INSTITUT FÜR
INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67 D–80538 München

Ludwig
Maximilians
Universität
München

LMU

Towards Data-Integration on the Semantic Web: Querying RDF with Xcerpt

Oliver M. Bolzer

Diplomarbeit

Beginn der Arbeit:	01.09.2004
Abgabe der Arbeit:	ABGABEDATUM
Betreuer:	Prof. Dr.François Bry
	Tim Furche
	Sebastian Schaffert

Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den ABGABEDATUM Oliver M. Bolzer

Zusammenfassung

Obwohl RDF als XML gespeichert wird, sind auf Grund der syntaktischen Variabilität und der Notwendigkeit des automatischen Schliessens XML-Anfragesprachen ungeeignet um RDF-Daten zu verarbeiten. Zur Zeit werden zahlreiche Anfragesprachen für RDF vorgeschlagen. Diese sind jedoch auf RDF spezialisiert und nicht in der Lage, beliebige XML Daten anzufragen. Um einheitlichen Zugriff auf das traditionelle Web und das neue Wissen des Semantischen Webs zu ermoöglichen, sind flexiblere Web-Anfragesprachen notwendig.

Diese Arbeit untersucht das Anfragen von RDF mit Xcerpt, einer deklarativen, regelbasierenden Anfrage- und Transformationssprache für XML und andere semistrukturierte Daten. Zwei Darstellungen von RDF Daten als Xcerpt Daten Terme werden vorgeschlagen, realisiert als Sichten über konkrete Serialisierungen in verschiedenen XML Formaten. Weiterhin wird eine Implementierung des automatische Schliessen von RDF Schema in Form von Xcerpt-Regeln erforscht. Schliesslich werden Anwendungsbeispiele betrachtet, um die Vorteile einer Anfragespraache hervorzuheben, die in der Lage ist sowohl das traditionelle Web als auch das Semantische Web anzufragen.

Abstract

Although RDF is serialized using XML, the many possible syntactic forms and the need for inferencing make it difficult to query RDF using existing XML query languages. Numerous new query languages for RDF with built-in knowledge about the semantics of particular inferencing formalisms like RDF Schema and OWL have been proposed or are currently under development. However most, if not all, are specific to RDF and not capable of querying arbitrary XML data. In order to better integrate and leverage both kinds of data on the Semantic Web, a new generation of versatile Web query languages is needed, allowing uniform querying and integration of both the traditional Web and the emerging new content of the Semantic Web.

This thesis investigates querying RDF using Xcerpt, a declarative, rule-based query and transformation language for XML and other semi-structured data on the Web. Two representations of RDF data as Xcerpt data terms are proposed, realized as views over concrete serializations of RDF data in arbitrary XML-based serialization formats. Furthermore, the possibility of implementing RDF Schema inferencing directly using Xcerpt rules is investigated. Finally, this thesis examines several use cases, demonstrating the advantages of a query language that can query and integrate the traditional Web and the Semantic Web.

Acknowledgements

I thank my supervisor, Prof. Dr. François Bry for many insightful conversations during the development of the ideas in this thesis.

My advisors, Tim Furche and Dr. Sebastian Schaffert, also made many helpful suggestions. Was it not for their professional advice and helpful comments, this thesis would not be what it is today.

Contents

CHAPTER ONE

Introduction

The World Wide Web is a vast collection of resources accessible on the Internet. Today, most of the resources are either documents marked up using HTML [1] or multimedia files like graphics and sounds, both intended for consumption by humans using visual user agents, such as web browsers and cellphones. With the Web rapidly growing on a daily basis, users rely heavily on search engines and portal sites to find web sites and documents related to their topic of interest.

In general, search engines collect as many documents as possible off the Web into a database, indexed by the words used in the documents. Users then query the search engine using one or more keywords to receive a list of documents containing these words. As words of natural languages tend to be ambiguous, the correct choice of keywords that yield the best results is left to the user. Portal sites on the other hand, collect information only from a limited number of web sites known to be relevant to the portal's topic. Specific information like headlines and price information is extracted by parsing the HTML and natural language of collected documents using handcrafted and heuristic rules, or by exchanging information using a small number of purpose-specific XML schemata. Utilizing XML [2], developers can easily define new markup languages that can be processed using generic XML tools. However, interpretation and semantics of such languages need to be individually coded into applications and compatibility between two schemata can only be achived through handcrafted conversion rules.

The Semantic Web [3] is a vision of the Web of tomorrow. Instead of relying on the handcrafted and heuristic extraction of data from natural-language documents, the Web is extended with data that has well-defined meaning, allowing machines to infer implicit information out of the data, enabling reuse and integration of data from large numbers of independent sources and across applications using automated tools. The *Resource Description Framework* (RDF) [4] is one of the corner stones of the Semantic Web. It is a framework for building web-based knowledge bases, consisting of a simple graph-based data model for asserting machine-processable statements about resources on the Web, an XML-based serialization format and a mechanism for defining custom vocab-

ularies. With the popularity of practical vocabularies such as the FOAF [5] vocabulary for describing personal data and the RSS [6] vocabulary for news headline syndication, the amount of RDF data available on the Web is rapidly increasing.

Even tough RDF data is frequently serialized using XML, differences between the data models of RDF and XML, peculiarities with the serialization format and the need for inferencing make it difficult to query RDF using existing XML query languages such as XQuery [7] and XSLT [8]. Numerous new query languages for RDF have been proposed or are currently under development. However most, if not all, are specific to RDF and are not capable of querying arbitrary XML data. In order to better integrate and leverage both kinds of data on the Web, a new generation of *versatile* Web query languages is needed, allowing uniform querying and integration of both the traditional Web and the emerging new content of the Semantic Web.

Xcerpt [9] is a declarative, rule-based query and transformation language for XML and other semi-structured data on the Web. With its native support for graph-based data and roots in logic programming, Xcerpt has the potential to become a versatile Web query language. This thesis investigates the suitability of Xcerpt and its underlying formalisms for querying and inferencing with both RDF and traditional XML data. The major contribution of this work is an abstraction of RDF data as Xcerpt data terms, creating a common representation of XML and RDF data, and a demonstration of Xcerpt's capabilities to perform inferencing with RDF data.

This thesis is structured into nine chapters. Chapter 2, "Preliminaries" introduces RDF and Xcerpt, the two technologies on which the thesis builds. Chapter 3, "Mission Statement" gives the motivations and goals of the thesis in form of a mission statement. Chapter 4, "State-of-the-Art and Related Work" summarizes the results of investigation into related research, namely on the diversity of serialization formats and query languages for RDF. Based on the findings, Chapter 5, "Querying RDF with Xcerpt" details how RDF can be queried using Xcerpt, with support for multiple representations of RDF data and arbitrary serialization formats. Chapter 6, "RDF Schema Inferencing with Xcerpt" demonstrates Xcerpt's capability to perform inferencing on RDF data. Use cases inspired by real world problems and solutions to them are shown in Chapter 7, "Use Cases". Based on limitations of Xcerpt uncovered during the research for this thesis, several extensions to the language are proposed in Chapter 8, "Proposed Extensions to Xcerpt". Finally, Chapter 9, "Conclusions" rounds the thesis off with concluding remarks.

CHAPTER TWO

Preliminaries

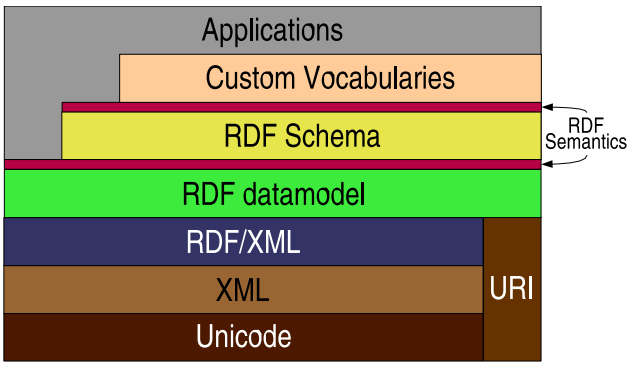
2.1 Resource Description Framework

The *Resource Description Framework* (RDF) is a family of standards from the World Wide Web Consortium (W3C) for asserting machine-processable statements about resources on the World Wide Web. Using the formalisms of RDF, computer programs can infer large amount of implicit information out of a much smaller set of explicit assertions using logical reasoning. RDF aims to aid with the evolution of the Web from a mere repository of documents to a global knowledge base.

The framework is described by six documents:

- RDF Concepts and Abstract Syntax [4]
- RDF/XML Syntax Specification [10]
- RDF Vocabulary Description Language 1.0: RDF Schema [11]
- RDF Semantics [12]
- RDF Test Cases [13]
- RDF Primer [14]

While the latter two are complementary documents intended to facilitate the adoption of RDF, the first four are actual specifications, each specifying a different layer of the framework. At the center lies the abstract data model of RDF accompanied by precisely defined semantics. RDF Schema allows the definition of new vocabularies which can be used with the data model. A format for serialization and exchange of RDF data is specified in form of RDF/XML. The whole framework is itself build on top of various Web standards such as XML, URIs and Unicode.



This chapter will first introduce the core data model of RDF and then continue on with describing RDF's vocabulary description language, *RDF Schema*, as well as the standard format for serializing RDF data: *RDF/XML*. Alternative serialization formats for RDF data and numerous proposals for querying RDF will be discussed in more detail in Chapter 4, "State-of-the-Art and Related Work".

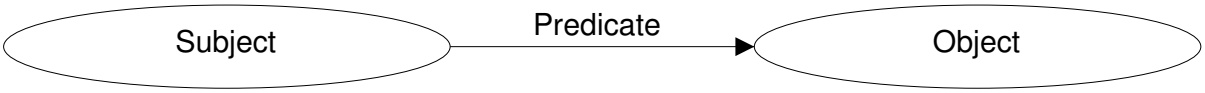
Note: Usage of URI references for identification purposes is one of the central concepts of RDF. For concise and improved readability, long URI references are abbreviated to the form `prefix:suffix`. The abbreviated URI references should be interpreted as the concatenation of the partial URI reference associated with the prefix and the suffix. The prefix `rdf` stands for the RDF namespace `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, and `rdfs` for the namespace of RDF Schema `http://www.w3.org/2000/01/rdf-schema#` . Others will be introduced as they appear.

2.1.1 The RDF Data Model

The data model of RDF is a simple graph-based model, describing resources on the Web and relationships between them. Specifically designed for integration of heterogeneous data on the Web, emphasis has been given to the simplicity and domain-neutrality of the data model, not limiting RDF to specific fields of application.

2.1.1.1 Triples: Describing Resources

The basic building blocks of RDF data are triples. Regularly also referred to as statements, each triple consists of three parts: a subject, a predicate, and an object. Similar to a minimal phrase in natural language, each triple describes a property of it's subject, whose value is the object of the triple. It can be said that a triple asserts a relationships between the two resources, denoted by the subject and the object. The nature of the relationship is identified by the predicate and is directed from subject to object. This is best illustrated by a simple diagram:



The individual components of triples are identified using *URI references*, Uniform Resource Identifiers [15] with optional fragment identifiers. URIs are identifiers that can be independently created by different organizations and individuals while avoiding clashes. They are most commonly used to identify and address network resources, but can refer to anything that needs to be identified, including persons and even abstract concepts. Any identifyable entity can thus be considered a resource.

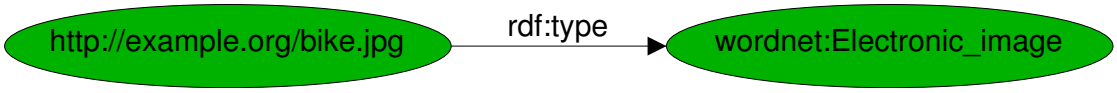
For instance, the natural language assertion

`http://example.org/bike.jpg is an electronic image .`

can be represented using an RDF triple that has

- `http://example.org/bike.jpg` as its subject,
- `rdf:type` as its predicate and
- `wordnet:Electronic_image`[1] as its object.

Or, in form of a diagram:



Note how URI references are used to identify not only the subject, but also the predicate and object, instead of the words “” and “mountain bike”. While the exact meaning of a word of a natural language can be ambiguous, the precise interpretation of an artificial identifier can simply be defined.

2.1.1.2 RDF Graphs: Collections of Triples

An RDF triple seldom comes alone. Usually a triple is part of a larger set of triples, together forming an RDF graph whose nodes are the subjects and object of the triples. The directed arcs connecting nodes correspond to the triples’ predicates.

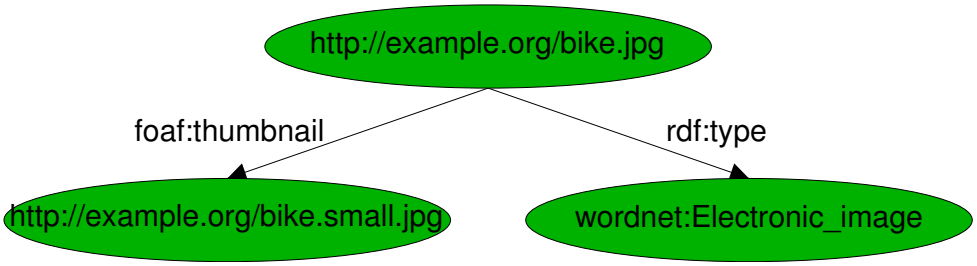
For example, the two statements

`http://example.org/bike.jpg is an electronic image .`

`http://example.org/bike.jpg has a thumbnail-image http://example.org/bike.small.jpg .`

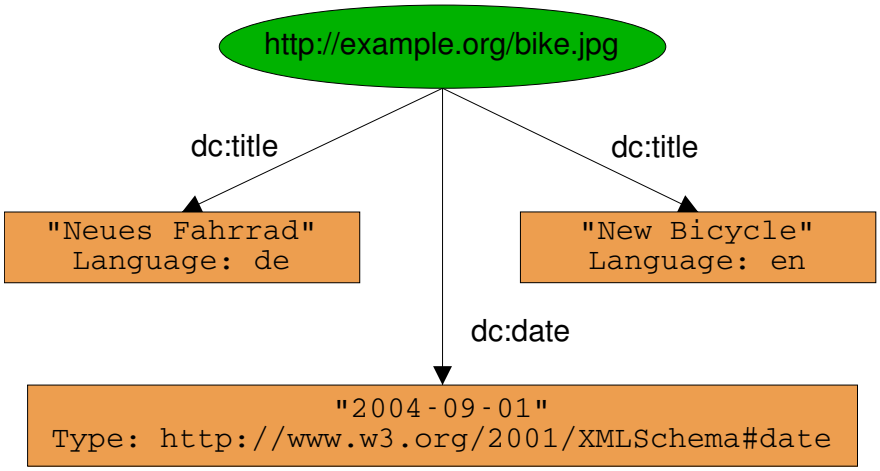
fall nicely together into a graph with three nodes and two arcs[2].

[1] `wordnet` prefix: `http://xmlns.com/wordnet/1.6/`
[2] `foaf` prefix: `http://xmlns.com/foaf/0.1/`



2.1.1.3 Literal Nodes

One of the node types supported by RDF besides URI references are *literals*, representing atomic values. For example, the title of the photograph introduced above can be specified using a so called plain literal, a string with an optional language specification. Typed literals on the other hand have a datatype associated with them, identified by an URI, and can represent any atomic value that is expressible using a string of characters. Literals can only be objects and are not allowed as predicates or subjects.



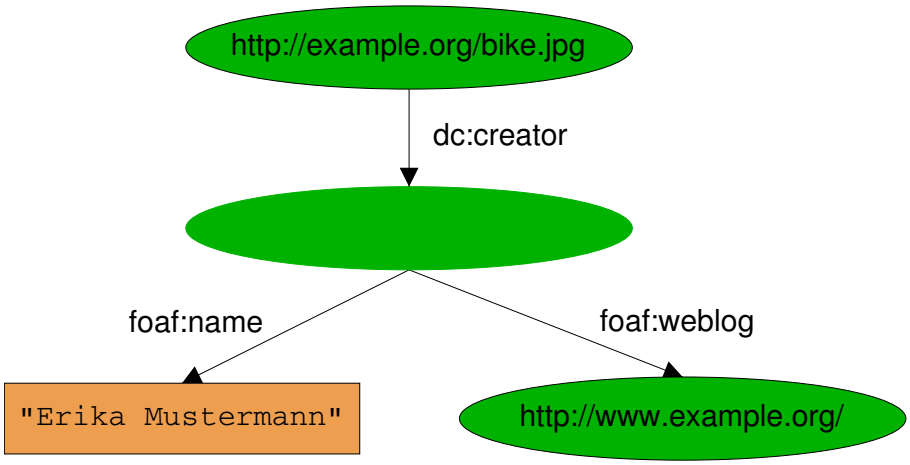
In this graph, the title and the creation date of the example photograph are asserted using the `dc:title`[3] and `dc:date` predicates. The title is asserted twice, once in English and in once German, using language codes defined by ISO 639-1 [16] to specify the language. The resource's creation date is "2004-09-10" and the type of the literal is identified by the URI `xsd:date`[4] , specifying the calendardate type defined by XMLSchema [17].

[3] `dc` prefix: `http://purl.org/elements/1.1/`
[4] `xsd` prefix: `http://www.w3.org/2001/XMLSchema`

2.1.1.4 Blank Nodes

The the third type of node supported by RDF are nodes without identifiers, called *blank nodes*. While literals can represent atomic values, data is frequently more complex and structured. For example, the creator of the example photograph could be asserted by specifying his or her name using an literal as object. However, it is more likely for the creator to be represented by a more structure resource, e.g. having additional properties besides a name, such as e-mail address and personal interests or relationships to other people. Such structured values are expressed by describing resources that have multiple properties. The nodes representing such resources can be identified by URIs, but it might not be necessary for them to be explicitly identified, as long as they are distinguishable from others in the realm of the a particular RDF graph. Blank nodes are used in such cases.

In the following graph, the creator of `http://example.org/bike.jpg` is only indirectly identified to be somebody named `Erika Mustermann` who keeps a weblog at `http://www.example.org/`.



Lacking an identifier, blank nodes can not be referenced form multiple RDF graphs. However, it does not mean that the resource represented by a blank node is unique to a single RDF graph. A blank node merely indicates that a resource having certain properties exists, without identifying it. If a node with the same properties is described in another RDF graph, the two nodes might or might not represent the same resource.

Given another RDF graph containing a blank node who's name is also asserted to be `Erika Mustermann` and has the e-mail address `mailto:erika@example.org`, it is impossible to determine whether the two Erikas are the same person or not. This poses a difficulty when merging multiple RDF graphs. If the two blank nodes were in fact referring to the same person, additional information about the creator of the photograph would become available by merging the two blank nodes. However, if the two blank nodes indeed represented two distinct persons, merging them would lead to

the false assertion regarding the weblog of the creator of the photograph. Therefore the blank nodes are not merged and kept separate to be on the safe side. Applications are allowed to merge blank nodes if they have additional knowledge, e.g. an membership management application knowing that two resources having the same membership ID number must represent the same member of a club.

2.1.2 Defining new Vocabularies: RDF Schema

RDF Triples express simple facts in a way that can be easily processed by machines. However, in order to act in some intelligent manner, machines needs to have at least some prior knowledge about the terms used in the triples they encounter. Note how the URIs for terms like "title" and "mbox" have not been made up in the preceding examples. Instead of inventing new URIs that nobody would know of, URIs defined by the FOAF [5], Wordnet [18] [19] and Dublin Core [20] projects have been (re)used. Each project defines a *vocabulary*, a set of URIs that represent specific terms for a certain purpose, using *RDF schema* [11].

Officially named the "*RDF Vocabulary Description Language*", RDF Schema is itself an RDF vocabulary for describing terms in an RDF vocabulary. By interpreting the terms of RDF Schema, systems that consume RDF data can check whether previously unknown terms correspond to any terms the system has specific knowledge of and is interested in.

RDF Schema allows the definition of two types of terms: classes and properties. These concepts are similar to those known from Object Oriented Programming, but differ in that classes do not describe what properties an individual resource must have, as all properties are optional in RDF. Instead, the properties that are known about a resource may determine what classes an individual resource belongs to. In contrast to Object Oriented Programming, individual resources may also be instances of not only one but any number of classes of completely distinct class hierarchies.

This section only touches on the definition of classes and properties, the two main aspects of RDF Schema. The language also includes additional vocabularies for annotating vocabulary definitions, as well as special classes for containers and lists as well as for reifying statements. Readers are directed toward the specification for details on these.

2.1.2.1 Classes and Class Hierarchies

Resources can be instances of one or more classes. A class is a grouping of resources with similar characteristics that individual resources belong to, in the same way that

an individual person belongs to the the group "Human". The fact that a resource is an instance of a class may asserted using the property `rdf:type`. For example,

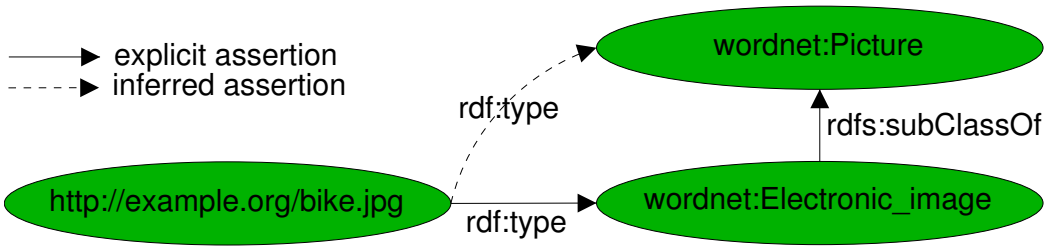
```
http://example.org/bike.jpg rdf:type wordnet:Electronic_image
```

asserts that the resource `http://example.org/bike.jpg` is an "Electronic Image" as defined by the Wordnet vocabulary. The actual human-readable description of instances of the class is asserted to be "an image represented as a two dimensional array of brightness values for pixels" in the vocabulary, using the `rdfs:label` property for vocabulary annotation.

Classes are themselves resources that are instances of the `rdfs:Class` class and can form hierarchies consisting of sub- and superclasses with all instances of a class also being instances of all superclasses of the class. RDF Schema provides the property `rdfs:subClassOf` for stating that class is a subclass of another. For example, the Wordnet vocabulary defines:

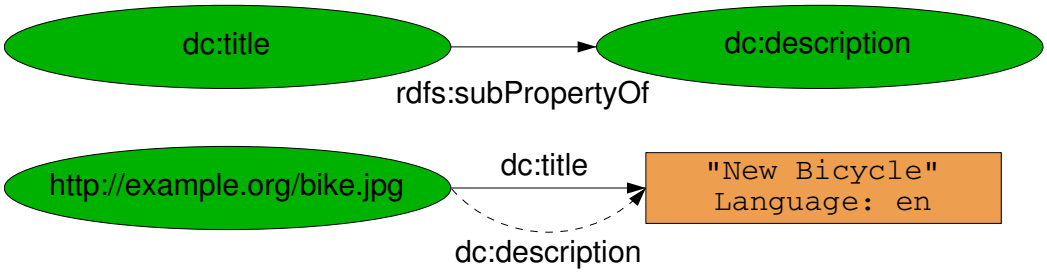
```
wordnet:Electronic_image rdfs:subClassOf wordnet:Picture
```

Based on this vocabulary description of `wordnet:Electronic_image` and the assertion that `http://example.org/bike.jpg` is an instance of that class, an RDF Schema-aware application searching for resources that are pictures, would also find `http://example.org/bike.jpg`, even though it is not explicitly asserted to be a picture. The additional information is only implied, but can be inferred by the application.



2.1.2.2 Properties and Property Hierarchies

Property URIs are instances of the `rdfs:Property` class. Analogous to class hierarchies, RDF Schema allows property hierarchies to be build, using `rdfs:subPropertyOf`. If a property P is an subproperty of another property P', a triple asserting a relationship between subject and object using the predicate P, also implies the relationship P' between the same subject and object. For example, a title of a resource is also an special form of a description:

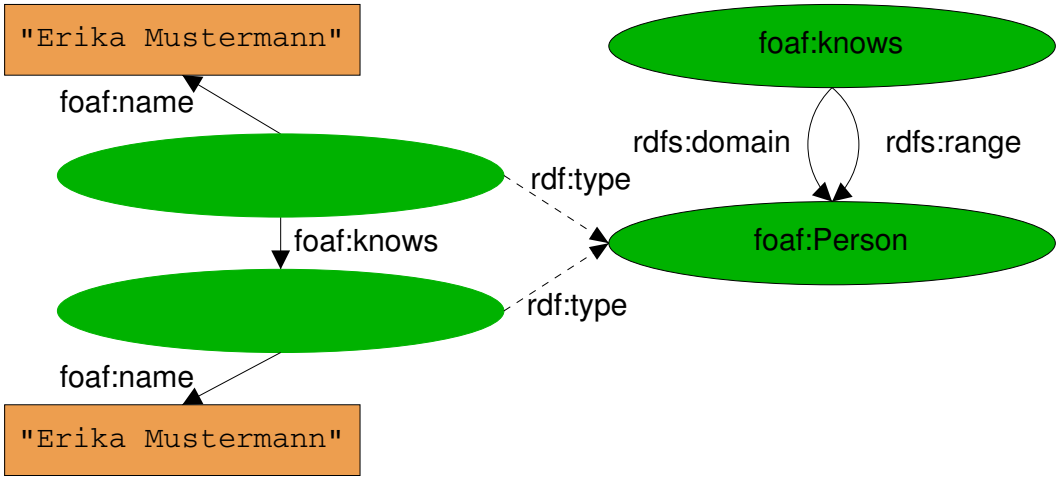


Besides other properties, a properties can also imply class memberships of the resources it relates with each other. The `rdfs:range` property of a resource specifies that the objects of triples that use the resource as property are instances of a specific class. `rdfs:domain` on the other hand, specifies a class that the subjects of such triples are instances of. For example, the FOAF vocabulary asserts

`foaf:knows rdfs:domain foaf:Person` and

`foaf:knows rdfs:range foaf:Person`,

defining that the `foaf:knows` property connects two resources that are both instances of the `foaf:Person` classs. So, if the blank node named "Erika Mustermann" from the previous section `foaf:knows` another blank node named"Max Mustermann", the implication is that both are `foaf:Person` instances.



