` on the one hand, and `AUTHOR = "Serge Abiteboul"` and `TITLE = "Data on the Web"` on the other hand. □

Xcerpt evaluation of query terms is based on a non-standard evaluation called *Simulation Unification*. This form of unification, which is intuitively described on examples in the present article, is formally addressed in [7].

3.3 The Construct $\leadsto$ ("as")

In Example 7, variables are "leafs" (i.e. atomic sub-terms) of the query term. In Xcerpt, a variable can also occur at a "higher position" in a query term and be assigned a "lower bound" in form of a sub-term constraining the bindings for the variable.

Example 8. The following Xcerpt query binds the variable `TITLE` to the compound element `title { ... }` (thus retrieving not the "leaf" but the parent element `title`):

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

The constraint `TITLE` $\leadsto$ `title` expresses that only terms of the form `title{...}` can be bound to the variable `TITLE`. Without this constraint, terms with another label than `title`, e.g. terms of the form e.g. `price{...}`, could be bound to `TITLE`. □

Example 9. A variable can also be constrained to a compound query term like `AUTHOR` in the following query term:

```
bib {{
  book {{
    TITLE ⇝ title,
    authors {{
      author { AUTHOR ⇝ author{{ last{ "Suciu" }, first{ FIRSTNAME } }} }
    }}
  }}
}}
```

□

The advantage of constraining a variable to a sub-term *within* a query term, instead of *outside* the query term through an additional constraint, is to better convey the overall structure of the considered query. The query term of Example 9 better convey this structure than e.g. the following (ternary) conjunction:

Example 10.

```
and{
  bib {{
    book {{
      TITLE,
      authors {{ author { AUTHOR } }}
    }}
  }},
  TITLE ⇝ title,
  AUTHOR ⇝ author{{ last{ "Suciu" }, first{ FIRSTNAME } }
}
```

□

At a glance, one grasps from the query term of Example 9 the intended structure. With the (conjunctive) query term of Example 10, this structure must be constructed in thoughts. Arguably, the possibility of constraining variables *within* query terms appropriately realizes the "query-as-form" metaphor (cf. Section 2).

3.4 The Construct *descendant*

The *descendant* construct is used to express that a sub-term occurs at an indefinite depth. It is a counterpart of the Kleene star operator of regular path expressions and of XPath's `//`.

Example 11. The following query terms retrieves the titles of books with an author "Stevens" at any depth:

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

□

In contrast to the Kleene operator of regular path expression, the *descendant* construct of Xcerpt makes it possible to specify *several* variables at once and to express their relative positions within a term.

Example 12. The following query term retrieves `bib` elements containing an `author` element with a given structure and specifies variables within these elements:

```
bib {{
    desc author {{ last{ "LASTNAME" }, first { FIRSTNAME } }}
  }}
}}
```

□

Arguably, the *descendant* construct of Xcerpt contributes to realize the "query-as-form" metaphor (cf. Section 2).

In a query term, multiple occurrences of a same term variable are possible. E.g. a possible answer to $a\{\{X \rightsquigarrow b\{\{c\}\}, X \rightsquigarrow b\{\{d\}\}\}$ is $a\{b\{c, d\}\}$. However, $a[[X \rightsquigarrow b\{c\}, X \rightsquigarrow f\{d\}]]$ has no answers, for labels b and f are distinct. The $\rightsquigarrow$ construct makes it possible to express "variable cyclic" query terms such as $a\{\{\ X \rightsquigarrow b\{\{X\}\}\ \}\}$. Such "variable cyclic" query terms are undesirable, for their variables cannot be bound to finite database terms. Therefore, "variable cyclic" query terms are forbidden in Xcerpt.

3.5 Construct Terms

Xcerpt Construct terms serve to re-assemble variables, the values (or bindings) 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 for expressing ordered and unordered elements respectively. The constructs [[ ]] and {{ }} are not allowed in construct terms because these constructs that serves to express partial matches (in query terms) do not make sense when data items are constructed. Variables as references to sub-terms specified in a query can also occur in construct terms. The construct $\rightsquigarrow$ is not allowed in construct terms, the rationale for this being 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 (cf. Section 2).

