INSTITUT FÜR INFORMATIK

Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67, D–80538 München

Querying the Web Reconsidered:
A Practical Introduction to Xcerpt

Sebastian Schaffert and François Bry

Querying the Web Reconsidered
A Practical Introduction to Xcerpt

Abstract

This article gives a practical introduction into the language Xcerpt, guided by many examples for illustrating language constructs and usage. Xcerpt is a rule-based, declarative query and transformation language for XML data. In Xcerpt, queries and the (re-)structuring of answer (also called "constructions") are expressed in terms of patterns instead of path navigations (like in XSLT and XQuery). Pattern queries are evaluated against XML documents using a non-standard form of unification (called "simulation unification" [BS02]). Furthermore, Xcerpt supports so-called rule chaining (with recursion), which makes it suitable even for complex query programs. Due to its foundations in logic programming, Xcerpt can also serve to implement reasoning algorithms for the Semantic Web.

Querying the Web Reconsidered
A Practical Introduction to Xcerpt

Table of Contents

Querying the Web Reconsidered
A Practical Introduction to Xcerpt

Sebastian Schaffert and François Bry

§ 1 Introduction

XML, the Extensible Markup Language, is nowadays used in a wide area of applications, ranging from text oriented documents over languages aimed at exchanging messages between peers to data centric formats for exchanging and storing data. Querying and transforming data in XML format has thus received much attention, and the languages XSLT [XSLT] and XQuery [XQuery] proposed by the W3C have become de facto standards for this purpose, with XSLT being the more widespread of the two.

Also, one of the major endeavors in current Web research is the so-called "Semantic Web" [SemanticWeb]. Briefly summarised, the goals of the Semantic Web can be described as enriching the existing Web with meta-data and data processing (and meta-data processing) so as to provide Web-based systems with advanced (so-called intelligent) capabilities, in particular with *context-awareness* and *decision support*.

Obviously, processing Semantic Web data also needs query languages, but although the Semantic Web is mostly XML based, XSLT and XQuery are not well suited for the task, as they lack reasoning capabilities. Therefore, several new query and reasoning languages for Semantic Web reasoning have been proposed. However, all current proposals are special purpose languages in that they only implement specific forms of reasoning, (e.g. those of a certain description logic [Ho99]), and are only capable of querying certain kinds of data (e.g. RDF or OWL).

This tendency to develop special purpose query and reasoning languages appears questionable, as XML query languages should be able to profit from semantic information in all possible formats and likewise, queries to Semantic Web data should be able to also query XML content. The query language Xcerpt presented in this article therefore aims at being capable of both, querying XML content and reasoning with Semantic Web data.

In contrast to XML query languages like XSLT and XQuery, Xcerpt is a declarative, rule based query language with much of its foundations in Logic Programming and thus appears very suitable for reasoning on the Semantic Web. As an additional bonus, Xcerpt's declarativeness supports non-programmers and programmers with limited experience in formulating queries and helps structuring complex query programs.

In contrast to special purpose Semantic Web query languages, Xcerpt is a general purpose language that can query any kind of XML data. Thus, in Xcerpt it will be possible to implement a wide range of reasoning methods as needed on the Semantic Web.

This article gives a practical introduction into expressing simple and advanced Web queries in Xcerpt. Section 2 gives an overview over design principles that guided the development of Xcerpt. Section 3 introduces the core concepts of the language, i.e. term patterns, rules and programs. Further extensions that are useful in practical applications are discussed in Section 4. Section 5 describes how to use rule chaining to build complex query programs. A brief overview over Xcerpt's semantics and implementation is given in Section 6. Section 7 introduces a visual language called visXcerpt, which takes advantage of the declarative and pattern based nature of Xcerpt in that it merely needs to *render* Xcerpt queries instead of developing a new language. Finally, Section 8 gives an overview over current and future work related to Xcerpt.

§ 2 Principles of Xcerpt

The following language principles have prevailed to the definition of the language Xcerpt:

2.1 *Declarative, Rule Based Querying*

Xcerpt programs consist of derivation rules (or if-then rules). An Xcerpt rule specifies declaratively a simple or compound query and relate it to a term giving a structure to (one speaks of "constructing") answers. They are often easy to use by novice users and enhance readability.

2.2 *Pattern Based Querying and Construction*

Patterns are a declarative specification of a query (and its answer) and have a structure that is very close to the queried data. Query patterns in addition support partial specifications, which allow to omit parts that are irrelevant to the query. Query and construction patterns can thus be considered as *examples* of XML documents. Arguably, this similarity between query and the queried resource also improves usability for both, the novice and the experienced user.

In Xcerpt, queries and answer structures, expressed by so-called "construct terms", are expressed by patterns except for special aspects like arithmetical conditions that are more conveniently expressed by other means such as (in)equations. For example, patterns are used both for the *selection* of data (in so-called query terms) and for the *construction* of new data (in so-called construct terms).

There are some exceptions to this principle where a pattern is either not possible or not desirable, like arithmetic constraints. Such extensions are discussed in Section 4

2.3 *Incomplete Queries*

As data on the Web often have no common schema or DTD, and schema information is often incomplete, it is necessary that query patterns allow an incomplete specification of the matched structure. Such incompleteness is desirable both in depth (i.e. for selecting subterms that are located at an arbitrary level of nesting, without taking into account other subterms on the path) and in breadth (i.e. for omitting such subterms at the same level that are irrelevant to the query).

In addition, queries referring to subterms that appear only in some data items are desirable. So as to conveniently cope with such optional subterms in queries, construct terms might refer to default values (e.g. retrieve email addresses in an address book if they exist or produce a string "unknown" otherwise).

2.4 *Separation of Querying and Construction*

Xcerpt consequently follows a strict separation of querying and answer structuring (i.e. "construction"). A query part is strictly limited to retrieving data and constraining the possible results. A construction part is strictly limited to reassembling data and creating new data, and never can impose restrictions on the query part.

2.5 *Backward and Forward Chaining*

Unlike most other rule-based XML query languages, rules in Xcerpt may interact with each other by *rule chaining*, i.e. the query part of one rule can query the results of another rule. Rule chaining allows to develop complex and well-structured programs. Xcerpt rules give rise to structure complex programms. They correspond to subroutines, procedures, or functions of other languages.

2.6 *Recursion*

An important aspect of rule chaining is that recursive Xcerpt programs are possible, i.e. a cycle of "rule queries / rule constructions" dependencies might exist in an Xcerpt program. Whereas many other XML query languages only allow a structural recursion over the input document (e.g. XSLT), rule chaining in Xcerpt can be used for all kinds of recursion, e.g. over *conceptual* structures (i.e. such structures that are not explicitly given but rather specified by the semantics of the data, like "all section titles") or over Web links (given in terms of URIs).

