Data Retrieval and Evolution on the (Semantic) Web: A Deductive Approach

François Bry, Tim Furche, Paula-Lavinia Pătrânjan, and Sebastian Schaffert

Institut für Informatik, Ludwig-Maximilians-Universität München
Oettingenstr. 67, D-80538 München, Germany

Abstract. To make use of data represented on the Semantic Web, it is necessary to provide languages for Web data retrieval and evolution. This article introduces into the (conventional and Semantic) Web query language *Xcerpt* and the event and update language *XChange*, and shows how their deductive capabilities make them well suited for querying, changing and reasoning with data on both the conventional and the Semantic Web. To this aim, small application scenarios are introduced.

1 Introduction

The *Semantic Web* is an endeavour aiming at enriching the existing Web with meta-data and data and meta-data processing to allow computer systems to actually *reason* with the data instead of merely *rendering* it. To this aim, it is necessary to be able to *query* and *update* data *and* meta-data. Existing Semantic Web query languages (like DQL[1] or TRIPLE[2]) are special purpose, i.e. they are designed for querying and reasoning with special representations like OWL[3] or RDF[4], but are not capable of processing generic Web data, and are furthermore restricted to a specific reasoning algorithm like a certain description logic (e.g. $\mathcal{SHIQ}$). In contrast, the language *Xcerpt* presented in this article (and more extensively in e.g. [1]) is a general purpose language that can query any kind of XML data, i.e. "conventional" Web as well as Semantic Web data, and at the same time provides advanced reasoning capabilities. It could thus serve to implement a wide range of different reasoning formalisms.

Likewise, the maintenance and evolution of data on the (Semantic) Web is necessary: the Web is a "living organism" whose dynamic character requires languages for specifying its evolution. This requirement regards not only updating data from Web resources, but also the propagation of changes on the Web. These issues have not received much attention so far, existing update languages (like XML-RL Update Language [2]) and reactive languages [3] developed for XML data offer the possibility to execute just simple update operations while important features needed for propagation of updates on the Web are still missing. The

[1] DAML Query Language, http://www.daml.org/dql.
[2] TRIPLE Language, http://triple.semanticweb.org.
[3] Web Ontology Language, http://www.w3.org/TR/owl-ref/
[4] Resource Description Framework, http://www.w3.org/TR/rdf-primer/

language *XChange* also presented in this article builds upon the query language Xcerpt and provides advanced, Web-specific capabilities, such as propagation of changes on the Web (*change*) and event-based communication between Web sites (*exchange*), as needed for agent communication and Web services.

This article is structured as follows: Section 2 summarises the design principles underlying the languages Xcerpt and XChange. Section 3 gives a brief introduction into the Web query language Xcerpt, and illustrates the use of recursion as a first step towards Semantic Web querying. Section 4 subsequently introduces the event language XChange that builds upon Xcerpt. Section 5 introduces a simple Semantic Web scenario and uses it to illustrate how Xcerpt and XChange can be used for querying and evolution on the Semantic Web. Finally, Section 6 concludes with a summary and perspectives for future research.

2 Design Principles of Xcerpt and XChange

2.1 Principles of Xcerpt

Pattern-Based Queries. Most query languages for the Web, like XQuery or XSLT, use a path-based or *navigational* selection of data items, i.e. a selection is specified in terms of path expressions (usually expressed in the language XPath) consisting of a sequence of location steps that specify how to reach nodes in the data tree in a stepwise manner. In contrast, Xcerpt uses a *positional* or *pattern-based* selection of data items. A query pattern is like a *form* that gives an *example* of the data that is to be selected, like the forms of the language QBE or query atoms in logic programming. As in logic programming, a query pattern can furthermore be augmented by zero or more variables, which serve to retrieve data items from the queried data.

Incomplete Patterns. As data on the Web (e.g. XML) often differs much in structure and does not necessarily conform to an (accessible) schema, query patterns in a Web query language like Xcerpt need to be much more flexible than in logic programming or relational databases. Therefore, Xcerpt allows to specify incompleteness in *breadth* (i.e. within the same parent in the data tree) as well as in *depth* (i.e. on paths in the data tree), and it is possible to consider *ordered* and *unordered* content.

