Chapter 0

QUERYING SEMANTIC WEB CONTENTS
A CASE STUDY

Loïc Royer[1], Benedikt Linse[2], Thomas Wächter[1], Tim Furche[2], François Bry [2] and Michael Schroeder[1]
[1]Biotec, Dresden University of Technology, Germany
[2]Institute for Informatics, University of Munich

Abstract:
Semantic web technologies promise to ease the pain of data and system integration in the life sciences. The semantic web consists of standards such as XML for mark-up of contents, RDF for representation of triplets, and OWL to define ontologies. We discuss three approaches for querying semantic web contents and building integrated bioinformatics applications, which allows bioinformaticians to make an informed choice for their data integration needs. Besides already established approach such as XQuery, we compare two novel rule-based approaches, namely Xcerpt - a versatile XML and RDF query language, and Prova - a language for rule-based Java scripting. We demonstrate the core features and limitations of these three approaches through a case study, which comprises an ontology browser, which supports retrieval of protein structure and sequence information for proteins annotated with terms from the ontology.

Key words: Bioinformatics, Semantic Web, UniProt, Protein Data Bank, PubMed, Gene Ontology, Prova, Prolog, Java, Xcerpt, logic programming, declarative programming, Web, query languages, XML, RDF, rules, semi-structured data, query patterns, simulation unification, XQuery, XPath, Relational Databases.

1. INTRODUCTION

Bioinformatics is a rapidly growing field in which innovation and discoveries often arise by the correlative analysis of massive amounts of data from widely different sources. The Semantic Web and its promises of intelligent integration of services and of information through 'semantics' can only be fulfilled in the life sciences and beyond if its technologies satisfy a minimum set of pragmatic requirements:

- Ease of use - A language must be as simple as possible. Users will go for a not so powerful but comfortable solution instead of a very rich language that is too complicated to use.
- Platform independence - Operating system idiosyncrasies are increasingly becoming a nuisance, the internet is universal, and so must be a language for the semantic web.
- Tool support - Nowadays, it is not enough to provide language specifications and the corresponding compilers and/or interpreters. Programmers require proper support tools like code-aware editors, debuggers, query builders and validation tools.
- Scalability - The volume of information being manipulated in bioinformatics is increasing exponentially, the runtime machinery of a language for integrating such data must be able to scale and cope with the processing needs of today and tomorrow.
- Modularity - Modularity is a very fundamental idea in software engineering and should be part of any modern programming language.
- Extensibility - Languages should be as user extensible as possible to accommodate unforeseen but useful extensions that users might need and be able to implement.
- Declarativeness - The language should be high-level and support the specification of what needs to be computed rather than how.

2. DATA INTEGRATION IN BIOINFORMATICS

The amount of available data in the life sciences increases rapidly and so does the variety of data formats used. Bioinformatics has a tradition for legacy text-based dataformats and databases such as UniProt [2] for protein sequences, PDB [3] for 3D structures of proteins, or PubMed [4] for scientific literature.

UniProt, PDB, PubMed

Today, many databases, including the above are available in Extensible Markup Language (www.w3.org/XML/).

Due to its hierarchical structure, XML is a flexible data format. It is a text-based format, is human-readable, and its support for Unicode ensures portability throughout systems. Together with XML a whole family of languages (www.w3.org/TR) support querying and transformation (XPath, XQuery, and XSLT). Additionally APIs such as JDOM (www.jdom.org), an implementation of the Document Object Model (DOM), and the Simple API for XML (www.saxproject.org) were developed in support of XML.

Beside the need of technologies for data handling, a major task in bioinformatics is the one of data integration. The required mapping between entities from different data sources can be managed through the use of an ontology.

Ontologies in Bioinformatics

Currently there is no agreed vocabulary used in molecular biology. For example, gene names are not used in a consistent way. EntrezGene [4] addresses this problem by providing aliases. EntrezGene lists for example eight aliases for a gene that is responsible for breast cancer (*BRCAI*; *BRCC1*; *IRIS*; *PSCP*; *RNF53*; *breast cancer 1, early onset*; *breast and ovarian cancer susceptibility protein 1*; and *breast and ovarian cancer susceptibility protein variant*).

At the time of writing, searching PubMed for *PSCP* returns 2417 relevant articles. Searching for *papillary serous carcinoma of the peritoneum*, returns 89 articles. However, searching for both terms returns only 19 hits. In general, there is a pressing need in molecular biology to use common vocabularies. This need has been addressed through the ongoing development of biomedical ontologies. Starting with the GeneOntology (www.geneontology.org) [1], the Open Biomedical Ontologies effort (obo.sourceforge.net) currently hosts 59 biomedical ontologies ranging from anatomy over chemical compounds to organism specific ontologies.

Gene Ontology (GO)

A core ontology is the Gene Ontology [1], which contains over 20000 terms describing biological processes, molecular functions, and cellular components for gene products. The biological process ontology deals with biological objectives to which the gene or gene product contributes. A process is accomplished via one or more ordered assemblies of molecular functions. The molecular function ontology deals with the biochemical activities of a gene product. It describes what is done without specifying where or when the event takes place. The cellular component ontology describes the places where a gene product can be active. The GO ontologies have become a de facto standard and are used by many databases as annotation vocabulary and are available in various formats: flat files, the Extensible Mark-up Language (XML), the resource description format (RDF), and as a MySQL database.