2.1.3 RDF/XML: Serialization Format for RDF

RDF/XML [10] is the W3C's standardized format for interchange of RDF, serializing RDF graphs into XML document trees. The basic idea of RDF/XML is to maps the nodes as well the edges of an RDF graph to nested XML elements.

2.1.3.1 Mapping nodes and edges to elements

Example 2.1.1 shows an RDF/XML serialization of the RDF graph involving Erika Mustermann presented earlier.

Example 2.1.1 RDF/XML Serialization of an RDF Graph

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:foaf="http://xmlns.com/foaf/0.1/" >
  <rdf:Description rdf:about="http://example.org/bike.jpg">
    <dc:creator>
      <rdf:Description>
        <foaf:name>Erika Mustermann</foaf:name>
        <foaf:mbox>
          <rdf:Description rdf:about="http://www.example.org" />
        </foaf:mbox
      </rdf:Description>
    </dc:creator>
  </rdf:Description>

</rdf:RDF>
```

`rdf:Description` elements represent the nodes of the RDF graph, indicating the URIs using `rdf:about` attributes. The blank node simply is not identified. The direct children of the `rdf:Description` elements are the properties describing each node. The namespace and local name together identify the URI of the predicate. The child element of such elements represents the accompanying object, which in turn can have child elements of it's own, should the resource also be the subject of additional triples itself.

This alternating nesting of elements representing nodes and edges of RDF graphs in RDF/XML is often characterized as "striping" [21]. The nesting can go on into arbitrary depth.

2.1.3.2 Abbreviations

In order to avoiding overly verbose serializations, and make RDF/XML more accessible to human readers of RDF/XML documents, RDF/XML allows to shorten the serialization using various abbreviations. For instance, a property with a literal as object can be expressed using an XML attribute. Also the resource that is the object of an statement can be named in the `rdf:resource` attribute of the element for the property, instead of an `rdf:Description` child element. Example 2.1.2 is also an RDF/XML serialization of the same graph as the previous example, but much shorter through the use of abbreviations.

The possibilities for abbreviation greatly complicates parsing of RDF/XML documents and extraction of individual triples, as it is impossible to determine whether a certain

Example 2.1.2 RDF/XML with Abbreviations.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:foaf="http://xmlns.com/foaf/0.1/" >
  <rdf:Description rdf:about="http://example.org/bike.jpg">
    <dc:creator>
      <rdf:Description foaf:name="Erika Mustermann">
        <foaf:mbox rdf:resource="http://www.example.org" />
      </rdf:Description>
    </dc:creator>
  </rdf:Description>

</rdf:RDF>
```

XML element represents a node or an edge of the RDF graph serialized into RDF/XML, without analyzing the whole path from the root element of the document to the element in question. Despite the intentions of the authors of RDF/XML, the format is not easy to parse, neither for machines nor humans.

2.2 Xcerpt

Xcerpt [9] is a declarative, rule-based query and transformation language for XML and other semi-structured data on the Web, developed at the Institute for Informatics of the Ludwig-Maximilians-Universität Munich.

In contrast to the path-navigational approach taken by XSLT [8] and XQuery [7], queries and constructions of answers are expressed using patterns in Xcerpt. One of the notables features of Xcerpt is rule-chaining, allowing complex queries to be split into multiple, simpler units. Furthermore, the language's foundation in logic programming makes it suited for reasoning.

2.2.1 Data Terms

Xcerpt has two different syntaxes for programs and semi-structured data: an XML-based syntax and the more compact Xcerpt syntax, an abstraction of XML into terms. Any XML document can be expressed in Xcerpt syntax. Such terms without any query-specific Xcerpt-constructs are referred to as *data terms* in contrast to *query terms* and *construct terms*, which will be introduced later.

A data term `t` corresponds to either an XML element or attribute and is of the form `namespace:label[`t_1`, ..., `t_n`]` or `namespace:label{`t_1`, ..., `t_n`}`. t_i are subterms of `t` and are either ordered when listed in square brackets or unordered when listed in curly brackets. Following XML terminology, subterms are often also referred to as child terms.

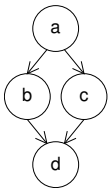
Example 2.2.1 Part of this Thesis' Bibliography in XML

```
<?xml version="1.0"?>

<file xmlns="http://bibtexml.sf.net/">
  <entry id="FOAF_spec_2004" version="1.0">
    <manual>
      <authorlist>
        <person>
          <first>Dan</first>
          <last>Brickley</last>
        </person>
        <person>
          <first>Libby</first>
          <last>Miller</last>
        </person>
      </authorlist>
      <title>FOAF Vocabulary Specification</title>
      ...
    </manual>
  </entry>
  <entry>....</entry>
  ...
</file>
```

Example 2.2.1 shows a small part from the bibliography of this thesis in it's raw XML form. Example 2.2.2 shows the corresponding data term in Xcerpt's own syntax. XML attributes are regular child terms in Xcerpt, but in order to distinguish them from XML child elements, they are grouped into the special subterm `attributes`. Note also that while regular child elements are ordered in XML in document order, attributes are not. The Xcerpt data term reflects this through the appropriate use of curly and square brackets.

Besides being more compact and allowing content to be unordered, the Xcerpt syntax includes a language construct that has no direct counterpart in XML: transparent references. Semi-structured data is often not only tree structured but graph structured, potentially including cycles. Although several reference and linking mechanisms for XML exist (ID/IDREF [2], XLink [22], etc.), all of them require explicit dereferencing. References in Xcerpt, on the contrary, are considered equivalent to regular parent-child relationships and are used to express graphs in a linear manner. Consequently, data terms can be seen as representations of rooted, node-labeled, directed graphs. The two data terms in Example 2.2.3 are equivalent in describing the graph show in Figure 2.1 .

Figure 2.1 A simple graph



Transparent dereferencing and the resulting ability to handle graphs is crucial to query-

Example 2.2.2 Xcerpt Data Term corresponding to Example 2.2.1

```
ns-default = "http://bibtexml.sf.net/"

file [
  entry [
    attributes{
      id { "FOAF_spec_2004" },
      version{ "1.0" }
    },
    manual [
      authorlist [
        person [
          first[ "Dan" ],
          last[ "Brickley" ]
        ],
        person [
          first[ "Libby" ],
          last[ "Miller" ]
        ],
      ],
      title[ "FOAF Vocabulary Specification" ],
      ....
    ]
  ],
  entry [ ... ],
  ...
]
```

Example 2.2.3 Two equivalent Data Terms using References

```
a{                          a{
  b { &id @ d{} },            b{ ^&id },
  c { ^&id }                  c{ &id @ d{} }
}                           }
```

ing graph-based, semi-structured data models such as RDF, as will be demonstrated later.

2.2.2 Rules

An Xcerpt programs consists of one or more rules. Each rule describes a single transformation and has the general form shown in Figure 2.2. As can be seen, a rule has two parts: a *construct term* (rule head) and a *query* (rule body).

Figure 2.2 General Form of a Rule

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

A complete rule is shown in Example 2.2.4 and will be used to explain the different parts of rules.

Example 2.2.4 an Xcerpt Rule

```
CONSTRUCT
  ul{
   all li { var TITLE ,
     ul {
       all li { var AUTHOR_F, var AUTHOR_L }
     }
   }
 }
FROM
  ns-default = "http://bibtexml.sf.net/"

  in{
    resource{ "file:bibliography.xml" },
    file {{
      entry{{
        manual{{
          title {{ var TITLE }},
          /author|authorgroup/{{
            person{{
              first{ var AUTHOR_F },
              last{ var AUTHOR_L }
            }}
          }}
        }}
      }}
    }}
  }
END
```

2.2.2.1 Queries

Queries in Xcerpt are formulated using patterns that are matched against data terms. They share the same basic structure as data terms, but contain one or more variables for selecting data items from the data terms they are matched against. They may have the following constructs in addition to those of data terms:

- partial specification (double square or curly brackets): only subterms related to the query need to be specified. The data term might contain other subterms that are ignored.
- subterm at arbitrary depth (`desc` keyword)
- Variables (`var` keyword): used to extract data from data terms that match the query. Depending on usage, the variable is bound to whole terms or only the labels or namespaces of them.
- optional subterms (`optional` keyword): the data term may or may not match against an optional query subterm. Variables are only bound when the optional query subterm actually matches.
- conjunctions and disjunctions (`or` and `and` keywords): multiple queries can be connected using *n*-ary boolean connectors.

- resource specifications (`resource` and `in` keywords): specify the document the query will be matched against. Queries without resource specifications query the results of other rules (see Section 2.2.3)

More advanced constructs like positional specification, negation, regular expressions for string and label matching are also available. For a more comprehensive list, consult the Xcerpt specification [9] and introductionary articles about Xcerpt [23] [24].

The query part of the rule shown in Example 2.2.4 is a query for selecting the title and authors of bibliographic entries from the BibTeXML file `bibliography.xml`. The title of an entry is bound to the variable `TITLE`, while the first and last name of one of the authors are bound to the variables `AUTHOR_F` and `AUTHOR_L`.

When applied to the data term of Example 2.2.2, the query returns the following sets of bindings:

1. {`TITLE` — "FOAF Vocabulary Specification", `AUTHOR_F` — "Dan", `AUTHOR_L` — "Brickley' }
2. {`TITLE` — "FOAF Vocabulary Specification", `AUTHOR_F` — "Libby", `AUTHOR_L` — "Miller"}

2.2.2.2 Construct Terms

Construct terms are used for assembling new data terms using variable bindings yielded by the associated query. On query match, the variables of an construct term are substituted with the term bound to the corresponding variable of the query. If there are multiple bindings, as many data terms as bindings are constructed. Additional language constructs specify how the variables are substituted. Most important is the `all` construct, which collects all combinations of bindings in it's scope into a list, while eliminating duplicates.

The construct term in Example 2.2.4 produces a HTML list of articles and their authors. The `all` around a term encompassing both variables leads to duplicate elimination and consequently the result is grouped by article. Applied to the previously presented variable bindings, the data term in Example 2.2.5 is constructed.

Example 2.2.5 Data Term constructed by Rule from Example 2.2.4

```
ul{
  li { "FOAF Vocabulary Specification",
    ul{
      li{ "Dan Brickley" },
      li{ "Libby Miller" }
    }
  }
}
```

2.2.2.3 `CONSTRUCT` Rules

`CONSTRUCT` rules associate a query and a construct term to form a logical unit of transformation. They can also be considered to be *views*, specifying an alternate representation of data that matches the query.

The rule from Example 2.2.4 produces only maximally one data term, due to it's use of `all`. Rules however can produce more than one result. For instance if the construct term of a rule does not contain any `all` constructs at all, it will produce as many results as bindings returned by the query.

2.2.2.4 `GOAL` Rules

A `GOAL` rule is a special rules that is associated with an output resource. Instead of making its results available for querying by other rules, a `GOAL` writes its result to the output resource, declared using the `out` and `resource` language constructs, analogous to the resource specification of queries. If the output resource is not explicitly specified, the result is written to the standard output. Additionally, the format of the output can also be selected to be either XML or Xcerpt's own syntax. Example 2.2.6 shows an `GOAL` rule who's result will be written in to the file `result.xml` as well-formed XML.

Example 2.2.6 A `GOAL` Rule with explicit Output Resource

```
GOAL
  out{
    resource{ "file:result.xml", "xml" },
    construct term
  }
FROM
  query
END
```

2.2.3 Rule Chaining

One of the most important aspects of Xcerpt is *rule chaining*, a unique feature that can only be found in Xcerpt and not in any of the major XML query languages, like XSLT 1.0 and XQuery 1.0. The basic idea of rule chaining is to allow rules to query not only explicit specified resources but the results of other rules. In Xcerpt, any rule who's query is not associated with an external resource is assumed to be querying the results of other rules or itself. The most notable benefits of rule chaining are:

Separation of Concerns One of the key principles when designing software applications is the *separation of concerns* [25], e.g. separating program logic from presentation, so that only small parts of a program have to be rewritten, if the requirements

change. While such separation is difficult in XSLT and XQuery, it is straightforward using multiple rules in Xcerpt: one or more rules specify how to extract and process data, while another set of rules constructs the presentation.

Mediation Complex programs often need to query information from multiple sources, each with it's own structure, and integrate the results. Using rule chaining as mediator, a developer can write rules to transform each into a common structure, which can then be queried, without the need to know about the specific representations of each source.

Recursion Not only can Xcerpt rules query the results of other rules, they can also qurey their own result recursively, allowing solutions to complex problems to be expressed in a concise and compact manner.

In general, rule languages commonly employ one of two different strategies to determine the order of rule chaining. *Forward Chaining* starts with a complete set of data and iteratively applies rules, evolving the data until saturation is achived and the data does not change any more. Then, the goal is evaluated against the saturated data. Forward Chaining is predominantly used in deductive databases. *Backward-Chaining* on the other hand, begins with the goal, searching for rules that might produce output, which could possibly match the goal's query. Further rules are then iteratively searched, until real data is met. Xcerpt uses backward-chaining for rule chaining, making it possible to query the entire Web, where it is infeasible to start with the complete set of data.

Consider the program of Example 2.2.7 for a rough, illustration of backward-chaining. The evaluation starts with examining the body of the `GOAL`-rule (①), establishing that the program is seeking data terms of the form `f{...}`. The evaluation then searches for rule heads that produce data terms that could potentially be matched by `f{{}}`, finding the `CONSTRUCT`-rule ② . The same procedure is repeated, reaching the rule ③ , which contains only a variable in it's head and thus might produce anything. The body of ③ is associated with an external resource, therefore ending the backward-chaining. A set of constraints collected during the backward-chaining are used to check whether the data term contained in `chaindemo.xml` matches the requirements of the rules on the path from the goal. Assuming `chaindemo.xml` contains the single data term `g{"foo"}`, the program outputs the data term `results{f{"foo"}}` as result.

Xcerpt requires programs to be range restricted and stratisfiable, in order for it's rule chaining to function properly (cf. [9], Chapter 6)

Example 2.2.7 Rule Chaining

```
GOAL ①
  results{ all var R }
FROM
  var R as f{{}}
END

CONSTRUCT ②
  f{ var X }
FROM
  g{{ var X }}
END

CONSTRUCT ③
  var ROOT
FROM
  in{
    resource{ "file:chaindemo.xml" },
    var ROOT
  }
END
```

CHAPTER THREE

Mission Statement

The REWERSE[1] Project I4 working group has recently published a paper [26] arguing that a new generation of *versatile* query languages for the Web, able to cope with both Web and Semantic Web data expressed in any web markup language, is essential for realizing the Semantic Web vision. The paper proposed a collection of design principles for such query languages including:

- Single Query Language for both, the Standard and the Semantic Web
- Integrated View of Standard and Semantic Web Data
- Pattern-based, Incomplete Queries
- Answers as Arbirary XML Data
- Declarative and Rule-based, with support for Chaining and Recursion
- Reasoning Capabilities

It is the conviction of this thesis' author that Xcerpt as introduced in Section 2.2 is a viable candidate for such a versatile Web query language. The language already reflects many of the design principles when querying XML. This thesis is an attempt at querying RDF using the XML query language Xcerpt, demonstrating the language's versatility and suitability as an universal query language for the Semantic Web. To achive this goal, the following objectives have been pursued in regards to processing and querying RDF data with Xcerpt:

Support for Multiple Representations of RDF Data. RDF data can be viewed in two ways: either as a flat set of triples or as a highly structured graph. Though both representations are equal in their expressiveness, the chosen representation greatly

[1] Reasoning on the Web with Rules and Semantics, a research "Network of Excellence" funded by the EU Commission and Switzerland, `http://rewerse.net/`

influences how queries can be formulated against RDF data. Chapter 2, "Preliminaries" reveals that both representation are equally popular for both storage and querying of RDF data.

Both representations are provided to users, using views, so that they can choose freely based on personal preference and the nature of the query to be performed.

Uniform Access to XML and RDF Data. A query language supporting both XML and RDF should provide access to both kinds of data in a natural manner. Xcerpt data terms are a representation of XML. This thesis proposes a representation of RDF data as Xcerpt data terms, with the aim to provide a natural form that can be queried intuitively while retaining the structure of RDF data as much as possible.

Independence from Serialization Format. Though RDF/XML is the standard format for storage and interchange of RDF data, Section 4.1 demonstrates that many different kinds of formats have been proposed by the Semantic Web community and are in actual use on the Web. Multiple serialization formats are transparently supported, demonstrating the ability to support arbitrary XML-based serialization formats using only the facilities provided by the language Xcerpt.

Enabling Inferencing. Even though Inferencing is one of the main focuses of the Semantic Web and RDF, Section 4.2 shows that only a small number of query languages actually have support for it. Even of those supporting some form of inferencing, most only do so via extension functions that are implemented using the underlying programming language, limiting themselves to predetermined sets of inference rules. Only a very small number actually support formulation of custom inference rules using facilities of the language itself. The thesis demonstrates, using the inference rules of RDF Schema, how inferencing can be flexibly implemented in Xcerpt. The implementation of RDF Schema inferencing in Xcerpt is

- implemented independently from concrete queries, so that queries need not concern themselves whether what they query is explicitly asserted information or implied information
- implemented using only language constructs of Xcerpt itself
- a natural translation of the inference rules specified in the RDF Schema specification

Where any deficiencies of Xcerpt have been uncovered, these are either addressed directly by proposing suitable extensions or modifications to the language, or a concise description of the problem is given, in order to encourage subsequent research.

CHAPTER FOUR

State-of-the-Art and Related Work

In this chapter, the state-of-the-art of RDF serialization formats and RDF query languages is presented, giving insight into the perceptions and expectations of the community surrounding RDF toward query languages.

4.1 RDF Serialization Formats

One of the RDF specifications by the W3C is a XML-based serialization format for RDF graphs: RDF/XML. Yet, in the five years after the initial release of RDF in 1999 [27], numerous alternative serialization formats have been proposed. Even though various reasons are given by the authors of the proposals for creating alternative serialization formats, almost all complain about structural weaknesses and incompleteness of RDF/XML. Some are additionally motivated by the need to add features that extend the RDF model.

This sections first summarizes the problems and weaknesses of RDF/XML and then presents an overview of the different types of proposals for alternative formats and the motivations behind their creation.

A separate report [28] written by the author of this thesis describes an extended list of serialization formats for RDF in more detail.

4.1.1 The Problems of RDF/XML

Because of the differences in the underlying data models of RDF and XML, RDF/XML specifies a mapping where resources and properties are both converted to XML elements and nested into each other. Section 2.1.3 introduced the main concept behind

RDF/XML, *striping* [21], together with the basic format and some possible abbreviations.

Authors of RDF/XML documents are free to choose which abbreviations they use when serializing RDF data into RDF/XML, leading to a multitude of RDF/XML serialization "styles". Even more variations are possible, as the aggregation of multiple statements about a single resource into children of a single element are not mandatory. Instead of deeply nesting resources and properties, it is also possible to create a shallow structure with all elements that represent nodes of the RDF graph directly under the root, like shown in Example 4.1.1 .

Example 4.1.1 Very verbose RDF/XML

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:foaf="http://xmlns.com/foaf/0.1/" >
  <rdf:Description rdf:about="http://example.org/bike.jpg">
    <dc:creator rdf:nodeID="blank01" />
  </rdf:Description>
  <rdf:Description rdf:nodeID="blank01">
    <foaf:name>Erika Mustermann</foaf:name>
    <foaf:weblog rdf:resource="http://www.example.org" />
  </rdf:Description>

</rdf:RDF>
```

Example 2.1.1, Example 2.1.2 and Example 4.1.1 all represent the exact same RDF graph. After being processed by a properly implemented RDF/XML parser, there is no difference between them. But from the view point of XML tools, each one is a completely different XML document. Because of this syntactic variability, it is very difficult to use standard XML tools like XML Schema, XPath, XSLT and XQuery with arbitrary RDF/XML documents.

Furtheremore, several other problems have been identified with RDF/XML, which all led to the development of the subsequent serialization formats. The following are among the most notable:

- It is impossible to distinguish an XML element for a node from one for an edge without knowledge about the striping.
- Individual triples are hard to identify.
- Many different things are used to to represent URIs: element names, attribute names and attribute values.
- Valid URIs ending in certain characters can not be used for predicates due to restrictions on characters allowed as element names in XML
- It is impossible to specify a single DTD or XML Schema that validates all RDF/XML documents.
- It is difficult to read and write for humans.

In 2001, the RDF Core Working Group was formed to fix inconsistencies in the RDF/XML syntax and clean up the whole RDF specification. This effort [29] led to the revised RDF/XML Syntax Specification [10] in early 2004, which is the current W3C standard for serialization of RDF graphs. However, though the specification has undergone major reorganization and the syntax is now specified in a cleaner and much more concise manner, the basic concepts remain unchanged and the problems arising from RDF/XML's structure have not been solved.

4.1.2 Graph-based XML formats

4.1.2.1 Unstriped Syntax

Only few month after the publication of the original RDF specification, Tim Berners-Lee started experimenting [30] with simplifications of RDF/XML. He considered a modification of RDF/XML without the alternation of node as edges, naming it "Unstriped Syntax". In it, XML elements are only used for the edges in the RDF graph, with the subjects specified using the newly introduced `rdf:for` attribute.

```
<dc:title rdf:for="http://www.example.org/bike.jpg">New Bicycle</dc:title>
```

Alternately, a default subject for all nested elements can be be given using a `rdf:about` attribute on the parent element, similar to RDF/XML. Blank nodes and deep nesting of elements are handled in the same ways in RDF/XML.

Example 4.1.2 Example 2.1.1 serialized using the Unstriped Syntax

```
<someelement rdf:about="http://www.example.org/bike.jpg">
  <dc:creator>
    <foaf:name>Erika Mustermann</foaf:name>
    <foaf:weblog rdf:about="http://www.example.org" />
  </dc:creator>
</someelement>
```

However, lacking a suitable parent element (`somelement` in the above example), use of the `rdf:Description` element is recommended, undermining the Unstriped Syntax's basic principle that elements are only to be used for edges. Because it addressed only the striping issue and none of the other problems with RDF/XML, the Unstriped Syntax has not been pursued further than it's "strawman draft" status.

4.1.2.2 RxML

RxML [31] is a serialization format by Adam Souzis developed as component of his Rx4RDF implementation suit of RDF-related technologies. It is unique in it's consistent use of XML element names to encode all URIs, extending the way properties are handled in RDF/XML to subjects and objects. Each child of the root `rx:rx` element

describes a resource. It's children are in turn the properties and the grandchildren are objects: either text children for literals or empty elements for resources. Nesting is not allowed, limiting the maximum depth of the XML tree to three levels. Blank nodes are represented as resources who's URIs begin with '`bnode:`'.

Example 4.1.3 Example 2.1.1 serialized using RxML

```
<rx:rx xmlns:rx='http://rx4rdf.sf.net/ns/rxml#'
        xmlns:bnode='bnode:'
        xmlns:ex='http://www.example.org/'
        xmlns:dc='http://purl.org/dc/elements/1.1/'
        xmlns:foaf='http://xmlns.com/foaf/0.1/' >
  <ex:bike.jpg>
    <dc:creator><bnode:1 /></dc:creator>
  </ex:bike.jpg>
  <bnode:1>
    <foaf:name>Erika Mustermann</foaf:name>
  </bnode:1>
</rx:rx>
```

While the consistent use of XML element names for URIs seems an elegant solution, it is accompanied by one fatal problem: some URIs can't be expressed due to restrictions in the characters allowed for element names in XML. The URI of Erika's weblog (`http://www.example.org/`) can't be turned into a RxML-style XML element name because of the trailing '/' and therefore had to be omitted in Example 4.1.3.

4.1.3 Plain-Text Formats

4.1.3.1 Notation 3

Based on a pseudo-syntax people started using in various discussion forums instead of RDF/XML, Tim Berners-Lee proposed Notation 3, also known as N3 [32]. Contrary to other serialization format proposals, N3 is not a XML based format. It focuses on the triples that make up a RDF graph and writes them down in a straight forward manner, listing subject, property and object in order. Triples are terminated with a period.