Rules. Xcerpt programs consist of deduction rules (*if* ... *then* ... rules) that may interact via (possibly recursive) rule chaining. Rules are advantageous as they are easy to comprehend even for novice users and can serve to structure a program into logical components. They are also well-suited for *deduction*, i.e. they allow to give meaning to data very much like rules in logic programming.

Backward Chaining. Rule-based query languages for traditional database systems (like *Datalog*) mainly use *forward chaining*, where rules are applied to the current database until saturation is achieved (it is thus *data driven*). On the Web, forward chaining is not always possible, for the database is the whole Web,

which is not feasible as an initial point for forward chaining. As a consequence, a *backward chaining* approach is sought for. Backward chaining is *goal driven*, i.e. only such resources are retrieved that are necessary to answer the query.

Separation of Querying and Construction. Most XML query languages (e.g. XQuery and XSLT, but also pattern-based approaches such as XML-QL) mix querying and construction by embedding data selection in construction patterns and by using subqueries inside construction patterns. In this way, however, the structure of the queried data is no longer apparent from the query. Therefore, a strict separation of querying and construction is favourable.

Reasoning Capabilities. A query language for the Web should be capable of querying both, XML data (on the standard Web) and meta-data (on the Semantic Web). Most currently available query languages are specific to a certain task, i.e. they are either capable of querying XML data, or capable of querying and reasoning with meta-data in a specific formalism. However, meta-data might be given in different formalisms (like OWL Light/DL/Full or RDF), wherefore it is desirable that a query language is generic enough to work with any kind of meta-data, and provides generic reasoning capabilities (e.g. like Prolog) that allow to implement a wide range of different formalisms.

2.2 Principles of XChange

Communication Between Web Sites. XChange uses *events* to communicate between Web Sites. An event is an XML document with a root element with label `event` and the four parameters (represented as child elements as they may contain complex content) `raising-time` (i.e. the time of the raising machine when the event is raised), `reception-time` (i.e. the time of the receiving machine when the event is received), `sender` (i.e. the URI of the site where the event has been raised), and `recipient` (i.e. the URI of the site where the event has been received). An `event` is an envelope for arbitrary XML content, and multiple `events` can be nested (e.g. to create trace histories).

Peer-to-peer communication. XChange events are directly communicated between Web sites without a centralised processing or management of events. All parties have the ability to initiate a communication. Since communication on the Web might be unreliable, synchronisation is supported by XChange.

No broadcasting. The approach taken in XChange excludes broadcasting of events on the Web, as sending events to *all* sites is not adequate for the framework which XChange has been designed for (i.e. the Web). Hence, before an event is sent, its recipient Web sites are determined.

Transactions as Updating Units. Since it is sometimes necessary to execute *complex updates* in an *all-or-nothing manner* (e.g. when booking a trip on the Web, a hotel reservation without a flight reservation is useless), the concept of transactions is supported by the language XChange. More precisely, XChange

transactions are composed of events posted on the Web and updates to be performed on one or more Web sites.

Complex applications specifying evolution of data and meta-data on the (Semantic) Web require a number of features that cannot always be specified by simple programs. In XChange transactions can also be used as *means for structuring* complex XChange programs.

Rule-Based Language. The language XChange aims at establishing reactivity, expressed by *reaction rules*, as communication paradigm on the Web. Reaction rules (also called *Event-Condition-Action* rules or active rules) are rules of the form `on` *`event`* `if` *`condition`* `do` *`action`*. At every occurrence of the event, the rule is triggered and the corresponding action is executed if the specified condition is satisfied. The components of an XChange Event-Condition-Action rule are:

- *Event* is an Xcerpt query against events received by the Web sites,
- *Condition* is an Xcerpt query against (local or remote) Web resources, and
- *Action* might be raising events and/or executing updates. These actions may be compound and considered as transactions. XChange considers *transactions* instead of isolated actions as active rules heads.

Pattern-Oriented Update Specifications. A metaphor for XChange (and at the same time one of the novelties of the update language) is to see XChange update specifications (i.e. Xcerpt queries against data terms, augmented with update operations) as forms, answers as form fillings yielding the data terms after update execution.