Example 13. Assuming that some query term (e.g. that of Example 7) specifies values for the variables `TITLE` and `AUTHOR`, the following construct term assembles these values in a `result` element:

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

With the variable bindings `TITLE = title { "TCP/IP Illustrated" }` and `AUTHOR = author { last { "Stevens" }, first { "W." } }`, this construct term yields the following database term:

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

□

3.6 The Construct *all*

Recall that evaluating an Xcerpt query in general yields several bindings for each of its variables. It is sometimes needed to collect all the bindings for a variable in an answer. The construct *all* serves this purpose.

Example 14. The following construct term collects in a `results` element all `result` elements resulting from the various bindings of the variables `TITLE` and `AUTHOR`:

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

If the variables `TITLE` and `AUTHOR` are bound by an evaluation of the query term of Example 7 against the database term of Example 2, then the construct term given above yields the following database term:

```
results {
  result {
    title { "TCP/IP Illustrated" },
    author { last { "Stevens" }, first { "W." } }
  },
  result {
    title { "Advanced Programming in the Unix environment" },
    author { last { "Stevens" }, first { "W." } }
  },
   result {
    title { "Data on the Web" },
    author { last { "Abiteboul" }, first { "Serge" } },
    author { last { "Buneman" }, first { "Peter" } },
    author { last { "Suciu" }, first { "Dan" } }
  }
}
```

□

Formally, *all* t denotes the collection of all instances of t obtained from all possible bindings of the variables that are *free* in term t, *all* t' sub-terms of t being (recursively) evaluated in the same way. A variable X is *free* in a term t, if X does not occur in t within the scope of an *all* expression.

The *all* t construct gives rise to a simple and intuitive expression of queries that are rather complex in languages requiring an explicit iteration over answers.

Example 15. The following construct term returns for each author the list of his titles:

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

Compare this construct term with the much more complex XQuery expression of Example 1. The Xcerpt construct term given above is close to the the English specification, the XQuery expression of Example 1 is not. □

Example 16. The construct term returning for each title all its authors is expressed in Xcerpt as follows:

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

□

The symmetry of the natural language specifications of Examples 15 and 16 ("for each author, all title" and "for each title, all authors") is kept in the Xcerpt construct terms, but not in the corresponding XQuery expressions. The Xcerpt expressions Examples are very simple 15 and 16 compared with their XQuery counterparts (cf. Example 1). The Xcerpt construct terms of Examples 15 and 16 are very close to the the English specification, their XQuery counterparts are not (cf. Example 1).

3.7 The Construct *some*

It is sometimes useful to non-deterministically select a number $n \geq 1$ of answers, i.e. variable bindings. The construct *some* serves this purpose.

Example 17. The following construct term collects in a `results` element two `result` elements resulting from some (non-deterministically selected) bindings of the variables `TITLE` and `AUTHOR`:

```
results {
  some 2 result { TITLE, some AUTHOR }
}
```

If the variables `TITLE` and `AUTHOR` are bound by an evaluation of the query term of Example 7 against the database term of Example 2, then the construct term given above might yield the following database term:

```
results {
  result {
    title { "Data on the Web" },
    author { last { "Abiteboul" }, first { "Serge" } }
  }
  result {
    title { "Advanced Programming in the Unix environment" },
    author { { last { "Stevens" }, first { "W." } } }
  }
}
```

□

The construct *some* imposes conditions on the scopes of variables in construct terms: Two occurrences of a variable X within and outside a *some* expression, respectively, denote different variables. The rational for this scoping rule is that each of the variable occurrences might denote different sub-terms.

Example 18. The pairs of variables `TITLE` and `AUTHOR` in each line of the following construct term might be bound differently:

```
results {
  sometwo {
    some 2 result { TITLE, AUTHOR }
  },
  result { TITLE, AUTHOR }
}
```

If the variables `TITLE` and `AUTHOR` are bound by an evaluation of the query term of Example 7 against the database term of Example 2, then the construct term given above might yield the following database term:

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

□

Xcerpt's construct *some* makes it possible not to return a complete collection of data items in those cases where only a sample is needed. In querying data on the Web, incomplete data samples are often needed, e.g. for determining the schema of a Web site. The construct *some* can be seen as a declarative counterpart to Prolog's cut (`!`) and `once` primitives.