```
<http://example.org/bike.jpg> <http://xmlns.org/foaf/0.1/depicts>
<http://xmlns.com/wordnet/1.6/Mountain_bike> .
```

Two shortcuts are provided to combine several statements: a semicolon introduces another property about the same subject and a comma introduces another object with the same property and subject. Blank nodes are identified by using square brackets as objects, putting the statements about that blank node inside the brackets. Additionally, N3 allows the use of short prefixes to abbreviate long URIs. Also the very common `rdf:type` predicate can be abbreviated to just an 'a'.

N3 is also not just an serialization format for RDF graphs. It has additional support beyond RDF, allowing multiple triples to be quoted as well as formulation of rules and

Example 4.1.4 Example 2.1.1 serialized using N3

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dc:   <http://purl.org/dc/elements/1.1/> .
@prefix foaf: <http://xmlns.org/foaf/0.1/> .

<http://www.example.org/bike.jpg> dc:creator [
      a foaf:Person
      foaf:name "Erika Mustermann";
      foaf:weblog <http://www.example.org/>

].
```

queries using variables and quantification. While most people only think of N3 as a serialization format, some people think of N3 as a rule language, while others consider it a query language. To avoid confusion, attempts have been made to define subsets of N3 according to capability: N3 RDF, N3 Rules and full N3.

Being easy to read for both humans and machines, N3 has been quickly adopted by the Semantic Web community as the format used for online discussions about RDF. Today, various tools and query language implementations for RDF accept some subset of N3 as input and output format in addition to RDF/XML.

4.1.3.2 Turtle

While more and more RDF-related tools adopted N3 in addition to RDF/XML, most of them implemented only a ad-hoc subset of N3, leaving out some of the more complex features that extend beyond the RDF data model. In light of such development, Dave Beckett proposed in the end of 2003 a new plain-text format Turtle [33], including only the commonly used and well understood features of N3, that are within the bounds of RDF model.

Among the features taken from N3 are short prefixes for long URIs and the abbreviations using commas and semicolons, as well as blank node creation using square brackets and collections.

Turtle is most likely to become the properly specified successor of N3 and is quickly gaining popularity as *the* plain-text serialization format for RDF graphs.

4.1.3.3 Quads

When aggregating RDF statements from multiple sources and saving them locally, tracking the origin of each statement becomes more and more important. Some storage systems store the URI of origin together with each triple, forming a "quad". Quads [34] is an extension of N3, adding an optional forth element for such provenance information.

```
<http://example.org/bike.jpg> foaf:depicts wordnet:Mountain_Bike
<http://www.example.org/pics.rdf> .
```

asserts the origin of the triple as `http://www.example.org/pics.rdf`.

4.1.3.4 TriG

TriG [35] is another plain-text format descending from N3. Described as "a compact and readable alternative to the XML-based TriX", TriG extends Turtle beyond the RDF model by adding support for serializing multiple graphs in one file, with the ability to give each a distinct name.

Example 4.1.5 Example 2.1.1 serialized and named using TriG

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dc:   <http://purl.org/dc/elements/1.1/> .
@prefix foaf: <http://xmlns.org/foaf/0.1/> .

<http://www.example.org/pics.rdf> {
   <http://www.example.org/bike.jpg> dc:creator [
         a foaf:Person
         foaf:name "Erika Mustermann";
         foaf:weblog <http://www.example.org/>
   ].
}
```

Example 4.1.5 shows the same graph as in the previous examples, naming it `http://www.example.org/pics.rdf`. The triples of the graph are grouped together using curly braces, with the graph's name prepended.

4.1.4 Triple-based XML formats

4.1.4.1 TriX

TriX [36] has been designed by Jeremy J. Carroll and Patrick Stickler as the XML-based answer to N3. TriX goes beyond the original RDF model by supporting literals as subjects and *Named Graphs* [37] .

A TriX document contains one or more graphs, each optionally with a name. Each graph consists of one or more triples. The `triple` element is the core of TriX, containing three children. The position of each child determines whether the child is the subject, the property or the object of the triple. The child elements name identifies its type.The `uri` element is used for unabbreviated URIs, the `id` element is used for identifying blank nodes, `plainLiteral` is used for string literals and `typedLiteral` is used for any other type of literal in combination with a `datatype` attribute.

Example 4.1.6 Example 2.1.1 serialized in TriX

```
<TriX xmlns="http://www.w3.org/2004/03/trix/trix-1/">
  <graph>
    <uri>http://www.example.org/pics.rdf</uri>
    <triple>
      <uri>http://www.example.org/bike.jpg</uri>
      <uri>http://xmlns.org/dc/elements/1.1/creator</uri>
      <id>blank1</id>
    </triple>
    <triple>
      <id>blank1</id>
      <uri>http://xmlns.org/foaf/0.1/name</uri>
      <plainLiteral>Erika Mustermann</plainLiteral>
    </triple>
    <triple>
      <id>blank1</id>
      <uri>http://xmlns.org/foaf/0.1/weblog</uri>
      <uri>http://www.example.org/</uri>
    </triple>
  </graph>
</TriX>
```

Example 4.1.6 demonstrates the example RDF graph serialized using the basic TriX format. The name `http://www.example.org/pics.rdf` is attached to it via the `uri` element directly under `graph`. Though being very verbose, individual triples are clearly identifiable.

Specifying an absolutely minimal base format without any abbreviations, TriX takes a unique approach by allowing syntactic extensions that make the syntax more human-friendly using a XML transformation language, XSLT in the case of TriX. One popular trick to increase readability of RDF serializations is to allow a XML QName-like abbreviation for URIs. By declaring an appropriate stylesheet processing instruction, TriX allows such syntactic sugar. Example 4.1.7 shows a triples serialized in TriX, using the `qname` element to abbreviate a long URIs. Other syntactic sugars demonstrated by the authors of TriX include the use of `xml:base` as another method for URI-abbreviation, tags for specific typed literals and collections. The authors even go as far as suggesting RDF/XML as an TriX extension, based on the possibility of writing an RDF/XML parser in XSLT.

Example 4.1.7 Syntactic Extensions in TriX using XSLT

```
<?xml-stylesheet type="text/xsl"
      href="http://www.w3.org/2004/03/trix/all.xsl" ?>

<TriX xmlns="http://www.w3.org/2004/03/trix/trix-1/"
      xmlns:foaf="http://purl.org/dc/elements/1.1/" >
  <graph>
    <triple>
      <uri>http://www.example.org/bike.jpg</uri>
      <qname>dc:creator</uri>
      <id>blank1</id>
    </triple>
  </graph>
</TriX>
```

4.1.4.2 RXR

RXR (Regular XML RDF) [38] is another Triple-based XML format, that strictly limits itself to the RDF data model. It has been proposed by Dave Beckett as an reaction to TriX's support for features beyond the original RDF model, it's dependency on XSLT and the resulting risk of ad-hoc extensions.

The `triple` element, containing three children, is at the heart of RXR, similar to TriX. But, instead of relying on the position, the children's roles are unambiguously identified by the elements `subject`, `predicate` and `object`. URIs are then given as value to the `uri` attribute, while literals are given as element content, with an optional `datatype` attribute. Blank nodes are specified with the `blank` attribute.

Example 4.1.8 Example 2.1.1 serialized in RXR

```
<graph xmlns="http://ilrt.org/discovery/2004/03/rxr/">
  <triple>
    <subject uri="http://www.example.org/bike.jpg" />
    <predicate uri="http://xmlns.org/dc/elements/1.1/creator" />
    <object blank="b1" />
  </triple>
  <triple>
    <subject blank="b1" />
    <predicate uri="http://xmlns.org/foaf/0.1/name" />
    <object>Erika Mustermann</object>
  </triple>
  <triple>
    <subject blank="b1" />
    <predicate uri="http://xmlns.org/foaf/0.1/weblog" />
    <object uri="http://www.example.org/" />
  </triple>
</graph>
```

Example 4.1.8 shows yet again the sample RDF graph, this time serialized using RXR. Despite it's verbosity, the triples are clearly recognizable. RXR does not allow any abbreviations of URIs or other complexities. One notable exception are collections, supported by RXR through the `collection` element. Multiple statements with the same subject and predicate can be aggregated using this facility. Three tripled are contained in Example 4.1.9. The same triples could also have been written separately, using three separate `triple` elements.

4.1.5 Observations

The dissatisfaction of the Semantic Web community with RDF/XML has brought forward numerous proposals for alternate serialization formats, of which the most representative have been described here. After various early attempts at directly mapping components of RDF graphs to XML elements failed to gain support, the current trends seems to go toward triple-listing formats, both XML and non-XML based. Turtle, TriX

Example 4.1.9 Collections in RXR

```
<graph xmlns="http://ilrt.org/discovery/2004/03/rxr/">
  <triple>
    <subject uri="http://example.org/box" />
    <predicate uri="http://example.org/contains" />
    <collection>
      <object>apple</object>
      <object>pear</object>
      <object>potato</object>
    </collection>
  </triple>
</graph>
```

and RXR are the current contestants for such formats, but it is still too early to speculate on which will prevail. However, considering the disputes concerning support for new features like graph naming and literals as subjects, it is likely that the world will see yet more format proposals in the near future. Until some consensus is reached, RDF/XML remains the only formally standardized format all implementations must support. Support for multiple formats and extensibility for supporting future formats is desirable for systems handling with RDF data.

4.2 Querying RDF

With more and more RDF data becoming available on the Web, a large number of RDF query languages have been proposed by researchers and implementers with varying backgrounds. Some attempts have also been made at using XML query languages for querying RDF. This section provides a brief description of a selection of the most representative of the different kinds of RDF query languages.

Curious minds are directed toward a recent report [39] by the REWERSE Project that identifies and evaluates more than 20 languages and formalisms for querying RDF.

4.2.1 Versa

Versa [40] is an RDF query language inspired by XPath, developed as a replacement for XPath when using XSLT to query and transform RDF graphs. The main concept of Versa is to select nodes from RDF graphs navigating through graphs using forward and backward traversals and specifying filters expressions:

Forward traversal. Starting from a list of resources, Versa allows the selection of resources that are reachable via given properties. For example, the Versa expression `all() - dc:creator -> *` selects all resource that are creators of other resources. Traversal targets can also restricted, e.g. to those being certain literals.

The `*` is a wildcard, indicating no restriction. Traversal expressions can also be chained.

Forward filters. The set of resources returned by a subexpression is limited through the use of filters. The Versa expression `all() |- foaf:name -> eq('Erika Mustermann')` selects all resource that are named "Erika Mustermann".

Backward traversal. Traversal in the backward direction from object resources to subject resources is also allowed by reversing the arrow. The selection above can also be written `* <- foaf:name - 'Erika Mustermann'` using backward traversal.

General traversal and filters. While the forward and backward traversal operators allow only the traversal of fixed length paths, the function `traverse` allows recursive traversal of arbitrary length. Similarly, the `filter` function provides a general filter, sequentially narrowing down the set of resources returned by evaluating the first argument using the expressions provided in the remaining arguments.

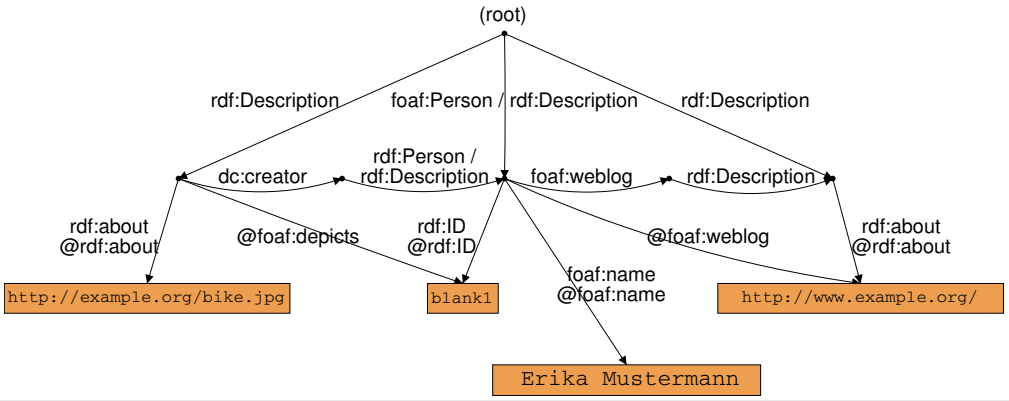
Additionally, a large number of functions are provided for such tasks as list generation and manipulation, boolean operators as well as numerous tests. Reasoning is provided by such functions, e.g. the `type` function that returns a list of resources that are instances of a specific class or one of its subclasses.

4.2.2 TreeHugger

An attempt to use XSLT to query and transform RDF is TreeHugger [41], working on a normal form of RDF graphs. Accessed using the XPath function `th:document()`, the normal form appears to users as an XML-like graph, similar to the RDF graph.

Figure 4.1 visualizes the TreeHugger normal form of the RDF graph depicted in Section 2.1.1.4, with an additional `rdf:type foaf:Person` assertion about the blank node. The basic structure is that of an graph that uses class names and property URIs as element names, informally describable as "RDF/XML graph". All resources of the RDF graph are direct children of the root element, reachable via an edge denoting the resource's class. If a resource is an instance of multiple classes, the edges can be addressed by the name of any of the resource's classes. The resource's URI is given as an `rdf:about` child element and an `rdf:about` attribute (prefixed with an @ in the figure), or an `rdf:ID` child element and attribute in case of a blank node. Each resource's properties are given as child elements named after the property pointing to the object

Figure 4.1 Normal form of TreeHugger



resource via an extra element named after the object's classes. In addition, an attribute named after the property directly points to the object's URI or blank node identifier.

Example 4.2.1 XPath Query against the TreeHugger Normal Form

```
/rdf:Description[dc:creator/foaf:Person/foaf:name/text() = "Erika Mustermann"]/@rdf:about

/rdf:Description[foaf:name/"Erika Mustermann"]/inv:dc:creator/rdf:about
```

Example 4.2.1 shows two XPath expressions that each selects the URIs of resources that have been created by Erika Mustermann when applied against the TreeHugger normal from of RDF graphs. The first expression is a straight forward implementation of the query, the second expression uses the additional `inv` axis provided by TreeHugger, that traverses the inverse of a property.

Inferencing is realized by the additional XPath functions `th:documentRDFS()` and `th:documentOWL()`, each providing access to normal forms that contain the additional edges resulting from the application of a specific, predefined set of inference rules.

4.2.3 Using XQuery for RDF: "The Syntactic Web"

During the development of XQuery 1.0 [7], Jonathan Robie proposed to use XQuery to also process and query RDF, so that a single tool can be used to access both XML and RDF. Calling his approach "The Syntactic Web" [42], Robie proposed to normalize RDF into a specific XML representation before querying, in order to overcome the syntactic variability of RDF/XML that makes process with standard XML tools difficult.

The normal form used as an illustration is similar in spirit to RXR, using `statement` elements with `subject`, `predicate` and `object` child elements. The actual transformation is performed by an external function library implementing the XQuery function `assertions()`, returning the set of known triples. Intended only as a sketch of the

approach, aspects such as blank nodes and typed literals are not covered. Example 4.2.2 shows an XQuery against the normal from, querying for URIs of resources that were created by resources named ”Erika Mustermann”.

Example 4.2.2 Using XQuery for RDF

```
FOR $creator IN assertions()//subject[../predicate="http://xmlns.com/foaf/0.1/"
                                      AND ../object="Erika Mustermann" ],
    $picture IN assertions()//statement[/object=$creator
                                      AND /predicate="http://purl.org/dc/elements/1.1/creator" ],
RETURN $picture/subject
```

Support for inferencing is provided by additional functions implemented in XQuery. For example, the function `rdf:instance-of-class` returns a list of all known resources that are instances of the specified class or any of its subclasses.

4.2.4 TRIPLE

TRIPLE [43] is a rule-based, query and inference language for RDF with a syntax close to F-Logic [44]. Conceptually based on Horn Logics, TRIPLE provides well-defined semantics and fairly powerful reasoning capabilities. Instead of including a pre-defined set of semantics, e.g. those of RDF Schema, TRIPLE allows the implementation of custom semantics in form of extension modules containing inference rules.

A triple, including provenance information, is expressed in TRIPLE using the following general form:

```
subject[predicate -> object]@graph
```

Based on this atomic form, complex logical formulae can be written, including variables that are introduced by quantifiers. TRIPLE supports the full range of logical connectors and quantifiers: $\wedge \vee \neg \forall \exists$

Assuming, that the RDF graph to be queried is identified by the URI `http://example.org/pics.rdf`, the TRIPLE program shown in Example 4.2.3 queries for resources that were created by `foaf:Person` instance resources named `Erika Mustermann`. The program first defines several identifiers that are used to abbreviate long URI references. Then, the actual query appears in form of a single formula. Note how all variables are introduced by quantifiers.

However, the query would fail, if the `rdf:type` property was not explicitly asserted in the graph and only implied by another property, e.g. a `foaf:knows`. Inferencing of `rdfs:domain` and `rdfs:range` can be implemented with the rules given in Example 4.2.4 in form of a module that augments an RDF graph. The first rule “copies” all triples from the graph the module extends, while the latter two implement the inference rules of `rdfs:domain` and `rdfs:range`. With this module, the query above

Example 4.2.3 A TRIPLE Program

```
dc   := 'http://purl.org/dc/elements/1.1/'.
foaf := 'http://xmlns.com/foaf/0.1/'.
rdf  := 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'
targetGraph := 'http://example.org/pics.rdf'.

FORALL S,C result(S) <-
    S[dc:creator -> C[ rdf:type -> foaf:Person;
                       foaf:name -> "Erika Mustermann"
                     ]

     ]@targetGraph.
```

only needs to be modified to query `@rdfschema(targetGraph)` instead of the the original graph that contains only explicitly asserted triples. The author of TRIPLE has demonstrated that the other parts of the RDF Schema semantics can be implemented as easily.

Example 4.2.4 Implementing Inferencing using TRIPLE

```
rdf  := 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'
rdfs := 'http://www.w3.org/200/01/rdf-schema#'

FORALL Mdl @rdfschema(Graph) {
  FORALL S,P,O S[P->O] <-
               S[P->O]@Graph
  FORALL S,T   S[rdf:type -> T] <-
               EXISTS P, O (S[P->O] AND P[rdfs:domain->T]).
  FORALL O,T   O[rdf:type -> T] <-
               EXISTS P, S (S[P->O] AND P[rdfs:range->T]).

}
```

4.2.5 Metalog

Metalog [45] is a query language notably different from other RDF query languages, as it uses a controlled subset of English instead of a completely synthetic language, allowing near-natural language queries. Anybody is supposed to be able to read and understand Metalog program, but writing one requires knowledge about the constructs Metalog understands.

Variables are written in capital letters in Metalog, URI references and literals are written as quoted strings. The list of keywords available includes `represents` for definition of variables, `then`, `imply` and `implies` for expressing logical implications as well as `equals` and `minus` for arithmetic, amongst others.

The Metalog program in Example 4.2.5 starts with defining the variables `DEPICTS` and `MOUNTAINBIKE`, each representing an URI reference. Then a single RDF triple is asserted using an URI reference and the defined variables. Note how the sentences differ slightly, as words that are neither variables, quoted strings or keywords are simply ignored. Lines starting with `comment:` are comments and are also ignored. Finally, a

Example 4.2.5 A Metalog Program

```
comment: variable declarations
DEPICTS represents the verb "depicts" from the vocabulary "http://xmlns.com/foaf/0.1/"
MOUNTAINBIKE represents a "Mountain_bike" from "http://xmlns.com/wordnet/1.6/".

comment: assertion
"http//example.org/bike.jpg" DEPICTS a MOUNTAINBIKE.

comment: query

Which PICTURE DEPICTS a MOUNTAINBIKE?
```

query is issued, asking for pictures that depict mountain bikes. A query in Metalog has the same structure as an assertion, but with a question mark rather than a dot at the end. As the variable `PICTURE` in the query doesn't represent anything, Metalog tries to find something that would make sense as the value for `PICTURE` in the given context.

4.2.6 SPARQL

SPARQL [46] is the query language for RDF currently in development by the RDF Data Access Work Group of the W3C, as an attempt to standardize and improving on a whole family of SQL-like RDF query languages descending from SquishQL [47] and RDQL [48].

To users familiar with SQL and relational databases, the basic principle of SPARQL is best described as each RDF graph being a three-column table with individual triples being the rows of such tables. Queries are specified using a list of triple patterns that when "AND"ed specify the structure of the RDF graph matched by the query. Reasoning capabilities are not part of SPARQL, but can be incorporated if the underlying implementation allows querying of "virtual" graphs that contain inferred triples.

Example 4.2.6 A SPARQL query

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc:   <http://purl.org/dc/elements/1.1/>
SELECT ?madebyerika
FROM <http://www.example.org/pics.rdf>
WHERE (?madebyerica dc:creator ?creator),

      (?creator foaf:name "Erika Mustermann")
```

Example 4.2.6 shows a SPARQL `SELECT` query for selecting the URIs of resources that were created by a resource named "Erika Mustermann". In it, the RDF graph to be queried is specified in the `FROM` part, the triple patterns specifying the actual query follows the `WHERE` keyword. Each component of a triple pattern is either an variable beginning with a question mark, a potentially abbreviated URI reference or a literal value. When multiple graphs are specified in the `FROM` part of the query, the merger of them is queried. If the `FROM` is omitted, the merger of all graphs known to the system is queried.

In addition to `SELECT` queries that return a subset of the variable bindings, SPARQL offers `ASK` queries that return a single boolean value whether the query matches or not and `CONSTRUCT` queries that return either an RDF subgraph that include all matching triples or an RDF graph constructed by substituting variables in a set of triple patterns.

Beware that the description of SPARQL in this section is based on an early working draft from October 2004. Syntax and concepts are expected to still change drastically as the language matures.

4.2.7 Observations

Even tough the query languages and formalisms introduced in this section represent a small selection of many currently available, a great diversity is clearly observable. Many languages approach RDF data as a set of triples, specifying queries as conjunctions of triple patters. Many others prefer the graph form of RDF data and specify queries as graph patterns. Yet others employ a navigational approach, specifying pathes through graphs. The support for inferencing also differs greatly. Some languages consider RDF data to only be a explicit set of data, requiring the underlying storage or retrieval system to provide pre-inferred data. Other languages provide inferencing through extension functions that need to be manually and specifically invoked, while a third class of languages directly incorporate inferencing capabilities, allowing inference rules to be specified using constructs of the language itself. And finally, some languages have been specifically designed for RDF, while others are modifications or extensions of existing XML tools. Some brave attempts have also been made at directly using XML tools for querying RDF.

The standardization work currently ongoing at the W3C will certainly lead to consolidation of the languages up to a certain degree. Still, a single language can only cover a limited set of the diverse philosophies and requirements for querying RDF. It is very likely that more than language will survive the natural selection by the increasing number of users finding more and more usages for RDF. A language that is flexible and powerful enough to accommodate the diverse interests but remains easy enough to understand and learn has yet to see the light of the world.

CHAPTER FIVE

Querying RDF with Xcerpt

RDF data on the Web exists predominantly in form of RDF/XML documents. Even though RDF/XML is an XML format, its syntactic variability (c.f. Section 4.1.1) makes it impractical to directly query RDF/XML as normal XML using XML query languages such as Xcerpt. In addition, use of other serialization formats is increasing. Thus an abstraction from serialization is necessary for RDF data to be uniformly queryable. This chapter proposes a formalism for querying RDF with Xcerpt, consisting of representations of RDF data as Xcerpt data terms and a rule "library" enabling access to RDF data in different serialization formats.

Section 5.1 defines two representations of RDF data as Xcerpt data terms, reflecting the differing views on RDF data: a graph representation and a triple representation. Section 5.2 discusses how provenance information is added to the representations without introducing syntactic overhead for applications not requiring such additional information. A framework for providing the two representations as queryable *views* over concrete serializations is introduced in Section 5.3, examining sets of rules for parsing RXR and RDF/XML in detail, serving as examples for supporting arbitrary serialization formats. Using this framework, users need only specify XML documents containing RDF data and query one of the two representations, not concerning themselves with the actual serialization used by the documents. Finally, the capabilities of Xcerpt and the proposed formalism are evaluated in Section 5.4 against a list of requirements proposed by the W3C for RDF query languages.

5.1 Two Representations: Triples and Graphs

RDF data can be seen in at least two ways: either as a set of triples or as a structured graph. Chapter 4, "State-of-the-Art and Related Work" shows that both representations enjoy similar popularity. Tough both representations are equal in their expressiveness,

the choice of representation greatly influences how queries are formulated. Queries against a triple representations need to be expressed in a relational manner, as conjunctions of structurally simple triple patterns. Queries against a graph representation on the other hand are expressed as highly structured graph patterns. Depending on the nature of a particular query, one or the other style might be more suited.

This section examines the difference between the RDF data model and that of Xcerpt data terms. Based on the findings, two mappings of RDF data to Xcerpt data terms are proposed: a triple representation and a graph representation, providing users with a choice between representations.