3. CASE STUDY: PROTEINBROWSER

Biological databases are growing rapidly. Currently there is much effort spent on annotating these databases with terms from controlled, hierarchical vocabularies such as the Gene Ontology. It is often useful to be able to retrieve all entries from a database, which are annotated with a given term from the ontology. The ProteinBrowser use-case shows how typically one needs to join data from different sources. The starting point is the Gene Ontology (GO), from which a hierarchy of terms is obtained. Using the Gene Ontology Annotation (GOA) database, the user can link GO terms to the UniProt identifiers of proteins that have been annotated with biological processes, molecular functions, and cellular components. After choosing a specific protein, the user can, remotely, query additional information from the UniProt database, for example the sequence of the protein. In turn, the PDB database can be remotely queried for still additional information. Finally, using the PubMed identifier of the publication in which the structure of the protein was published, one can query PubMed and obtain the title and abstract of the publication.

As shown in Fig. 0-1, the ProteinBrowser example is specified by the following workflow:

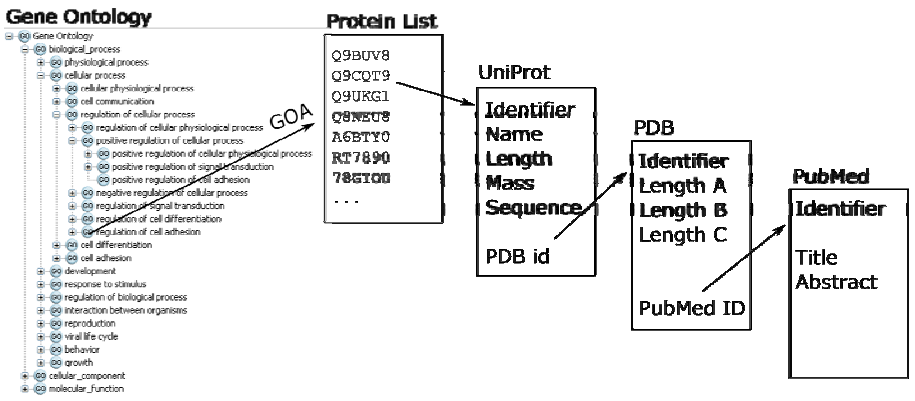
Gene Ontology
Gene Ontology
biological_process
physiological process
cellular process
cellular physiological process
cell communication
regulation of cellular process
regulation of cellular physiological process
positive regulation of cellular process
positive regulation of cellular physiological process
positive regulation of signal transduction
positive regulation of cell adhesion
negative regulation of cellular process
regulation of signal transduction
regulation of cell differentiation
regulation of cell adhesion
cell differentiation
cell adhesion
development
response to stimulus
regulation of biological process
interaction between organisms
reproduction
viral life cycle
behavior
growth
cellular_component
molecular_function
GOA
Protein List
Q9BUV8
Q9CQT9
Q9UKG1
Q8NEU8
A6BTY0
RT7890
78GIOO
...
UniProt
Identifier
Name
Length
Mass
Sequence
PDB id
PDB
Identifier
Length A
Length B
Length C
PubMed ID
PubMed
Identifier
Title
Abstract

Figure 0-2. ProteinBrowser Workflow: from GO to PubMed via GOA, UniProt and PDB.

- A term is chosen from the Gene Ontology tree. The Gene Ontology exists in various formats: MySql database, XML, RDF.

- All relevant proteins associated through the GOA (http://www.ebi.ac.uk/GOA/) database are listed.

- A protein is chosen from the list.

- UniProt is queried for information about this protein. The protein's name, its sequence length, mass, sequence, and corresponding PDB identifier can be retrieved by querying the XML file linked by the following parameterized URL:

  http://www.ebi.uniprot.org/entry/<UniprotId>? format=xml&ascii

- PDB is queried for additional information. The three lengths *width*, *height* and *depth* and the PubMed identifier of the publication in which the structure was described, can be obtained by querying the XML file linked by the following parameterized URL:

  http://www.rcsb.org/pdb/displayFile.do?fileFormat=XML&structureId=<PDBid>

- Retrieve PubMed abstract title and text where the structure was published. This uses the Pubmed ID (if available) and queries the website of NCBI with the PubMed Id at this address:

  http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=pubmed&retmode=xml
  &rettype=full&id=<PubMedId>

As shown in Fig. 0-3, this workflow involves accessing local and remote databases, in the form of files, possibly in XML format and of 'pragmatic' web-services in the form of parametrized URLs linking to XML files (also known as REST-style Web Services).

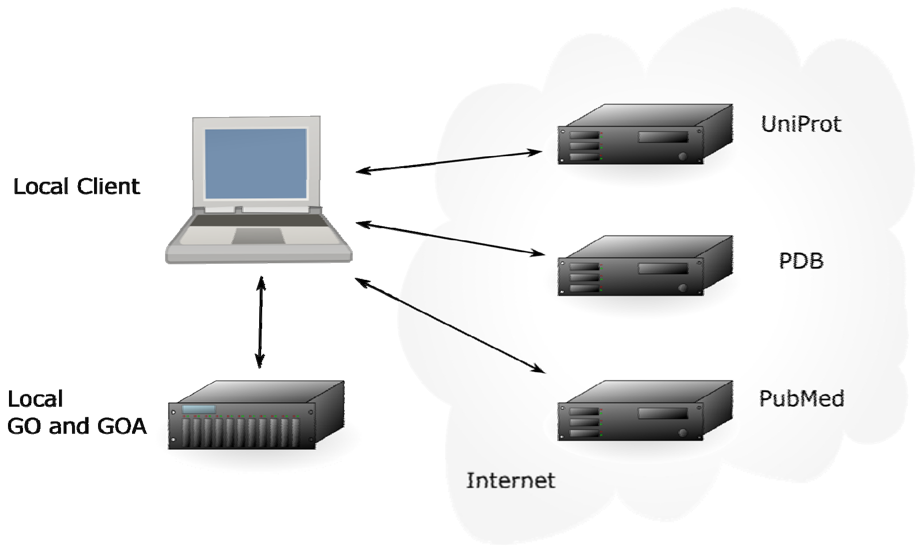

Figure 0-4. ProteinBrowser: integrates data from GO, UniProt, PDB and PubMed.