4 Construct-Query Rules

Xcerpt construct-query rules (short: rules) relate queries consisting in (*and* and *or*) connected query terms, and construct terms. In Xcerpt, both *and* and *or* may have an arbitrary arity $a \geq 1$ and may be nested. The connectives *and* and *or* have the intuitive meaning: *and* requires all sub-queries to be satisfied, *or* requires at least one of the sub-queries to be satisfied. In most cases, it is worth requiring that each variable occurring in the construct term of a rule is bound by every evaluation of the rule's query, i.e. variable in construct-query rules are "range-restricted" or "allowed". An Xcerpt rule has the form `rule{ cons { c }, query { q } } }` (also denoted `c` ← `q`) where `c` and `q` respectively denote a construct term and a query. The left hand-side, i.e. the construct term, of a rule is called the rule's "head", the right hand-side of a rule, the rule's "body". In contrast to the body of a Prolog clause, the body of an Xcerpt rule cannot be empty.

Example 19. The following construct-query rule relates the query and construct terms of Examples 7 and 15:

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

□

Note the *in* construct in the body (or query part) of the rule. This construct specifies the (URI of the) resource against which the rule's query is to be evaluated. A compound query might retrieve data from distinct resources. In the following example, the rule's body is a conjunction of two queries against distinct resources.

Example 20. The following rule collect for all books offered at both sites `bn.com` and `amazon.com`, their titles and their `bn.com` and `amazon.com` prices:

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

□

The conjunction of query terms in example 20 expresses an equijoin on book titles `TITLE`.

Example 21. The following rule collects all possible title/price pairs from each site `amazon.com` and `bn.com` and returns them in a unified format:

```
rule { cons {
    books {
      all book { title { TITLE }, price { PRICE } }
    }
  },
  or {
    query {
      in { "bn.com" },
      bib {{
        book {{ title { TITLE }, price { PRICE } }}
      }} },
    query {
      in { "amazon.com" },
      reviews {{
        entry {{ title { TITLE }, price { PRICE } }}
      }} }
  }
}
```

□

4.1 Ordered vs. Unordered Boolean Connectives

The connectives *and* and *or* may be used both with curly brackets { }, indicating that the evaluation order is of no importance as well as square brackets [ ], indicating that the query terms have to be evaluated in the specified the order. While `and {...}` and `or {...}` might be seen as more declarative and often give rise to optimization through reordering of their arguments, `and [...]` and `or [...]` allows a programmer a finer control over the evaluation. Note that in most programming languages, e.g. SML, Haskell, Prolog, Pascal, the Boolean connectives `and` and `or` have ordered arguments.

4.2 Boolean Connectives Within Query Terms

Consider the following compound query retrieving from a single resource `db` `country` elements with waters described by `sea`, `lake`, or `river` subelements:

Example 22.

```
or {
  query {
    in { "db" },
    desc country {{      COUNTRYNAME ⇝ name,
      sea
    }}
  },
  query {
    in { "db" },
    desc country {{
      COUNTRYNAME ⇝ name,
      lake
    }}
  },
  query {
    in { "db" },
    desc country {{
      COUNTRYNAME ⇝ name,
      river
    }}
  }
}
```

□

Xcerpt allows the following expression of the same query in which the *or* connective occurs within a single query term:

Example 23.

```
query {
  in { "db" },
  desc country {{
    NAME ⇝ name,
    or {
      sea,
      lake,
      river
    }
  }}
}
```

□

For the ease of programming as well as for an efficient query evaluation the query of Example 23 is preferable to the query of Example 22, for it stresses both, the common structure of the three alternatives of the disjunction, and the common resource against which each alternative of the disjunction is to be evaluated.

Like *or* connectives, *and* connectives are allowed within query terms. The semantics of a query term such as that of Example 23 is that of the correspond compound query (Example 22 in case of Example 23). The issue is addressed formally and in more details in [9].

5 Goals and Programs

An Xcerpt program consists of one or several rules and of one or several goals, i.e. queries, the answers to which are to be determined.

The following example is an Xcerpt program consisting of

1. a rule collecting in a "view" element `books` elements `book` giving in a common format the title and all the authors of each book available at each of the sites `amazon.com` (cf. Example 3) and `bn.com` (cf. Example 2).
2. a rule returning in a "view" element `bibliography` for each author in the "view" `books` the collection of the books he or she authored
3. a goal requesting to output in the resource `buneman-bibl.xml` the bibliography of the author Buneman,
4. a goal requesting to output in the resource `authors.xml` the authors of the book entitled "Data on the Web".