5.1.1 Graph Representation

Even tough Xcerpt data terms are graph structured, a direct representation of RDF graphs as Xcerpt data terms is not possible, as graphs induced by Xcerpt data terms (Xcerpt graphs) are rooted, node-labeled graphs, while RDF graphs have labeled edges as well as nodes and no root node. The structure and characters allowed for the labels are also different. This section presents a mapping that mirrors the structure of RDF graphs as closely as possible and enable intuitive query formulation.

5.1.1.1 Resource nodes

The most significant difference between RDF graphs and Xcerpt graphs is how nodes are labeled. The nodes of an Xcerpt graph are labeled with a namespace and a label, strongly hinting at Xcerpt's roots in XML. While the namespace of a node can be an arbitrary string, only a limited range of characters are allowed for the label, a restriction inherited from element names and namespaces in XML [49]. In contrast, the resource nodes of an RDF graph are either identified by an URI reference or are blank nodes without permanent identifiers.

Fortunately, the difference is small enough that an almost straight-forward mapping is possible, as nodes in both types of graphs are labeled with two components: namespace and label for Xcerpt data terms, type (URI reference or blank) and identifier for resource nodes in RDF. An intuitive mapping would be to indicate a node's type using the namespace and its identifier using the label. However, as XML tags are fairly restricted in the characters they permit, URIs can not be used directly as labels of data terms. Tough relaxion of the restriction is being discussed by the author's of Xcerpt, a decision has not yet been made. Therefore, node identifiers are mapped to namespaces and the node's type is indicated through the label. The labels used are `uri` for nodes with URI references and `blank` for blank nodes.

A resource node identified by the URI reference `http://example.org/foo` is thus mapped to an Xcerpt data term with the namespace and label pair

```
"http://example.org/foo":uri
```

and a blank node with the blank node identifier `b1` is mapped accordingly to

```
"b1":blank
```

5.1.1.2 Literal Nodes

Literals in Xcerpt are plain strings. Literals in RDF on the other hand have a literal value and optionally a language or datatype specification. A direct mapping to Xcerpt literals would be most convenient for querying, but is only possible if the additional properties are disregarded. Therefore, literals are mapped to structured data terms, indicating the type through the label `literal`. The namespace remains empty, while the literal value becomes the data term's only child term. Language and datatype are preserved as attributes of the data term. The URI references for `xml:lang` and `rdf:datatype` are used as attributes names, respectively.

For example, the plain literal "Hello World" is simply mapped to

```
literal{ "Hello World" }
```

and the typed literal "`42`" with the datatype `int` from the XML Schema datatypes is mapped to

```
literal{
  attributes{
    "http://www.w3.org/1999/02/22-rdf-syntax-ns#datatype":uri{
      "http://www.w3.org/2001/XMLSchema#int"
    }
  },
  "42"
}
```

In case of XML literals, the child directly represents the XML tree in form of an Xcerpt data term. For example, if an XML literal was an XHTML table, it would be mapped to:

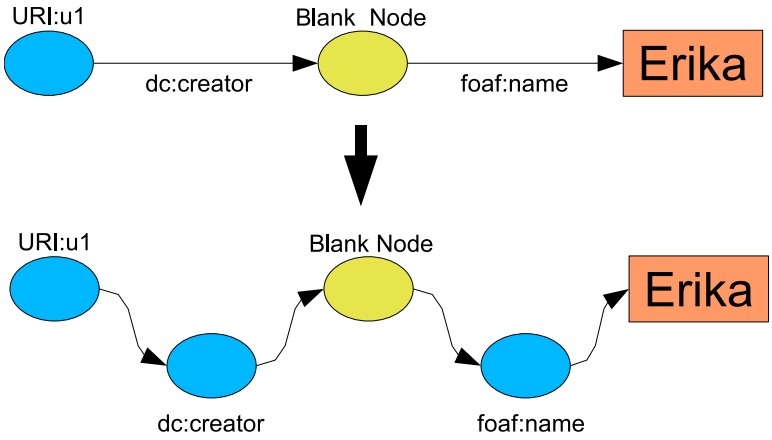
```
literal{
  ns-default = "http://www.w3.org/1999/xhtml"

  attributes{
    "http://www.w3.org/1999/02/22-rdf-syntax-ns#:datatype":uri{
      "http://www.w3.org/1999/02/22-rdf-syntax-ns#XMLLiteral"
    }
  },
  table{
    tr{ ... }
  }
}
```

5.1.1.3 Labeled Edges

Another large structural difference between Xcerpt graphs and RDF graphs is the handling of edges. Xcerpt graphs have unlabeled edges implicitly connecting parent and child terms. The edges of RDF graphs on the contrary are labeled, corresponding to the predicates of the statements that induce the graph.

A mapping is established by transforming each labeled edge into one labeled node and two unlabeled edges. One edges connects the original edge's source to the newly created node and another connects the newly created node to the original edge's target. The node inherits the label of the original edge. Figure 5.1 visualizes the conversion of the labeled edges of a simple RDF graph into nodes and unlabeled edges.

Figure 5.1 conversion of labeled edges to labeled nodes



The newly created nodes are mapped to Xcerpt data terms according to the node mapping previously defined. Example 5.1.1 shows the mapping applied to the RDF graph depicted in Figure 5.1. Note that the mapping at this point has a "striped" structure, making it difficult to determine whether a data term corresponds to a node or an edge from the original RDF graph. The issue is addressed in the next section.

Example 5.1.1 Figure 5.1 mapped to Xcerpt data term

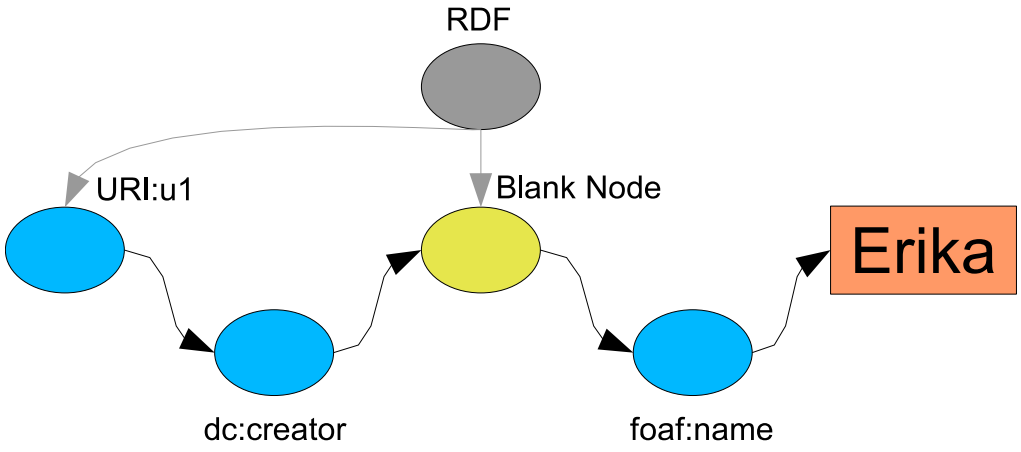
```
"u1":uri{
  "http://purl.org/dc/elements/1.1/creator":uri{
    "b1":blank{
      "http://xmlns.com/foaf/0.1/name":uri{
        literal{ "Erika Mustermann" }
      }
    }
  }
}
```

5.1.1.4 Root Node

The third structural difference between RDF graphs and Xcerpt graphs is the existence of *root nodes*. While RDF graphs might contain multiple disjoint subgraphs, Xcerpt graphs are always connected, having a designated node from which there is a path to each and every node of the graph.

Instead of searching for a node that fulfills the criteria of a root node,which might not exist, e.g. if the RDF graph is disjoint, an artificial root node labeled `RDF` is added to the graph together with edges from the root node to each node representing a resource. Figure 5.2 illustrates the added root node and the edges from it.

Figure 5.2 adding a root node



The final mapping of RDF graphs to Xcerpt data terms is shown in Example 5.1.2. Note that references needed to be used in order to express the graph structure. As each resource's identifier is unique within an RDF graph, they have also been used as identifiers for Xcerpt's reference mechanism. Note also how the issue with "striping" has disappeared as the data terms representing URI and blank nodes from the original RDF graph are clearly identifiable as to be those being direct children of the root node.

Example 5.1.2 Figure 5.2 mapped to a Xcerpt data term

```
RDF{
  u1 @ "u1":uri{
    "http://purl.org/dc/elements/1.1/creator":uri{
      ^u1
    }
  },
  b1 @ "b1":blank{
    "http://xmlns.com/foaf/0.1/name":uri{
      literal{ "Erika Mustermann" }
    }
  }
}
```

5.1.1.5 Querying the Graph Representation

Queries against the graph representation are expressed in form of graph patterns. Example 5.1.3 shows a simple example of such a query, selecting the URIs of identified resources created by other resources named "Erika Mustermann" into the variable `URI`. Example 5.1.4 shows a slightly more complex query, this time selecting resources "Erika Mustermann" is interested in and optionally the names of any people she knows who share the same interest, using the terms `foaf:interest` and `foaf:knows`. As Xcerpt currently does not have support for abbreviating long URIs, all URIs must be written in their entirety, making the queries verbose and decreasing their readability.

Example 5.1.3 A simple Query against the Graph Representation

```
CONSTRUCT
...
FROM
  RDF{{
    var URI:uri{{
      "http://purl.org/dc/elements/1.1/creator":uri{
        /.*/:/.*/{{
          "http://xmlns.com/foaf/0.1/name":uri{
            literal{ "Erika Mustermann" }
          }
        }}
        }
    }}
  }}
END
```

Example 5.1.4 Another Query against the Graph Representation

```
CONSTRUCT
...
FROM
  RDF{{
    /.*/:/.*/{{
      "http://xmlns.com/foaf/0.1/name":uri{
        literal{ "Erika Mustermann" }
      },
      "http://xmlns.com/foaf/0.1/interest":uri{ var INTEREST },
      optional "http://xmlns.com/foaf/0.1/knows":uri{
        /.*/:/.*/{{
          "http://xmlns.com/foaf/0.1/name":uri{
            literal{ var FRIEND }
          },
          "http://xmlns.com/foaf/0.1/interest":uri{ var INTEREST }
        }}
      }
    }}
  }}
END
```

More examples of queries against the graph representation are demonstrated in Chapter 7, "Use Cases".

5.1.2 Triple Representation

The other popular approach to RDF is to view an RDF graph simply as the set of it's triples. The inference rules of RDF Schema are also defined in terms of triples. Therefore it is desirable to provide users also a triple-based view of their RDF data, which they can use to formulate queries similar to those of the triple-based query languages presented in the previous chapter.

5.1.2.1 Triples as Data Terms

A triple is a data structure with three elements: a subject, a predicate and an object. Such a structure can be intuitively expressed using an Xcerpt data term with three children:

```
triple[ subject, predicate, object ]
```

This structure is similar to the way triples are handled in triple-centric, XML-based RDF serialization formats, such as RXR and TriX. Accordingly, there are two possibilities how to mark each child's role: either implicitly through the ordering of the children, similar to TriX, or by explicitly labeling the children of an unordered term, in the same manner as RXR.

For reasons of consistency, we choose to represent each component in the same way as in the graph representation, using an ordered term labeled `RDF-TRIPLE` to hold them. A triple whose elements are all URIs would thus look like:

```
RDF-TRIPLE[ "subj_uri":uri{}, "pred_uri":uri{}, "obj_uri":uri{} ]
```

5.1.2.2 Querying the Triple Representation

Queries against the triple representation are expressed as conjunctions over multiple triple patterns, using variables to *join* the triple patterns. Example 5.1.5 shows such a query. It is very similar to queries in other, triple-based query languages. The query is comparably slow, due to the overhead of joining the results of the individual triple patterns. Again, more examples of queries against the triple representations are demonstrated in Chapter 7, "Use Cases".

Example 5.1.5 A Query against the Triple Representation

```
CONSTRUCT
...
FROM
  and{
    RDF-TRIPLE[ var SUBJECT:uri{}, "http://purl.org/dc/elements/1.1/creator":uri{}, var X ],
    RDF-TRIPLE[ var X, "http://xmlns.com/foaf/0.1/name":uri{}, var Z ]
  }
END
```

5.2 Multiple RDF Graphs and Provenance Information

RDF allows anyone to make statements about any resource. In a very simple view, the Semantic Web is a collection of knowledge, stored in documents retrievable over the Internet, each holding an RDF graph. The *RDF Semantics* [12] specification defines how multiple RDF graphs are to be merged, in order to integrate heterogeneous knowledge into an all-embracing gigantic knowledge base. However, practice has shown that the ability to track the origin of triples is essential when dealing with knowledge from heterogeneous sources. Such provenance information is necessary, amongst other things, for the management of trust and conflict resolution.

5.2.1 Provenance Information of Triples

In the triple representation of RDF data as Xcerpt data terms, provenance information is attached in form of `origin` attributes to the `RDF-TRIPLE` data terms.

Query terms without any specification of attributes default to `attributes{{}}` (any attributes) in Xcerpt. The query

```
CONSTRUCT
...
FROM
  RDF-TRIPLE[ var S, var P, var O ]
END
```

thus matches all triples known to the program, regardless of their origin. Should the query be restricted to triples of a specific origin, then the origin is simply specified:

```
CONSTRUCT
...
FROM
  RDF-TRIPLE[
    attributes{ origin{ "http://example.org/sample.rdf" } },
    var S, var P, var O
  ]
END
```

If a conjunction needs to be restricted to triples from a single RDF graph, a variable is specified for the origin, forcing all matching triples to have the same origin. A query for any two resources that have a transitive relationship in a single RDF graph would thus be:

```
CONSTRUCT
...
FROM
  and{
    RDF-TRIPLE[ var X, var Y, var Z, attributes{ origin{ var ORIGIN } } ],
    RDF-TRIPLE[ var Z, var Y, var Z, attributes{ origin{ var ORIGIN } } ]
  }
END
```

5.2.2 Provenance Information of Graphs

The origin of an RDF graph is also attached as an `origin` attribute on the `RDF` root data term that holds the graph. A query against `RDF` data terms without specifying any `origin` attribute queries for a single graph that matches the given pattern. By specifying an origin, the query is restricted to a specific graph.

Unlike the triple representation however, querying the combined knowledge from all graphs known to the systems is more laborious, as the separate graphs can not be queried at once with a single graph pattern. For example, if the two triples

```
RDF-TRIPLE[ "http://fakeroot.net/person/Oliver":uri{},
            "http://xmlns.com/foaf/0.1/name":uri{},
            literal{ "Oliver M. Bolzer" } ]
```

and

```
RDF-TRIPLE[ "http://fakeroot.net/person/Oliver":uri{},
            "http://xmlns.com/foaf/0.1/weblog":uri{},
            "http://slashdot.jp/%7EOliver/journal":uri{} ]
```

came from two different origins, a graph pattern querying for URIs of weblogs of person(s) named "Oliver" would not match them:

```
GOAL
  blogs{
   all blog{ var BLOGURI }
  }
FROM
  RDF{{
    /.*/:/.*/{{
      "http://xmlns.com/foaf/0.1/name":uri{
        literal{ /Oliver .*/ }
      },
      "http://xmlns.com/foaf/0.1/weblog":uri{
        var BLOGURI:uri{{}}
      }
    }}
  }}
END
```

Instead, the query needs a tedious conjunction, quite similar to the way queries formulated against the triple representation.

```
GOAL
...
FROM
  and{
    RDF{{
      var XI:varXT{{
        "http://xmlns.com/foaf/0.1/name":uri{
          literal{ /Oliver/ }
        }
      }}
    }},
    RDF{{
      var XI:var XT{{
        "http://xmlns.com/foaf/0.1/weblog":uri{
```

```
          var BLOGURI:uri{{}}
        }
      }}
    }}
  }
END
```

Requiring such conjunctions obliterates the benefits of the graph representation over the triple representation. Therefore, an additional `RDF` term constituting the union of all RDF graphs is made available by a variation of the rules defined in Section 5.3.4. It's `origin` attributes has the value `*`. The above query can thus be written as:

```
GOAL
  blogs{
   all blog{ var BLOGURI }
  }
FROM
  RDF{{
    attributes{ origin{ "*" } },
    /.*/:/.*/{{
      "http://xmlns.com/foaf/0.1/name":uri{
        literal{ /Oliver/ }
      },
      "http://xmlns.com/foaf/0.1/weblog":uri{
        var BLOGURI:uri{{}}
      }
    }}
  }}
END
```

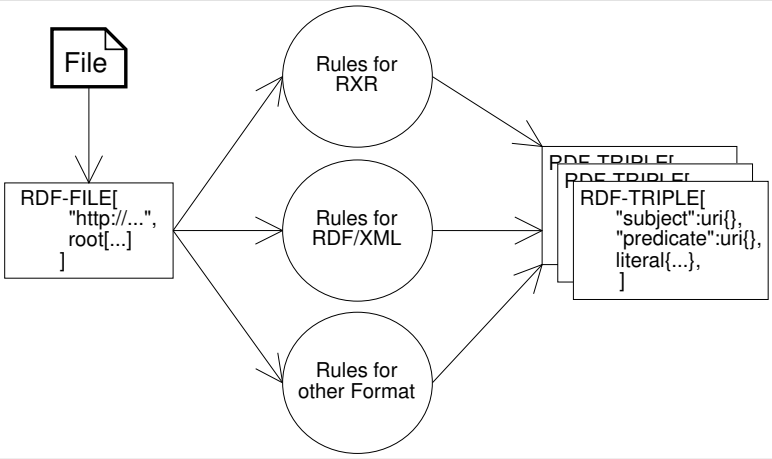
5.3 Supporting Arbitrary Serialization Formats

As summarized in Chapter 4, "State-of-the-Art and Related Work", RDF data on the Internet is stored using numerous serialization formats. Any system for querying RDF data should at least be able accept input in the the most common and only standardized format: RDF/XML. Support for other formats is desirable, as their popularity is rapidly increasing. Nevertheless, it is still not clear which format will prevail, so implementations should also support easy extension of the formats they support. Using Xcerpt, adaption to new XML-based formats is easily possible by writing rules that extract the triples out of XML-files in that new format.

With support for multiple formats, users should only need to instruct the system to load specific files and not concern themselves with the actual format the file is in. This section proposes a framework to do so. First, the contents of XML files containing RDF data is wrapped in `RDF-FILE` data terms, containing the URI of the file. Then, rules for specific serialization formats transform the documents' data terms into `RDF-TRIPLE` data terms retaining the provenance information. Figure 5.3 illustrates these steps.

As an example to writing rules for arbitrary XML-based serialization formats, two sets of rules are presented in this section: one for RDF/XML, a tree-based format requiring

Figure 5.3 Support for Arbitrary Formats



structural recursion in its parsing, and another for RXR, a much simpler triple-listing format.

5.3.1 Loading files

An XML document containing RDF data is loaded into the system using a rule based on the following template:

```
CONSTRUCT
  RDF-FILE[ "URL_of_XML_file_with_RDF_data", var Root ]
FROM
  in{
   resource{ "URL_of_XML_file_with_RDF_data", "xml" },
   var Root
  }
END
```

In this rule, the root of the document is wrapped in an `RDF-FILE` data term, together with the URI of the document. This wrapping marks the contents of the document to be interpreted as RDF data and retains provenance information, as Xcerpt does not automatically keep track about the origin of a data term. The rules that actually parse the content of the file can then use the provided URI for provenance information, unless the serialization format includes explicit provenance information.

5.3.2 Rules for RXR

Recalling the summary from Section 4.1.4.2, the triples of an RDF graph are listed in an RXR file in the form:

```
<rxr:triple>
  <rxr:subject uri="subject_uri" />
```

```
  <rxr:predicate uri="predicate_uri" />
  <rxr:object uri="object_uri" />
</rxr:triple>
```

As the triples are easily recognizable, the Xcerpt rules for extracting them are also fairly simple. Provenance information is directly carried forward from the `RDF-FILE` data term that wraps the RXR document root term. The triples are extracted using a rule that mimics the general structure of triples in RXR.

```
CONSTRUCT
  RDF-TRIPLE[
    attributes{ origin{ var ORIGIN } },
    var SUBJECT, var PREDICATE:uri{}, var OBJECT
  ]
FROM
  ns-prefix rxr = "http://ilrt.org/discovery/2004/03/rxr/"

  and[
    RDF-FILE[ var ORIGIN,
      rxr:graph {{
        rxr:triple {
          var S as rxr:subject{{}},
          rxr:predicate{ attributes{ rxr:uri{ var PREDICATE } } },
          var O as rxr:object{{}}
        }
      }}
    ],
    RXR-RDFNODE[ var S, var SUBJECT ],
    RXR-RDFNODE[ var O, var OBJECT ]
  ]
END
```

The only complicating factor in this rule is the encoding of the subjects and objects' type into attribute names. There are two possible variations for the subject (URI or blank node) and three for the object (literal, URI or blank node), adding up to six different combinations in total. Instead of requiring one rule for each case, resulting in six nearly identical rules, the rule given here queries intermediate `RXR-RDFNODE` data terms for the Xcerpt mapping of the subjects and the objects. These are constructed by two additional rules.

```
CONSTRUCT
  RXR-RDFNODE[ var NODE, var IDENT:var TYPE{} ]
FROM
  ns-default = "http://ilrt.org/discovery/2004/03/rxr/"

  RDF-FILE[ /.*/,
    graph{{
      desc var NODE as /(subject|object)/{
        attributes{{ var TYPE{ var IDENT } }}
      }
    }} where { var TYPE = "uri" or var TYPE = "blank" }
  ]
END
```

The first rule constructs an `RXR-RDFNODE` data term for each `subject` or `object` element of the RXR file, holding the element itself and the corresponding Xcerpt mapping of the RDF node that the element represents. The other rule handles literal objects with

optional datatype and language declarations.

```
CONSTRUCT
 RXR-RDFNODE[ var NODE,
              literal{
                var TEXT,
                attributes{{
                 optional "http://www.w3.org/1999/02/22-rdf-syntax-ns#datatype":uri{ var TYPE },
                 optional "http://www.w3.org/1999/02/22-rdf-syntax-ns#datatype":uri{ var LANG }
                }}
              }
            ]
FROM
  ns-default = "http://ilrt.org/discovery/2004/03/rxr/"

  RDF-FILE[ /.*/,
    graph{{
      desc var NODE as object{
        attributes{{
          optional datatype{ var TYPE },
          optional xml:lang{ var LANG }
        }},
        var TEXT as /.*/
      }
    }}
  ]
END
```

Rules for other triple-listing, XML-based serialization formats, e.g. TriX, would likely be very similar.

5.3.3 Rules for RDF/XML

RDF/XML is a complex serialization format, that slices an RDF graph into an XML tree by nesting XML elements in arbitrary depth. Recalling Section 2.1.3, the basic structure of RDF/XML is the alternation between elements representing nodes and elements representing edges of the RDF graph the document serializes. Due to its complexity, a large number of rules are necessary for extracting the triples from RDF/XML documents. Only some highlights are explained here. The whole set of rules is given in Appendix A .

5.3.3.1 Structural Recursion

Parsing RDF/XML is complicated by the the many possible abbreviations the format allows and the fact that there exist cases where it is not possible to identify whether an XML element represents, the subject, the predicate or the object of a triple, except by examining the whole path from the root to the element in question. RDF/XML thus needs to be parsed using structural recursion.

The recursion starts with a rule that selects the direct children of the `rdf:RDF` root element, as they certainly represent nodes in the RDF graph. These are collected into

intermediatory RDFXML-SUBJECT data terms, used to hold the RDF/XML document's origin URL and subtrees of the document who's top most element represent potential subject nodes of the RDF graph serialized into the document.

```
% Top-Level Subjects
CONSTRUCT
  RDFXML-SUBJECT[ var ORIGIN, var SUB ]
FROM
  ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

  RDF-FILE[
    var ORIGIN,
    rdf:RDF{{
      var SUB
    }}
  ]
END
```

In the most general case, triples are represented in RDF/XML by nesting elements that represent the subject, the predicate and the object of a triple into each other in the order subject-predicate-object. As the objects are also potential subjects of other triples, a rule collects them into RDFXML-SUBJECT data terms. Again, the origin URL of the document that contains the subtree is carried over.

```
% Recursion, subject-predicate-object nesting
CONSTRUCT
  RDFXML-SUBJECT[ var ORIGIN, var OBJECT ]
FROM
  ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

  and[
    RDF-FILE[
      var ORIGIN,
      rdf:RDF{{ desc identity var IDENT var OBJECT as /.*/:/.*/{{}} }}
    ],
    RDFXML-SUBJECT[ var ORIGIN,
      /.*/:/.*/{{      % subject
        /.*/:/.*/{{    % predicate with child as object
          without attributes{{ rdf:parseType{ "Literal" } }},
          without attributes{{ rdf:parseType{ "Resource"} }},
          identity var IDENT var OBJECT
        }}
      }}
    ]
  ]
END
```

The rdf:parseType attribute influences the striping of RDF/XML documents and requires its own rules for recursion. See the respective sections of Appendix A for them.

5.3.3.2 Nodes: Four different Appearances

Elements that represent nodes of the RDF graph have one of four possible forms: they either have one of the identifying attributes (rdf:about, rdf:ID, rdf:nodeID) that specifies the node's type together with an identifier, or they do not have any identifying

attribute and represent a blank node without any explicit blank node identifier.

