Query Processing in the TAMBIS Bioinformatics Source Integration System

Norman W. Paton[1], Robert Stevens[1], Pat Baker[2], Carole A. Goble[1],
Sean Bechhofer[1] and Andy Brass[2]

Department of Computer Science[1], School of Biological Sciences[2]
University of Manchester
Oxford Road, Manchester M13 9PL, UK
tambis@cs.man.ac.uk

Abstract

Conducting bioinformatic analyses involves biologists in expressing requests over a range of highly heterogeneous information sources and software tools. Such activities are laborious, and require detailed knowledge of the data structures and call interfaces of the different sources. The TAMBIS (Transparent Access to Multiple Bioinformatics Information Sources) project seeks to make the diversity in data structures, call interfaces and locations of bioinformatics sources transparent to users. In TAMBIS, queries are expressed in terms of an ontology implemented using a description logic, and queries over the ontology are rewritten to a middleware level for execution over the diverse sources. This paper describes query processing in TAMBIS, focusing in particular on the way source-independent concepts in the ontology are related to source-dependent middleware calls, and describing how the planner identifies efficient ways of evaluating user queries.

1. Introduction

Bioinformatics is the use of computational techniques for the consolidation and analysis of experimental data in biology. The bioinformatics community is distributed, and has a history of sharing both data and software tools. This means that much useful information is available, and that powerful analysis tools are readily accessible to biologists on the internet. The development of a global bioinformatics infrastructure has, however, been piecemeal, and disparate sources and tools are often poorly integrated and difficult to use together. Furthermore, the bioinformatics community has enthusiastically embraced the WWW as a way of making individual sources more accessible to remote users, but this has often led to an emphasis on interactive browsing that mitigates against effective interoperation or programmatic access to information.

Information sources for bioinformatics thus exhibit classical characteristics of autonomous, heterogeneous environments. As many bioinformatic analyses need to make use of multiple information sources, the problem of providing effective integrated access to such sources has become an important one in the community. Different approaches have been adopted. For example, SRS [9] provides an integrated browser interface and a rudimentary query language for a range of important information sources, but does not hide from users the formats and conventions of the integrated sources. OPM [7] uses a database view mechanism to define a global schema over a collection of sources that can be relational databases or structured files. OPM views can be used to hide individual sources, although, unlike TAMBIS, the global schema is described using an object data model, and the integration methodology is very much bottom-up. CPL [5] comes with a library of functions that supports a syntactically consistent view of biological sources, plus a comprehension based query language for combining information derived from the sources. P/FDM [12] combines database functionality with source specific entity types and functions that are used to wrap a range of sources, which are generally databases. Current support for source integration in CPL and P/FDM tends not to emphasise source transparency, which is an aim of the work described here.

This paper describes how queries are processed in TAMBIS (Transparent Access to Biological Information Sources [2]). TAMBIS is an ontology centred system for evaluating queries that require access to multiple heterogeneous bioinformatics sources. In TAMBIS, queries are written using a form-based interface over a source-independent ontology of bioinformatics concepts. The ontology [3] is expressed using the description logic GRAIL [16], and the TAMBIS system transforms source independent GRAIL queries to source dependent query plans in CPL. A GRAIL query is essentially a description of a set of instances, but no instances are stored with the concept model. It is thus the role of the query processor to construct an execution plan from a GRAIL query so that the illusion is provided to the user that they have access to a single, local instance store. This paper describes how user queries written in terms of the ontology are transformed into source-dependent CPL queries, taking into account the sources available and the likely efficiency of the generated plans.

The TAMBIS system can thus be seen as being quite ambitious both in terms of what it seeks to do and in terms of the technologies on which it builds. TAMBIS allows users to express queries over sources without the users needing to be aware of the location, capabilities, data types or programming interfaces of the sources. TAMBIS also seeks to use modern knowledge representation technologies to provide users with a clear, conceptual level view of bioinformatics, rather than the more logical or physical views provided by most earlier proposals. The TAMBIS system is currently being evaluated over the WWW by a range of biologists; it is hoped to provide general public access later in the year[1].