Example 24.

```
rule { cons {
    books {
      all book { title { TITLE }, authors { all AUTHOR } }
    }
  },
  or {
    query {
      in { "bn.com" },
      bib {{
        book {{ title { TITLE }, desc author { AUTHOR } }}
      }} },
    query {
      in { "amazon.com" },
      reviews {{
        entry {{ title { TITLE }, desc author { AUTHOR } }}
      }} }
    }
  }

rule {
  cons {
    bibliography {
      all author { AUTHOR, all book{ TITLE } }
    }
  },
  query {
    books {{
      book {{ TITLE ⇝ title, authors {{ AUTHOR ⇝ author }} }}
    }}
  }
```

```
goal{
  out{ "buneman-bib.xml" },
  bibliography { desc author { desc "Buneman" } }
}

goal{
  out{ "authors.xml" },
  books { book {  title { "Data on the Web" }, authors } }
  }
```

□

In the Xcerpt program of Example 24, the query of the second rule is assigned no resources. This means that the "working space" of the evaluator is the resource to query. With the construct *out*, the goals are assigned resources where the computed results are to .

In contrast to a program's goal, a rule's query is not necessarily to be evaluated. If it is to be evaluated, the values occurring in the program's goals might constrain the rule evaluation. E.g. with the program of Example 24, an efficient Xcerpt processor would query the sites `amazon.com` and `bn.com` only for the book entitled "Data on the Web" or for books one authors of which is "Buneman". Such a backchaining of values from goals, i.e. data retrieval requests, to the queries eventually posed to resources is essential for an efficient evaluation of queries on the Web. Without such a backchaining, complex queries would soon induce querying by far more Web sites and collecting more data than possible. Thus, Xcerpt rules specify views, in the database sense, that are virtual in the sense that they are not necessarily evaluated during the evaluation of a query.

Both a forward chaining (as in deductive databases) and a backchaining (as in Prolog) are possible for processing Xcerpt programs. Note that backchaining requires to "unify" query and construct terms that might both contain unbound variables. To this aim, Xcerpt relies on a non-standard unification called "Simulation Unification', cf. Section 9.1 and [7].

Allowing several goals in an Xcerpt program makes a combined evaluation and a sharing of intermediate results needed for answering several goals possible, thus giving rise to efficient evaluations.

6 Rule Chaining

Xcerpt allows to "chain" rules, i.e. to refer to the head of a rule in the body of another rule. Remind (cf. the previous Section 5) that Xcerpt rules specify virtual views that in general are never fully computed during the evaluation of the goals. Therefore, chaining rules is a convenient mean for a stepwise expression of complex queries and/or transformations which does not necessarily result in an inefficient evaluation.

Example 25. Consider the rule of Example 20. Assume the data retrieved is to be further transformed into two different formats, HTML [10] (suitable for a PC screen) and WML [11] (suitable for the small screen of a Personal Digital Assistant). This can be expressed adding the following rules to that of Example 20 that both query the "result" of the rule of Example 20.

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

□

7 Further Constructs of Xcerpt

7.1 Attributes

In Xcerpt's simplified syntax, attributes are expressed as follows.

Example 26. The following `entry` element has two attributes named `language` and `isbn`. The values of these attributes are the book's language (denoted "`en`") and the book's ISBN number respectively.

```
entry {
   att{ language { "en" }, isbn { "1-55860-622-X" } }
   title { "Data on the Web" },
   price { "34.95" },
   review { "A good discussion of semi-structured database systems and XML." },
  }
```

□

XML `ID` and `IDREF` attributes have special notations in Xcerpt's simplified syntax: An `ID` attribute `a` attached to element *elt* is denoted by `a : ` *elt* and an `IDREF` attribute referring to *elt* is denoted by ↑`a`.