Instead of distinguishing the four forms in each rule that concerns itself with extracting triples, the mapping of these elements to their corresponding Xcerpt data term representation without children is delegated to four specific rules queried by other rules using rule-chaining, each covering one of the four forms.

```
% regular URI reference
CONSTRUCT
  RDFXML-NODE[ var ORIGIN, var NODE, var URI:uri{} ]
FROM
    ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

    RDFXML-SUBJECT[
      var ORIGIN,
      var NODE as /.*/:/.*/{{
        attributes{{ rdf:about{ var URI } }}
      }}
    ]
END
```

The given rule maps elements representing URI nodes, by extracting the URI from the value of the element's `rdf:about` attribute. Only such elements determined to be representing nodes by the structural recursion are considered. The result is a data term labeled `RDFXML-NODE` that contains the element, the URL it originated and the corresponding mapping.

5.3.3.3 Extraction of Triples

The actual extraction of triples is performed by a large number of rules, required by the great number of variations allowed in RDF/XML. In the most verbose form, a triple is represented by nested elements that represent the subject, the predicate and the object.

```
CONSTRUCT
  RDF-TRIPLE[
    attributes{ origin{ var ORIGIN} },
    var SUBJECT,
    &join( var P_NS, var P_LN):uri{},
    var OBJECT
  ]
FROM
  ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

  and[
    RDFXML-SUBJECT[
      var ORIGIN,
      identity var SIDENT var SNODE as /.*/:/.*/{{
        var P_NS:var P_LN{
          without attributes{{ rdf:parseType{{}} }},
          identity var OIDENT var ONODE as /.*/:/.*/{{}}
        }
      }}
    ],
    RDFXML-NODE[ var ORIGIN, identity var SIDENT var SNODE, var SUBJECT ],
    RDFXML-NODE[ var ORIGIN, identity var OIDENT var ONODE, var OBJECT ]
```

```
  ]
END
```

The rule given here covers that case. It constructs a `RDF-TRIPLE` out of the nested elements, using a conjunction to extract the mappings for the elements representing the subject and the object from `RDFXML-NODE` data terms introduced in the previous section. The URI of the predicate is assembled from the corresponding element's namespace and local name. Provenance information is preserved in `origin` attributes of the constructed `RDF-TRIPLE` data terms.

The other sections of Appendix A each covers a specific abbreviation from the RDF/XML syntax specification.

5.3.4 From Triples to Graphs

The rules for RXR and RDF/XML presented so far only cover the triple representation, converting from the serialized form to a set of `RDF-TRIPLE` data terms. Instead of providing another set of rules to convert from serialized format directly to the graph representation, a rule is used to create the graph representation as a view on to the triple representation:

```
CONSTRUCT
  RDF {
    attributes{ origin{ var ORIGIN } },
    all var S_IDENT @ var S_IDENT:var S_TYPE {
      all optional var P_URI {
        ^var O_IDENT
      },
      all optional var P_URI {
        var LITERAL
      }
    }
  }
FROM
  or{
    RDF-TRIPLE[
      attributes{ origin{ var ORIGIN } },
      var S_IDENT:var S_TYPE{},
      var P_URI:uri{},
      optional var LITERAL as literal{{}},
      optional var O_IDENT:/uri|blank/{{}}
    ],
    RDF-TRIPLE[
      attributes{{ origin{ var ORIGIN } }},
      /.*/:/.*/{{}},
      /.*/:/.*/{{}},
      var S_IDENT:var S_TYPE{{}}
    ]
  }
END
```

The rule collects all resources that are subjects or objects of triples from a single origin into the two variables `S_IDENT` and `S_TYPE`, holding the resource's type and identifier.

These two are used to construct the direct children of the `RDF` root node of the graph representation, eliminating duplicates. The resources' URIs and blank node identifiers are used to set anchors. Then for each resource, the predicates and objects of triples having the resource as subject are collected and used to connect resources, using the objects' identifiers to create references to the previously set anchors.

In addition to the above rule that creates one graph representation for each RDF graph loaded into the system, another almost identical rule is needed to create the merger graph described in Section 5.2.2. Except specifying `!/\*/` (all but `*`) as the origin in the query, and `*` as the origin in the construction, the two rules are exactly the same.

5.4 Evaluation against "RDF Data Access Use Cases and Requirements"

The W3C's *RDF Data Access Working Group* (DAWG) is currently working on standardizing the query language SPARQL(c.f. Section 4.2.6) together with an access protocol for retrieval of RDF data. As part of its on-going work, the working group has published a working draft a document titled "*RDF Data Access Use Cases and Requirements* [50], collecting the requirements, use cases and objectives that the group aims to meet with its language.

Even tough the DAWG limits its scope to a query language specific to RDF data that does not cover inferencing and integration with arbitrary XML data, the group's observations still serve as a valuable guideline for any query language intended to be used on RDF data.

This section will evaluate Xcerpt and the formalism proposed in this chapter one by one against the design objectives and requirements outlined in the *Use Cases and Requirements* document. Some of the use cases outlined in the same document have been implemented and are presented in Chapter 7, "Use Cases".

5.4.1 Requirements

RDF Graph Pattern Matching – Conjunction Because all Xcerpt queries are patterns, queries against any of the the both representations of RDF data as data terms are graph patterns. Queries can be specified either as a conjunction of triple patterns, or as structured graph patterns when querying the graph representation.

Variable Binding Results Xcerpt's variable mechanism allows arbitrary numbers of variables to be bound by a query, resulting in a set of bindings for each possible way the queries data matches the query. Arbitrary data terms can be constructed using the bindings, including for example the SPARQL Variable Binding Results XML Format [51].

Extensible Value Testing Xcerpt includes a number of arithmetic and comparison functions that can be used for aggregation, ordering, binding filtering by semantic conditions and computation of new values. An extension mechanism is being considered, but detail have not yet been worked out.

Subgraph Results The form of the result of an query in Xcerpt depends on the bound variables and the construction using them. Variables can be selectively bound to either namespaces, labels, literal values or complete terms. By binding terms, a subgraph of the original queried graph can be extracted, containing all nodes reachable from the query target.

Example 5.4.1 shows such a query, binding subgraphs to the variable `PERSON`, namely those subgraphs that consist of resources that are reachable from resources representing persons that have the first name Peter.

Example 5.4.1 Binding a subgraph to a variable

```
CONSTRUCT
  ...
FROM
  RDF{{
    var PERSON → /.*/{{
      "http://xmlns.com/foaf/0.1/firstName" {{
        "Peter"
      }}
    }}
  }}
END
```

Local Queries As demonstrated in this chapter, it is possible to define rules for extracting the triples out from any XML based serialization format. External resources to be queried are specified using the `resource` construct, in which the resource is identified by an URI. The Xcerpt prototype implementation recognizes the `http` and `file` schemes and retrieves the resource either remotely using the HTTP protocol or from locally stored files.

Optional Match Parts of a query that do not necessarily need to match in order for the query to succeed, can be specified using the `optional` construct.

Limited Datatype Support Xcerpt natively supports strings as well as integer and floating point numbers as literal values. A number of functions for arithmetics and string manipulation, as well as comparison and aggregation functions are defined as standard function library. A complete type system, however, has not been integrated into Xcerpt yet and is the subject of ongoing research.

Result Limits The maximal number of results produced by a Xcerpt rule can be restricted by using the `some` grouping construct together with a numeral parameter in the rule's construct term.

Example 5.4.2 limiting the number of results to 5

```
CONSTRUCT
  authorlist{
    some 5 author{ var AUTHOR }
  }
FROM
 query
END
```

Streaming Results This particular requirement is not applicable, as Xcerpt does not concern itself with an access protocol for a client-server environment. Instead, the final result from the execution of a Xcerpt program is a single XML document. The streaming of the results is orthogonal to language concepts.

RDF Graph Pattern Matching – Disjunction Disjunctions are supported through the `or` construct.

5.4.2 Design Objectives

Human-friendly Syntax Xcerpt provides both an user-friendly, text-based syntax and a XML-based syntax.

Data Integration and Aggregation Despite the generic title, this objective limits itself to the support of querying provenance information and restricting a query based on the origin of the RDF data queried. Section 5.2 details how this thesis proposes to handle provenance information.

Non-existent Triples Xcerpt supports two different kinds of negation: *query negation* (`not` keyword) and *subterm negation* (`without` keyword). Depending on context,

either one or the other can be use to query for non-existence. For example, a query for resources that have a certain property but not a specific other property, can be formulated like in Example 5.4.3, which queries for the URIs of all resources whose `firstName` property is known, but not their `lastName` property.

Example 5.4.3 querying for non-existence of a property

```
CONSTRUCT
  construct term
FROM
  RDF {{
    var URI{{
      "http://xmlns.com/foaf/0.1/firstName"{{}},
      without "http://xmlns.com/foaf/0.1/lastName"{{}}
  }}
END
```

Bandwidth-efficient Protocol The objective is obviously intended for the access protocol and not the query language the working group is designing, and thus inapplicable to Xcerpt and the formalism for querying RDF proposed in this thesis. It should be noted however, that the amount of data returned by a Xcerpt query can be reduced by limiting the number of results using `some` or excluding certain subterms from variable bindings using the `except` construct.

Literal Search Xcerpt supports POSIX-style Regular Expressions for extremely flexible searches on string literals.

Yes-No Queries Yes-No queries can not be directly expressed in Xcerpt, as queries that do not match do not return a result. Nevertheless, equivalent results can be achieved using *conditional queries*, by querying for either a "Yes" or a "No" depending on the success of the actually intended query, as shown in Example 5.4.4.

Addressable Query Results The objective is intended for the access protocol than for the query language the working group is designing, and thus inapplicable to Xcerpt and this thesis' formalism. However, it should be easily possible to encode the query into an URI using the encoding scheme for nonpermissible characters described in [15], similar to the handling how parameters are passed with `GET` requests in Web applications. An idea to associate rules or sets of rules with URIs to make them adressable is also currently being investigated by the authors of Xcerpt and might be supported in the future.

Example 5.4.4 a simulated Yes-No query in Xcerpt

```
GOAL
  var RESULT
FROM
if
  query
then
  var RESULT → yes{}
else
  var RESULT → no{}
END

CONSTRUCT
  yes{}
END

CONSTRUCT
  no{}
FROM
```

CHAPTER SIX

RDF Schema Inferencing with Xcerpt

> inference: the drawing of a conclusion from known or assumed facts or statements; esp. in Logic, the forming of a conclusion from data or premisses, either by inductive or deductive methods;
>
> —Oxford English Dictionary

The Semantic Web promises to make the Web more useful by expressing information in a machines processable manner using RDF. The RDF data model has been carefully designed to allow inferencing to be performed, allowing implicit information to be deduced from a set of explicit assertions by applying inference rules. Nevertheless, only a small number of query language actually support inferencing. Even among them, most only do so via extension constructs that are implemented using the underlying programming language, limiting themselves to predetermined sets of inference rules. Only a very small number actually support formulation of custom inference rules using facilities of the language itself, a feature indispensable for truly versatile Semantic Web query languages.

This chapter demonstrates Xcerpt's ability to perform inferencing on RDF data by means of the RDF Schema inference rules. First, a direct translation of the inference rules is shown in Section 6.1, which, though comprehensible and elegant, does not function as expected. After careful analysis of the reasons for its failure in Section 6.2, a practical solution is presented in Section 6.3, implementing a exemplary subset of the inference rules of RDF Schema.

6.1 A Naive Approach to RDF Schema Inferencing

RDF Schema provides a vocabulary with complex semantics for defining new vocabularies (cf. Section 2.1.2). Figure 6.1 shows the properties and classes defined in the

RDF Schema vocabulary. The semantics of these terms are defined as a set of axiomatic triples and a number of inference rules [11]. This section shows an attempt at converting them to Xcerpt rules as directly as possible. With Xcerpt being a rule language with native support for rule chaining, such an approach seems the natural way to implement RDF Schema inferencing.

Figure 6.1 Terms defined by RDF Schema.

```
rdfs:domain rdfs:range rdfs:Resource rdfs:Literal
rdfs:Datatype rdfs:Class rdfs:subClassOf
rdfs:subPropertyOf rdfs:member rdfs:Container
rdfs:ContainerMembershipProperty rdfs:comment
rdfs:seeAlso rdfs:isDefinedBy rdfs:label rdf:XMLLiteral
rdf:Property rdf:type
```

6.1.1 Axiomatic Triples

Axiomatic triples are triples that are always asserted. Those defined by RDF Schema assert the semantics of many of the terms of RDF Schema using the vocabulary of RDF Schema itself, reducing the number of inference rules necerssary to define the semantics of all terms.

The list of axiomatic triples for RDF Schema include assertions about the `rdfs:domain` and `rdfs:range` of RDF Schema's properties and `rdfs:subClassOf` relationships between RDF Schema's classes, amongst other things. One such axiomatic triple is

```
rdf:type rdfs:range rdfs:Class
```

asserting that all object resources of triples having `rdf:type` as predicate are classes.

As axiomatic triples are always asserted, they could be implemented as `CONSTRUCT` rules without bodies, rules that always match, one rule per axiomatic triple in the following style:

```
CONSTRUCT
  RDF-TRIPLE[
    "http://www.w3.org/1999/02/22-rdf-syntax-ns#type":uri{},
    "http://www.w3.org/2000/01/rdf-schema#range":uri{},
    "http://www.w3.org/2000/01/rdf-schema#Class":uri{}
  ]
END
```

Alternately, a file containing all axiomatic triples in a supported serialization format could be imported:

```
CONSTRUCT
  RDF-FILE[ "axiom", var Root ]
FROM
  in{
```

```
    resource{ "file:axioms.rdf", "xml" },
    var Root
  }
END
```

In practice however, both approaches are unable to assert all axiomatic triples, as RDF Schema defines an infinite number of axiomatic triples for specifying the semantics of infinitely many properties for asserting ordered membership in contains, e.g. `rdf:_1`, `rdf:_2`, ...

6.1.2 Inference Rules

The semantics of the core vocabulary of RDF Schema is specifies by inference rules named `rdfs1` through `rdfs13` [11]. Besides defining semantics of the `rdfs:domain`, `rdfs:range`, `rdfs:subClassOf` and `rdfs:subPropertyOf` properties, the rules also define automatic class memberships of resources in the classes defined by RDF schema, such as that all resources used as predicates of triples are instances of the class `rdfs:Property`. The rules are all stated with the form *add a triple to a graph when it contains triples conforming to a pattern*, defining what the triples in the pattern imply.

For example, the inference rule `rdfs11` defines the transitivity of `rdfs:subClassOf` as

> If [the graph] contains:
>
> `uuu rdfs:subClassOf vvv` and `vvv rdfs:subClassOf xxx`
>
> then add:
>
> `uuu rdfs:subClassOf xxx`

The clear separation of the conditions necessary for the rule to match and the consequence of a match invites a natural translation into an Xcerpt rule, whose body and head directly correspond to those of the inference rule:

```
CONSTRUCT
  RDF-TRIPLE[ var UUU, "http://www.w3.org/2000/01/rdf-schema#subClassOf":uri{}, var XXX ]
FROM
  and{
    RDF-TRIPLE[ var UUU, "http://www.w3.org/2000/01/rdf-schema#subClassOf":uri{}, var VVV ],
    RDF-TRIPLE[ var VVV, "http://www.w3.org/2000/01/rdf-schema#subClassOf":uri{}, var XXX ]
  }
END
```

Such straight-forward rules would work perfectly with a forward-chaining evaluation. However, with the backward-chaining evaluation used by the current prototype of Xcerpt, recursive rules of this style lead to non-termination as in any backward-chaining logic programming language and can not be used.

6.2 The Pitfalls of Recursion with Backward-Chaining

The seemingly simple and comprehensible implementation of an inference rule as Xcerpt rule leads to non-termination of the whole Xcerpt program. In order to properly implement RDF Schema-style inferencing, the problem needs to be carefully analyzed.

The problematic rule tries to implement the semantics of ***add*** *a triple to a graph when it contains triples conforming to a pattern*. Triples are added as long as the pattern matches, potentially using a newly added triple as the input to another application of the rule. With a forward-chaining evaluation, such a semantics would be easily achieved, as it is exactly how forward-chaining works (cf. Section 2.2.3). Consider the following Xcerpt program consisting of a single data term, a single `CONSTRUCT` rule and a `GOAL` rule:

```
% data term
CONSTRUCT
  f{ f{ "foo" } }
END

% f-Unwrapper
CONSTRUCT
  f{var X}
FROM
  f{ f{ var X } }
END


% query
GOAL
  f-text{ all var F }
FROM
  f{ var F as /.*/ }
END
```

If Xcerpt is implemented using forward-chaining, the evaluation would start with the only data term and search for rules whose bodies match the data term, finding the only `CONSTRUCT` rule of the program. The rule unwraps the data term and constructs the new data term `f{"foo"}`, increasing the number of data terms in the "universe" to two. At this point, saturation is achieved as the new data term does not match the body of the `CONSTRUCT` rule and reapplying the rule to the initial data term would not yield another, non-duplicate data term. Finally the body of the `GOAL` rule is evaluated and matched against the two data terms, returning `f-text{"foo"}` as the result of the whole program.

The backward-chaining evaluation actually employed by the current Xcerpt prototype works in the other direction. When using the body of a rule as search pattern, not only are data terms matched against the pattern, but also rules the rule "depends" on, rules that potentially produce data that could match the pattern, are searched. When such rules are found, their bodies become the new search patterns, enhanced with additional constraints based on the original pattern. The process is repeated until no further data terms or rules are matched. With recursive rules, however, comes the danger of non-

termination. Again, consider the Xcerpt program introduced above. The backward-chaining starts with the body of the GOAL rule, searching for rule heads and data terms that match the pattern

```
f{ var F as /.*/}
```

The data term does not match, but the head of the CONSTRUCT rule (`f{var X}`) might produce a data term that matches, if the bindings of the variable X were of the form `/.*/`. Applying the constraint to the rule's body, the new search pattern becomes

```
f{ f{ var F as /.*/} }
```

The data term now matches this new pattern with `foo` being a binding of the variable F. At the same time, the head of the CONSTRUCT rule also matches the new pattern, if the bindings of the variable X were of the form `f{/.*/}`. Even tough *an* answer has already been found with the data term, the search is continued, as *all* possible results are queried, using the pattern with new constraint applied.

```
f{ f{ f{ var F as /.*/} } }
```

This new pattern does not match any data term, but again matches the head of the CONSTRUCT rule. The pattern endlessly continues to grow with each recursion of the CONSTRUCT rule through the cyclic dependency, even tough there are no data terms that can potentially match it. Figure 6.2 illustrates the steps of the forward- and backward evaluations.

Figure 6.2 Forward- and Backward-Chaining Evaluations

Forward-Chaining:

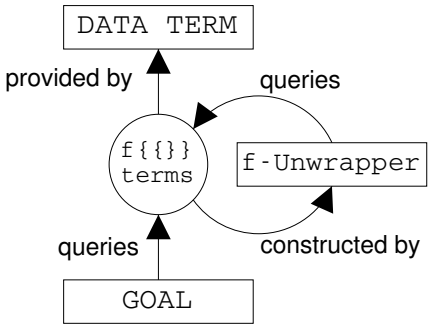
```
f{f{"foo"} → f{"foo"} → {F "foo"}
```

Backward-Chaining:

```
f-text{ all var F}
  to
f{ var F as /.*/ }
  to
f{f{ var F as /.*/}} → f{f{"foo"} → {F "foo"}
  to
f{f{f{ var F as /.*/}}}
  to
f{f{f{f{ var F as /.*/}}}}
  to
...
```

As demonstrated by this simple example, backward-chaining starts with the GOAL and traverses the chain of the dependencies between rules backward, finding matching data terms on the way. Rules are recursive if they are part of a cycle in the dependency chain, using their own output directly or indirectly as input to themselves. Such cycles lead to endless traversal of the dependency chain and thus to non-termination of the program. Figure 6.3 visualizes the cyclic dependencies of the rules of this example. The naively implemented inference rule suffers from the same problem, with both subqueries of the conjunction being recursive.

Figure 6.3 Cyclic Rule Dependency



For recursive rules to terminate with backward-chaining, a *base case* is needed to limit the cyclic traversal of the dependency chain. The following rule implements the transitive closure of simplified RDF triple-like data terms with `subClassOf` predicates in a terminating manner:

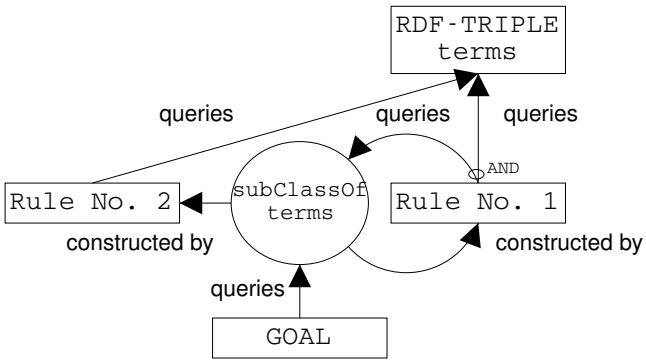
```
% Rule No. 1
CONSTRUCT
  subClassOf[ var CLS, "subClassOf", var SUPR_CLS]
FROM
  and[
    TRIPLE[ var CLS, "subClassOf", var X ],
    subClassOf[ var X, "subClassOf", var SUPR_CLS ]
  ]
END

%Rule No. 2
CONSTRUCT
  subClassOf[ var CLS, "subClassOf", var SUPR_CLS]
FROM
  TRIPLE[ var CLS, "subClassOf", var SUPR_CLS ]
END
```

Instead of "adding" the result back into the set of `TRIPLE` data terms, the inferred triples as well as the explicit triples asserting subclass relationships are collected into another set of data terms named `subClassOf`. By using an ordered conjunction and a non-recursive query as the first subquery, the recursive second subquery is only evaluated if the first subqueries matches, in effect breaking the recursion when there is no explicit triple to support another iteration. Figure 6.3 visualizes the limited cyclic dependencies of the two rules, together with a `GOAL` querying for `subClassOf` terms. Note how the cycle is limited by data untouched by the cycle.

6.3 A Practical Approach to RDF Schema Inferencing

This section demonstrates a working implementation of a subset of the inference rules of RDF Schema. First, the dependencies between the inference rules are analyzed, in order to identify potentially problematic cyclic dependencies described in the previous

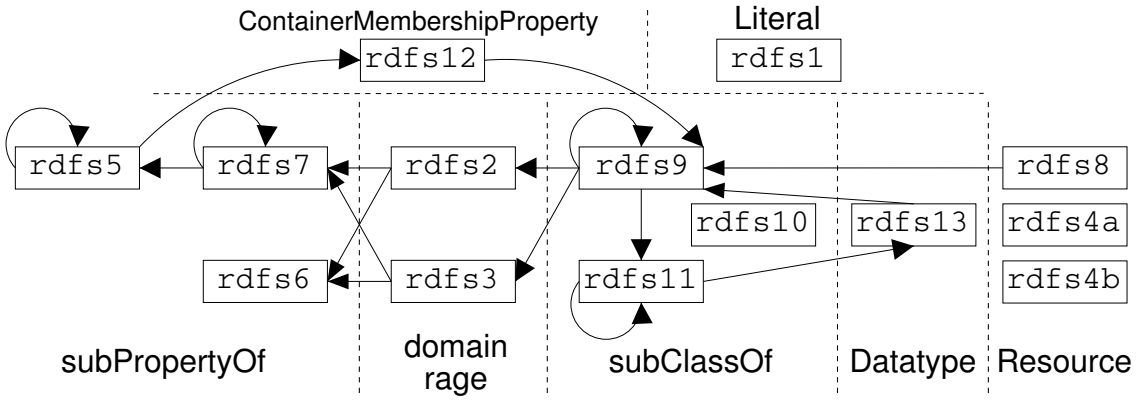
Figure 6.4 Restricted Cyclic Rule Dependency



sections. Based on the findings, sets of Xcerpt rules are presented, each implementing a different aspect of RDF Schema.

Figure 6.5 shows the interdependencies of the inference rules implementing the semantics of RDF Schema. In addition of the charted dependencies, all rules depend on the set of explicitly asserted triples, as the rules match regardless whether a triple they depend on is explicitly asserted or only implied. The recognizable dependency cycles, indicating recursion and thus need be carefully handled, include local cycles of the rules responsible for the transitivity of `rdfs:subPropertyOf` and `rdfs:subClassOf` as well as non-local cycles involving `rdfs:ContainerMembershipProperty` instance properties and type inferencing for literals. It is assumed that the vocabulary itself is not extended. For example, if an superclass of `rdfs:Resource` was added, a dependency from `rdfs11` to `rdfs8` would emerge, adding another non-local cycle.