§ 3 Xcerpt Core Constructs

The constructs of the language Xcerpt can be divided into *simple constructs* and *advanced constructs*. The simple constructs, which are introduced in this section, are those constructs that are necessary for structural querying and construction. This basically includes query and construction patterns, rules and goals. Advanced constructs (Section 4) are constructs that are not strictly necessary for the pure task of querying and rearranging data, but make Xcerpt suitable and convenient for practical use. This includes, among others, constructs like functions and aggregations, negation and regular expressions.

An Xcerpt program may consist of at least one *goal* and of some (maybe zero) *rules*. Goals and rules are composed of data, query and construct terms representing respectively XML documents, query and XML documents constructed from the answers to queries. In addition to the "compact" syntax presented in the following, Xcerpt also provides an XML syntax. An non-XML "compact" syntax is desirable for readability reasons, as programming with XSLT that has no concrete syntax sufficiently demonstrates.

3.1 *Data, Query and Construct Terms*

Common to all terms is that they represent tree-like (or graph-like) structures. Square brackets (i.e. [ ]) denote *ordered term specification* (as in standard XML), i.e. the matching subterms in the queried resource are required to be in the same order as in the query term. Curly braces (i.e. { }) denote *unordered term specification* (as is common in databases), i.e. the matching subterms in the queried resource may be in arbitrary order.

Single (square or curly) braces (i.e. [ ] and { }) are used to denote that a matching term must contain matching subterms for all subterms of a term and may not contain additional subterms (*total term specification*). Double braces (i.e. [[ ]] and {{ }}) are used to denote that the data term may contain additional subterms as long as matching partners for all subterms of the query term are found (*partial term specification*).

Graph structure is expressed using a reference mechanism. The construct id @ t is a *defining occurrence* of the identifier id and the construct ^id is a *referring occurrence*.

3.1.1 Data Terms

Data terms are used to represent XML documents ("XML in disguise") and the data items of a semistructured database. They are similar to *ground* functional programming expressions and logical atoms. Data terms may only contain the single square and curly braces described above, but no partial specifications, as a data item is always considered to be complete.

In this context, a *database* is a (multi-)set of data terms (e.g. the Web). Note, however, that a single data term is often used to represent what is commonly referred to as a "database": an address book which might contain a lot of entries can be represented in a single data term (or XML document), whereas a relational database would store each entry as a separate item.

Example: The following data term represents this article in Xcerpt syntax. For obvious reasons, most parts are omitted. Note that some parts of this document use unordered term specification (e.g. the author entries), as the order is irrelevant.

```
paper [
   front [
      title [ "Querying the Web Revisited" ],
      author {
         fname [ "Sebastian" ], surname [ "Schaffert" ],
         address { ... },
         bio [ ... ]
      },
      author {
         fname [ "François" ], surname [ "Bry" ],
         address { ... },
         bio [ ... ]
      },
   ],
   body [
      section [
         title [ "Introduction" ],
      ],
      ...
   ],
   rear [
      acknowl [ ... ],
      bibliog {
         bibitem [
            bib [ "XQuery" ],
            pub [ "XQuery: The XML Query Language ..." ]
         ],
         ...
      }
   ]
]
```

Data terms induce a graph in a straightforward manner. Figure 1 shows an (incomplete) graph representation of the data term used in the previous example.
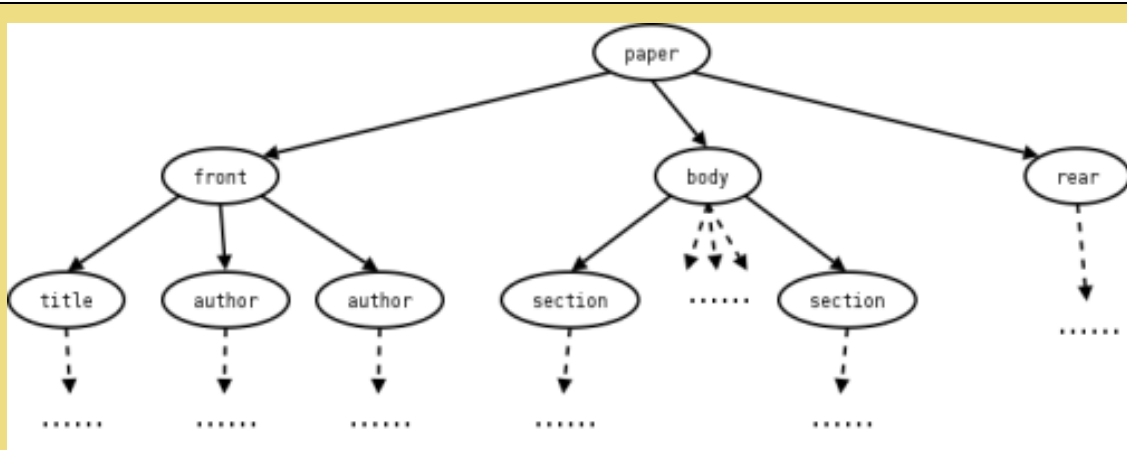
Figure 1

Graph induced by the data term that represents this article. References to parts not illustrated are shown as dashed arrows.

As usual, graphs and trees are represented with the root at the top and the leaves at the bottom. In this article, *breadth* accordingly always refers to the maximal branching level in the tree, wheras *depth* always refers to the length of the maximal path in the tree or graph. Be aware that the pretty-printed textual representation of XML documents or data terms might induce a converse denomination, where the depth refers to the length of the document and the breadth refers to the nesting level.

XML attributes are represented in Xcerpt data terms as a special kind of subterms. An XML attribute of the form name="value" is represented as a subterm of the form name{"value"}. So as to distinguish proper subterms from attributes, all attributes are grouped inside a subterm with the reserved label attributes, which has to appear as the first subterm of a term (regardless of whether the term is unordered or ordered).

Example: The XML fragment on the left (describing a figure with a unique id attribute in the Extreme Markup Languages DTD) corresponds to the data term on the right hand side.

```
<figure id="fig:graph">            figure [
                                      attributes { id{"fig:graph"} },
<graphic scale="80"                   graphic [
        figname="figure1"/>             attributes {
                                          scale{"80"},
                                          figname{"figure1"} }
                                      ]
</figure>                          ]
```

Note that data terms do not cover all constructs found in XML. Constructs like Processing Instructions are intentionally left out because they do not add important information to the data represented.

Xcerpt deliberately uses a non-XML syntax. Experience with XSLT shows that many programmers are uncomfortable with expressing programming language constructs in a very verbose XML notation, and programs arguably become cluttered and difficult to grasp merely due to the verbose notation. The term syntax used in Xcerpt is more lightweight, and can be easily transformed into XML and vice versa using a simple lexer (with a stack to remember the nesting level for the closing tags). Such a transformation can even be performed transparently by an advanced editor, like in the language visXcerpt introduced below.