Example 27. Example 2 is modified such that the data about the authors are stored only once and referenced to at different places:

```
bib {
  a1: author { last{ "Stevens" }, first { "W." } },
  a2: author { last{ "Abiteboul" }, first { "Serge" } },
  a3: author { last{ "Buneman" }, first { "Peter" } },
  a4: author { last{ "Suciu" }, first { "Dan" } },

  book {
    title { "TCP/IP Illustrated" },
    authors [ ↑ a1 ],
    publisher { "Addison-Wesley" },
    price { "65.95" }
  },
  book {
    title { "Advanced Programming in the Unix environment" },
    authors [ ↑ a1 ],
    publisher { "Addison-Wesley" },
    price { "65.95" }
  },
  book {
    title { "Data on the Web" },
    authors [ ↑ a2, ↑ a3, ↑ a4 ],
    publisher { "Morgan Kaufmann Publishers" },
    price { "39.95" }
  }
}
```

□

7.2 String and Label Retrieval

Variables and regular expressions on strings for both, text data and labels (i.e. XML tags), are supported by Xcerpt.

7.3 Groups

Group constructs make it possible to specify *in* and/or *out* resources for several rules and/or goals.

Example 28.

```
group{
  out { "output1.xml" }, in { "db1.xml",  "db2.xml" },
  rule {  cons { c1 { ... } }, query { q1  { ... } }  },
  rule {  cons { c2 { ... } }, query { q2  { ... } }  },
  rule {  cons { c3 { ... } }, query { in { "db3.xml" }, q3  { ... } }  },
  goal{  c1 { ... }  },
  goal{  out{ "output2.xml", "output3.xml" }, c2 { ... }  }
  }
```

□

The *in* and *out* resources specified in the `in` and `out` elements of the `group` hold for the rules and goals in the group: The answers to both goals are to be stored in `output1.xml` and the queries are to be evaluated against (both of) the resources `db1.xml` and `db2.xml`. The scope of the*in* and *out* specifications is the group in which they occur.

The third rule's query, however, mention another *in* resources, `db3.xml`, which overrides *for this rule's query* `db1.xml` and `db2.xml` that are specified at a higher level. Similarly, the *out* resources `output2.xml` and `output3.xml` specified in the second goal overrides *for this goal* the `output1.xml`: The answers to this goal are to be stored in (both of) `output2.xml` and `output3.xml`.

8 Elements for a Comparison of Xcerpt and XQuery

Queries expressed in XQuery are sometimes more compact than their counterpart in Xcerpt. However, XQuery queries are often more difficult to comprehend than Xcerpt queries. XQuery expression often do not well convey the structure of the data items to be retrieved like e.g. in the following examples.

Example 29. Consider the database term of Example 2. In the following, the left query lists for a title, all its authors, and the right query lists all authors for a title. Expressed in XQuery, the queries are as follows:

```
<results>
{
 for $b in
  document
    ("http://bn.com")/bib/book
 return
   <result>
     { $b/title }
     { $b/author  }
   </result>
}
</results>
```

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

The same queries are expressed in Xcerpt as follows:

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

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

□

The *only* difference between both Xcerpt rules is the position of the *all* construct in the rules' construct term. Reflecting a common understanding of the natural language requests, the rules' query parts are identical. In contrast, XQuery requires two completely distinct queries.

Furthermore, the meaning of the Xcerpt rules is arguably easier to grasp als that of the XQuery expressions.

9 Outlook into Xcerpt's Semantics

This section is an informal introduction into Xcerpt's evaluation strategy. Refer to [7] for a more detailed presentation. Xcerpt's evaluation strategy has been implemented in a prototype which can be found at [12]. Xcerpt's Procedural Semantics is closely related to evaluation strategies used in Constraint and Logic Programming.

9.1 Procedural Semantics: Simulation Unification

Xcerpt relies on a non-standard unification called "Simulation Unification" [7]. This non-standard unification is indispensable for the following reasons:

- *partial patterns* in a query term (expressed in Xcerpt with the double brackets {{ }} and [[ ]]) have to be properly processed selecting terms that might have subtermds not explicitely mentioned in the query term,
- the *descendant* construct must be dealt with,
- *ordered* and *unordered* terms (expressed in Xcerpt with the brackets { } and [ ]) have to be handled according to their semantics.

A so-called *simulation preorder* [7] as the underlying theory provides with the necessary semantics to handle these properties. A *simulation* is a relation between two graphs where intuitively one graph simulates in the other if all its vertices and corresponding edges can be found in this graph (see Figure 1). Two ground query terms t_1 and t_2 are in the simulation preorder, written $t_1 \preceq t_2$, if t_1 can be simulated in t_2. For two terms t_1 and t_2, a simulation unifier is a substitution σ such that $t_1\sigma \preceq t_2\sigma$.