We will compare three approaches to implement this workflow. The first is based on a novel hybrid object-oriented and declarative programing language, Prova. The second is based on standard XML technologies such as XQuery and XPath. The third is based on a novel declarative query language for XML documents: Xcerpt.

• Prova	http://www.prova.ws
• XQuery/XPath	http://www.w3.org
• Xcerpt	http://www.xcerpt.org

3.1 Prova

Prova [5] is a rule-based Java scripting language. The use of rules allows the declarative specification of integration needs at a high-level, separately from implementation details. The transparent integration of Java caters for easy access and integration of database access, web services, and many other Java services. This way Prova combines the advantages of rule-based programming and object-oriented programming. Prova satisfies the following design goals:

- Combine the benefits of declarative and object-oriented programming;

- Merge the syntaxes and semantics of Prolog, as rule-based language, and Java as object-oriented languages;
- Expose logic as rules;
- Access data sources via wrappers written in Java or command-line shells like Perl;
- Make all Java API from available packages directly accessible from rules;
- Run within the Java runtime environment;
- Be compatible with web- and agent-based software architectures;
- Provide functionality necessary for rapid application prototyping and low cost maintenance.

Workflow solved with Prova

The Prova code closely resembles a declarative logic program. Rules are written down in the form **`conclusion :- premise`** where `:-` is read 'if'. Instead of relying on an internal knowledge base, which needs to be loaded entirely into memory, Prova can access external knowledge wrapped as predicates. Thus there is a clean separation between the details needed to access the external data and the way this data is joined in the workflow. Prova applies so-called backward-chaining to evaluate queries.

Wrapping the Gene Ontology and the Gene Ontology Annotation

For the Prova implementation of the ProteinBrowser we use the Gene Ontology and the protein annotations in their relational database format. As shown on Fig. 0-5 accessing databases from Prova is very simple.

```
% Imports some utility functions
:-eval(consult("utils.prova")).

% Define database location
location(database,"GO","jdbc:mysql://server","guest","guest").

% T2 is-a T1 if in the term2term table of the database
isaDB(T2,T1) :-
   dbopen("GO",DB),
   concat(["term1_id=",T1," and relationship_type_id=2"],
   WhereClause),
   sql_select(DB,term2term,[term2_id,T2],[where, WhereClause]).

% A term T is-a T
isa(T,T).

% Recursive definition of is-a:
% A term T2 is a T1 if T3 is a T1 and T2 is a T3
isa(T2,T1) :-
   isaDB(T3,T1),
   isa(T2,T3).
```

Figure 0-6. Wrapping the Gene Ontology database and the isa relationship.

After importing some utility predicates for connecting to databases, the `location` predicate is used to define a database location, the `dbopen` predicate is used to open a connection to the database, and the `sql_select` predicate provides a nice and practical declarative wrapping of the select statement of relational databases. In order to obtain all sub-terms of a given term, we simply compute the transitive closure of the sub-term relationship defined by the recursive predicate `isa`.

Finally, in order to retrieve the UniProt identifiers corresponding to a given gene ontology term, we need the `name2uniProtId` predicate (see Fig. 0-7).

```
    name2UniProtId(Term,UniProtId) :-
      dbopen("GO",DB),
      concat(["uni.GOid = ", Term],Where),
      concat(["go.term as term, goa.goa_human as uni"],From),
     sql_select(DB,From,['uni.DB_Object_ID',UniProtId],
     [where,Where]).
```

Figure 0-8. Wrapping the Gene Ontology Annotation database.

Wrapping UniProt, PDB and Medline

The three databases UniProt, PDB and Medline can be remotely accessed through a very simple web interface: a parameterized URL links to an XML file containing the relevant information for a given identifier.

As shown in Figure 0-9 The three predicates `queryUniProt`, `queryPDB`, `queryPubMed`, wrap the downloading and parsing of the XML files in a few lines:

```
urlUniProtPrefix("http://www.ebi.uniprot.org/entry/")
urlUniProtPostfix("?format=xml&ascii")
urlPDB("http://www.rcsb.org/pdb/displayFile.do?fileFormat=XML&s
 tructureId=")
urlPubMed("http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.
 fcgi?db=pubmed&retmode=xml&rettype=full&id=")

% Query UniProt by giving a UniProt Id and getting the length,
 mass, sequence, and PDB id
queryUniProt(UniProtId,Name,Length,Mass,Sequence,PDBId):-
  urlUniProtPrefix(URLpre),
  urlUniProtPostfix(URLpost),
  concat([URLpre,UniProtId,URLpost],URLString),
  retrieveXML(URLString,Root),
  children(Root,"entry",EntryNode),
  children(EntryNode,"protein",ProteinNode),
  descendantValue(ProteinNode,"name",Name),!,
  descendant(EntryNode,"sequence",SequenceNode),
  nodeAttributeByName(SequenceNode,"length", Length),
  nodeAttributeByName(SequenceNode,"mass", Mass),
  nodevalue(SequenceNode,Sequence).

% Query PDB by giving a PDB Id and getting three lengths a,b,c
 and a PubMed id of a publication
queryPDB(PDBId,LA,LB,LC,PMID):-
  urlPDB(URL),
  concat([URL,PDBId],URLString),
  retrieveXML(URLString,Root),
  descendantValue(Root,"PDBx:length_a",LA),!,
  descendantValue(Root,"PDBx:length_b",LB),!,
  descendantValue(Root,"PDBx:length_c",LC),!,
  descendantValue(Root,"PDBx:pdbx_database_id_PubMed",PMID).

 % Query pubMed by giving a PubMed Id and getting the text of
 the abstract
queryPubMed(PMID,AbstractTitle, AbstractText):-
  urlPubMed(URL),
  concat([URL,PDBId],URLString),
  retrieveXML(URLString,Root),
  descendantValue(Root,"ArticleTitle",AbstractTitle),!,
  descendantValue(Root,"AbstractText",AbstractText),!.
```

Figure 0-10. Wrapping UniProt, PDB and Medline.