3.1.2 Query Terms

Query terms are partial patterns that are matched with data terms, augmented by an arbitrary number of variables for selecting data items from a data term. In addition to the constructs used in data terms, query terms have the following additional properties:

- *partial specifications* omitting subterms irrelevant to the query are possible (indicated by double square brackets or curly braces),
- it is possible to specify subterms at *arbitrary depth* (indicated by the keyword *desc*).
- they may contain *term variables* and *label variables* to "select" data from the database (variables are written in upper case letters below)

Thus, query terms are similar to *non-ground* functional programming expressions and logical atoms.

The Xcerpt construct X -> t (read "as") serves to associate a query term to a variable, so as to specify a restriction of its bindings. The Xcerpt construct *desc* (read "descendant") is used to specify subterms at arbitrary depth.

Example: Suppose that all articles of the Extreme04 conference are contained in a proceedings04 element. The following query term selects title and author pairs for each article:

```
proceedings04 {{
   article {{
     front {{
       var T -> title {{,}}
       var A -> author {{'}}
     }}
   }}
}}
```

The reference mechanism using ^id and id @ t is resolved internally when a query is evaluated. Programmers thus do not need to distinguish between usual term-subterm relationships and references, they are conceptually treated in the same manner.

Query terms are *unified* with data or construct terms using a non-standard unification called *simulation unification*, which is described in [BS02]. Simulation unification is based on *graph simulation* [ABS00] which is a relation similar to graph homomorphisms.

The outcome of unifying a query term with a data term is a set of substitutions for the variables in the query term, where each substitution represents an alternative solution. Applying each of these substitutions to the query term yields a ground query term which is simulated (in the sense of [ABS00]) in the data term.

3.1.3 Construct Terms

Construct terms serve to reassemble variables (the bindings of which are specified in query terms) so as to construct new data terms. They may only contain single brackets and variables, but no partial specification or variable restrictions. The rationale of this is to keep variable specifications within query terms, ensuring a strict separation of purposes between query and construct terms.

Example: The following construct term creates an Author-Title pair wrapped in a "result" element:

```
result {
  var A, var T
}
```

In a construct term, the Xcerpt construct all t serves to collect (in the construct term) all instances of t that can be generated by alternative substitutions for the variables in t (returned by the associated query terms in which they occur). Likewise, some n t serves to collect at most n instances of t that can be generated in the same manner.

Example: In the following construct term, the all construct is used to create a list of publications for each author:

```
results {
  result {
    var A,
    all var T
  }
}
```

If several variables are in the scope of an all construct, all combinations that are justified in a query are listed. The following construct term collects all title/author pairs for the previous query:

```
results {
  all result { var A, var T }
}
```

The constructs all and some n may be nested to form more complex results. Assuming the previous query, the following construct term collects all titles for each author:

```
results {
  all result {
    var A,
    all var T
  }
}
```

Positioning the nested all around the A yields "all authors for each title" as a result:

```
results {
  all result {
    all var A,
    var T
  }
}
```

3.2 *Queries*

A *query* is a connection (using ANDs and ORs) of query terms. Empty queries are possible. A query is always (implicitly or explicitly) associated with a *resource*. A resource may be the program itself, an external Xcerpt program or an (XML or other) document specified by a URI (uniform resource identifier).

Variables occurring in more than one query terms in an *and* connected query evaluate similar to an equijoin in the language SQL (for relational databases).

Example: The following query selects all such authors that published a paper in the Extreme Markup Languages proceedings of both 2003 and 2004. Again, it is assumed that the articles are all contained in a proceedings03 resp. proceedings04 element:

```
and {
    in { resource { "file:proceedings03.xml" },
        desc author {{
            fname { var First }, surname { var Last }
        }}
    },
    in { resource { "file:proceedings04.xml" },
        desc author {{
            fname { var First }, surname { var Last }
        }}
    }
}
```

If a query does not explicitly have an associated resource, the resource specification is implicit and inherited from the parent. If none of the parents have a resource specification, or the query does not have a parent, the queried resource is the program itself (i.e. the heads of the rules and possibly data terms contained in the program - see "Rule Chaining" below).

Note that it is possible to use curly and square braces in and and or connections to specify that the evaluation order is of importance or not. This may serve as an indication to the evaluation engine whether certain optimizations are applicable or not.

3.3 *Construct-Query Rules and Goals*

An Xcerpt program consists of zero or more *construct-query rules*, one or more *goals* and zero or more data terms. Both rules and goals have the form

```
CONSTRUCT
    construct term
FROM
    query part
END
```

where a construct term is constructed depending on the evaluation of a query part.

A rule can be seen as a "view" (in the sense of databases) specifying how documents shaped in the form of the construct term can be obtained by evaluating the query part against a Web resource (e.g. an XML document or a database).

In addition to the form above, goals are always (explicitly or implicitly) associated with an output resource. This resource specifies where to "write" the resulting data terms. If not explicitly specified, the output resource defaults to *stdout*, writing all output to the console.

§ 4 Further Elements of Xcerpt

4.1 *Functions and Aggregations*

Many Web applications require to *compute* information in addition to arranging it in new structures, like adding the VAT (value added tax) to a price, calculate totals, concatenate strings, compute the average of all results, etc. Likewise, it is desirable to express conditions on such computed data in a cohesive manner, like "the price should be lower than 50". Such computations are in general expressed using *functions* and *aggregations*.

As both, normal functions and aggregations, are *constructing* new data, they are only allowed in construction parts of rules (i.e. the rule or goal heads).

Example: The following Xcerpt rule counts the number of articles containing the keyword "XML" in the proceedings04.xml file by using the aggregation function count in the construct part:

```
CONSTRUCT
  nr_of_articles {
    count ( all var Paper )
  }
FROM
  in {
    resource{ "proceedings04.xml", "xml" },
    var Paper -> paper {{
      keywords {{
        keyword { "XML" }
      }}
    }}
  }
END
```

4.2 *Non-Pattern Conditions*

One of Xcerpt's main design principles is pattern based querying and many query conditions can be expressed in Xcerpt query terms by using partial patterns and variable restrictions. However, such conditions are limited to structural properties, like the parent-child or sibling relationships, and there are only few possibilities to work with the content (i.e. the semantics) rather than the structure of the data in such patterns.

Often such *semantic* conditions are desirable, e.g. to select all books that are published later than a certain date, etc. For such expressions, Xcerpt uses a so-called *condition box* which may be attached to query specifications (and parts of it) in rules by using the keyword where.

Example: Assume the XML database of an online book store which might contain books with title, author(s) and price. The following Xcerpt program selects all books that are published in or later than 2003:

```
CONSTRUCT
  books {
    all var Book
  }
FROM
  in {
    resource{ "bib.xml", "xml" },
    bib {{
      var Book -> book {{
        year { var Year }
      }}
    }}
  } where {
    var Year >= 2003
  }
END
```

Conditions in this condition box only apply to the variables contained in the associated query specification, and there to each alternative binding separately, like the WHERE-part in SQL queries. As a consequence, there is no possibility to collect all alternatives using constructs like all or some.