Figure 6.5 Interdependencies of RDF Schema Inference Rules



For an Xcerpt program to terminate, all rules need to be either non-recursive, or if recursive, they need to be restricted using the method described in the previous section. It is not possible to simply add inferred triples back to the set of "raw", explicitly asserted triples in a forward-chaining manner. Instead, inferred triples must be kept separately, so that the set of explicitly asserted triples can serve as base case. Therefore, RDF

Schema inferencing with Xcerpt is implemented by manually ordering the application of rules. For each group of inference rules, a new sets of data terms is built, consisting of newly inferred triples and triples produces by previous sets of inference rules. These intermediary sets can then be used as base cases for recursive rules.

Fortunately, the interdependencies of the inference rules for the four main vocabulary-defining properties of RDF Schema (`rdfs:subClassOf`, `rdfs:domain`, `rdfs:range` and `rdfs:subPropertyOf`) allow such an grouping and ordering. Considered as group, the rules for `rdfs:subPropertyOf` do not depend on any output from the other rules. The rules for `rdfs:doman` and `rdfs:range` depend on the triples inferred by the `rdfs:subPropertyOf` rules but not those of the `rdfs:subClassOf` rules. The rules for `rdfs:subClassOf` in turn depend on triples having `rdf:type` as their predicate, potentially inferred by the `rdfs:doman` and `rdfs:range` rules.

Figure 6.6 Dependencies between Xcerpt Rules implementing RDF Schema Inferencing

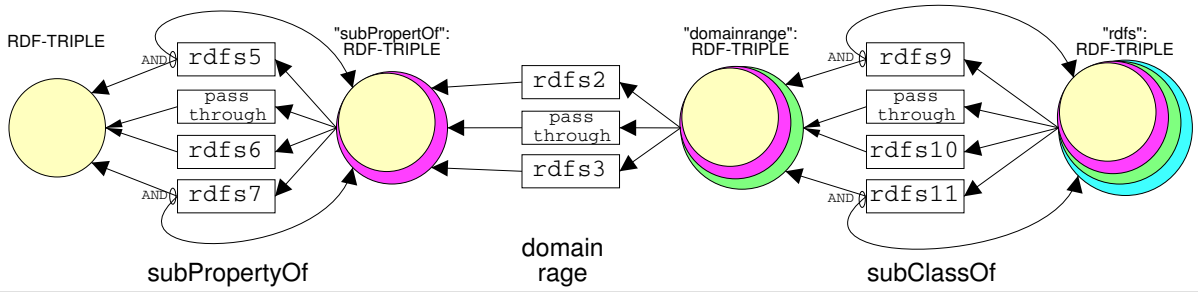


Figure 6.6 shows the interdependencies of the Xcerpt rules implementing the subset of RDF Schema inference rules for the main-vocabulary defining properties and the sets of data terms used as intermediaries. The intermediary triples are also labeled `RDF-TRIPLE` and have the same structure as explicitly asserted triples, but belong to a different namespaces.

6.3.1 `rdfs:subPropertyOf` Inferencing

The semantics of `rdfs:subPropertyOf` is defined by three inference rules, specifying the transitivity of property hierarchies and how triples imply other triples with super-properties as predicates.

rdfs5 `uuu rdfs:subPropertyOf vvv` and `vvv rdfs:subPropertyOf xxx`

-¿ `uuu rdfs:subPropertyOf xxx`

rdfs6 `uuu rdf:type rdf:Property`

-¿ `uuu rdfs:subPropertyOf uuu`

rdfs7 `aaa rdfs:subPropertyOf bbb` and `uuu aaa yyy`

-¿ `uuu bbb yyy`

The rules `rdfs5` and `rdfs7` are both recursive. `rdfs5` queries for triples with the predicate `rdfs:subPropertyOf` and constructs such triples. `rdfs7` consumes arbitrary triples whose predicates have superproperties and produces triples with predicates that might also have a superproperty. However, `rdfs7` does not need to be recursive, as all indirect superproperties of a property become direct superproperties by first repeatedly applying `rdfs5` and building the complete transitive closures of properties. `rdfs6` is non-recursive and does not affect the other two rules, as triples inferred by it do not imply additional triples when matched by the other two rules..

The following Xcerpt rules implement the three inferencee rules, producing a view, having the namespace `subPropertyOf`, of the set of explicitly asserted triples and triples implied by them through the `rdfs:subPropertyOf` inference rules.

```
% pass-through all explicit triples
CONSTRUCT
  "subPropertyOf":RDF-TRIPLE[ var S, var P, var O]
FROM
  RDF-TRIPLE[ var S, var P, var O]
END

% rdfs5, transitive closure of subPropertyOf
CONSTRUCT
 "subPropertyOf":RDF-TRIPLE[
   var PROP, "http://www.w3.org/2000/01/rdf-schema#subPropertyOf":uri{}, var SUPERPROP
 ]
FROM
  and[
    RDF-TRIPLE[
      var PROP, "http://www.w3.org/2000/01/rdf-schema#subPropertyOf":uri{}, var X
    ],
    "subPropertyOf":RDF-TRIPLE[
      var X, "http://www.w3.org/2000/01/rdf-schema#subPropertyOf":uri{}, var SUPERPROP
    ]
  ]
END

% rdfs7, subPropertyOf inferencing
CONSTRUCT
  "subPropertyOf":RDF-TRIPLE[ var S, var SUPERPROP, var O ]
FROM
  and[
    RDF-TRIPLE[ var S, var PROP, var O ],
    "subPropertyOf":RDF-TRIPLE[
      var PROP, "http://www.w3.org/2000/01/rdf-schema#subPropertyOf":uri{}, var SUPERPROP
    ]
  ]
END

% rdfs6, property is subPropertyOf self
CONSTRUCT
  "subPropertyOf":RDF-TRIPLE[
    var PROP, "http://www.w3.org/2000/01/rdf-schema#subPropertyOf":uri{}, var PROP
  ]
```

```
FROM
  RDF-TRIPLE[
    var PROP,
   "http://www.w3.org/1999/02/22-rdf-syntax-ns#type":uri{},
   "http://www.w3.org/1999/02/22-rdf-syntax-ns#Property":uri{}
  ]
END
```

6.3.2 `rdfs:domain` and `rdfs:range` Inferencing

The semantics of `rdfs:domain` and `rdfs:range` are defined by one inference rule each, specifying how class memberships of subject and object resources of triples with certain predicates are implied.

rdfs2 `aaa rdfs:domain xxx` and `uuu aaa yyy`

-¿ `uuu rdf:type xxx`

rdfs3 `aaa rdfs:range xxx` and `uuu aaa vvv`

-¿ `vvv rdf:type xxx`

Both rules match arbitrarty triples, potentially including those inferred by the inference rules for `rdfs:subPropertyOf` and thus need to query the `"subPropertyOf":RDF-TRIPLE` data terms constructed in the previous section. The three rules are implemented by the following Xcerpt rules, producing a view of the set of previously inferred triples and triples implied by them through the `rdfs:doman` and `rdfs:range` inference rules, having the namespace `domainrange`.

```
% pass-through all explicit and previously inferred triples
CONSTRUCT
  "domainrange":RDF-TRIPLE[ var S, var P, var O]
FROM
  "subPropertyOf":RDF-TRIPLE[ var S, var P, var O]
END

% rdfs2, rdfs:domain inferencing
CONSTRUCT
  "domainrange":RDF-TRIPLE[
    var S, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type":uri{}, var CLASS
  ]
FROM
  and[
    "subPropertyOf":RDF-TRIPLE[
      var P, "http://www.w3.org/2000/01/rdf-schema#domain":uri{}, var CLASS
    ],
    "subPropertyOf":RDF-TRIPLE[ var S, var P, var O ]
  ]
END

% rdfs3, rdfs:range inferencing
CONSTRUCT
```

```
  "domainrange":RDF-TRIPLE[
    var O, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type":uri{}, var CLASS
  ]
FROM
  and[
    "subPropertyOf":RDF-TRIPLE[
      var P, "http://www.w3.org/2000/01/rdf-schema#range":uri{}, var CLASS
    ],
    "subPropertyOf":RDF-TRIPLE[ var S, var P, var O ]
  ]
END
```

6.3.3 `rdfs:subClassOf` Inferencing

The semantics of `rdfs:subClassOf` is defined by three inference rules, specifying the transitivity of class hierarchies and how membership of a resource in a class also implies membership in all of the class' superclasses.

rdfs9 `uuu rdfs:subClassOf xxx` and `vvv rdf:type uuu`

-¿ `vvv rdf:type xxx`

rdfs10 `uuu rdf:type rdfs:Class`

-¿ `uuu rdfs:subClassOf uuu`

rdfs11 `uuu rdfs:subClassOf vvv` and `vvv rdfs:subClassOf xxx`

-¿ `uuu rdfs:subClassOf xxx`

The rules match triples with `rdf:type` as predicate, potentially inferred by the inference rules for `rdfs:domain` and `rdfs:range`. Therefore, they need to query the `"domainrange":RDF-TRIPLE` data terms constructed in the previous section. Being similar to the three rules for `rdfs:subPropertyOf`, the rules are also implemented in the same manner, creating a view having the namespace `rdfs`.The name was chosen, because this view contains all triples inferred by the Xcerpt rules rules presented in this chapter.

```
% pass-through all explicit and previously inferred triples
CONSTRUCT
  "rdfs":RDF-TRIPLE[ var S, var P, var O]
FROM
  "domainrange":RDF-TRIPLE[ var S, var P, var O]
END

% rdfs9, type inferencing with subClassOf
CONSTRUCT
  "rdfs":RDF-TRIPLE[
    var RESOURCE, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type":uri{}, var SUPERCLASS
  ]
```

```
FROM
  and[
    "domainrange":RDF-TRIPLE[
      var RESOURCE, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type":uri{}, var CLASS
    ],
    "rdfs":RDF-TRIPLE[
      var CLASS, "http://www.w3.org/2000/01/rdf-schema#subClassOf":uri{}, SUPERCLASS
   ]
  ]
END

% rdfs10, classes are subClassOf self
CONSTRUCT
  "rdfs":RDF-TRIPLE[
    var CLASS, "http://www.w3.org/2000/01/rdf-schema#subClassOf":uri{}, var CLASS
  ]
FROM
  "domainrange":RDF-TRIPLE[
    var CLASS,
    "http://www.w3.org/1999/02/22-rdf-syntax-ns#type":uri{},
    "http://www.w3.org/1999/02/22-rdf-syntax-ns#Class":uri{}
  ]
END

% rdfs11, transitive closure of subClassOf
CONSTRUCT
 "rdfs":RDF-TRIPLE[
   var CLASS, "http://www.w3.org/2000/01/rdf-schema#subClassOf":uri{}, var SUPERCLASS
 ]
FROM
  and[
    "domainrange":RDF-TRIPLE[
      var CLASS, "http://www.w3.org/2000/01/rdf-schema#subClassOf":uri{}, var X
    ],
    "rdfs":RDF-TRIPLE[
      var X, "http://www.w3.org/2000/01/rdf-schema#subClassOf":uri{}, var SUPERCLASS
    ]
  ]
END
```

6.3.4 Axiomatic Triples

> "Well, don't look at me; it's the tribbles that are breeding. And if we don't get them off this ship, we're going to be hip deep in them."
>
> —Dr. McCoy, StarTrek - The Trouble With Tribbles

RDF Schema defines several dozen axiomatic triples asserting the `rdfs:domain` and `rdfs:range` of properties defined in the `rdfs` and `rdf` vocabularies, along with a class hierarchy of the classes defined by the two vocabularies. Additionally, an infinite number of axiomatic triples for specifying the semantics of infinitely many container membership properties are defined.

```
rdf:_1 rdf:type rdfs:ContainerMembershipProperty .
rdf:_1 rdfs:domain rdfs:Resource .
rdf:_1 rdfs:range rdfs:Resource .
rdf:_2 rdf:type rdfs:ContainerMembershipProperty .
```

```
rdf:_2 rdfs:domain rdfs:Resource .
rdf:_2 rdfs:range rdfs:Resource .
...
```

The finite subset of axiomatic triples can be asserted using the two methods described above in the naive approach (cf. Section 6.1.1). The infinite subset however can not be materialized in the same manner. Recently, ter Horst has proven [52] that a finite subset of the axiomatic triples concerning container membership properties suffices to perform equivalent inferencing on a RDF graph, by omitting axiomatic triples describing `rdf:_`*i* properties that do not appear in the graph, as long as the axiomatic triples for at least one `rdf:_`*i* are asserted.

The following Xcerpt rules construct the axiomatic triples for `rdf:_`*i* properties, with *i* larger then 1, only if the are present in the graph on which inferencing is performed. The axiomatic triples for `rdf:_1` are added to the finite set of always asserted triples to ensure that the axiomatic triples for at least one `rdf:_`*i* are always asserted.

```
CONSTRUCT
  RDF-TRIPLE[
    &join("http://www.w3.org/1999/02/22-rdf-syntax-ns#_", var INT):uri{},
    "http://www.w3.org/1999/02/22-rdf-syntax-ns#type":uri{},
    "http://www.w3.org/2000/01/rdf-schema#ContainerMembershipProperty":uri{}
  ]
FROM
  RDF-TRIPLE[
    var SUBJECT,
    /http://www.w3.org/1999/02/22-rdf-syntax-ns#_(var INT as [1-9][0-9]*)/:uri{},
    var OBJECT,
  ] where { var INT > 1 }
END

CONSTRUCT
  RDF-TRIPLE[
    &join("http://www.w3.org/1999/02/22-rdf-syntax-ns#_", var INT):uri{},
    "http://www.w3.org/2000/01/rdf-schema#domain":uri{},
    "http://www.w3.org/2000/01/rdf-schema#Resource":uri{}
  ]
FROM
  RDF-TRIPLE[
    var SUBJECT,
    /http://www.w3.org/1999/02/22-rdf-syntax-ns#_(var INT as [1-9][0-9]*)/:uri{},
    var OBJECT,
  ] where { var INT > 1 }
END


CONSTRUCT
  RDF-TRIPLE[
    &join("http://www.w3.org/1999/02/22-rdf-syntax-ns#_", var INT):uri{},
    "http://www.w3.org/2000/01/rdf-schema#range":uri{},
    "http://www.w3.org/2000/01/rdf-schema#Resource":uri{}
  ]
FROM
  RDF-TRIPLE[
    var SUBJECT,
    /http://www.w3.org/1999/02/22-rdf-syntax-ns#_(var INT as [1-9][0-9]*)/:uri{},
    var OBJECT,
  ] where { var INT > 1 }
```

```
END
```

CHAPTER SEVEN

Use Cases

The characteristics of a query language are most clearly demonstrated through use cases portraying real world problems and solutions. Diverse input data and varying intentions each highlight needs for different aspects of a language. A number of use cases have been implemented, using Xcerpt and the formalism for querying RDF with it proposed in this thesis. This chapter presents them in the hope to convey a feeling of the language to readers and to identify its strength and weaknesses.

Also, as Xcerpt is still under active development, the language's development state described in [9], together with the extensions proposed in Chapter 8, "Proposed Extensions to Xcerpt", has been used to implement the use cases.

7.1 Use Cases developed by the W3C RDF DAWG

The RDF Data Access Work Group of the W3C has collected a large number of such use cases for the evaluation of the RDF query language SPARQL it is developing. As these use cases overlap in many parts, only a small selection is described here. The use cases are roughly reordered based on their complexity, guardedly introducing more and more aspects, such as querying of multiple sources and inferencing.

The sample data used for many of the use cases are verbatim copies of the samples of the DAWG Use Cases, others have been generated specifically for this section. Following the DAWG, the data is presented in Notaion 3 form (cf. Section 4.1.3.1).

7.1.1 Finding an Email Address (Personal Information Management)

Despite its original positioning as a meta data format, RDF is increasingly used for immediate representation of data. The use case envisions storage of an address book

in RDF, using the FOAF [5] vocabulary for description of personal information and relationships. The whole address book is stored as a single RDF graph, with each entry of the address book represented by a blank node and a number of statements directly describing the node.

```
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .
[]
     foaf:name    "Johnny Lee Outlaw" ;
     foaf:mbox    <mailto:jlow@example.com> .
```

The address book is queried by specifying values for one or more properties, returning the value of another property of entries matching the query. In the given use case, the name of a person is specified in order to query his/her e-mail address, returning a list of matches.

```
GOAL
  email{ all address{ var MAILTO } }
RDF{{
  /.*/:/.*/{{
    "http://xmlns.com/foaf/0.1/name":uri{ literal{"Johnny Lee Outlaw"} },
    "http://xmlns.com/foaf/0.1/mbox":uri{ var MAILTO:uri{{}} }
  }}
}}
END
```

This query is formulated against the graph representation of RDF graphs. It specifies a node of arbitrary type and identifier with two properties, a name and an e-mail address. The name is specified as a literal value, while the e-mail address, which is an URI, is unknown and bound to the variable `MAILTO`. The construction term of the rule has been omitted, as the use cases does not specify a specific result form.

```
GOAL
  email{ all address{ var MAILTO } }
FROM
  and{
   RDF-TRIPLE[ var X, "http://xmlns.com/foaf/0.1/name":uri{}, literal{"Johnny Lee Outlaw"} ],
   RDF-TRIPLE[ var X, "http://xmlns.com/foaf/0.1/mbox":uri{}, var MAILTO:uri{} ]
  }
END
```

The second query queries not the graph representation but triple representation of the same graph. In it, two triple patterns are joined using the common variable `X`.

7.1.2 Finding Information about Motorcycle Parts (Supply Chain Management)

The use case describes data retrieval from a parts database that includes dependency information between individual parts, stored as a single RDF graph.

```
@prefix triumph:  <http://triumph.example/schema/#> .
@prefix rdfs:     <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
```

```
<http://triumph.example/part/0d92ie433>
    rdf:type    triumph:part ;
    rdfs:label  "Accelerator Cable MK3" ;
    triumph:depends-on  <http://triumph.example/part/329i2dk39> ;
    triumph:part-for  <http://triumph.example/2004/SpeedTriple> ;
    triumph:part-number  "LCD 100-04BSPT" .

<http://triumph.example/part/329i2dk39>
    rdfs:label  "Mounting Bracket" ;
    triumph:requires
        [ triumph:has-number  "4" ;
          triumph:part-number  "149028ab-MT" ;
          triumph:type  triumph:screwx

        ] .
```

A query is sought that retrieves the human-readable description of a part, given it's URI, and all dependent parts that must be replaced simultaneously. The use case differs from the previous one in that it requests information about more than one resource. A simple HTML page that is displayed to the user is constructed as the result of the query. The function `&join` is used to concatenates multiple string literals into a single literal subterm.

```
GOAL
  html{
    head{ title{ "Search Results" } },
    body{
      h1{ &join( var PART_LBL, "(", var PART_NUM, ")" ) },
      p{ "Part for:" },
      ul{
        all li{ var VEHICLE_LBL }
      },
      h2{ "Dependencies:" },
      optional ul{
        all li{
          &join( var DEP_LBL, "(", var DEP_NUM, ")" )
        }
      } with default "-"
    }
  }
FROM
  RDF{{
    "http://triumph.example/part/0d92ie433":uri{{
      "http://www.w3.org/2000/01/rdf-schema#label":uri{
        literal{ var PART_LBL }
      },
      "http://triumph.example/schema/#part-number":uri{
        literal{ var PART_NUM }
      },
      "http://triumph.example/schema/#part-for":uri{
        /.*/:uri{{
          "http://www.w3.org/2000/01/rdf-schema#label":uri{
            literal{ var VEHICLE_LBL }
          }
        }}
      },
      optional "http://triumph.example/schema/#depends-on":uri{
        desc (/.*/:uri > "http://triumph.example/schema/#depends-on":uri)* /.*/:uri{{
          "http://www.w3.org/2000/01/rdf-schema#label":uri{
            literal{ var DEP_LBL }
          },
```

```
          "http://triumph.example/schema/#part-number":uri{
            literal{ var DEP_NUM }
          }
        }
      }}
    }}
  }}
END
```

The query retrieves the human-readable label and part number of the part identified by the URI `http://triumph.example/part/0d92ie433` and optionally any dependent parts from the graph representation, utilizing the qualified descendant construct (see Section 8.3) to recursively select the complete dependency tree. A simple HTML document is constructed using the selected information, demonstrating two major advantages of Xcerpt: the ability to construct arbitrary XML documents and the complete separation of query and construction, allowing independent modification of one or the other.

The same query against the set of triples is more complicated. The construction is the same as above, but in the query the transitive closure of the dependencies need to be collected separately using additional recursive rules. Two `CONSTRUCT` rules collect the information into ordered data terms labeled `DEPENDENCY`, each with two children: the dependent part as the first child and one of its dependencies as the second child. One rule collects all direct dependencies, the other recursively collects all indirect dependencies. The `GOAL` rule, the rule that constructs the final result, queries these `DEPENDENCY` data terms for any parts that the part identified by the URI `http://triumph.example/part/0d92ie433` depends on. Due to the backward-chaining nature of Xcerpt, only the dependencies of this part are actually collected, instead of the dependencies of all parts.

```
GOAL
  html{
    head{ title{ "Search Results" } },
    body{
      h1{ &join( var PART_LBL, "(", var PART_NUM, ")" ) },
      p{ "Part for:" },
      ul{
        all li{ var VEHICLE_LBL }
      },
      h2{ "Dependencies:" },
      optional ul{
        all li{
          &join( var DEP_LBL, "(", var DEP_NUM, ")" )
        }
      } with default "-"
    }
  }
FROM
  and{
    RDF-TRIPLE[ "http://triumph.example/part/0d92ie433":uri{},
                "http://www.w3.org/2000/01/rdf-schema#label":uri{},
                literal{ var PART_LBL } ],
    RDF-TRIPLE[ "http://triumph.example/part/0d92ie433":uri{},
                "http://triumph.example/schema/#part-number":uri{},
                literal{ var PART_NUM } ],
```

```
    RDF-TRIPLE[ "http://triumph.example/part/0d92ie433":uri{},
                "http://triumph.example/schema/#part-for":uri{},
                var VEHICLE ],
    RDF-TRIPLE[ var VEHICLE,
                "http://www.w3.org/2000/01/rdf-schema#label":uri{},
                literal{ var VEHICLE_LBL } ],

    optional DEPENDENCY[ "http://triumph.example/part/0d92ie433":uri{}, var DEP ],
    optional RDF-TRIPLE[ var DEP,
                "http://www.w3.org/2000/01/rdf-schema#label":uri{},
                literal{ var DEP_LBL } ],
    optional RDF-TRIPLE[ var DEP,
                "http://triumph.example/schema/#part-number":uri{
            literal{ var DEP_NUM }
          }
  }
END

CONSTRUCT
  DEPENDENCY[ var PART, var DEP ]
FROM
  RDF-TRIPLE[ var PART, "http://triumph.example/schema/#depends-on":uri{}, var DEP]
END

CONSTRUCT
  DEPENDENCY[ var PART, var DEP ]
FROM
  and[
    RDF-TRIPLE[ var PART, "http://triumph.example/schema/#depends-on":uri{}, var X ],
    DEPENDENCY[ var X, var DEP]
  ]
END
```

The queries are dominated by long URIs, making them overly verbose. Even tough Xcerpt supports an abbreviation mechanism for XML namespaces, it is not suited for abbreviating the URIs used with RDF. Future research will look into a more generic abbreviation mechanism that will increase the readability of queries such as those shown in this thesis.

7.1.3 Finding Unknown Media Objects (Publishing)

The use cases introduces a multinational media conglomerate that manages a large, constantly updated database of media objects, such as books, movies and music tracks. As the conglomerate has rapidly expanded over the recent years, different parts of the database use different ontologies. To bridge the gap between these different parts, appropriate assertions using the vocabulary of RDF Schema have been added.

```
@prefix baf:     <http://big-accounting-firm.example/scheme/1.0/#> .
@prefix bmc:     <http://big-media-conglomerate.example/ontology/#> .
@prefix dc:      <http://purl.org/dc/elements/1.1/> .

[]
    baf:dollarPrice  "29.99" ;
    bmc:objectName  "J to the LO" ;
    dc:author   <http://big-media.example/author/1929/> .
```

One of the conglomerate's editors needs to be informed about new titles that match certain criteria such as author, title and price range. For this purpose he creates a query that is regularly executed executed against the conglomerate's database and notifies the editor in form of web pages describing the matched items.