3 Xcerpt: Querying the Web

An Xcerpt program consists of at least one *goal* and some (possibly zero) *rules*. Rules and goals contain query and construction patterns, called *terms*. Terms represent tree-like (or graph-like) structures. The children of a node may either be *ordered*, i.e. the order of occurrence is relevant (e.g. in an XML document representing a book), or *unordered*, i.e. the order of occurrence is irrelevant and may be chosen by the storage system (as is common in database systems). In the term syntax, an *ordered term specification* is denoted by square brackets `[ ]`, an *unordered term specification* by curly braces `{ }`.

Likewise, terms may use *partial term specifications* for representing incomplete query patterns and *total term specifications* for representing complete query patterns (or data items). A term t using a partial term specification for its subterms matches with all such terms that (1) contain matching subterms for all subterms of t and that (2) might contain further subterms without corresponding subterms in t. Partial term specification is denoted by *double* square brackets `[[ ]]` or curly braces `{{ }}`. In contrast, a term t using a total term specification does not match with terms that contain additional subterms without corresponding subterms in t. Total term specification is expressed using *single* square brackets `[ ]` or curly braces `{ }`.

Data Terms represent XML documents and the data items of a semistructured database, and may thus only contain total term specifications (i.e. single square brackets or curly braces). They are similar to *ground* functional programming expressions and logical atoms. A *database* is a (multi-)set of data terms (e.g. the Web). A non-XML syntax has been chosen for Xcerpt to improve readability, but there is a one-to-one correspondence between an XML document and a data term.

Example 1. The following two data terms represent a train timetable (from `http://railways.com`) and a hotel reservation offer (from `http://hotels.net`).

At site `http://railways.com`:

```
travel {
  last-changes-on { "2004-04-30" },
  currency { "EUR" },
  train {
    departure {
      station { "Munich" },
      date { "2004-05-03" },
      time { "15:25" }
    },
    arrival {
      station { "Vienna" },
      date { "2004-05-03" },
      time { "19:50" }
    },
    price { "75" }
  },
  train {
    departure {
      station { "Munich" },
      date { "2004-05-03" },
      time { "13:20" }
    },
    arrival {
      station { "Salzburg" },
      date { "2004-05-03" },
      time { "14:50" }
    },
    price { "25" }
  },
  train {
    departure {
      station { "Salzburg" },
      date { "2004-05-03" },
      time { "15:20" }
    },
    arrival {
      station { "Vienna" },
      date { "2004-05-03" },
      time { "18:10" }
    }
  }...
}
```

At site `http://hotels.net`:

```
voyage {
  currency { "EUR" },
  hotels {
    city { "Vienna" },
    country { "Austria" },
    hotel {
      name { "Comfort Blautal" },
      category { "3 stars" },
      price-per-room { "55" },
      phone { "+43 1 88 8219 213" },
      no-pets {}
    },
    hotel {
      name { "InterCity" },
      category { "3 stars" },
      price-per-room { "57" },
      phone { "+43 1 82 8156 135" }
    },
    hotel {
      name { "Opera" },
      category { "4 stars" },
      price-per-room { "106" },
      phone { "+43 1 77 8123 414" }
    },
  ...
  },
...
}
```

Query Terms are (possibly incomplete) patterns matched against Web resources represented by data terms. They are similar to the latter, but may contain *partial* as well as *total* term specifications, are augmented by *variables* for selecting data items, possibly with *variable restrictions* using the $\rightsquigarrow$ construct (read *as*), which restricts the admissible bindings to those subterms that are matched by the restriction pattern, and may contain additional query constructs like *position matching* (keyword `position`), *subterm negation* (keyword `without`), *optional subterm specification* (keyword `optional`), and *descendant* (keyword `desc`).

Query terms are "matched" with data or construct terms by a non-standard unification method called *simulation unification* that is based on a relation called *simulation*. In contrast to Robinson's unification (as e.g. used in Prolog), simulation unification is capable of determining substitutions also for incomplete and unordered query terms. Since incompleteness usually allows many different alternative bindings for the variables, the result of simulation unification is not only a single substitution, but a (finite) *set of substitutions*, each of which yielding ground instances of the unified terms such that the one ground term matches with the other.