This paper is organised as follows. Section 2 provides an overview of the TAMBIS system, and in particular the components other than those relating to query processing. Section 3 outlines the sources and services model, which associates terms from the ontology with functions in CPL. Section 4 describes how GRAIL queries are translated into an internal form used for optimisation. Section 5 describes both the optimiser algorithm and the mappings it carries out. Section 6 describes how CPL programs are generated from the optimised internal form. Section 7 compares query processing in TAMBIS with query processing in other knowledge based information integration systems. Section 8 presents some conclusions and pointers to future work.

2. TAMBIS Overview

The architecture of TAMBIS is illustrated in figure 1. The solid arrows represent the passage of a query through the system, and the dashed arrows represent the use of information models. Details on the different components are provided throughout the paper.

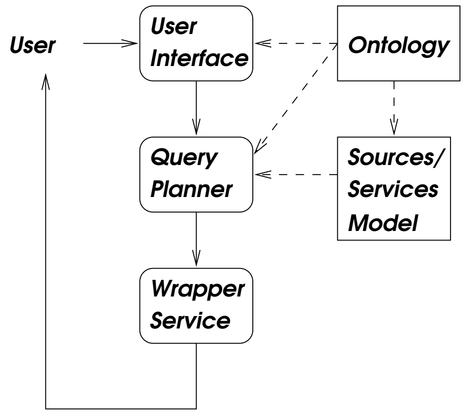

Figure 1. TAMBIS architecture.

2.1. The Ontology

The TAMBIS ontology is expressed using the description logic GRAIL [16]. A description logic (DL) [4] is a structural knowledge representation language in which concepts are automatically arranged in a lattice on the basis of the 'subsumption' relationship between their descriptions. This essentially means that instead of asserting that $C_1\ is-a-kind-of\ C_2$, the modeller describes the concepts C_1 and C_2 and the system infers the $is-a-kind-of$ relationship. A GRAIL model can be considered to consist of three parts:

1. *Assertions:* Assertions can be used to introduce new concepts and to place concepts explicitly in the subsumption lattice. For example:

```
BioPolymer newSub Protein
BioPolymer newSub NucleicAcid
NucleicAcid newSub DNA
```

   declares that `Protein` and `NucleicAcid` are both kinds of `BioPolymer`, and that `DNA` is a kind of `NucleicAcid`. It is considered good practice when modelling using description logics to make minimal use of asserted subsumption relationships.

2. *Concept forming operations and reasoning services:* Concepts can be described in GRAIL using the single concept formation operation `which`. For example:

```
Motif which isComponentOf Protein
```

   describes a particular kind of `Motif` (a motif is a pattern in a sequence) that is found within a `Protein`. If there is a need to refer to this concept regularly, it can be given a name:

```
Motif which isComponentOf Protein
name ProteinMotif
```

   There are also motifs that are found in things other than proteins, for example, in DNA:

```
Motif which isComponentOf DNA
name DnaMotif
```

[1] Details of the WWW release are accessible from `http://img.cs.man.ac.uk/tambis`.

The subsumption algorithm will automatically classify both `ProteinMotif` and `DnaMotif` as $kinds-of$ motif, and will know that a `ProteinMotif` is not a kind of `DnaMotif`, and vice-versa. To provide some terminology that will be used later in the paper, in the definition of `DnaMotif`, `Motif` is the *base concept*, `isComponentOf` is a *role*, `DNA` is a *role filler*, and the combination of the role and its filler in `isComponentOf DNA` is a *criterion*. A concept with criteria is known as a *complex concept*. A concept without criteria is known as a *primitive concept*.

In general, a new concept can be defined as follows, where Con is the name of a concept and and each C_i is a criterion:

Con `which` C_1 ... C_n

As concept descriptions can be quite complex, a rich lattice of concepts can be constructed, in which some concepts have many parents, and many concepts have many children. GRAIL is not a particularly expressive DL, however. For example, it supports neither negation nor disjunction in concept descriptions, and query processing in TAMBIS operates in the context of the closed world assumption. This considerably simplifies query processing compared with more complex DLs, which is particularly helpful in the context of heterogeneous external sources.