4.3 *Optional Subterms*

One of the main characteristics of Web sites and semistructured databases is that they are heterogeneous in structure. Nonetheless, data items of the same kind (e.g. address book entries or student information) are often very similar and differ only in that for certain entries some information is missing. For instance in an address book, the general structure of entries is very similar, but some entries might have no email address and others might have no telephone number, either because the respective persons do not have an email address/telephone number, or because the information is simply unknown.

Querying such entries with partial query patterns is very inconvenient when information is missing in some cases but is to be retrieved in the cases where it is available. For this reason, Xcerpt provides the possibility to specify subterms of a query term as *optional*, in which case the pattern matches even if the optional subterm does not have a match in the database.

Xcerpt uses the keyword optional to mark a subterm as optional. Occurrences of optional may also be nested. If a query term has a subterm that is marked as optional, this subterm usually also contains variables. Obviously, the optional construct yields cases where there are no variable bindings for these variables. For this reason, such variables must be marked as optional in the construction part of a rule as well.

Example: The following Xcerpt program creates an HTML table listing all authors in the Extreme 2004 proceedings, with email address if available, or the string "not given" otherwise, sorted by last and first name:

```
CONSTRUCT
  table [
    tr [ td [ "Author Name" ], td [ "Email (optional)" ] ],
    all tr [
      td [ var First, var Last ],
      td [ optional var Email with default "not given" ]
    ] order by [Last,First]
  ]
FROM
  in {
    resource{ "extreme2004.xml", "xml" },
    desc author {{
      fname { var First },
      surname { var Last },
      optional email { var Email }
    }}
  }
END
```

4.4 *Position Specification*

In some applications it is desirable to query subterms only at a certain position while still being able to use partial query patterns, or to query the position of subterms in the database. For example, a query to this Extreme 2004 article could select the first and second element of all sections:

```
in {
  resource{ "extreme2004.xml", "xml" },
  desc section {{
    position 1 var First,
    position 2 var Second
  }}
}
```

Obviously, a position specification is only reasonable in partial query patterns which are matched against ordered terms (otherwise there is no position associated with the subterm in the database).

The position specification may be of the following kinds:

- *positive number.* A positive number specifies the position of the matched subterm below

its parent term, where 1 is the position of the first subterm

- *negative number.* A negative number specifies the position relative to the last subterm of the parent, where -1 is the position of the last subterm.
- *variable.* A variable matches with a subterm with any position and binds the variable to the position of this subterm as a positive integer number.

Under the "pattern" paradigm, it is straightforward to also permit variables at position specifications. This allows to build very complex queries. The following Xcerpt rule queries the cells in an HTML table and builds the totals for each row and column. Note how the bindings of the variables Row and Column are used to group the results with the help of nested all constructs.

```
CONSTRUCT
  table [
    all tr [
      attributes { row { var Row } },
      all td [ var Value ],
      td [ sum (all var Value) ]
    ],
    tr [
      all td [
        attributes { col { var Col } },
        sum (all var Value)
      ],
      td [ sum (all var Value) ]
    ]
  ]
FROM
  in {
    resource { "sum-table.html", "html" },
    html {{
      desc table {{
        position var Row tr {{
          position var Col td {{ var Value }}
        }}
      }}
    }}
  }
END
```

4.5 *Negation*

Sometimes it is useful to know what is *not* in the database. For instance, a query might ask whether there is *no* train between Munich and Paris at a certain time. Also, it might be interesting which students did *not* submit their homework or did *not* fill out the registration form completely. To achieve this, it is possible to negate Xcerpt query patterns.

An important observation is that querying semistructured data requires two different notions of negation which are equivalent in expressiveness but used differently. The first kind of negation is a *subterm negation* and is indicated by the keyword without. The second kind of negation is a *query negation* which behaves like the negation in logic programming and is indicated by the keyword not.

Both negation constructs implement "negation as failure" (NaF). Since full NaF is infamous for its various, not always intuitive, and often enough diffiult to use declarative semantics, Xcerpt in its current version imposes a certain syntactical condition called *stratification* [ABW88], which in a simplified sense disallows recursions over negation. The semantics of NaF is (not are) a wide subject. Dissussing it in detail in the context of Xcerpt is out of the scope of this paper.

4.5.1 Subterm Negation: The without Construct

Subterm negation allows to express that a certain subterm should *not* be found at a certain position (the root node is excluded). It is thus a construct only used in query terms. A subterm is negated by prefixing it with the keyword without. The following Xcerpt rule uses without to find all unresolved citations in an Extreme 2004 article:

```
CONSTRUCT
  unresolved_citations {
    all var Citation
  }
FROM
  report {{
    desc var Citation -> cite {
      attributes { ref { var Ref } }
    },
    desc bibliography {{
      without entry {{
        attributes {{ id { var Ref } }}
      }}
    }}
  }}
END
```

Note that the subterm negation is only applicable to subterms (as it specifies that a term *must not have* a certain subterm) and may not be used at the root level. Furthermore, subterm negation is only reasonable in partial term specifications, and order does not have influence on the negated subterms (only on all positive subterms).

Such negation is quite common in semistructured data, as such data is often heterogeneous. For example, a query might ask for ``all students that did not submit their homework'' (i.e. all student elements that do not contain an element indicating that they submitted their homework). Note that in relational database systems, this negation is very similar to querying for NULL values.

Subterms negated by without may contain variables, but such an occurrence can never yield variable bindings, i.e. it is not possible to list all subterms that do *not* occur. Consequently, all variables that occur in the scope of a without have to appear elsewhere outside the scope of a negation construct.

Subterm negation is an *existential* negation: As long as there exists at least one term which does not contain the negated subterm (and matches with the remainder of the pattern), all other terms are irrelevant. There might be terms that contain the negated subterm, those simply do not match.

4.5.2 Query Negation: The not Construct

Query negation specifies that the negated query must not be fulfilled. Such negation is e.g. used to express that there is no train from Munich to Paris at a certain time, or that the student database does not contain a certain student.

Query negation is expressed as part of the query specification. A query is negated by prefixing it with the keyword not. The following query selects all such authors that have an article in the Extreme 2004 proceedings but not in the Extreme 2003 proceedings (compare with Section 3.2):

```
and {
  in { resource { "file:proceedings04.xml" },
    desc author {{
      fname { var First }, surname { var Last }
    }}
  },
  not in { resource { "file:proceedings03.xml" },
    desc author {{
      fname { var First }, surname { var Last }
    }}
  }
}
```

Query negation is a *universal* negation. If the query is negated, there must not exist a term with which it matches, i.e. all terms are required to not match with the pattern.

4.6 *Regular Expressions*

Query languages for Web data need not only be able to query based on the *database structure*, it is

---

often also desirable to query parts of the plain text contained in the database. This is of particular importance when the document contains mainly text content (like this paper), but is also useful for many database applications.