Construct Terms serve to reassemble variables (the bindings of which are specified in query terms) so as to construct new data terms. Again, they are similar to the latter, but augmented by *variables* (acting as place holders for data selected in a query) and the *grouping construct* `all` (which serves to collect all instances that result from different variable bindings). Occurrences of `all` may be accompanied by an optional sorting specification.

Example 2. Left: A query term retrieving departure and arrival stations for a train in the train document. Partial term specifications (partial curly braces) are used since the train document might contain additional information irrelevant to the query. *Right:* A construct term creating a summarised representation of trains grouped inside a `trains` term. Note the use of the `all` construct to collect all instances of the `train` subterm that can be created from substitutions in the substitution set resulting from the query on the left.

```
travel {{
  train {{
    departure {{
      station { var From } }},
    arrival {{
      station { var To }   }}
  }}
}}
```

```
trains {
  all train {
    from { var From },
    to   { var To }
  }
}
```

Construct-Query Rules (short: rules) relate a construct term to a query consisting of AND and/or OR connected query terms. They have the form

`CONSTRUCT` *`Construct Term`* `FROM` *`Query`* `END`

Rules can be seen as "views" specifying how to obtain documents shaped in the form of the construct term by evaluating the query against Web resources (e.g. an XML document or a database). Queries or parts of a query may be further restricted by arithmetic constraints in a so-called condition box, beginning with the keyword `where`.

Example 3. The following Xcerpt rule is used to gather information about the hotels in Vienna where a single room costs less than 70 Euro per night and where pets are allowed (specified using the `without` construct).

```
CONSTRUCT
  answer [ all var H ordered by [ P ] ascending ]
FROM
  in {
    resource { "http://hotels.net"},
    voyage {{
      hotels {{
        city { "Vienna" },
        desc var H ⇝ hotel {{
          price-per-room { var P },
          without no-pets {}
        }}
      }}
    }}
   } where var P < 70
END
```

An Xcerpt query may contain one or several references to *resources*. Xcerpt rules may furthermore be *chained* like active or deductive database rules to form complex query programs, i.e. rules may query the results of other rules. Recursive chaining of rules is possible. In contrast to the inherent structural recursion used e.g. in XSLT, which is essentially limited to the tree structure of the input document, recursion in Xcerpt is always explicit and free in the sense that any kind of recursion can be implemented. Applications of recursion on the Web 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 HTML documents 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 4. The following scenario illustrates the usage of a "conceptual" recursion to find train connections, including train changes, from Munich to Vienna.

The `train` relation (more precisely the XML element representing this relation) is defined as a "view" on the train database (more precisely on the XML document seen as a database on trains):

```
CONSTRUCT
  train [ from [ var From ], to [ var To ] ]
FROM
  in {
    resource { "file:travel.xml" },
    travel {{
      train {{
        departure {{ station { var From } }},
        arrival   {{ station { var To }   }}
      }}
    }}
  }
END
```

A recursive rule implements the transitive closure `train-connection` of the relation `train`. If the connection is not direct (recursive case), then all interme-

diate stations are collected in the subterm `via` of the result. Otherwise, `via` is empty (base case).

```
CONSTRUCT
  train-connection [
    from [ var From ],
    to   [ var To ],
    via  [ var Via, all optional var OtherVia ]
  ]
FROM
  and {
    train [ from [ var From ], to [ var Via ] ],
    train-connection [
      from [ var Via ],
      to   [ var To  ],
      via  [[ optional var OtherVia ]]
    ]
  }
END
CONSTRUCT
  train-connection [
    from [ var From ],
    to   [ var To ],
    via  [ ]
  ]
FROM
  train [ from [ var From ], to [ var To ] ]
END
```

Based on the "generic" transitive closure defined above, the following rule retrieves only connections between Munich and Vienna.

```
GOAL
  connections {
    all var Conn
  }
FROM
  var Conn ⇝ train-connection [[ from { "Munich" } , to { "Vienna" } ]]
END
```

4 XChange: Evolution on the Web

XChange is a declarative language for specifying evolution of data and meta-data on the (Semantic) Web.

Events. As mentioned in section 2.2, XChange events are XML data, hence a generic data exchange between Web sites is supported by XChange, simplifying the transfer of parameters (e.g. raising time, recipient(s)) and thus the execution of actions in a user-defined synchronised manner.