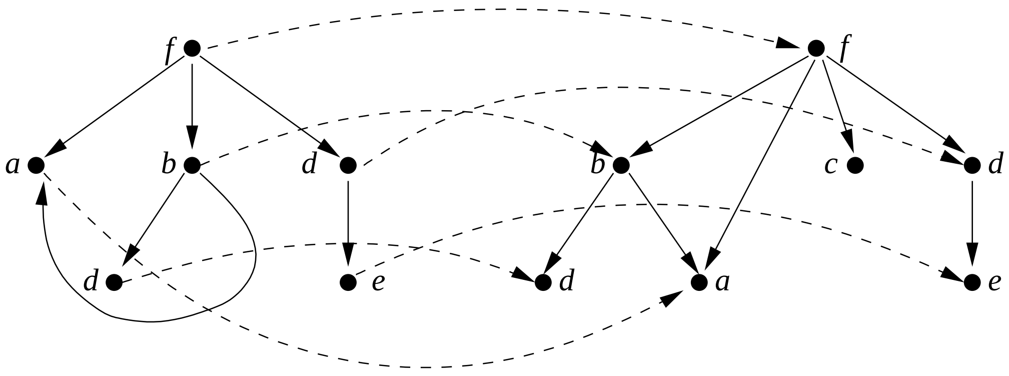

Fig. 1. Ground Query Term Simulation between two graphs
The simulation relation is illustrated by the dashed arrows. Note that the graph on the right hand side may contain additional nodes not represented in the left graph.

As a consequence of using the simulation preorder $\preceq$ instead of equality as in standard unification, two terms t_1 and t_2 several incomparable "unifiers" . This is trhe case e.g. if $t_1 = f\{\{X\}\}$ and $t_2 = f\{a, b\}$: The two substitutions $\sigma_1 = \{X = a\}$ and $\sigma_2 = \{X = b\}$ are relevant).

Instead of returning each of these substitutions separately, the simulation unification algorithm described in [7] returns a formula consisting of "and" and "or" conected constraints on the variables (in the above mentioned case, this formula is $\sigma = X = a \vee X = b$).

9.2 Procedural Semantics: Inferences

Using Simulation Unification, several form of rule processing are conceivable. Forward chaining, as in deductive databases, could be an appropriate way to perform e.g. a transformation of a whole XML database. Backward chaining appears more adequate for answering queries posed to the Web or to a very large database, cf. Section 6.

In either way, a chaining of rules can be realised using a constraint solver. Instead of immediately committing variables to a binding (e.g. $\sigma = X = a \vee X = b$), variables are kept in inequations as long as possible (e.g. $\sigma = X \preceq a \vee X \preceq b$). A (simple) constraint solver is then used for detecting and removing inconsistencies. In the formula $a \preceq X \wedge (X \preceq a \vee X \preceq b)$, the branch $X \preceq b$ is removed, yielding $a \preceq X \wedge X \preceq a$. As a final step, all inequations that provide an upper bound for a variable are replaced with a binding of that variable to this upper bound (i.e. $a \preceq X \wedge X = a$).

9.3 Declarative Semantics

Xcerpt's declarative semantics is described in [13]. It is defined in terms of a model theory à la Tarski, i.e. the declarative semantics of a compound expression is defined in terms of its components.

10 Perspectives

In this article, the query and transformation language Xcerpt has been introduced on examples. Its design principles and its basic constructs have been introduced and illustred.

While the basic constructs introduced above are sufficient for elementary queries and transformations, a query language also needs additional tools (e.g. arithmetics and aggregations functions) as well as advanced features (e.g. a type system making a type-checking at compile time possible). The following additional features of Xcerpt are currently under development. (Note that the following list is not complete.)

Basic datatypes. In this article, only string data are mentioned, although Xcerpt's support of various basic scalar types such as different kinds of numbers (e.g. integers and reals) is under development. The (current) view is that Xcerpt will support the "simple types" of XML Schema [14,15,16] including basics operations on these types (such as e.g. basic arithmetics on number types).