A powerful construct to support such text querying are *regular expressions*. Xcerpt uses regular expressions as defined in POSIX [POSIX RE], but with certain Xcerpt specific extensions. Like in Perl, regular expressions are enclosed in slashes (i.e. "/").

Regular expression matching integrates well with the pattern matching paradigm of Xcerpt, with one exception: so as to retrieve the data that matched a pattern, POSIX defines so-called subexpressions that are enclosed in single parentheses, i.e. (...). The matching substring is later referred to by the position of the subgroup (for instance, Perl uses the variables $1, $2, ..., $9 for this purpose). This way of retrieving matches is straightforward in imperative languages but awkward in a language that is mainly based on pattern matching and rules.

Therefore, Xcerpt extends POSIX regular expressions by *named subexpressions*, where the name is the variable to which the matching substring is bound. Subexpressions of the form (...) are written in Xcerpt as (var X ->...), if the matching substring is to be retrieved in the variable X.

Example: Assume that the "Extreme Markup Languages" organisers want to publish a list of authors with email addresses, if available, out of the collected proceedings. Due to the massive spam problems, it would be unwise to publish the email addresses as they are. Rather, a common practice is to create a "spamvertised form", where the "@" sign is replaced by something different, e.g. "<at>". The following Xcerpt rule uses a regular expression to separate the user from the domain part of email addresses and creates an author list with such spamvertised email addresses (ordered by last and first name):

```
CONSTRUCT
    table [
      tr [
        td{ "Name" },
        td{ "Email" }
      ],
      all [
        td{ var First, var Last },
        optional td{ var User, "<at>", var Domain }
      ] order by [Last,First]
    ]
FROM
    in {
      resource { "file:proceedings04.xml" },
      proceedings04 {{
        paper {{
          front {{
            author {{
              fname { var First }, surname { var Last },
              optional email {
                /^(var User ->[^@]+)@(var Domain ->[a-zA-Z0-9.-]+)/
              }
            }}
          }}
        }}
      }}
    }
END
```

§ 5 Rule Chaining and Recursion

One of the major design principles of Xcerpt is *rule chaining*, an aspect in which Xcerpt departs significantly from other rule-based query languages, like the XML query languages XML-QL [DFFLS98] and UnQL [BFS00], or the relational query language SQL, all of which do not support proper rule chaining (although querying VIEWS in SQL can be seen as a restricted form of rule chaining).

Conceptually, rule chaining means that a rule may interact with another rule by querying the "result" of the other rule. In this way, it is possible to reuse code, to write recursive queries (like a Web crawler recursively querying (for some data patterns) Web sites whose URI appear in

previoulsy queried Web sites), to structure complex "query programs" appropriately (so that they are easier to maintain), and even to build external "rule libraries". The following sections highlight important aspects that can be achieved by rule chaining.

5.1 *Code Reuse*

Many applications have code that is used in several places in a program. Repeating this code at every such occurrence is very inefficient and prone to errors. Therefore, programming languages have long been using concepts like *procedures*, *functions* or *methods*.

Xcerpt's equivalent of a procedure is a rule, which can be "called" by using rule chaining. Like a procedure in imperative programming language, a rule can serve to encapsulate auxiliary computations that are referred to at several positions in a program into a separate construct.

5.2 *Separation of Concerns*

When designing applications, a good principle is *separation of concerns*, e.g. separating program logic from presentation. Whereas this separation is impossible with a single XSL stylesheet (as all results are immediately written out and cannot be queried within the same program), and very inconvenient in XQuery (by using XQuery's function mechanism), Xcerpt's rule chaining makes separation of concerns straightforward: one rule is used to specify how to process and transform the data, while another rule focuses on presentation aspects (like a stylesheet).

Example: Conferences like Extreme Markup Languages usually publish their program as an HTML page. With personal digital assistants becoming more widespread, it is however also useful to publish the program in a format more suited for such mobile devices with small screens, e.g. the "wireless markup language" (WML). The following Xcerpt program uses an intermediate rule to create the program from the proceedings and a timetable (i.e. the "program layer"), and two different goals to format the result in HTML and WML (i.e. the "presentation layer"), both of which call the rule that creates the conference program via rule chaining:

```
GOAL
  out {
    resource { "file:program.html" ,  "html" },
    html [
      head [
        title [ "Program for Extreme Markup Languages 2004" ]
      ],
      body [
        h1 [ "Conference Program" ],
        table [
          tr [
            td [ "Time"    ],
            td [ "Title"   ],
            td [ "Authors" ]
          ],
          all tr [
            td [ var Time  ],
            td [ var Title ],
            td [ all var Author ]
          ] order by [Time]
        ]
      ]
    ]
  }
FROM
  program [[
    talk [[
      title [ var Title ],
      author [ var Author ],
      time  [ var Time ]
    ]]
  ]]
END

GOAL
  out {
    resource { "file:prices.wml" ,  "xml" },
    wml [
      all card [
```

```
              "Time: " , var Time,
              "Title: " , var Title ,
              "Authors: " , all var Authors
            ]
          ]
        }
      ]
FROM
    program [[
      talk [[
        title [ var Title ],
        author [ var Author ],
        time  [ var Time ]
      ]]
    ]]
END


CONSTRUCT
    program [
      all talk [
        title [ var Title ],
        all author [ var First, var Last ],
        time [ var Time ]
      ]
    ]
FROM
    and {
      in {
        resource { "file:proceedings04.xml" },
        proceedings04 [[
          paper [[
            front [[
              title { var Title },
              author {{
                fname { var First }, surname { var Last }
              }}
            ]]
          ]]
        ]]
      },
      in {
        resource { "file:timetable.xml" },
        timetable [[
          entry [[
            time [ var Time ],
            title [ var Title ]
          ]]
        ]]
      }
    }
END
```

Note also that both query terms in the and-connected query part share the variable Title to join the information from the proceedings database with the information of the timetable.

5.3 *Mediation*

Complex applications often need to query several databases in order to retrieve information, all of which might have different structure. For example, the booking system of a travel agency might need to query the flight schedules of different airlines. Rule chaining can be used as a mediator by using Xcerpt rules that transform data from different resources into a common structure, which can then be queried by other rules in the program.

Example: The following rules create a unified author list for authors that published an article either in the 2003 or in the 2004 proceedings. The second and the third rule create from the proceedings04.xml and proceedings03.xml files author terms containing first and last name of an author (note that they do not contain the collection construct all and thus yield several alternative data terms as answers, one for each author). The first rule (goal) collects all such author terms within an authors term.

```
GOAL
    out {
      resource { "file:authors.xml" },
      authors [
        all var Author
      ]
    }
FROM
    var Author -> author {{ }}
```

```
END

CONSTRUCT
  author { var First, var Last }
FROM
  in {
    resource { "file:proceedings04.xml" },
    proceedings04 [[
      paper [[
        front [[
          author {{
            fname { var First }, surname { var Last }
          }}
        ]]
      ]]
    ]]
  }
END
CONSTRUCT
  author { var First, var Last }
FROM
  in {
    resource { "file:proceedings03.xml" },
    proceedings03 [[
      paper [[
        front [[
          author {{
            fname { var First }, surname { var Last }
          }}
        ]]
      ]]
    ]]
  }
END
```

5.4 *Recursion*

Besides querying other rules, Xcerpt rules can also call themselves recursively. In contrast to the inherent structural recursion in XSLT, which is limited to the tree structure of the input tree 1, Xcerpt allows all kinds of recursions, not only structural recursion over XML documents.