Example 5. Assume that a train has 30 minutes delay. The control point that observes this raises the following event and sends it to `http://railways.com`:

```
event {
  recipient { "http://railways.com" },
  delay {
    train { departure { station { "Munich" }, date { "2004-09-23" },
                        time { "21:30" } },
            minutes-delay { "30" } }
  }
 }
```

The recipient sites (e.g. `http://railways.com` in Example 5) process the incoming events in order to execute (trans)actions or to raise other events. Thus only the information of interest for these sites is used (e.g. the time stamps may be used to decide whether the events are too old or not). The processing of events is specified in XChange by means of *event-raising rules*, *event-driven update rules*, and *event-driven transaction rules*.

Event-Raising Rules. The body of an event-raising rule may contain Xcerpt queries to incoming events, Xcerpt queries to XML resources (local or remote), and conditions that variables (specified in the queries to incoming events or XML resources) should satisfy. The head of an event-raising rule contains resource specifications (i.e. the resources to which events shall be sent) and event specifications (used to construct event instances). In the following, an example of an XChange event-raising rule is given.

Example 6. Mrs. Smith uses a travel organiser that plans her trips and reacts to happenings that might influence her schedule. The site `http://railways.com` has been told to notify her travel organiser of delays of trains Mrs. Smith travels with:

```
RAISE
  event {
    recipient { "http://travelorganizer.com/Smith" },
    delay {
      train { departure { var M, estimated-time { var DT + var Min } },
              arrival { var U, estimated-time { var AT + var Min } }
    }
  }
  }
ON
 event {{
   delay {{
     train {{ departure { var M ⇝ station { "Munich" },
                          var Date ⇝ date { "2004-09-23" },
                          time { var DT ⇝ "21:30" } },
              minutes-delay { var Min } }}
   }}
  }}
FROM
 in {
    resource { "http://railways.com" },
    travel {{ train {{ departure {{ var M, var Date, time { var DT } }},
                       arrival {{ var U ⇝ station { "Vienna" }, time { var AT } }}
              }}
     }}
   }
END
```

Update Rules. The XChange update language uses rules to specify intensional updates, i.e. a description of updates in terms of queries. The notion of update rules is used to denote rules that specify (possibly complex) updates (i.e. insertion, deletion, and replacement). The body of an XChange update rule may (and generally does) contain Xcerpt queries (to XML resources and/or incoming events), which specify bindings for variables and conditions that variables should satisfy. The head of an XChange update rule contains resource specifications for the data that is to be updated, update specifications, and relations between the desired updates.

An XChange update specification is a (possibly incomplete) pattern for the data to be updated, augmented with the desired update operations. The notion of update terms is used to denote such patterns containing update operations for the data to be modified. An update term may contain different types of update operations. The head of an update rule may contain one or more update terms.

Example 7. At `http://railways.com` the train timetable needs to be updated as reaction to the event given in Example 5:

```
UPDATE
  in {
    resource { "http://railways.com" },
    travel {{
      last-changes-on { var L replaceby var RTime },
      train {{
        departure {{ station { var DS }, var Date, time { var DT },
                     insert estimated-time { var DT + var Min } }},
        arrival {{ time { var AT }, insert estimated-time { var AT + var Min } }}
      }}
    }}
  }
ON
  event {{
    raising-time { var RTime },
    delay {{
      train {{ departure { station { var DS }, var Date, time { var DT } },
               minutes-delay { var Min } }}
    }}
  }}
END
```

Synchronisation of Updates. XChange provides the capability to specify relations between complex updates and execute the updates synchronously (e.g. when booking a trip on the Web one might wish to book an early flight *and* of course the corresponding hotel reservation, *or* a late flight *and* a shorter hotel reservation). As the updates are to be executed on the Web, network communication problems could cause failures of update execution. To deal with such problems, an explicit specification of synchronisation of updates is possible with XChange, a kind of control which logic programming languages lack. Means to realise synchronisation of updates on the Web: dependent updates (specified by means of XChange synchronisation operations, which express ordered/unordered *conjunction* of updates, or ordered/unordered *disjunction* of updates), time specification for updates (expressing e.g. an explicit time reference, or a timeout for update execution), and user specified commitment (realised by supporting transactions in XChange).