The previous predicates use the following utility predicates:

```
retrieveXML(URLString,Root):-
  URL = java.net.URL(URLString),
  Stream = URL.openStream(),
  ISR = java.io.InputStreamReader(Stream),
  XMLResult = XML(ISR),
  Root = XMLResult.getDocumentElement().
```

Figure 0-11. XML retrieval.

The `retrieveXML` predicate downloads an XML file from a specified URL, and returns the root DOM (Document Object Model) tree representation of the XML file.

In Fig. 0-12, a set of predicates provide functionality to query nodes and values from the DOM tree:

```
% Simulates an XPath traversal.
descendantsValue(Current,Name,Value):-
  descendants(Current,Name,Node),
  nodeValue(Node,Value),!.

% Descendant (any depth), similar XPath: //*
descendants(Node,Node).
descendants(Element,S2):-
  children(Element,S1),
  descendants(S1,S2).

% Descendant with given name, similar XPath: //Name
descendants(Node,Name,Descendant):-
  descendants(Node,Descendant),
  nodeName(Descendant,Name).

% Definition for a direct child, similar XPath: /*
children(Element,Child):-
  Childs = Element.getChildNodes(),
  Childs.nodes(Child).

% Child with a given name, similar XPath: /Name
children(Node,Name,Child):-
  children(Node,Child),
  nodeName(Child,Name).

nodeName(Node,Name):-
  Name = Node.getNodeName().

nodeValue(Node,Value):-
  Data = Node.getFirstChild(),
  Raw = Data.getNodeValue(),
  Value = Raw.trim().
```

Figure 0-13. XML Querying.

Assembling the Workflow

Now that we have wrapped the GO and GOA databases, as well as the remote XML ressources for UniProt, PDB and PubMed. We can proceed with the assembly of the ProteinBrowser workflow, as shown in Fig. 0-14.

```
workflowStep1(GoTermName,UniProtId):-
  name2term(GoTermName,GoTerm),
  isa(GoTerm,Descendant),
  name2UniProtId(Descendant,UniProtId),
  java.lang.System.out.println(UniProtId).

workflowStep2(UniProtId):-
  queryUniProt(UniProtId,Name,Length,Mass,Sequence,PDBId),
  java.lang.System.out.println(Name),
  java.lang.System.out.println(Length),
  java.lang.System.out.println(Mass),
  java.lang.System.out.println(Sequence),
  queryPDB(PDBId,LA,LB,LC,PMID),
  java.lang.System.out.println(LA),
  java.lang.System.out.println(LB),
  java.lang.System.out.println(LC),
  queryPubMed(PMID,AbstractTitle, AbstractText),
  java.lang.System.out.println(AbstractTitle),
  java.lang.System.out.println(AbstractText).

  % Given the name N, get the term id T
name2term(N,T) :-
  dbopen("GO",DB),
  concat(["name like ",N],WhereClause),
  sql_select(DB,term,[id,T],[where, WhereClause]).
```

Figure 0-15. Workflow.

The first step is simply to enumerate all UniProt identifiers *UniProtId* annotated with terms and subterms of a given Gene Ontology term *GoTermName*. The second step uses the chosen protein UniProt identifier and starts a cascade of three remote queries to the UniProt, PDB and PubMed web sites. All relevant information collected is printed out.

3.2 XQuery and XPath

XPath allows the user to address certain parts of an XML document. Beside many applications it is used in XQuery, which is a declarative query- and transformation language for semi-structured data. It is widely used to formulate queries on RDF and XML documents. These documents can be provided as XML files, as XML views onto a XML database or created by a middleware. XQuery 1.0 is a W3C Candidate Recommendation and is already supported by many software vendors (e.g. IBM DB2, Oracle 10g Release 2, Tamino XML Server).

The Workflow Solved with XQuery

An XQuery implementation of the workflow works on XML data only and can be realized with all program logic specified as XQuery. We note that XQuery as described in the language standard is expressive enough to aggregate data from different data sources, locally or remotely.

Recursive traversal of the Gene Ontology

With XQuery the recursive traversal of the GO has to be programmed explicitely. In Fig. 0-16 the functions `local:getDescendants` and `local:getChildren` show how this simple recursion can be specified with XQuery. The locally available GO OWL file is loaded using the `doc()` function, which also works for remote resources of plain XML content. By using XQuery from within Java it is possible to preserve the DOM tree, so that it only has to be loaded once.

```
declare function local:getChildren( $term , $context)
{
  for $my_term in $context//go:term
  where $my_term/go:is_a/@rdf:resource = $term/@rdf:about
  return
     $my_term
};

declare function local:getDescendants( $term, $context)
{
  for $my_term in local:getChildren($term, $context)
  return
  <descendants>
  {
     local:getDescendants($my_term , $context), $my_term
  }
  </descendants>
};
```

Figure 0-17. Recursive XQuery to create the transitive closure over the sub-class relations.