The ability to infer knowledge not explicitly included in a graph is one of the most important principles of RDF. This use case utilizes the `subPropertyOf` property from RDF Schema to specify relationships between properties, so that queries using a specific property match not only statements using the exact property but also for statements expressed using its subproperties. Two different approaches are available for incorporating inferencing into RDF queries using Xcerpt. One approach is to manually implement the desired inferencing on top of the raw RDF data that includes only explicitly asserted statements.

```
GOAL
  ns-default = "http://www.w3c.org/1999/xhtml"

  html{
    head{ title{ &join("Watched Media Item:", var TITLE) } },
    body{
      h1{ "uery Match!" },
      ul{
       li{ "Title: ", var TITLE },
       li{ "Author: ", var AUTHOR },
       li{ "Price: USD", var PRICE }
    }
  }
FROM
  and{
    optional RDF{{
      var OBJNAME:uri{{
         "http://www.w3.org/2000/01/rdf-schema#subPropertyOf":uri{{
           desc ( /.*/:uri > "http://www.w3.org/2000/01/rdf-schema#subPropertyOf":uri)*
            "http://big-media-conglomerate.example/ontology/#objectName":uri{{}}
         }}
      }}
    }},
    RDF{{
      identity var ID /.*/:/.*/{{
        or{
         "http://big-media-conglomerate.example/ontology/#objectName":uri{
            literal{ var TITLE as /Money/ }
         },
         var OBJNAME:uri{
           literal{ var TITLE as /Money/ }
         },
        },
        "http://purl.org/dc/elements/1.1/author":uri{
          literal{ var AUTHOR}
        },
        "http://big-accounting-firm.example/scheme/1.0/#dollarPrice":uri{
          var PRICE
        } where { var PRICE < 30 }
      }}
    }}
  }
END
```

In this example query, the URIs of properties that have `objectName` in their transitive closures over `subPropertyOf` predicates are bound to the variable `OBJNAME`. The query searches for media objects with a price less than 30 dollars and a title containing the word "Money". The title is checked against statements that have either `objectName` directly or one of its subproperties as property. The subproperties are specified as those containing the `objectName` property in their transitive closures over `subPropertyOf` statements and are bound to the variable `OBJNAME`. A simple XHTML page detailing the query result is constructed for each match.

The other approach to inferencing is to query the fully expanded RDF graph that contains all statements that are inferrable from explicitly asserted statements. By directing the query towards the graph built from triples augmented using the rules presented in Chapter 6, "RDF Schema Inferencing with Xcerpt", the user needs only to query for the `objectName` property and not concern himself with the inferencing.

```
GOAL
  ns-default = "http://www.w3c.org/1999/xhtml"

  html{
    head{ title{ &join("Watched Media Item:", var TITLE) } },
    body{
      h1{ "uery Match!" },
      ul{
       li{ "Title: ", var TITLE },
       li{ "Author: ", var AUTHOR },
       li{ "Price: USD", var PRICE }
    }
  }
FROM
  "rdfs":RDF{{
    identity var ID /.*/:/.*/{{
      "http://big-media-conglomerate.example/ontology/#objectName":uri{
        literal{ var TITLE as /Money/ }
      },
    },
    "http://purl.org/dc/elements/1.1/author":uri{
      literal{ var AUTHOR}
    },
    "http://big-accounting-firm.example/scheme/1.0/#dollarPrice":uri{
      var PRICE
    } where { var PRICE < 30 }
  }}
END
```

Instead of periodically evaluating the same query against a database that might or might not have changed since the last evaluation, a reactive system that queries changes in the database as they occur might be better suited for use case of this kind. XChange [53] is a reactive language for distributed event processing on the Web, based Xcerpt. A significant amount of the methods for querying RDF using Xcerpt is likely to be directly applicable to XChange and poses an interesting direction for future research.

7.1.4 Customizing Content Delivery (Device Independence)

A mapping service provides mobile users with navigational maps that can be displayed on mobile phones. In the use case's scenario, a user requests a map of the surroundings of a race track he's headed to. In order to chose an image adequate for the user's phone, the mapping service dereferences an URI where a profile describing the capabilities of the user's phone is stored, merging it with a more specific profile the user has sent along with his request.

The device profile is described using the UAProf schema defined by the Open Mobile Alliance, an industry consortium of mobile phone makers. Profiles using this schema are already being distributed by most major mobile phone makers. Below is an excerpt of an actual profile, available at the URI `http://mobileinternet.panasonicbox.com/UAprof/GD67/04.xml` in RDF/XML format.

```
@prefix prf:     <http://www.openmobilealliance.org/tech/profiles/uaprof/ccppschema-20021212#>.
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#>.
[]
    prf:BitsPerPixel "8"^^xsd:int ;
    prf:ColorCapable "true"^^xsd:boolean ;
    prf:CPU "Arm 7" ;
    prf:ImageCapable "true"^^xsd:boolean ;
    prf:ScreenSize "101x80" ;
    prf:SoundOutputCapable "false"^^xsd:boolean ;
    prf:Vendor "Panasonic" ;
    prf:Model "GD67" .
```

The mapping service has a large collection of maps in various sizes and numbers of colors, stored on the service's web servers. The maps are described by an RDF graph identified by the URI `http://example.com/maps.rdf`, including assertions about which maps depict a certain resource.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix img: <http://example.org/image#>.

<http://example.org/racetrack>
  foaf:depiction <http://example.org/racetrackmap.color.jpg>;
  foaf:depiction <http://example.org/racetrackmap.bw.jpg> .

<http://example.org/racetrackmap.bw.jpg>
  img:width "120"^^xsd:int ;
  img:height "100"^^xsd:int ;
  img:hasColor "false"^^xsd:boolean.

<http://example.org/racetrackmap.color.jpg>
  img:width "100"^^xsd:int ;
  img:height "75"^^xsd:int ;
  img:hasColor "true"^^xsd:boolean.
```

The user submits the following information in order to receive an image showing the desired map:

- The URI of his phone's default profile (`http://mobileinternet.panasonic`

box.com/UAprof/GD67/04.xml)

- An RDF graph describing his phone's current settings, overriding his phone's default profile. It is temporally stored at the location file:///tmp/36e7ba4f.rdf
- The URI of the resource he is requesting a map of (http://example.org/racetrack). How the phone maps a real-world location to this URI is out of scope of this use case.

The use cases integrates RDF data from multiple sources, including locally stored data, data retrieved over the Internet and data provided on a query-by-query basis by the user. The query shown here implements the use case using six Xcerpt rules. The first three rules specify the RDF graphs to be queried: the user's phone's basic profile, a temporarily stored local file containing the user's override profile and the mapping service's map description database. The fourth and fifth rules extract the actual property-value pairs from the default and override profiles into DEFAULT-PROFILE and USER-PROFILE data terms, which are merged into PROFILE data terms in rule six, using a conditional query. The GOAL rule implements the actual query, querying the merged profile for the screen size and color capabilities of the user's phone, employing a regular expression to extract height and width of the screen from a literal containing both, and uses the data to search for a map that is adequate for it.

```
% Rule No. 1, load default device profile
CONSTRUCT
  RDF-FILE[ "http://mobileinternet.panasonicbox.com/UAprof/GD67/04.xml", var Root ]
FROM
  in{
   resource{ "http://mobileinternet.panasonicbox.com/UAprof/GD67/04.xml" },
   var Root
  }
END

% Rule No. 2, load override profile
CONSTRUCT
  RDF-FILE[ "file:///tmp/36e7ba4f.rdf", var Root ]
FROM
  in{
   resource{ "file:///tmp/36e7ba4f.rdf" },
   var Root
  }
END

% Rule No. 3, load database of maps
CONSTRUCT
  CONSTRUCT
  RDF-FILE[ "http://example.com/maps.rdf", var Root ]
FROM
  in{
   resource{ "http://example.com/maps.rdf" },
   var Root
  }
END

% Rule No. 4, abstract data out from default profile
CONSTRUCT
```

```
  DEFAULT-PROFILE[ var Prof,  var VALUE ]
FROM
  RDF{{
    attributes{{
      origin{ "http://mobileinternet.panasonicbox.com/UAprof/GD67/04.xml" }
    }},
    /.*/:/.*/{{
      /http://www.openmobilealliance.org/tech/profiles/uaprof/.*#(var Prof as .*)/:uri{
       literal{ var VALUE }
    }}
  }}
END

% Rule No. 5, abstract data out from override profile
CONSTRUCT
  USER-PROFILE[ var PROFILE, var VALUE ]
CONSTRUCT
    RDF{{
      attributes{{ origin{ "file:///tmp/36e7ba4f.rdf" } }},
      /.*/:/.*/{{
        /http:www.openmobilealliance.org/tech/profiles/uaprof/.*#(var PROF as .*)/:uri{
       literal{ var VALUE }
     }}
    }}
  ]]
END

% Rule No. 6, merge default and override profiles
CONSTRUCT
  PROFILE[ var KEY, var VALUE ]
FROM
  if USER-PROFILE[ var KEY, var VALUE ] then
     USER-PROFILE[ var KEY, var VALUE ]
  else
     DEFAULT-PROFILE[ var VENDOR, var MODEL, var KEY, var VALUE]
END

% Rule No. 7, use merged profiled data to query appropriate maps
GOAL
  var IMAGE_URI
FROM
  and[
    PROFILE[ "ScreenSize", /(var SCREEN_W as [:digit:]+)x(var SCREEN_HT as [:digit:]+)/ ],
    PROFILE[ "ColorCapable". var COLOR ],
    RDF{{
      "http://example.org/racetrack":uri{{
        "http://xmlns.com/foaf/0.1/depiction":uri{
          var IMAGE_URI:uri{{
            "http://example.org/image#width":uri{ literal{ var IMAGE_W } },
            "http://example.org/image#height":uri{ literal{ var IMAGE_H } },
            "http://example.org/image#color":uri{ literal{ var COLOR } }
          }} where { var IMAGE_W <= var SCREEN_W and
                     var IMAGE_H <= var SCREEN_H
          }
        }
      }}
    }}
  ]
END
```

7.2 Use Cases that integrate RDF and XML

7.2.1 Querying XML documents and their meta data

A publisher has accumulated a large collection of technical articles it supplies to numerous affiliated web sites. The company keeps track of the articles, using an RDF knowledge base that describes title, authors and subjects of each article.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix dc: <http://xmlns.com/dc/elements/1.1/>.
@prefix pub: <http://example.com/publishing/>.

<http://exampletechupdate.com/article/how-to-sink-a-ship.html>
  rdf:type pub:Article;
  dc:subject "Ship";
  dc:title "How to sink a Pirate Ship";
  dc:creator _:author1 .

_:author1
  foaf:name "Peter Pan";
  foaf:make <http://exampletechupdate.com/article/how-to-sink-a-ship.html> .
```

One day, the chief editor of the company receives a letter from a law firm, warning the company of inappropriate use of the trade mark KOLA™ in numerous of its articles, without naming any particular articles. The chief editor instructs his staff to quickly prepare a list of all articles making use of the trade mark sorted by author and citing the paragraphs containing the trademark.

The use cases demonstrates how a Xcerpt can be used to control a query against XML documents using meta data described using RDF. The query obtains meta data as well as the URIs of articles from the company's RDF knowledge base and uses the URIs as arguments to the `resource` specifications, in order to retrieve the articles over the Internet and search them for paragraphs that contain the allegedly infringing trademark. A HTML report in is constructed using the selected information.

```
GOAL
  out{
    resource{ "file:///home/editor/greylist.html" },
    html{
      head{ title{ "Potentially Infringing Articles" } },
      body{
        h1{ "Articles containing the word KOLA" },
        ul{
          all li{
            var TITLE,
            p{ "Author(s):", &join( all var AUTHOR ) },
            all blockquote{ var PARA }
          }
        }
      }
    }
  }
FROM
  and{
    RDF{{
```

```
      var URI:uri{{
        "http://www.w3.org/1999/02/22-rdf-syntax-ns#type":uri{
          "http://example.com/publishing/Article":uri{{}}
        },
        "http://purl.org/dc/elements/1.1/creator":uri{{
          /.*/:/.*/{{
            "http://xmlns.com/foaf/0.1/name":uri{ literal{ var AUTHOR }  }
          }}
        }},
        "http://purl.org/dc/elements/1.1/title":uri{ literal{ var TITLE } }
      }}
    }},
    in{
      resource{ var URI },
      desc var PARA as p{ /.* KOLA .*/ }
    }
  }
END
```

7.2.2 Transparently integrating XML and RDF Querying

One of the editors of the same publisher is planning to use a photograph of a ship as background image for an article he is currently working on. He uses his company's article database to search for pictures used in past articles that covered ships. As he is not sure yet what kind of ships he wants to use, he includes hyponyms of "ship" in his search. To simplify his query, the editor uses rules from a library of Xcerpt rules created by his company's IT Staff, copying rules providing a simple interface to articles in the company's knowledge base and an interface for hyponyms into his program.

The use case demonstrates how parts of the query can be outsourced as re-usable libraries of rules, providing simple views on complex data, freeing users of the need to concern themselves with details. One of the rules used in this use case provides an XML-like view of RDF data, for users familiar only with XML. Another rule provides a simple collection of hyponyms, that is actually extracted from the Wordnet vocabulary. Developers of such libraries can in turn draw upon the rules presented in this thesis, to gain independence from RDF serialization formats and perform RDF Schema inferencing. It also showcases the advantages of the ability to uniformely query both RDF and XML, leveraging information from XML data, RDF meta data about it and an RDF ontology.

```
% Rule No. 1, query for <img/>-es in articles about ships
GOAL
  out{
    resource{ "file:///home/editor/shipimages.html" },
    html{
      head{ title{ "Images related to Ships" } },
      body{
        h1{ "Images related to Ships" },
        all p{
          img{ src{ var FILENAME } }
        }
      }
```

```
    }
  }
FROM
  and[
    HYPONYM[ "Ship", var HYPONYM ],
    Document{{
      uri{ var URI },
      subject{ var HYPONYM },
    }},
    in{
      resource{ var URI },
      desc img{{
        attributes{{ src{ var FILENAME } }}
      }}
    }
  ]
END

% Rule No. 2, a xml-like view of specific RDF data
CONSTRUCT
  Document{{
    uri{ var URI },
    title{ var TITLE},
    all subject{ var SUBJECT },
    all author{ var AUTHOR }
  }}
FROM
  RDF{{
    var URI:uri{{
      "http://www.w3.org/1999/02/22-rdf-syntax-ns#type":uri{
        "http://example.com/publishing/Article":uri{{}}
      },
      "http://purl.org/dc/elements/1.1/subject":uri{ var SUBJECT },
      "http://purl.org/dc/elements/1.1/creator":uri{{
        /.*/:/.*/{{
          "http://xmlns.com/foaf/0.1/name":uri{ literal{ var AUTHOR }  }
        }}
      }},
      "http://purl.org/dc/elements/1.1/title":uri{ literal{ var TITLE } }
    }}
  }}
END

% Rule No. 3, hyponym database, based on the Wordnet vocabulary
CONSTRUCT
  HYPONYM[ var WORD, var HYPONYM]
FROM
  and[
    "rdfs":RDF-TRIPLE[
      var HYPONYM_URI as /http://xmlns.com/wordnet/1.6/.*/:uri{},
      "http://www.w3.org/2000/01/rdf-schema#subClassOf":uri{},
      var WORD_URI as /http://xmlns.com/wordnet/1.6/.*/:uri{}
    ],
    "rdfs":RDF-TRIPLE[
      var HYPONYM_URI,
      "http://www.w3.org/2000/01/rdf-schema#label":uri{},
      literal{ var HYPONYM }
    ],
    "rdfs":RDF-TRIPLE[
      var WORD_URI,
      "http://www.w3.org/2000/01/rdf-schema#label":uri{},
      literal{ var WORD }
    ]
  ]
```

```
END
```

CHAPTER EIGHT

Proposed Extensions to Xcerpt

Several limitations of Xcerpt have been encountered during the course of this thesis, ranging from the inconvenience of performing simple structural recursion to the inability to construct graphs as the result of queries. In this chapter, extensions to Xcerpt are proposed, both in form of completely new constructs as well as changes to existing constructs.

The previous chapters have been written using an extended Xcerpt language that included the features proposed here on top of the currently specified language.

8.1 Term Identity Specification

When doing structural recursion over an input document, it is often necessary to differentiate between two subterms that share the exact same structure but appear at different positions in the input document. For instance, without considering the identity of subterms, the `all` construct would coalesce structurally and value equivalent bindings and join operations using `and` could lead to unexpected results.

A new construct `identity` is proposed, allowing access to identifiers uniquely identifying each subterm. A query term of the form `identity` *`var ID`* `t{{}}` denotes that the query matches the same data terms as the query `t{{}}`, binding an identifier unique to each data term to the variable `ID`.

Identity specification is admissible in any kinds of query term specifications, e.g in ordered and unordered as well as total and partial query terms as well as in combination with constructs like `desc`. Note, however, that it is not possible to specify a constant instead of a variable, as the structure and form of the identifiers is implementation-dependent and the identifiers are only suited for comparison between multiple bindings.

The identity of a data term is a property of the term that does not change for the entire lifetime of the data term, similar to other properties like sorted/unsorted or total/incomplete. The uniqueness of the identifiers defined as follows:

- an identifier for a data term originating in an external resource is always the same, even if the same resource is retrieved in multiple rules
- the identifier does not change when a data term is matched by and passed through a rule
- when multiple structurally and value equivalent data terms are mergen by `all`, the resulting data term and its subterms have the identity of any one of the merged data terms and its subterms.
- the identifier for a single data term only needs to be unique for a single query of a single instance of an Xcerpt implementation

One possible implementation is to construct the identifier for dataterms from external resources using the URI of the resource and a data term counter based on document order. Identifiers of data terms constructed in rule heads could be generated using a rule identifier and a running counter for each newly constructed. data term.

Example 8.1.1 Grouping by Identity

```
GOAL
  list{
    all var NAME
  } group by { var IDENT }
FROM
  desc identity var IDENT var NAME as name{{}}
END
```

Example 8.1.1 is a query listing all `name` subterms, using the `identity` construct for grouping, in order to avoid the elimination of duplicates by `all`.

Other possible uses for term identity include the generation of unique values for ID attributes or the identification of data terms that are matched by two different rules. Example 8.1.2 shows such a `GOAL` rule that matches data terms that satisfy two different conditions, each checked in it's own `CONSTRUCT` rule. Without the *join* over the identity, the result would include such subterms that match only one condition, while another subterm with identical structure matches the other condition.

Example 8.1.2 A *join* over Identity

```
GOAL
  R{ all var TERM }
FROM
  and[
   CONDITION_A[ identity var IDENT var TERM ],
   CONDITION_B[ identity var IDENT var TERM ]
  ]
END
```

8.2 Negation of Regular Expressions

Regular Expressions are a powerful and convenient tool for specifying advanced queries against namespaces, labels and strings. Xcerpt currently supports the POSIX regular expression language [54] with some Xcerpt-specific extensions. One feature lacking from the POSIX regular expression language, and therefore from Xcerpt, is the ability to negate an expression, in order to accept everything *except* the strings accepted by the expression.

Using existing Xcerpt constructs, "all except" queries can be expressed using a conjunction over a query matching all terms and another negated query matching the undesired terms. Example 8.2.1 shows such a query, which selects all chapter titles of a text document that do *not* contain the substring "Xcerpt". Using conjunctions, however, is redundant in the wording of the query and performs poorly performance-wise, as two almost identical queries have to be independently performed and then joined.

Example 8.2.1 "all except' Query using Conjunction

```
and[
  desc chapter{{ title{ var TITLE } }}
  not desc chapter{{ title{ var TITLE → /.*Xcerpt*/ } }}
]
```

In order to simplify writing such common queries, a simple method of negating a regular expression is proposed, allowing negation of regular expressions by prefixing the expression with an exclamation mark (`!`), an operator that is used commonly by programming languages for negation. A query containing a negated regular expression matches a data term only when the regular expression does not accept the appropriate namespace, label or string it is matched against. Like in negated subterms and queries, negated regular expressions may contain variables but these do not yield bindings and accordingly need to also appear elsewhere in the query in a non-negated context.

Example 8.2.2 "all except" Query with negated Regular Expression

```
desc chapter{{
  title {{ !/.*Xcerpt.*/ }}
}}
```

Example 8.2.2 shows the same query as before, this time using a negated regular expression. Not only is this query much more concise, it can also be evaluated more efficiently.

Alternately, Xcerpt could adopt a regular expression language more powerful than the POSIX regular expression language. Many modern regular expression engines like PCRE [55], Oniguruma [56] and the regexp library for Haskell [57], amongst others, support *forward lookahead asserts* (and the negation thereof) as part of their regular expression languages. Forward lookahead asserts have the form `(?=sub)` and match zero characters that are followed by "sub". The negated form `(?!sub)` matches zero

characters that are not followed by "sub", effectively allowing subexpression negation. Nevertheless, extended regular expression languages might have a very high complexity and therefore, careful consideration of the consequences is necessary before such an extended language can be accepted as part of the Xcerpt language. The simple negation proposed here, on the other hand, would not add any additional complexity and is easy to understand.

8.3 Qualified Decendant

Structural recursion is a method commonly used for querying and transformation of XML documents. Not only is structural recursion useful for processing documents with recursive structure definitions, like RDF/XML, it is also necessary for executing queries like "all male descendants" against hierarchical databases. Query languages like XSLT [8] are actually built around the concept of structural recursion, recursively applying templates to subtrees based on the notion of contexts. Xcerpt on the other hand is context-free and always matches against whole data terms, making it difficult to write queries which require structural recursion. A new construct is proposed, simplifying common cases of structural recursion and greatly improving performance.

8.3.1 Existing Methods for Structural Recursion

For example, given the data term of Figure 8.1, and the need to list all "a-descendants" under `root`, there are three possible strategies for formulating a query in Xcerpt. With "a-descendant" meaning all subterms solely reachable via zero or more terms labeled `a`. The path to such subterms could be described as `(a)*`, using regular expression-like syntax. In each case, the desired result would be the subterms `a{b{"1"}}`, `b{"1"}`, `a{c{a{"2"}}}`, `c{a{"2"}}` and `d{"3"}`, but not include `a{"2"}`.

Figure 8.1 Sample Data Term

```
root{
  a{
    b{ "1" },
  },
  a {
    c {
      a{ "2" }
    }
  },
  d{ "3" }
}
```

8.3.1.1 Recursive Rules

One possible strategy is to use multiple, recursive rules, as shown in Example 8.3.1. In this example, all a-descendants are collected into top-level data terms labeled ADESC. The direct children of the root (zero a-terms on the path) are collected in the rule marked ①, while the children of each a-child of them are recursively collected in the rule marked ②, using a join with data from outside the recursion due to the requirement of range restriction, in order for the rule to be backward-chainable (cf. [9], Section 8.3). This requirement is not only unintuitive for users, it is also very ineffective performance-wise, as a join against all subterms of the input must be executed for each recursion.

Example 8.3.1 Structural Recursion

```
GOAL
  R { all var AD }
FROM
  ADESC[[ var AD ]]
END

%%%% base case  %%%%%
CONSTRUCT ①
  ADESC[ var CHILD ]
FROM
  root {{ var CHILD }}
  }}
END

%%%% recursion, all a-children of a-descendants %%%%
CONSTRUCT ②
  ADESC[ var A-CHILD ]
FROM
  and[
    root{{ desc var A-CHILD }},
    ADESC[ a{{ var A-CHILD }} ]
  ]
END
```

8.3.1.2 Recursive Rules using Explicit Simulation Unification

The overhead of joins can be avoided by reformulating the rules to use *explicit simulation unification*, denoted by the operator :<. Explicit simulation unification is an experimental feature that allows to restrict the structure of the term occurring to the right of the operator to those that are matched by the query term to the left of the operator. It can be abused to formulate rules similar to functions of functional programming.

Example 8.3.2 shows an "a-descendant" query using explicit simulation unification. The GOAL (①) rule selects all direct children under the root for querying and then uses them as "argument" to the ADESC "function". The second child of ADESC contains the "return value". One rule (②) denotes the base case of the recursion: when INPUT is not an a term, the result is empty. Another rule (③) handles the recursive case: when INPUT

has the form `a{{}}`, the result includes all children of the `a` term and the result of recursively applying `ADESC` to each child.

Example 8.3.2 Structural Recursion using Explicit Simulation Unification

```
GOAL ①
 R{ all optional var BRANCHES, var AD }
FROM
 and[
   root{{ var BRANCHES }},
   ADESC[ var BRANCHES, result{{ optional var AD }} ]
 ]
END

%% base case: no result for non-a children
CONSTRUCT ②
  ADESC[ var INPUT, result{} ]
 FROM
  !/a/{{}} :< var INPUT
END

%% recursion: result is all a-children and ADESC of them
CONSTRUCT ③
  ADESC[ var INPUT, result{{ all var ACHILD, all optional var ACHILD_ADESC  }} ]
FROM
  and[
   a{{ var ACHILD }} :< var INPUT,
   ADESC[ var ACHILD, result{{ optional var ACHILD_ADESC }} ]
  ]
END
```

However, *explicit simulation unification* is an experimental feature that is considered a *hack* by its author and unlikely to become part of the standard language, as it is only usable with backward-chaining implementations and lacks clear semantics. Its use is discouraged.

8.3.1.3 Using Complementation

Maarten Marx has shown in [58] that subsets of nodes of XML-like trees definable using first-order logic formulas can be expressed in *Core XPath* extended with complementation. The "a-descendant" query expressed as first-order logic formula is the set of all terms y such that

x `descendant` $y \wedge \forall x((x$ `descendant` $z \wedge z$ `descendant` $y) \rightarrow$ `label`$(z) =$ "a")

where `x` and `z` are terms, `descendant` is the binary descendant-relation between terms and `label` is a function returning a term's label.

Translating the corresponding XPath expression to Xcerpt yields the query shown in Example 8.3.3, which subtracts all subterms that have a non-`a` ancestor from the set of all descendants. The query is obviously not straight-forward to write and nowhere near efficient due to the nesting of `desc /.*/{{}}` and negation.