Transaction Rules. Transaction rules are very similar to XChange update rules, with the important difference that transactions (consisting of event and/or update specifications that should be raised and/or executed in an all-or-nothing manner) are specified in the head of reaction rules. In case of transaction abort, a rollback mechanism that undoes partial effects of a transaction is to be used.

XChange transactions are transactions executed on user requests or as reactions to incoming XChange events. The latter transactions are specified in the head of XChange event-driven transaction rules. An example of an event-driven transaction rule is given next.

Example 8. The travel organiser of Mrs. Smith uses the following rule: if the train of Mrs. Smith is delayed such that her arrival will be after 23:00h then book a cheap hotel at the city of arrival and send the telephone number of the hotel to her husband's address book. The rule is specified in XChange as:

```
TRANSACTION
  and [
     update {
       in { resource { "http://hotels.net" },
        reservations {{
          insert reservation { var H, name { "Christina Smith" },
                               from { "2004-09-23" }, until { "2004-09-24" } }
         }}   }    },
     update {
       in { resource { "address-book://addresses/my-husband" },
        addresses {{
          insert my-hotel { phone { var Tel },
                                     remark { "I'm staying in Vienna over night!" } }
      }}     }   }
   ]
ON
  event {{
    sender { "http://railways.com" },
      delay {{
        train {{ arrival { station { var City ⇝ "Vienna" }, estimated-time { var ETime } }
        }}
      }}
     }} where var ETime after 23:00
FROM
  in {
   resource { "http://hotels.net" },
   voyage {{
     hotels {{ city { var City },
               desc var H ⇝ hotel {{ price-per-room {var P}, phone { var Tel } }} }}
     }}
   } where var P < 70
END
```

5 Querying and Evolution on the Semantic Web

The vision of the Semantic Web is that of a Web where the semantics of data is available for processing by automated means. Based on standards for representing Semantic Web (meta-)data, such as RDF and OWL, the need for a expressive, yet easy-to-use query language to access the large amounts of data expected to be available in the Semantic Web is evident. However, in contrast to most current approaches for querying the Semantic Web, we believe that it is crucial to be able to access both conventional and Semantic Web data within the same query language. The following examples illustrate based on a small set of RDF data some of the peculiarities and pitfalls of a Semantic Web query language and how these can be handled in Xcerpt and XChange.

Example 9. The data term shows a small excerpt from a book database together with a sample ontology over novels and other literary works. Some of the concepts used are drawn from the "Friend of a Friend" (foaf) project[5]. The rest of this paper uses prefixes to abbreviate the URLs for RDF, RDFS and OWL properties.