Applications of recursion in Web query languages are manifold:

- structural recursion over the input tree (like in XSLT) is necessary to perform transformations that preserve the overall document structure and change only certain things in arbitrary documents (e.g. replacing all <em> elements in an HTML document by <strong> elements).
- recursion over the conceptual structure of the input data (e.g. over a sequence of elements) is used to iteratively compute data (e.g. create a hierarchical representation from flat structures with references).
- recursion over references to external resources (hyperlinks) is desirable in applications like a Web crawler that recursively visit Web pages.

Example: The following rule implements a Web crawler that recursively visits Web sites, retrieves all links (i.e. a elements) from it and in turn queries each of these by recursively calling itself and with each call passes a new binding for the variable Resource.

Note that this Web Crawler provides only minimal functionality. For instance, it does not terminate in case of cyclic hyperlinks.

```
CONSTRUCT
  crawler [
    linklist {
      all link { var Link1 },
      all link { var Link2 }
    },
    var Resource
  ]
FROM
  and {
    in {
      resource { var Resource, "html" },
      desc a {{
```

```
            attributes {{ href { var Link1 } }}
        }}
    },
    crawler [
        linklist {{ var Link2 }},
        var Link1
    ]
  }
END
```

5.5 *Backward and Forward Chaining*

Rule languages in general allow to evaluate rules in two directions, so-called "forward chaining" and "backward chaining".

- Forward chaining is *data driven*. Rules are evaluated iteratively against the current set of data terms until saturation is achieved. Forward chaining is useful for instance for materialising views and for view maintenance, and is widely used in deductive databases.
- Backward chaining is *goal driven*. Beginning with the query part of the goal, program rules are selected if they are relevant for "proving" a query term. The query term in question is then replaced by the query part of the selected rule. Backward chaining is useful when the expected result is small in comparison with the number of possible results of the program. It is mainly used in logic programming languages like Prolog.

In a Web environment, both forward and backward chaining are desirable. A forward chaining approach is e.g. useful when creating a static Web site (consisting of one or more Web pages) from an input document and an Xcerpt program used as a "stylesheet", i.e. "materialising the HTML view" on the input data. On the other hand, backward chaining is necessary when querying large collections of documents, in particular the Web itself, where it is impossible to begin with the complete database, which might be the whole Web.

The current Xcerpt prototype implements backward chaining only. Different aspects of backward chaining have been investigated in [BSS04]. A forward chaining algorithm is specified, but currently not implemented.

§ 6 Aspects of the Runtime System

One of the objectives in the developement of Xcerpt is to develop its declarative and procedural semantics in parallel so as to avoid discrepencies between them. The declarative semantics is given in form of a Tarski style model theory and first results have been published in [BS03a]. The operational semantics are defined in terms of a non-standard unification algorithm and a constraint solver, the concepts of which are both briefly introduced below. A more detailed presentation can be found in [BS02] and [BSS04].

Parts of the language have been implemented in a prototypical runtime system for evaluating Xcerpt programs. This prototype is available for free use under **http://www.xcerpt.org**.

6.1 *Simulation Unification*

Matching query with data or construct terms is based on a non-standard, asymmetric unification algorithm called *simulation unification* first described in [BS02]. This unification algorithm departs from Robinson's unification [Rob65] (used e.g. in Prolog) in that it is capable of handling Xcerpt's partial patterns and is not restricted to fixed arity and order. Simulation unification is based on a graph relation called *Simulation*:

Definition: Let $G_1 = (V_1, E_1, r_1)$ and $G_2 = (V_2, E_2, r_2)$ be two rooted graphs and let $\sim \subseteq V_1 \times V_2$ be an equivalence relation on labels. A relation $\leq \subseteq V_1 \times V_2$ is a rooted simulation with respect to $\sim$, iff:

1. $r_1 \leq r_2$
2. If $v_1 \leq v_2$, then $v_1 \sim v_2$.
3. If $v_1 \leq v_2$ and $(v_1, v'_1) \in E_1$, then there exists $v'_2 \in V_2$ such that $v'_1 \leq v'_2$ and $(v_2, v'_2) \in E_2$
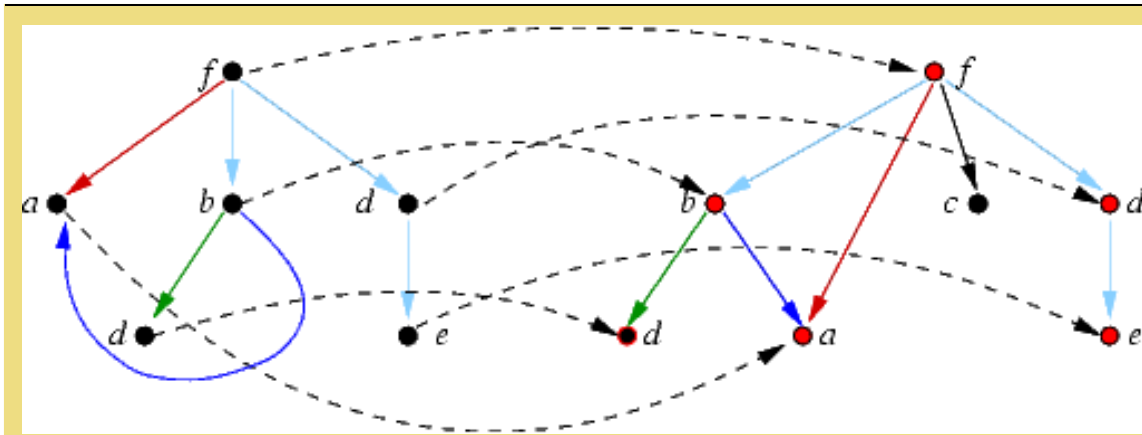

Figure 2

A rooted simulation of the rooted graph on the right hand side into the rooted graph on the left hand side. The simulation relation is indicated by dashed arrows. Note that the graph on the right contains a node labelled "c" which has not corresponding node in the graph on the left. This property can be used to match a query term containing partial term specifications with a data term.

A simulation unifier for two terms t_1 and t_2 is a *set of substitutions* that, when applied to t_1 and t_2, yields two sets of ground terms T_1 and T_2 with the property that for every term t'_1 in T_1, the graph induced by t'_1 simulates into the graph induced by a t'_2 in T_2. Simplified, a simulation unifier thus defines a set of alternative variable bindings such that simulation holds between the ground instances.