Assembling the Workflow

Fig. 0-18 shows the complete workflow as a batch process. Given a GO accession number like "GO:0000001" an XML document is created which contains all proteins associated with the specified term or any of its child terms. For all these proteins additional information is retrieved from UniProt. Further, database references to structural data in PDB is used, if found in UniProt. For the interactive browser these parts are separated and the functions are called once the GO term or protein is selected in the GUI.

```
xquery version "1.0";
declarenamespace go = "http://www.geneontology.org/dtds/go.dtd#";
declare namespace rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#";
declare namespace fn = "http://www.w3.org/2005/xpath-functions";
declare namespace uniprot = "http://uniprot.org/uniprot";
declare namespace PDBx = "http://deposit.pdb.org/pdbML/pdbx.xsd";
declare namespace xsi="http://www.w3.org/2001/XMLSchema-instance";

declare variable $GO as xs:string external;

(: function from www.w3c.org :)
declare function local:distinct-nodes-stable ($arg as node()*) as node()*
{
   for $a at $apos in $arg
   let $before_a := fn:subsequence($arg, 1, $apos - 1)
   where every $ba in $before_a satisfies not($ba is $a)
   return $a
};

declare function local:getChildren( $term , $context) { ... };
declare function local:getDescendants( $term, $context) { ... };
declare function local:queryUniprot($uniprotID) { ... };
declare function local:queryPDB($pdbID) { ... };

(: Construct a result set for one GO term :)
<terms>
{
    let $root :=doc("/data/go_200605-assocdb.rdf-xml")
    for $term in $root//go:term
    where $term/go:accession/text() = $GO
    return
        <result query_term_acc="{$term/go:accession/text()}">
         {
         let $terms := ($term, local:getDescendants($term,$root))
         for $d_term in $terms
         return
             for $dbxref in $d_term//go:dbxref
             where $dbxref/go:database_symbol/text()="UniProt"
             return
                     for $uniprot_id in local:distinct-nodes-
                     stable($dbxref/go:reference)
                     return
                            local:queryUniprot($uniprot_id/text())
        }
        </result>
}
</terms>
```

Figure 0-19. Recursive XQuery to aggregate proteins associated with a GO term or any of its children. The result gets enriched with Uniprot and PDB data.

Obtain additional information for proteins

For all proteins identified, the UniProt database is queried selecting data sets for a specific UniProt identifier (see Fig. 0-20). Additional information from the PDB is retrieved as shown in Fig. 0-21.

```
declare function local:queryUniprot($uniprotID)
{
    let $url := concat(concat("http://www.ebi.uniprot.org/entry/",
                          $uniprotID), "?format=xml&amp;ascii")
    for $entry  in doc($url)//uniprot:entry
    let $sequence:= $entry/uniprot:sequence
    return
        <protein uniprot_id="{$uniprotID}">
            {
            for $name in $entry/uniprot:protein//uniprot:name
            return
                <name>{$name/text()}</name>
             }
            <sequence_length>{$sequence/@length}</sequence_length>
            <sequence_mass>{$sequence/@mass}</sequence_mass>
            <sequence>{ $sequence/text() }</sequence>
             {
                 For $pdbID in $entry//uniprot:dbReference[@type="PDB"]/@id
                 return
                    local:queryPDB($pdbID)
             }
        </protein>
};
```

Figure 0-22. Querying the Uniprot database with XQuery for information on the names, sequence, sequence length, sequence mass and structures of a protein

```
declare function local:queryPDB($pdbID)
{
    let $url := concat("http://www.rcsb.org/pdb/downloadFile.do?
                        fileFormat=xml&amp;compression=NO&amp;struc
                        tureId=",$pdbID)
    for $item  in
               doc($url)/PDBx:datablock/PDBx:cellCategory/PDBx:cell
    return
        <pdb_structure pdb_id="{$pdbID}">
            <length_a>{$item/PDBx:length_a/text()}</length_a>
            <length_b>{$item/PDBx:length_b/text()}</length_b>
            <length_c>{$item/PDBx:length_c/text()}</length_c>
        </pdb_structure>
};
```

Figure 0-23. Querying the PDB database with XQuery.

3.3 Xcerpt

Xcerpt [7] is a declarative rule based query- and transformation language for semi-structured data in general and for RDF and XML in particular. Xcerpt does not natively query relational data bases, but relies on the XML, RDF or OWL serializations of the Gene Ontology and the Protein Databank. These serializations in general being graph structured and highly heterogeneous, Xcerpt provides a comfortable way to query possibly incomplete subpatterns of the data.

Xcerpt builds upon *simulation unification* and rule chaining for program evaluation. Xcerpt uses three kinds of terms to carry out its computations: *data terms*, *query terms* and *construct terms*. Data terms are semi-structured data serving as an abstraction from various tree- and graph shaped data-formats such as RDF and XML. Dataterms can be used to represent any kind of semi-structured data, while still taking care of XML specificities such as attributes, namespaces and references.

Query terms are data terms augmented by logical variables and enriched by constructs that allow the specification of various forms of incompleteness, which are used to match highly heterogeneous data. Incompleteness specifications include incompleteness in depth (the descendant construct and arbitrary length traversal path expressions), incompleteness in breadth (there may be more subterms in the queried data than which are specified by the query term) and optional subterms. Query terms are matched with data terms via simulation unification to produce \emph{substitution sets} (sets of sets of variable bindings). Substitution sets are then applied to construct terms by filling in the bindings for variable occurrences.