Example 8.3.3 "a-descendant" using Complementation

```
GOAL
  R {
   all var AD
  }
FROM
  root{{
    and[
      desc var AD as /.*/{{}},
      not desc /.*/{{
        !/a/ {{
          desc var AD as /.*/{{}}
        }}
      }}
    ]
  }}
END
```

8.3.2 Proposal

Structural recursion in Xcerpt is neither easy to write for users, nor efficient. The descendant construct `desc` offers simple structural recursion over a data term. The query `desc t{{}}` matches all data terms that contain a subterm that is matched by `t{{}}`, at arbitrary depth. From an operational view, the `desc` construct recursively searches for `t{{}}`, recursing into all subterms with unlimited recursion depth. It is very useful, but not suited for uses such as the one portrayed above.

An extension to the descendant construct is proposed, allowing to restrict it in the subterms it recurses into and in recursion depth by specifying a *path pattern*. The *qualified descendant* `desc` *pattern* `t{{}}` matches all data terms that contain a subterm that is matched by `t{{}}` and the path to it is matched by *pattern*. Path patterns are specified as sequence of term specifiers separated by > and a repetition quantifier.

Figure 8.2 Genealogy database

```
<Person name="John" bloodtype="O">
  <son>
    <Person name="Peter" bloodtype="O">
      <son><Person name="Michael"></son>
    <Person name="Fred" bloodtype="A">
      <son><Person name="Robert" bloodtype="AB"></son>
    </Person>
  </son>
  <daughter>
    <Person name="Elizabeth" bloodtype="A">
      <son><Person name="George" /></son>
    </Person>
  </daughter>
</Person>
```

For example, given the genealogy database shown in Figure 8.2 (written in XML notation for the compact representation of attributes), a query for all descendants on male lines (`Person` terms connected only by `son` terms) would be

```
desc (Person > son)* var MD as Person{{}}
```

matching John, Peter and Fred, but not George.

8.3.2.1 Term Specifiers

Term specifiers are patterns of the form

namespace:*label*[*attributens*:*attributename*=*vale*]

specifying namespace, label and optionally attributes that a term must have in order to match the corresponding part of the sequence. Each component is given either as a constant or a POSIX regular expression. Any part other than the label defaults back to wildcard if omitted. The label can not be omitted.

If the query for all descendants on male lines was to be restricted to those branches of the family that inherit the bloodtypes O, the pattern needs to be changed to

```
(Person[bloodtype="O"] > son)*
```

Multiple alternative term specifies are also allowed by separating with the OR operator(|). A pattern for the branches that inherit the bloodtypes B or O would be

```
(Person[bloodtype="O"] | Person[bloodtype="B"] > son)*
```

8.3.2.2 Repetition

How often the pattern must/can be repeated is specified after a bracked-enclosed sequence of term specifiers in form of a repetition quantifier. Allowed values are

- * (zero or more times)
- + (one or more times)
- {*n*} (exactly *n* times)
- {*n*,*m*} (at least *n* but not more than *m* times)
- {*n*,} (at least *n* times)
- {,*m*} (*m* times at the most)

A pattern for descendants on male lines up to the fifth degree would be

```
(Person > son){,5}
```

8.3.2.3 Grammar

The formal grammar of the qualified descendant construct is defined in Figure 8.3 using a simple Extended Backus-Naur Form (EBNF) notation [59]. Expressions between < and > are non-terminal symbols, while terminal symbols are marked using double quotes. The non-terminal symbols `query-term`, `label` and `string` correspond to the respective constructs of the overall Xcerpt language and are not characterized in further detail here. The non-terminal symbol `regexp-without-var` stands for the regular expression language used by Xcerpt, but without variables. The qualified descendant construct may be used anywhere within queries where the traditional descendant construct is allowed.

Figure 8.3 EBNF Grammar for the Qualified Descendant Construct

```
<qdesc>      ::= "desc" <pattern> <query-term>
<pattern> ::= "(" <sequence> ")" <repetition>
<sequence>   ::= <alternative> (">" <alternative>)*
<alternative>::= <termspec> ( "|" <termspec>)*
               | "(" <termspec> ( "|" <termspec>)* ")"
<termspec>   ::= (<namespace> ":")? <localname> (<attrspec>)*
<attrspec>   ::= "[" (<namespace> ":")? <localname> ("=" <attvalue>)? "]"

<repetition> ::= "*" | "+" | "{" <numspec> "}"
<numspec>    ::= <posint> | <posint> "," <posint>
               | <posint> "," | "," <posint>

<posint>     ::= [0-9]+
<namespace>  ::= <label> | <string> | <regexp-without-var>
<localname>  ::= <label> | <regexp-without-var>

<attvalue>   ::= <label> | <string> | <regexp-without-var>
```

8.3.2.4 A-Descendant using Qualified Descendant

Example 8.3.4 shows the "a-descendant" query from above, implemented using the qualified descendant construct.

Example 8.3.4 "a-descendant" using Qualified Descendant

```
GOAL
  R { all var ADESC }
FROM
  root{{
    desc (a)* var ADESC
  }}
END
```

8.4 Construction of Graphs

Xcerpt is a query language for semi-structured data, including cyclic graphs. References are used to textually represent cross-connecting edges. They are automatically dereferenced when queried and are undistinguishable to regular parent-child relationships. Xcerpt supports loading and querying of external resources that contain data terms with such references. However, only trees can be constructed as the result of a query, because references are currently not allowed in construct terms. This restriction has proven to be an obstacle during this thesis while reconstructing RDF-graphs from their serializations.

It is proposed to lift the above-mentioned restriction and allow references in construct terms, so that graphs can be constructed. As rules already may output graphs, by passing graph structures originating from external resources unchanged, no adverse side-effects to the whole of Xcerpt are apparent. In addition to static identifiers allowed in data terms, variables should be allowed in place of identifiers. Their respective bindings then become identifiers, when the rule is evaluated against a data term, allowing results from multiple bindings to reference each other, when used in combination with `all`. Section 5.3.4 uses this to transform RDF triples into a RDF graph.

This extensions not only opens the possibility of creating graphs as the result of rules for the first time, it also greatly simplifies the construction of trees in some cases which would otherwise require complex recursion. Such a case has been previously described by Michael Kraus in his collection of Use Cases for Xcerpt [23].

CHAPTER NINE

Conclusions

Starting from a discussion about how RDF data is perceived by its users, a method for querying RDF using Xcerpt has been developed in this thesis. Two different representations of RDF data as Xcerpt data terms are proposed, together with a collection of rules implementing the representations as queryable views over diverse serialization formats. In contrast to other proposals for querying RDF, either using languages specifically designed for querying RDF or XML query languages, Xcerpt and the methods proposed in this thesis allow access to both XML and RDF data in a natural and uniform manner. The flexibility to support multiple representations of RDF and XML data as well as the ability to extend support to arbitrary serialization formats using rules written in the language itself, is unique amongst query languages for the Web.

Beyond the querying of static RDF data, a subset of RDF Schema inferencing has also been implemented using Xcerpt. Instead of realizing inferencing using the underlying programming languages of evaluation engines, the approach allows adoption to and integration of heterogeneous data, including custom inferencing for specific applications.

It is the strongest conviction of the author of this thesis that Xcerpt is a suitable candidate for a truly versatile Web query language, allowing efficient and effective access to any kind of data on the Web: effective in the integration and leveraging of data from different data models and representations and efficient in its flexibility to adopt to new technologies and formalisms, covering a broad spectrum with a single language. This thesis hopes to have contributed to the further development of Xcerpt by demonstrating its potential for querying and integrating data on the Semantic Web.

APPENDIX A

Rules for Parsing RDF/XML

A.1 Structural Recursion

```
% Top-Level Subjects
CONSTRUCT
  RDFXML-SUBJECT[ var ORIGIN, var SUB ]
FROM
  ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

  RDF-FILE[
    var ORIGIN,
    rdf:RDF{{
      var SUB
    }}
  ]
END

% Recursion, subject-predicate-object nesting
CONSTRUCT
  RDFXML-SUBJECT[ var ORIGIN, var OBJECT ]
FROM
  ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

  and[
    RDF-FILE[
      var ORIGIN,
      rdf:RDF{{ desc identity var IDENT var OBJECT as /.*/:/.*/{{}} }}
    ],
    RDFXML-SUBJECT[ var ORIGIN,
      /.*/:/.*/{{      % subject
        /.*/:/.*/{{    % predicate with child as object
          without attributes{{ rdf:parseType{ "Literal" } }},
          without attributes{{ rdf:parseType{ "Resource"} }},
          identity var IDENT var OBJECT
        }}
      }}
    ]
  ]
END
```

A.2 Mapping RDF node to Xcerpt data term

```
% regular URI reference
CONSTRUCT
  RDFXML-NODE[ var ORIGIN, var NODE, var URI:uri{} ]
FROM
    ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

    RDFXML-SUBJECT[
      var ORIGIN,
      var NODE as /.*/:/.*/{{
        attributes{{ rdf:about{ var URI } }}
      }}
    ]
END

% relative URI reference
CONSTRUCT
  RDFXML-NODE[ var ORIGIN, var NODE, &join( var ORIGIN, "#", var ID):uri{} ]
FROM
    ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

    RDFXML-SUBJECT[
      var ORIGIN,
      var NODE as /.*/:/.*/{{
        attributes{{ rdf:ID{ var ID } }}
      }}
    ]
END

% blank node with explicit identifier
CONSTRUCT
  RDFXML-NODE[ var ORIGIN, var NODE, var NODEID:blank{} ]
FROM
    ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

    RDFXML-SUBJECT[
      var ORIGIN,
      var NODE as /.*/:/.*/{{
        attributes{{ rdf:blankID{ var NODEID } }}
      }}
    ]
END

% blank node without explicit identifier
CONSTRUCT
  RDFXML-NODE[ var ORIGIN, var NODE, var NIDENT:blank{} ]
FROM
    ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

    RDFXML-SUBJECT[
      var ORIGIN,
      identity var NIDENT var NODE as /.*/:/.*/{{
        attributes{{
          without rdf:about{{}},
          without rdf:nodeID{{}},
          without rdf:ID{{}}
        }}
      }}
    ]
END
```

A.3 Node and Property Elements, Node as Object

implements RDF/XML Syntax Specification Sections 2.2, 2.3

```
CONSTRUCT
  RDF-TRIPLE[
    attributes{ origin{var ORIGIN} },
    var SUBJECT,
    &join( var P_NS, var P_LN):uri{},
    var OBJECT
  ]
FROM
  ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

  and[
    RDFXML-SUBJECT[
      var ORIGIN,
      identity var SIDENT var SNODE as /.*/:/.*/{{
        var P_NS:var P_LN{
          without attributes{{ rdf:parseType{{}} }},
          identity var OIDENT var ONODE as /.*/:/.*/{{}}
        }
      }}
    ],
    RDFXML-NODE[ var ORIGIN, identity var SIDENT var SNODE, var SUBJECT ],
    RDFXML-NODE[ var ORIGIN, identity var OIDENT var ONODE, var OBJECT ]
  ]
END
```

A.4 Node and Property Elements, Literal as Object

implements RDF/XML Syntax Specification, Sections 2.2, 2.3, 2.7, 2.9

```
CONSTRUCT
  RDF-TRIPLE[
    attributes{ origin{var ORIGIN} },
    var SUBJECT,
    &join( var P_NS, var P_LN):uri{},
    literal{
      attributes{{
        optional "http://www.w3.org/XML/1998/namespacelang":uri{  var LANG },
        optional "http://www.w3.org/1999/02/22-rdf-syntax-ns#datatype":uri{ var TYPE }
      }},
      var O_TEXT,
    }
  ]
FROM
  ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  ns-prefix xml = "http://www.w3.org/XML/1998/namespace"

  and[
    RDFXML-SUBJECT[
      var ORIGIN,
      identity var SIDENT var SNODE as /.*/:/.*/{{
        var P_NS:var P_LN{
          var O_TEXT as /.*/,
          attributes{{
            optional xml:lang{ var LANG },
```

```
            optional rdf:datatype{ var TYPE }
          }}
        }
      }}
    ],
    RDFXML-NODE[ var ORIGIN, identity var SIDENT var SNODE, var SUBJECT ]
  ]
END
```

A.5 Empty Property Elements

implements RDF/XML Syntax Specification, Section 2.4

```
CONSTRUCT
  RDF-TRIPLE[
    attributes{ origin{var ORIGIN} },
    var SUBJECT,
    &join( var P_NS, var P_LN):uri{},
    var O_URI:uri{}
  ]
FROM
  ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

  and[
    RDFXML-SUBJECT[
      var ORIGIN,
      identity var SIDENT var SNODE as /.*/:/.*/{{
        var P_NS:var P_LN{
          attributes{{ rdf:resource{ O_URI } }}
        }
      }}
    ],
    RDFXML-NODE[ var ORIGIN, identity var SIDENT var SNODE, var SUBJECT ]
  ]
END
```

A.6 Property Attributes

implements RDF/XML Syntax Specification, Section 2.5

```
CONSTRUCT
  RDF-TRIPLE[
    attributes{ origin{var ORIGIN} },
    var SUBJECT,
    &join( var P_NS, var P_LN):uri{},
    literal{
     optional attributes{{
       "http://www.w3.org/XML/1998/namespacelang":uri{ var LANG }
     }},
     var O_TEXT
    }
  ]
FROM
  ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  ns-prefix xml = "http://www.w3.org/XML/1998/namespace"
```

```
  and[
    RDFXML-SUBJECT[
      var ORIGIN,
      identity var SIDENT var SNODE as /.*/:/.*/{{
        attributes{{
          var P_NS:P_LN{ var O_TEXT },
          optional xml:lang{ var LANG }
        }} where {
          var P_NS != "http://www.w3.org/1999/02/22-rdf-syntax-ns#" and
          var P_NS != "http://www.w3.org/XML/1998/namespace"
        }
      }}
    ],
    RDFXML-NODE[ var ORIGIN, identity var SIDENT var SNODE, var SUBJECT ]
END
```

A.7 XML literals, `parseType=Literal`

implements RDF/XML Syntax Specification, Section 2.8

```
CONSTRUCT
  RDF-TRIPLE[
    attributes{ origin{var ORIGIN} },
    var SUBJECT,
    &join( var P_NS, var P_LN):uri{},
    literal{
      attributes{ "http://www.w3.org/1999/02/22-rdf-syntax-ns#datatype":uri{
        "http://www.w3.org/1999/02/22-rdf-syntax-ns#XMLLiteral"
      }},
      var O_XML
    }
  ]
FROM
  ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

  and[
    RDFXML-SUBJECT[
      var ORIGIN,
      identity var SIDENT var SNODE as /.*/:/.*/{{
        var P_NS:P_LN{{
          attributes{{ rdf:parseType{ "Literal" } }},
          var O_XML as /.*/:/.*/{{}}
        }}
      }}
    ],
    RDFXML-NODE[ var ORIGIN, identity var SIDENT var SNODE, var SUBJECT ]
END
```

A.8 Omitting Blank Nodes, `parseType=Resource`

implements RDF/XML Syntax Specification, Section 2.11

```
% triple extraction
CONSTRUCT
```

```
  RDF-TRIPLE[
    attributes{ origin{var ORIGIN} },
    var SUBJECT,
    &join( var P_NS, var P_LN):uri{},
    var O_IDENT:blank{}
  ]
FROM
  ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

  and[
    RDFXML-SUBJECT[
      var ORIGIN,
      identity var SIDENT var SNODE as  /.*/:/.*/{{
        identity var O_IDENT var P_NS:P_LN{{
          attributes{{ rdf:parseType{ "Resource" } }},
        }}
      }}
    ],
    RDFXML-NODE[ var ORIGIN, identity var SIDENT var SNODE, var SUBJECT ]
  ]
END

%% recursion
CONSTRUCT
  RDFXML-SUBJECT[ var ORIGN, var BLANK ]
FROM
  and[
    rdf:RDF{{ desc identity var IDENT var BLANK as /.*/:/.*/{{}} }},
    RDFXML-SUBJECT[
      var ORIGIN,
      /.*/:/.*/{{
        var BLANK as /.*/:/.*/{{
          attributes{{ rdf:parseType{ "Resource" } }},
        }}
      }}
    ]
  ]
END
```

A.9 Proerpty Attributes on an empty Property Element

implements RDF/XML Syntax Specification, Section 2.12

```
%% subject with object as empty property element
CONSTRUCT
  RDF-TRIPLE[
    attributes{ origin{var ORIGIN} },
    var SUBJECT,
    &join( var P_NS, var P_LN):uri{},
    var O_IDENT:blank{}
  ]
FROM
  ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

  and[
    RDFXML-SUBJECT[
       var ORIGIN,
      identity var SIDENT var SNODE as  /.*/:/.*/{{
        identity var O_IDENT var P_NS:P_LN{
          attributes{{
```

```
            without rdf:resource{{}},
            without rdf:parseType{{}},
            var A_NS:/.*/{{}} % any other attribute
          }}
        } where {
          var P_NS != "http://www.w3.org/1999/02/22-rdf-syntax-ns#" and
          var P_NS != "http://www.w3.org/XML/1998/namespace"
        }
      }}
    ],
    RDFXML-NODE[ var ORIGIN, identity var SIDENT var SNODE, var SUBJECT ]
  ]
END

%% empty property element as subject to it's attributes
CONSTRUCT
  RDF-TRIPLE[
    attributes{ origin{var ORIGIN} },
    var S_IDENT:blank{},
    &join( var P_NS, var P_LN):uri{},
    link{ var O_TEXT }
  ]
FROM
  ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

  RDFXML-SUBJECT[
    var ORIGIN,
    /.*/:/.*/{{
      identity var S_IDENT /.*/:/.*/{
        attributes{{
          without rdf:resource{{}},
          without rdf:parseType{{}},
          var P_NS:var P_LN{ var O_TEXT } % any other attribute
        }} where {
          var P_NS != "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        }
      }
    }}
  ]

END
```

A.10 Typed Node Elements

implements RDF/XML Syntax Specification, Section 2.13

```
%% case 1, identified subject
CONSTRUCT
  RDF-TRIPLE[
    attributes{ origin{var ORIGIN} },
    var SUBJECT,
    "http://www.w3.org/1999/02/22-rdf-syntax-ns#type":uri{},
    &join( var S_CLASS_NS, var S_CLASS_LN ):uri{}
FROM
  ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

  and[
    RDFXML-SUBJECT[
      var ORIGIN,
      identity var SIDENT var SNODE as var S_CLASS_NS:var S_CLASS_LN {{
        without attributes{{ rdf:parseType{ "Resource" } }}
```

```
      }} where {
        var S_CLASS_NS != "http://www.w3.org/1999/02/22-rdf-syntax-ns#" and
        var S_CLASS_LN != "Description"
      }
    ],
    RDFXML-NODE[ var ORIGIN, identity var SIDENT var SNODE, var SUBJECT ]
  ]
END
```

A.11 Unimplemented Features

- Container Membership Property Elements (RDF/XML Specification Section 2.15)
- Collections `rdf:pareType=Collection` (RDF/XML Specification Section 2.16)
- Reifying Statements (RDF/XML Specification, Section 2.17)

Bibliography

[1] W3C, *HTML 4.01 Specification*, December 1999.

[2] W3C, *Extensible Markup Language (XML) 1.1*, February 2004.

[3] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," in *Scientific American*, May 2001.

[4] W3C, *Resource Description Framework (RDF): Concepts and Abstract Syntax*, February 2004.

[5] D. Brickley and L. Miller, *FOAF Vocabulary Specification*, May 2004.

[6] G. Beged-Dov, D. Brickley, R. Dornfest, I. Davis, L. Dodds, J. Eisenzopf, D. Galbraith, R. Guha, K. MacLeod, E. Miller, A. Swartz, and E. van der Vlist, *RDF Site Summary (RSS) 1.0*, 2000.

[7] W3C, *XQuery 1.0: An XML Query Language (Working Draft)*, July 2004.

[8] W3C, *XSL Transformations (XSLT) Version 1.0*, November 1999.

[9] S. Schaffert, *Xcert: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, Institute for Informatics, Ludwig-Maximilians-Universität München, October 2004.

[10] W3C, *RDF/XML Syntax Specification (Revised)*, February 2004.

[11] W3C, *RDF Vocabulary Description Language 1.0: RDF Schema*, February 2004.

[12] W3C, *RDF Semantics*, February 2004.

[13] W3C, *RDF Test Cases*, February 2004.

[14] W3C, *RDF Primer*, February 2004.

[15] T. Berners-Lee, R. Fielding, and L. Masinter, "RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax," August 1998.

[16] ISO/IEC, *ISO/IEC 639-1:2002 – Codes for the representation of names of languages – Part 1: Alpha-2 code*, 2002.

[17] W3C, *XML Schema Part 2: Datatypes*, May 2001.

[18] C. Fellbaum, ed., *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[19] D. Brickley, " Wordnet for the Web," August 2001.

[20] "DCMI term declarations represented in RDF schema language," March 2003.

[21] D. Brickley, "RDF: Understanding the Striped RDF/XML Syntax," October 2001.

[22] W3C, *XML Linking Language (XLink) Version 1.0*, June 2001.

[23] S. Kraus, "Use Cases für Xcerpt: Eine positionelle Anfrage- und Transformationssprache für das Web," 2004.

[24] S. Schaffert and F. Bry, "Querying the Web Reconsidered: A Practical Introduction to Xcerpt," in *Proceedings of Extreme Markup Languages 2004*, 2004.

[25] E. Dijkstra, *A Discipline of Programming*. Prentice Hall, 1976.

[26] F. Bry, T. Furche, L. Badea, C. Koch, S. Schaffert, and S. Berger, "Querying the Web Reconsidered: Design Principles for Versatile Web Query Languages," in *Journal of Semantic Web and Information Systems*, 2005.

[27] W3C, *Resource Description Framework (RDF) Model and Syntax Specification*, February 1999.

[28] O. Bolzer, "A Brief History of RDF Serialization Formats," July 2004.

[29] D. Beckett, "A retrospective on the development of the RDF/XML Revised Syntax," June 2003.

[30] T. Berners-Lee, "A strawman Unstriped syntax for RDF in XML," May 1999.

[31] A. Souzis, "RxML," November 2003.

[32] T. Berners-Lee, "Notation 3, an RDF language for the Semantic Web."

[33] D. Beckett, *Turtle - Terse RDF Triple Language*, February 2004.

[34] S. Russell, "Quads (rough draft)," 2002.

[35] C. Bizer, *The TriG Syntax*, April 2004.

[36] J. Carroll and P. Stickler, "TriX: RDF Triples in XML," May 2004.

[37] J. Carroll, C. Bizer, P. Hayes, and P. Stickler, "Named Graphs, Provenance and Trust," May 2004.

[38] D. Backett, "Modernising Semantic Web Markup," April 2004.

[39] T. Furche, F. Bry, S. Schaffert, R. Orsini, I. Horrocks, M. Krauss, and O. Bolzer, "Survey over Existing Query and Transformation Languages," Tech. Rep. I4-D1, 2004.

[40] M. Olson and U. Ogbuji, "Versa Specification," June 2004.

[41] D. Steer, "TreeHugger 0.1," September 2003.

[42] J. Robie, "The Syntactic Web: Syntax and Semantics on the Web," in *XML Conference and Exposition*, December 2001.

[43] M. Sintek and S. Decker, "TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web," in *Proceedings of the International Semantic Web Conference 2002*, pp. 364–378, 2002.

[44] M. Kifer, G. Lausen, and J. Wu, "Logical Foundations of Object-Oriented and Frame-Based Languages," vol. 42, pp. 741–843, July 1995.

[45] M. Marchiori, "Towards a People's Web: Metalog," April 2004.

[46] W3C, *SPARQL Query Language for RDF (W3C Working Draft)*, October 2004.

[47] L. Miller, A. Seaborne, and A. Reggiori, "Three Implementations of SquishQL, a Simple RDF Query Language," in *International Semantic Web Conference*, June 2002.

[48] A. Seaborne, "RDQL – A Query Language for RDF," January 2004.

[49] W3C, *Namespaces in XML 1.1*, February 2004.

[50] W3C, *RDF Data Access Use Cases and Requirements (W3C Working Draft)*, October 2004.

[51] W3C, *SPARQL Variable Binding Results XML Format (W3C Working Draft)*, December 2004.

[52] "Extending the RDFS Entailment Lemma," in *Proceedings of the International Semantic Web Conference 2004*, pp. 77–91, 2004.

[53] F. Bry and P.-L. Pătrânjan, "Reactivity on the Web: Paradigms and Applications of the Language XChange," in *Proceedings of 20th Annual ACM Symposium on Applied Computing*, March 2005.

[54] IEEE, *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Shell and Utilities, Issue 6*, 2001.

[55] P. Hazel, "PCRE - Perl Compatible Regular Expressions," October 2004.

[56] K. Kosako, "Oniguruma Regular Expressions," October 2004.

[57] M. Sage, "Regular Expression Library in Haskell," tech. rep., December 1996.

[58] M. Marx, "First order path in ordered trees," in *Proceedings ICDT 2005*, 2005.

[59] ISO/IEC, *ISO/IEC 14997:1996, Information technology – Syntactic metalanguage – Extended BNF*, 1996.