[5] http://www.foaf-project.org/

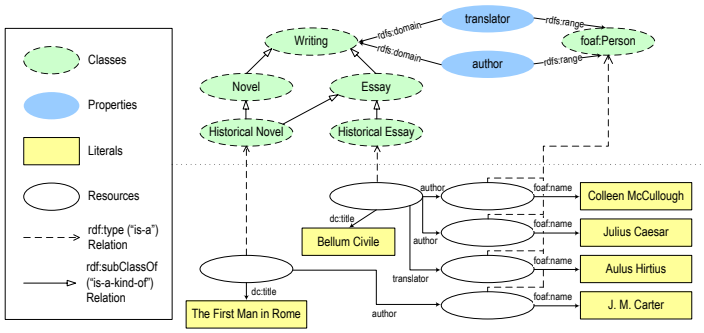

Fig. 1. RDF Graph for Example 9

At site `http://bookdealer.com`:

```
RDF {
  Historical_Novel {
    author {
      foaf:Person {
        foaf:name{"Colleen McCullough"}
      }
    },
    dc:title{"The First Man in Rome"}
  }
  Historical_Essay {
    author {
      foaf:Person {
        foaf:name { "Julius Caesar" }
      },
      foaf:Person {
        foaf:name { "Aulus Hirtius" }
      }
    },
    dc:title { "Bellum Civile" },
    translator {
      foaf:Person {
        foaf:name { "J. M. Carter" }
    } } }
  &author @ rdf:Property {
    rdfs:domain {
      ^&writing
    }
    rdfs:range {
      ^&foaf:Person
    }
  }
  &translator @ rdf:Property {
    rdfs:domain {
      ^&writing
    }
    rdfs:range {
      ^&foaf:Person
    }
  }
  &historical_novel @ rdfs:Class {
    rdfs:label { "Historical_Novel" },
    rdfs:subClassOf {
      &novel @ rdfs:Class {
        rdfs:label { "Novel" },
        rdfs:subClassOf {
          &writing @ rdfs:Class {
            rdfs:label { "Writing" }
          } } }
    }
    rdfs:subClassOf {
      &historical_essay @ rdfs:Class {
        rdfs:label { "Historical_Essay" }
        rdfs:subClassOf {
          &essay @ rdfs:Class {
            rdfs:label { "Essay" }
            rdfs:subClassOf {
              ^&writing
        } } } }
    }
  }
}
```

This data term represents the RDF graph shown in Figure 1: There are two books in the data, the first one is classified (via `rdf:type`) as a *Historical Novel*, a term defined in the sample ontology. Furthermore, it has an `author` that is a `foaf:Person` with `foaf:name` "Colleen McCullough". The second one also has a `translator` and several `author`s. The sample ontology is basically a conceptual hierarchy for a (small subset of) terms used to classify books and other literary works. The terms are related by `rdfs:subClassOf`, indicating that, e.g., a *Historical Novel* is a kind of *Novel* that, in turn, is a kind of *Writing*. Note the Xcerpt notation `id @ ...` (`^id`, resp.) for representing `ID` (`IDREF`, resp.) attributes.

For reasons of brevity and readability, a representation of the RDF graph as an Xcerpt data term is used that is very close to the syntactic representation of

RDF graphs in XML. In this respect, our approach is similar to [4] for querying RDF data with XQuery. However, as is illustrated in the following, there are several peculiarities of Semantic Web data that most query languages for the conventional Web do not support easily.

Properties are optional and multi-valued. In RDF, relations such as `author` or `rdf:type` between objects (also referred to as resources) are called properties. In contrast to traditional knowledge representation techniques, such as Frames, all properties in RDF are optional and multi-valued: it is not possible to formally restrict the number of properties of the same type between two resources. E.g., a `Writing` may have no translator, one translator, or any number of translators.

In Xcerpt, optional and multi-valued properties can easily be retrieved by using `all` and `optional`, as shown in Example 10.

Example 10. Retrieve all writings from `http://bookdealer.com` together with their title only if they have a title. Also return any authors or translators for each book, if there are any. `subClassOf[ var BookType, "Writing"]` expresses that the type of the resource must be a subclass of `Writing` (cf. Example 12).

```
CONSTRUCT
  RDF {
    var BookType {
      optional dc:title{ var Title },
      all optional author{ var Author },
      all optional translator{ var Translator }
    }
  }
FROM
  and {
    in {
      resource { "http://bookdealer.com" },
      RDF {{
        desc var BookType {
          optional author{ var Author },
          optional dc:title{ var Title },
          optional translator{ var Translator }
        }
      }}
    },
    subClassOf[ rdfs:Class {{ rdfs:label{var BookType} }},
                rdfs:Class {{ rdfs:label{"Writing"} }}     ]
  }
END
```

Inference. One of the most fundamental promises of the Semantic Web is that, given a machine-processable semantics for data, new data can be inferred from existing one automatically.

Example 11. All persons that have published together can be retrieved by the following program. The reflexivity of the co-author relation is expressed using unordered subterm specification (i.e. curly braces), as in `co-author{var X, var Y}`.

```
CONSTRUCT
  co-author{var X, var Y}
FROM
  and {
    in {
      resource { "http://bookdealer.com" },
      RDF {{
        var BookType {{
          author { var X },
```

```
          author { var Y },
        }}
      }}
    },
    subClassOf[ rdfs:Class {{ rdfs:label{var BookType} }},
                rdfs:Class {{ rdfs:label{"Writing"} }}     ]
  }
END
```

More interesting is the kind of inference that arises from traversing the structure of the RDF graph recursively, similar to the train connections in Example 4. This is required, e.g., to compute the closure of transitive relations. RDF Schema defines two such transitive relations `rdfs:subClassOf` and `rdfs:subPropertyOf`, OWL allows the definition of additional transitive properties by classifying them as subclasses of `owl:TransitiveProperty`.