The Workflow solved with Xcerpt

In order to select all proteins produced by a certain term referenced in the Gene Ontology, the following Xcerpt rules could be used. Since we are not only interested in the proteins produced by exactly the term provided by the user, but also in those proteins which are produced by processes which are subterms of the given term, and in additional information obtained from UniProt, PDB and PubMed, the task is split into several parts:

Extracting subterm relationships from the Gene Ontology Database

In a first step (realized by Fig. 0-24), the direct subterm relationships are extracted from the database. They are retrieved from the `is_a` elements given in the Gene Ontology. In the special `attributes`-element the form of the `rdf:resource`-attribute of the `is_a`-element is specified, demanding that it ends with a GO-Term identifier. Note that since Xcerpt programs are evaluated in a backward chaining manner, the binding of the logical variable `Term2` is passed on from the second and third rule below. Curly braces in the query term indicate that the order in which the siblings occur within the data is not important. This concept is called *Incompleteness with respect to order*.

Double curly braces are used to allow also further siblings in the data besides those explicitly specified - this concept is known as *incompleteness in breadth* in Xcerpt. Xcerpt's `desc` construct matches with descendants of the enclosing term that exhibit the specified pattern (*incompleteness in depth).* Since there is no enclosing element for the `go:term` element in the query term, it matches with all data nodes that have at least a `go:accession` and a `go:is_a` sub-element (of the specified form).

```
CONSTRUCT
  subterm { var Term1, var Term2 }
FROM
  in {
     resource {
       "http://archive.godatabase.org/full/2006-05-01/
        go_200605-assocdb.rdf-xml.gz" },
     desc go:term {{
       go:accession { var Term1 },
       go:is_a{{
         attributes{{
            rdf:resource {
              "http://www.geneontology.org/go#"++var Term2
            }
       }}
     }}
  }
END
```

Figure 0-25. Extracting subterm relationships from the Gene Ontology.

Computing the transitive closure of the subterm relationship

In a second rule (given in Fig. 0-26), the transitive closure of the subterm relationship is computed. Since all direct subterms are considered as transitive subterms, the second disjunct of the body of this second rule matches with the head of the first rule.

```
CONSTRUCT
  transitive_subterm { var Term1, var Term3 }
FROM
  or {
    and {
      subterm { var Term1, var Term2 },
      transitive_subterm { var Term2, var Term3 }
    },
    subterm { var Term1, var Term3 }
  }
END
```

Figure 0-27. Computing the transitive closure of the subterm-relationship with an Xcerpt rule.

Finding all the proteins associated with a term of the Gene Ontology

In the third rule (see Fig. 0-28) for each of the subterms of the given term `Term`, the associated proteins are looked up in the GOA database and rendered as a list of links to their Uniprot entries in an HTML file. The binding for the variable `Term` is provided by the user as a command line parameter (e.g. `xcerpt -D Term=GO:0051260`, where `GO:0051260` is the identifier of *protein homooligomerization*).

The first conjunct of the body of this rule matches with the second rule above and passes the `Term`-variable on to the head of the second rule. In this way, all of its subterms are bound to the variable `SubTerm`.

The second conjunct of the rule looks up all associated proteins for the subterm, which have a Gene Ontology database symbol of type `UNIPROT`. Each of these proteins is bound to the variable `PROTEIN`.

Note that also the second conjunct of the query term may match multiple times with the database for a single binding of the variable `SubTerm`, thus producing a set of pairs of variable bindings in which `SubTerm` is always bound to the same variable given in the query, and `Protein` is bound once for each protein produced by the given concept.

In the construct part of the rule (framed by the keywords `GOAL` and `FROM`) the proteins are grouped by the subterms which they are associated with in the Gene Ontology. This is achieved by the grouping construct `all`. The string-concatenation function "++"' is used to construct the URL pointing at the Uniprot entry. The construct term is a template of the HTML page rendered by the browser to form part of the user-interface.

```
GOAL
  html [
    head [ title [ "Proteins produced by" ++ var Term ] ],
    body [
      all span [
  h3 [ "Proteins produced by the subterm " ++ var SubTerm ],
  ul [
    all li [
      attributes{ href {
        "http://www.ebi.uniprot.org/entry/" ++ var Protein ++
     "?format=xml&ascii" } },
      var Protein ]
  ]
      ]
  ] ]
FROM
  and {
    transitive_subterm { var SubTerm, var Term },
    in {
      resource {
  "http://archive.godatabase.org/full/2006-05-01/
   go_200605-assocdb.rdf-xml.gz" },
      desc go:term{{
  go:accession{ var SubTerm },
  go:association{{
    go:gene_product{{
      desc go:database_symbol{ "UNIPROT" },
      desc go:reference{ var Protein }
    }}
  }}
      }}
    }
  }
END
```

Figure 0-29. Constructing an HTML list of proteins for a GO term.