Simulation Unification is an algorithm that computes a simulation unifier for two terms. The simulation unification algorithm is given as a set of simplification rules that operate on a constraint store initialised with a single inequation $t_1 \leq t_2$. The algorithm is described in more detail in [BS02] and [BSS04].

6.2 *Constraint Reasoning*

The evaluation of Xcerpt programs is described in terms of a constraint solver which applies simplification rules to a constraint store consisting of conjunctions and disjunctions of constraints. The constraint store is initialised with a disjunction of all goals in a program ("folded queries"). The program is then evaluated by successively applying unfolding rules that evaluate query parts against other rules (and add the query part of those rules as folded queries) or external resources. This process continues until sufficiently many solutions are found to output a result.

In the Web context, a constraint reasoning approach has several advantages. Most importantly, the non-deterministic nature of the algorithms allow the evaluation engine to apply optimisations that e.g. take into account the cost of retrieving external resources. For example, the evaluation may be based on advanced search algorithms like A* instead of depth first search, which is used in the SLD resolution underlying the language Prolog. Furthermore, constraint reasoning allows to evaluate the program in a "lazy" fashion: variables need only to be bound to concrete values at the very end of evaluation. Up to this point, it is sufficient to work with inequations that ensure the simulation relation between different bindings and may be used to detect inconsistencies.

Several different approaches for simplification rules are again discussed in more detail on [BSS04].

6.3 *Declarative Semantics*

Besides the operational semantics described above, Xcerpt has also a declarative semantics that provides a very precise specification of a program's meaning against which implementations can be verified. In this declarative semantics, an Xcerpt program is seen as a logic formula (in the sense of classical logic, but departing from it in several important aspects) for which Tarski-style models can be given. An answer to an Xcerpt program is correct in this model theory, if it is a necessary consequence of the program (i.e. "entailed" by the program).

First results of the declarative semantics have been published in [BS03a].

§ 7 visXcerpt: A Visual Language as Rendering of a Textual Language

The Web context demands for query technology that is easy to use even by non-programmers, since there are always queries not forseen by developers. A visual query language serves this purpose since it avoids many common errors by abstracting from the textual syntax, and would likely be well accepted among many Web users.

For visual query languages it is considered to be important to have a strong visual relationship between queries and queried data or query results. A natural approach is to provide some sort of example of a valid result as query as first presented in QBE. Xcerpt query patterns with positional variables can be seen as samples of valid source data items, where some parts are left out and others represented by variables. Construction patterns can be seen as samples or templates of result data items.

As part of the Xcerpt project, the visual language visXcerpt [BBSW03] has been developed. The syntax and semantics of Xcerpt as a whole is well suited as foundation for a visual language. As a consequence, *visXcerpt* can be conceived as a mere *rendering* of Xcerpt programs instead of a fully novel language. This rendering might be seen as an advanced (because of the dynamic features) layout. Figure 3 illustrates this rendering on a small Xcerpt program.

Figure 3

visXcerpt rule that selects all title/author pairs from the proceedings04.xml file
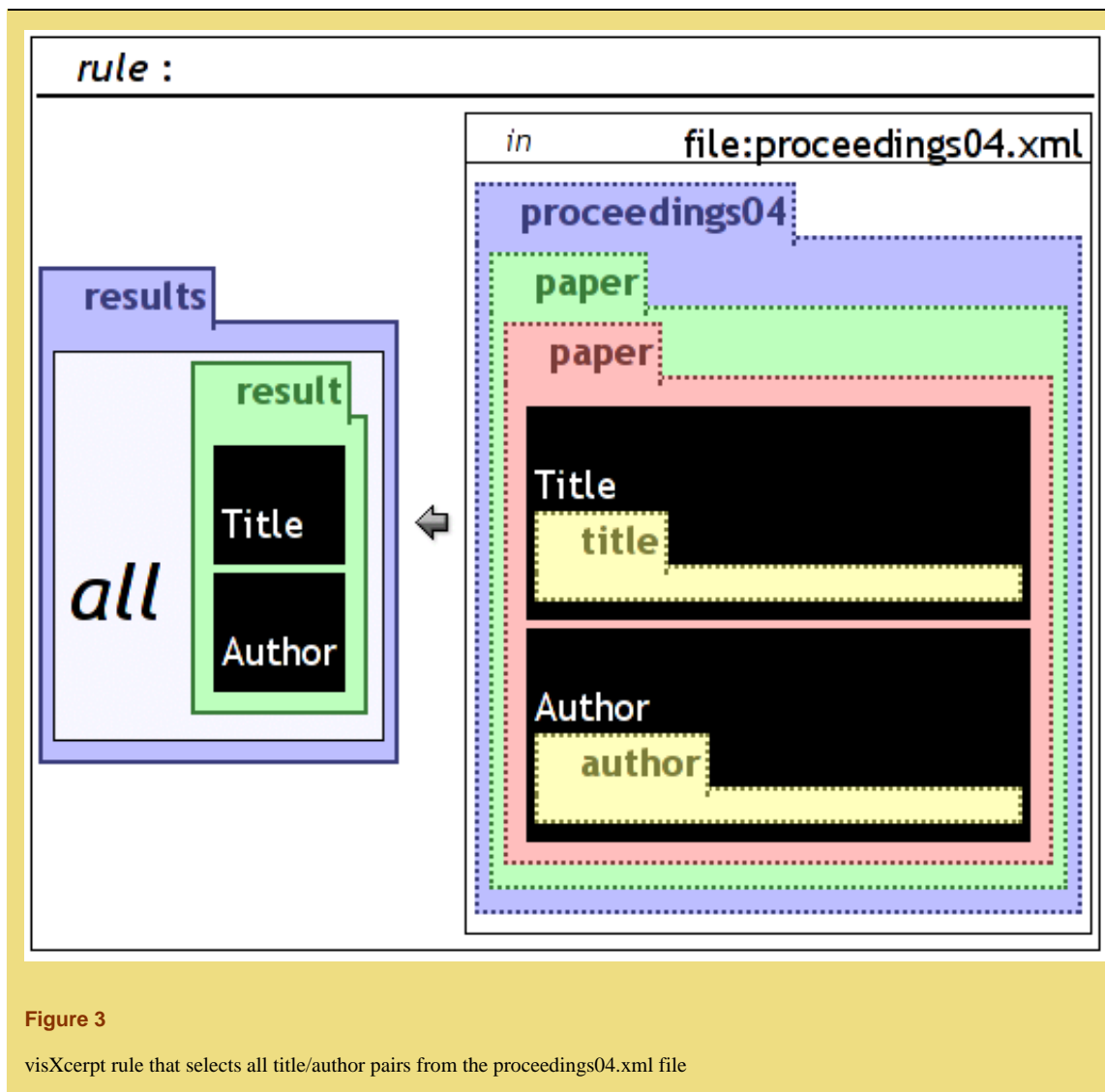
§ 8 Perspectives

Research on Xcerpt currently focusses on two areas: refinement and definition of language constructs that are necessary in practical applications (much of this work is presented in this article) and final specification of language semantics (an overview of which are given in Section 6 of this article). Furthermore, several current research projects investigate topics related to Xcerpt:

8.1 *Type System and Static Type Checking*

Modern programming languages (like the functional languages Haskell or SML) feature a static type system with type inference. Type systems are advantageous for both, the programmer (as they help to avoid errors) and the evaluation engine (as they enable type based optimisations like using machine registers or applying optimised algorithms). Type inference adds the advantages of untyped languages: the compiler or interpreter can in almost all cases determine the type of an expression even if it is not explicitly given.