Support for RDF Schema. RDF Schema extends RDF with a set of predefined properties and specifies a (partial) semantics for these properties. Most notably, means for defining a subsumption hierarchy for concepts and properties are provided by `rdfs:subClassOf` and `rdfs:subPropertyOf`. These properties are transitive. E.g., if a query asks for all `Writings`, also resources that are classified as `Novels` or `Historical_Essays` should be returned (under the sample ontology specified above).

Example 12. The semantics of, e.g., `rdf:subClassOf` can be easily implemented in Xcerpt as demonstrated by the following program. The transitive closure of `rdf:subClassOf` is computed using recursion (cf. Example 4).

```
CONSTRUCT
  subClassOf[ var Subclass, var Superclass ]
FROM
  or { RDF {{
          desc var Subclass ⇝ rdfs:Class {{
            rdfs:subClassOf {
              var Superclass ⇝ @ rdfs:Class {{ }}
            }
          }}
        }},
        and [ RDF {{
                desc var Subclass ⇝ rdfs:Class {{
                  rdfs:subClassOf {
                    var Z ⇝ rdfs:Class {{ }}
                  }
                }}
              }},
              subClassOf[ var Z, var Superclass ]    ]
    }
END
```

Other predefined relations from RDF (Schema) such as `rdf:type`, `rdfs:domain`, or `rdfs:range` can be implemented in a similar manner.

Evolution. Evolution and reactivity are at the core of the Semantic Web vision. It is crucial that relevant changes to data that has been used by a Semantic Web agent, e.g. in deciding which book to buy or what train to book, are consistently and rapidly propagated to all interested parties.

Example 13. Mrs. Smith is very interested in Essays and therefore wants to be notified about any new book that is classified as an Essay once it is added to the

list of books managed by `http://bookdealer.com`. The following two XChange programs illustrate this scenario, the left hand shows the event specification, the right hand an update that triggers the event.

```
RAISE
  event {
    recipient { "http://.../Smith" },
    new_book {
      type { var Type },
      all optional var Title
    }
  }
ON
  event {
    sender {"http://bookdealer.com"},
    RDF {{
      insert var Type {{
        optional var Title ⇝ dc:title{{}}
      }}
    }}
  }
FROM
  subClassOf[
    rdfs:Class {{ rdfs:label{var Type} }},
    rdfs:Class {{ rdfs:label{"Essay"} }}
  ]
END
```

```
UPDATE
in {
  resource { "http://bookdealer.com" },
  RDF {{
    insert Historical_Novel {
      dc:title { "Ein Kampf um Rom" }
      author {
        foaf:Person {
          foaf:name { "Felix Dahn" }
        }
      }
    }
  }}
}
END
```

6 Perspectives and Conclusion

This article first introduced into the deductive query language Xcerpt and the event and update language XChange. It furthermore illustrated how the deductive capabilities of these languages make them well suited for querying and evolution on the *conventional* as well as the *Semantic* Web.

Xcerpt and XChange are ongoing research projects. Whereas the language Xcerpt is already in an advanced stage [1] (cf. also `http://www.xcerpt.org`), work on the language XChange has begun more recently, but first results are promising. Applying both languages to more complex Semantic Web applications is currently investigated and will likely result in the implementation of (partial) reasoners for certain ontologies. Also, including native support for certain reasoning constructs might increase usability and/or performance.

Acknowledgement. This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

References

1. Schaffert, S., Bry, F.: Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In: Research Report PMS-FB-2004-7. (2004)
2. Liu, M., Lu, L., Wang, G.: A Declarative XML-RL Update Language. In: Proc. Int. Conf. on Conceptual Modeling (ER2003). LNCS 2813, Springer-Verlag (2003)
3. Papamarkos, G., Poulovassilis, A., Wood, P.T.: Event-Condition-Action Rule Languages for the Semantic Web. In: Workshop on Semantic Web and Databases, Berlin, VLDB'03 (2003)
4. Robie, J.: The Syntactic Web: Syntax and Semantics on the Web. In: XML Conference and Exposition 2001, Orlando, Florida (2001)