Extracting relevant information about Proteins from the Uniprot and PDB Files

Xcerpt's patterns are well-suited to extract the name, length, mass and the sequence of amino acids for a given protein from the UniProt database and to reassemble them within an HTML fragment as specified in Fig. 0-30. The second conjunct of the same rule is used to additionally extract information from the PDB database about the physical dimensions of the crystals of the Protein and PubMed identifiers of research papers dealing with the given protein. This data is to be combined with the information from UniProt. Note that the `PDB_ID` is extracted from the UniProt database, which means that the first conjunct is evaluated before the second one. The rule could be called via a system call from within a CGI script. Many of the PDB files about proteins additionally supply PubMed identifiers of research articles treating the protein, but this is not mandatory. Xcerpt's `optional`-construct allows to select optional data that does not have to be present for the query to succeed. Since their may be multiple references to PubMed identifiers, these references are wrapped into an unordered `HTML` list using the grouping construct `all`. These references could be easily encoded as hyperlinks in a similar way as in Fig. 0-31, which has been omitted for brevity.

```
CONSTRUCT
  div [
    h3 [ 'Information about protein', span[ var Protein ] ],
    p [ "Name: " ++ var Name ],
    p [ "Length: " ++ var Length ],
    p [ "Mass: " ++ var Mass ],
    p [ "Sequence: " ++ var Sequence ],
    p [ "length_a: " ++ var LengthA ],
    p [ "length_b: " ++ var LengthB ],
    p [ "length_c: " ++ var LengthC ],
    optional p [ 'PubMed References', ul [ all li[ var PubMedID
    ] ] ]
  ]
FROM
  and {
    in {
      resource {
  "http://www.ebi.uniprot.org/entry/" ++ var SubTerm ++
    "?format=xml&ascii" },
      entry {{
  protein {{ name {{ var Name }} }},
  sequence {{
    attributes {{ length { var Length }, mass { var Mass } }},
    var Sequence
  }}
  dbReference { attributes {{
    type { "pdb accesion" },
    value { var PDB_ID }
  }} }
      }}
    },
    in {
      resource {
  "http://www.rcsb.org/pdb/downloadFile.do?fileFormat=xml&
    compression=NO&structureId=" ++ var PDB_ID },
      PDBx:datablock {{
  desc PDBx:cell {{
    PDBx:length_a{{ var LengthA }},
    PDBx:length_b{{ var LengthB }},
    PDBx:length_c{{ var LengthC }}
  }},
  optional PDBx:pdbx_database_id_PubMed { var PubMedID }
    }
  }
END
```

Figure 0-32. Combining information from the PDB and the UniProt database for the same Protein.

Retrieving the PubMed Abstract and Title

The final step in the workflow of the Protein Browser consists of retrieving the PubMed abstract and title for a given PubMed identifier retrieved by the rule in Fig. 0-33. The PubMed identifiers may either be queried directly from the PDB file of a given protein or they may originate from the results of the previous rule. In Fig. 0-34 the second alternative is presented.

```
    CONSTRUCT
      html [ head [ title [ 'Articles for Protein' ++ var Protein ] ],
       body [
         all p [ h3 [ var Title ], div [ var Abstract ] ]
       ]
      ]
    FROM
      and (
        div [[ h3 [[ span [ var Protein ] ]],
         p [[ ul [[ li [ var PubMedId ] ]] ]]
        ]],
        in {
           resource{
 'http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=pubmed&ret
 mode=xml&rettype=full&id=' ++ var PubMedId },
           PubMedArticle {{
       desc AbstractText { var Abstract },
       desc ArticleTitle { var Title }
           }}
        }
    END
```

Figure 0-35. Retrieval of Abstract and Titles of PubMed entries.

The given rule finds all PubMed identifiers from the previously created HTML fragment, retrieves the PubMed documents for these articles and assembles a new HTML page containing a list of article titles and abstracts.

4. COMPARISON

In the following, we compare the three approaches according to several criteria. Some criteria are subjective, for example how easy or difficult it is to learn and use the approach. Other criteria are of a pragmatic nature and relate to the availability of supporting tools like editors and debuggers. From a technical point of view, it is also important to evaluate the scalability, modularity, and extensibility of an approach.

Learning curve

Prova requires basic understanding of both Prolog and Java. This might make it more complicated to understand than Java or Prolog separately. The Prova syntax integrates aspects from both paradigms in a very elegant way. If one assumes basic knowledge in both Java and Prolog, Prova is then a good way to profit from both worlds.

XQuery adapts standard programming paradigms like FOR loops or IF-THEN-ELSE statements and uses XPath to address nodes in the Document Object Model (DOM) tree. Nevertheless the syntax and especially the usage of functions requires some time to learn.

Xcerpt can be used to query and transform any XML application, thus also XML serializations of RDF and Topic Maps. Therefore it is very well-suited for data integration. Being a very declarative pattern- and rule-based language, potential errors are kept to a minimum and authoring queries in Xcerpt is straightforward. Xcerpt is especially easy to learn for users with experience in logic programming or with pattern based query languages such as Query By Example or to a certain extent XPath.

Platform independence

Prova is Java-based and as such is platform-independent.

XQuery and XPath standard implementations are available as libraries written in Java (http://saxon.sourceforge.net/) and can be used from any platform which supports Java. Additionally many database systems come with XPath or XQuery build in. Xcerpt is currently implemented in Haskell and compiled with the Glasgow Haskell Compiler, which is available for Linux, Solaris, Windows, FreeBSD and MacOS X. Thus Xcerpt can be used on any of these platforms. Future versions of Xcerpt will be written in Java to further increase platform independence.

Availability

Prova is a GNU Lesser General Public License (LGPL) open source project and thus can be used in any context, it can be freely downloaded from www.prova.ws.

XQuery, XPath and RQL are available within commercial products or for free under the Berkeley Software Distribution (BSD) license.

Xcerpt is current at a prototype stage of development and is

available at `www.xcerpt.org` under the terms of the GNU General Public License.

Tool support

Prova, because of its relative youth, has almost no support for editing or debugging tools.

XPath is simple enough to be written with a plain text editor. However it is strongly recommended to use specialized editors for XQuery. There exist mature tools for several software platforms which come with editing support, validation and debugging functionalities.

Xcerpt is accompanied by a visual query authoring and execution tool called `visXcerpt`. It features a web-based graphical interface, running on top of a web server such as the Apache HTTP server (`http://www.apache.org/`) and allows to dynamically browse both XML data and the Xcerpt rules. Support for debugging and code completion in Xcerpt is not available yet.

Scalability

Prova is arguably at most as scalable as Java and its libraries. Java is itself a very mature language in terms of performance. Starting with version 1.3, the Java Virtual Machine has been based on HotSpot, a technology that allows dynamic compilation of performance bottlenecks at execution time. For this reason Java itself cannot be thought as an interpreted language. So even though the rule engine behind Prova is essentially interpreted, all the heavy duty work can be delegated to Java classes and one can thus expect near-compiled performance.