Elementary text processing primitives. In addition to the primitives forseen in XML Schema [14,15,16] for the simple types "string", "normalizedString", and "token", Xcerpt includes primitives (inspired from Perl [17]) for an an elementary text processing – among others, regular expressions for text selection.

Aggregation. In re-assembling answers to queries into new data items, one often needs to collect several answers (as with the `all` construct, cf. Section 3.5) or to compute values (such as an average) from collected answers. In addition to the `all` construct, Xcerpt supports standard aggregation primitives such as `avg` (average), `max`, `min` for number datatypes, and `concat` (concatenation) for text datatype.

User defined constraints. Xcerpt allows the user to specify additional constraints to variables occuring in query terms. These user defined constraints may be expressed in terms of simulation unification (cf. Section 9.1) or using system or user-defined functions.

system and user-defined functions. It is possible to refer in an Xcerpt program to functions specified (e.g. by the user) outside the Xcerpt program.

Polymorphic Type system. A type system has two advantages: Programming errors can be detected at compile time (thus supporting program development) and the processing of queries can be more efficient. A extensible polymorphic type system à la ML [18] for Xcerpt is under development using which user defined types are expressed in an XML Schema syle [14,15,16].

Declarations and shadowing. Variable and type declarations local to part of an Xcerpt program make it possible that some definitions and names are local to a program part. Shadowing makes it possible to to differently binds same names within different program parts.

Modules. Modules aqrte under developments using which parts of Xcerpt programs can be imported and exported so as to combine parts of programs in different manners and to hide parts of programs that have no global relevance.

Xcerpt is an experimental language. First experiments with practical examples, e.g. that of the XQuery Use Cases [4], suggest that Xcerpt's main features are convenient. In the furture, some aspects of Xcerpt might be revised and/or refined, depending on the experience collected. Experiments with a prototype show that pattern queries, i.e. one the salient features of Xceprt, can be efficiently evaluated.

References

1. Abiteboul, S., Buneman, P., Suciu, D.: Data on the Web. From Relations to Semistructured Data and XML. Morgan Kaufmann (2000)
2. World Wide Web Consortium (W3C) http://www.w3.org/. (2002)
3. Fernandez, M., Siméon, J., Wadler, P.: XML Query Languages: Experiences and Examplars. Communication to the XML Query W3C Working Group (1999)
4. Chamberlin, D., Fankhauser, P., Marchiori, M., Robie, J.: XML query use cases. W3C Working Draft 20 (2001)
5. Olteanu, D., Meuss, H., Furche, T., Bry, F.: XPath: Looking Forward. In: Proceedings of Workshop on XML Data Management (XMLDM), http://www.pms.informatik.uni-muenchen.de/publikationen/#PMS-FB-2002-4, Springer-Verlag LNCS (2002)
6. 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
7. Bry, F., Schaffert, S.: 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, Springer-Verlag LNCS (2002)
8. Maier, D.: Database Desiderata for an XML Query Language. In: Proceedings of QL'98 - The Query Languages Workshop. (1998) http://www.w3.org/TandS/QL/QL98/.
9. Bry, F., Olteanu, D., Schaffert, S.: Grouping constructs for semistructured data. In: Proceedings of DEXA 2001, Munich (2001)
10. W3C http://www.w3.org/TR/xhtml1/: XHTML 1.0: The Extensible HyperText Markup Language. (2000)
11. WAP Forum http://www.wapforum.org: Wireless Markup Language (WML). (2000)

12. Schaffert, S.: Xcerpt Prototype, http://demo.xcerpt.org. (2002)
13. Bry, F., Schaffert, S.: The XML Query Language Xcerpt: Design Principles, Examples and Semantics. Technical report, Institut für Informatik, LMU München (2002)
14. W3C http://www.w3.org/TR/xmlschema-0/: XML Schema Part 0: Primer. (2001)
15. W3C http://www.w3.org/TR/xmlschema-1/: XML Schema Part 1: Structures. (2001)
16. W3C http://www.w3.org/TR/xmlschema-2/: XML Schema Part 2: Datatypes. (2001)
17. Wall, L., et al: Practical Extraction and Report Language, http://www.perl.com/. (1987-2002)
18. Lucent Technologies, Bell Labs http://cm.bell-labs.com/cm/cs/what/smlnj/: Standard ML of New Jersey. (1996)