Static type systems with type inference are currently in development for Xcerpt (in Munich and in

Linköping [WD03]). Both will be able to use schema information of documents if available and allow to explicitly specify types for Xcerpt rules. Besides the general advantages described above, a type system for Xcerpt can be used to compute the type (i.e. schema) of the resulting data terms.

8.2 *TXcerpt: Temporal Types*

Querying and reasoning on the (Semantic) Web often involves notions of time, like the last modification or the validity period of a document, or also explicit time referred to in a personal time planner or cinema timetable. Exemplary applications that need to reason with time are e.g. a personal digital assistant that automatically arranges a meeting between three business people living in London, Athens and Tokyo or informs about available evening entertainments in the city where a person arrives in the afternoon.

However, notions of time are much more difficult to reason with than normal numbers. On the Web as a globally available resource, reasoning languages have to take into account different calendar systems, different regional holidays, different ways to denote time, etc. TXcerpt aims at defining temporal types that are capable of handling such different notions in a straightforward manner. Times specified in different calendar systems becomes comparable without difficult transformation rules and reasoning with time thus becomes feasible. While this work is still at the beginning, first results in this direction have already been obtained [BS03b] [BLOS03].

8.3 *XChange: Updates and Events based on Xcerpt*

Limitations of current languages for evolution and reactivity on the Web make it difficult to realise some complex applications that emerge in the Web context. The objective of the XChange project is to investigate means to enhance the language Xcerpt with advanced, Web-specific capabilities, such as propagation of updates on the Web (*change*) and event-based communications between Web sites (*exchange*). This research will be done in a logical framework so as to yield a powerful logic programming language for evolution and reaction on the Web.

Notes

1. Note that XSLT does in principle support arbitrary recursions by means of the call-templates element. However, using call-templates is arguably very inconvenient.

Acknowledgements

Bibliography

[ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. Data on the Web. From Relations to Semistructured Data and XML. Morgan Kaufmann, 2000.

[ABW88] Krzysztof Apt, Howard Blair and Adrian Walker. Towards a Theory of Deductive Knowledge. In: Jack Minker (editor). *Foundations of Deductive Databases and Logic Programming*, Morgan-Kaufmann, 1988.

[BBSW03] Sacha Berger, François Bry, Sebastian Schaffert and Christoph Wieser. Xcerpt and visXcerpt: From Pattern-Based to Visual Querying of XML and Semistructured Data. *Int. Conference on Very Large Databases (VLDB)*, Berlin, Germany, 2003.

[BFS00] 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.

[BLOS03] François Bry, Bernhard Lorenz, Hans-Jürgen Ohlbach and Stephanie Spranger. On Reasoning on Time and Location on the Web. *Workshop on Principles and Practice of Semantic Web Reasoning*, Mumbai, India, 2003.

[BS02] François Bry and Sebastian Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proceedings of the International Conference on Logic Programming (ICLP)*, Copenhagen, Denmark, July 2002. Springer-Verlag, LNCS 2401.

[BS03a] François Bry and Sebastian Schaffert. An Entailment for Reasoning on the Web. *Proceedings of Rules and Rule Markup Languages for the Web (RuleML03)*, Sanibel Island (Florida), USA, 2003.

[BS03b] François Bry and Stephanie Spranger. Temporal Constructs for a Web Language. *Proceedings of the 4th Workshop on Interval Temporal Logics and Duration Calculi*, Vienna, Austria, 2003.

[BSS04] François Bry, Sebastian Schaffert, and Andreas Schroeder. A contribution to the Semantics of Xcerpt, a Web Query and Transformation Language. *Proceedings of 18. Workshop Logische Programmierung*, Potsdam, Germany, 2004.

[DFFLS98] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML. Technical report, W3C, 1998.

[Ho99] Ian Horrocks. FaCT and iFaCT. *Proceedings of the International Workshop on Description Logics*, Linköping, Sweden, 1999

[POSIX RE] IEEE / The Open Group. POSIX Regular Expressions. *The Open Group Base Specifications Issue 6*, 2001.

[Rob65] J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *ACM Journal* 12(1), January 1965.

[SemanticWeb] W3C Semantic Web Activity, **http://www.w3.org/2001/sw/**

[WD03] Artur Wilk and W#odzimierz Drabent. On Types for XML Query Language Xcerpt. *Workshop on Principles and Practice of Semantic Web Reasoning*, Mumbai, India, 2003.

[XQuery] XQuery: A Query Language for XML. W3C Working Draft, March 2004. **http://www.w3.org/TR/xquery/**

[XSLT] Extensible Stylesheet Language (XSL). W3C Recommendation, 2000. **http://www.w3.org/Style/XSL/**

The Authors

Sebastian Schaffert
University of Munich, Institute for Informatics
Sebastian.Schaffert@ifi.lmu.de
http://www.wastl.net

Sebastian Schaffert, born 1976, holds a Computer Science degree from the University of Munich (2001). He is currently employed as a teaching and research assistant at the Institute for Computer Science of the University of Munich, where he is working on his PhD thesis. His research interests are in XML and semistructured data, as well as functional and logic programming. He has several publications on the language Xcerpt and related topics. Besides this, Sebastian is interested in Linux and Open Source software, and likes sailing, biking and mountaineering.

François Bry
University of Munich, Institute for Informatics
Francois.Bry@ifi.lmu.de
http://www.pms.ifi.lmu.de/mitarbeiter/bry/

François Bry, born 1956, is currently investigating methods and applications emphasizing XML, semistructured data, document modeling, and query answering. Formerly, he worked on deductive databases, automated theorem proving, and logic programming. Since 1994, he is a full professor at the Institute for Computer Science of the University of Munich, Germany. From 1984 through 1993, he was with the European Computer-Industry Research Centre (ECRC), Munich. Before 1983, he worked in a few companies in Paris, France, among others on an early word processor. He received in 1981 a PhD from the University of Paris. He has been visiting at several university and research centers, including ICOT in Tokyo. He gave invited talks at many major research centers, universities, and scientific conferences. He contributed to scientific conferences as an author, program committee member or chairman. He has been or is involved in research projects founded by the European Community (ESPRIT, FP6) and in doctoral schools founded by the German Foundation for Research (DFG). He likes biking, hiking, and traveling.