On a machine powered by a Intel Xeon 3GHz, Saxon's XQuery engine needs approximately 50 seconds to prepare the 300 MB large Gene Ontology RDF file for XQuery execution.

Xcerpt programs are currently being evaluated in memory. Thus it is not yet possible to process large amounts of XML data. With 512 megabytes of random access memory, an XML file of a size at most 40 megabytes can be effectively processed. Research geared toward more efficient implementations is being carried out.

Modularity

Prova inherits the modularity of Java. XQuery allows for user-defined functions that can be used to modularize the code and improve its maintainability. Xcerpt is being developed with a module system.

Extensibility

Prova is based on Java and can construct Java objects and call any of their methods. Xcerpt being available under an open source license, it can be easily extended and adapted to ones own needs.

5. DISCUSSION AND CONCLUSION

In this article we have shown how the combination and integration of biological data from different resources on the Web may be realized with different technologies. XML is a suitable way for sharing and exchanging data across different systems interconnected over the Internet. XML query languages are an accepted means for extracting relevant information and for processing and transforming XML data.

XML and best practices.

Biological data is often stored in relational database engines and must be serialized before it can be processed by XML query languages. Additionally, huge amounts of biological data are already available and transferring entire databases over the network takes a significant amount of time. As a result, XML queries should be processed close to the data they operate on as far as possible, taking advantage of relational database indexes. Several commercial database products already support the local execution of XQuery programs. To minimize transfer and processing time, only the results of locally executed queries should be transferred over the network as XML. In many cases, however, queries cannot be executed locally in their entirety, because joins over entries located at different sites are necessary.

As can be seen in the exemplary workflow described previously, several transformations of XML data may be stringed together to achieve complex restructuring tasks. In such cases it is advisable to minimize intermediate serializations of XML data independently of the query language being used. In other words embedding several Xcerpt, XQuery or XSLT programs taking XML as input and producing XML as output in a host language is inefficient when compared to joining these programs to a single one, because processing time is lost for parsing and serializing XML data.

The advantages of using XML query languages for data integration versus the direct usage of relational databases increase with the amount of different data sources that must be integrated and with the degree of heterogeneity of the encountered data. The more heterogeneous the data, the harder it is to fit it into a relational database schema. Moreover, XML query languages (especially Xcerpt) provide a rich set of language constructs to deal with various kinds of heterogeneity of the data, which means that several SQL queries operating on a relational database can be combined to form a single Xcerpt query on XML data.

In picking the right XML technology for a bioinformatics project, maturity of the language is an important issue. Xcerpt being a research prototype, is currently not recommended for use in large projects. On the other hand XQuery is a W3C recommendation and several robust implementations are already available.

Beyond XML ?

It is not yet clear if XML will eventually become the universal format for data exchange. Relational databases, flat files, and other idiosyncratic formats might subsist and limit, in practice, the applicability of pure XML query languages. We have shown how practical Prova is for assembling workflows involving heterogeneous sources of data. Prova is also able to delegate XML processing tasks to XQuery which has itself a Java implementation based on the Saxon library (http://saxon.sourceforge.net/). Xcerpt will also be eventually reimplemented in Java, and thus it will also be possible in the future to run Xcerpt queries from a Prova program. It can be argued that the need for a generic and possibly declarative programming language will remain. Simply because from a pragmatic point of view, there will always be some tasks that will be simply too cumbersome to deal with any specialized languages. A user should always be able to fall-back to a standard programming approach.

Conclusion

In all cases, it is clear that independently of the technologies used, the trend is toward remote querying of data. Maintaining and synchronizing local databases is cumbersome and should not be necessary. As we have seen, several databases like UniProt, PDB and PubMed offer their data through URL links in XML format. Prova, Xquery/Xpath and Xcerpt are ready to process them.

Acknowledgements: We acknowledge the financial support of the EU projects Sealife (FP6-IST-027269) and REWERSE (FP6-IST-506779).

[1] The Gene Ontology (GO) project in 2006.*Nucleic Acids Research*, 34(Database issue):D322–6, 12 2005.

[2] BairochA, ApweilerR,WuCH, BarkerWC, BoeckmannB, FerroS, GasteigerE, HuangH, LopezR, MagraneM, MartinMJ, NataleDA, O'DonovanC, Redaschi N, andYeh LS. The Universal Protein Resource(UniProt). *Nucleic Acids Res.*, 33:D154–159, 2005.

[3] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig,

I. N. Shindyalov, andP. E. Bourne. The protein data bank. *Nucleic Acids Res*, 28(1):235–242, 2000.

[4] WheelerDL,ChappeyC,LashAE,LeipeDD, MaddenTL, SchulerGD,Tatusova TA, andRappBA. Database resourcesof the National Center for Biotechnology Information. *Nucleic Acids Res.*, 28:10–4, 2000.

[5] AlexanderKozlenkovand Michael Schroeder.PROVA: Rule-basedJava-Scripting for a Bioinformatics Semantic Web. In E. Rahm, editor, *International Workshop on Data Integration in the Life Sciences DILS*, Leipzig, Germany, 2004. Springer.

[6] Donna Maglott, Jim Ostell, Kim D Pruitt, and Tatiana Tatusova.Entrez Gene: gene-centered information at NCBI. *Nucleic Acids Res*, 33(Database issue):D54–D58, 2005.

[7] Sebastian Schaffert and Franc¸ois Bry. Querying the Web Reconsidered: APractical Introduction to Xcerpt. In *Proceedings of Extreme Markup Languages 2004, Montreal, Quebec, Canada (2nd–6thAugust 2004)*, 2004.