3. *Sanctions:* The concept forming operations cannot be used in an unconstrained manner, but are restricted by a number of sanctions that are applied to the model. In essence, it would not be legitimate to create a new concept `Motif which isComponentOf Protein` unless the modeller has introduced a sanctioning statement to indicate that it is plausible to think of motifs as being components of proteins.

   The sanctioning mechanism is used extensively during query construction to prevent users from building semantically meaningless query expressions, but has no direct role during query processing, and so is not discussed further here. It does, however, ensure that only biologically sensible queries are available to be rewritten.

The ontology is managed by the Grail terminology server [11], which provides a range of concept representation and reasoning services (e.g. for creating a concept, finding its parents, etc). Both the user interface and the query processor run as clients of this server.

The current TAMBIS ontology, which is described in [3], consists of around 1800 concepts and their relationships, focusing in particular on proteins and nucleic acids (and their various children – DNA, RNA, Gene, Enzyme, etc) and things that can be said about these core concepts in the bioinformatics sources. At the time of writing, the WWW release of the TAMBIS system provides query processing over bioinformatic sources for a reduced model containing around 250 concepts and their relationships. These models provide descriptions of bioinformatics concepts at a level of detail broadly consistent with that provided by bioinformatics sources.

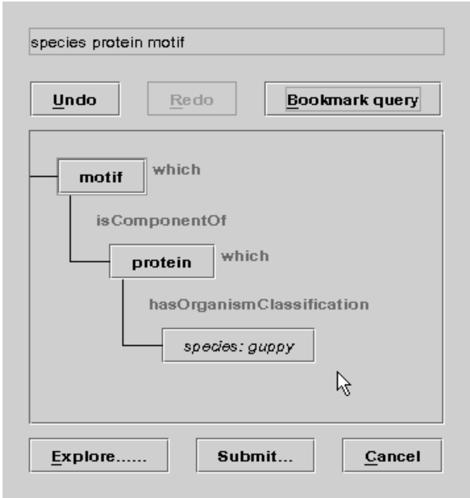

Figure 2. Query builder with motif query.

2.2. The User Interface

The user interface for query construction is a Java applet within which users explore the model, adding concepts and criteria into the current query as they go. In description logics, a concept can be viewed as a description of a set of instances, and thus also viewed as a query.

The following is an example query, which asks for motifs that are components of guppy proteins:

```
Motif which isComponentOf
  Protein which hasOrganismClassification
    Species:guppy
```

`Species:guppy` indicates that the user-defined value for the concept `Species` in the query is `guppy`. This query is built incrementally in the user interface. Firstly, the user selects the base concept `Motif`, which appears in a query builder window, an example of which is given in figure 2. The user can then restrict the motifs in which they are interested to those that are associated with specific criteria. In this case, a `Motif` is only of interest if it `isComponentOf` a `Protein`. This restriction is placed on `Motif` by selecting a *restrict* option from a popup menu on the `Motif` button in the query builder. This leads to a list of the criteria defined on *motif* being displayed, from which the user can make selections. A screenshot of the final representation of the query is given in figure 2.

2.3. The Wrappers

The output from the TAMBIS system is a query plan written in CPL [5]. CPL (Collection Programming Language) is a comprehension based language with data types for representing arbitrarily nested sets, bags, lists, records and variants. An example CPL query, which retrieves all motifs in guppy proteins, is as follows:

```
{m |
  \p<-get-sp-entry-by-os("guppy"),
  \m<-do-prosite-scan-by-entry-rec(p)}
```

In the query, the part before the | is the projection expression, which in this case indicates that only the motifs `m` are of interest. The two function calls in the body of the query to the right of the | are generators, which retrieve values from distinct, wrapped sources. The first line in the query body indicates that the new variable `p` is to be bound in turn to each of the values that result from the evaluation of the function `get-sp-entry-by-os` with the parameter `guppy`. The function name can be read as *get SwissProt entry by organism source*, where SwissProt is an information source containing data about proteins. The second function call binds the variable `m` in turn to each of the motifs of the proteins bound to `p`. The function name can be read as *scan the prosite database for motifs in the given protein record*. Running this example query using TAMBIS in Manchester (connecting to remote sources – SwissProt is in Switzerland) typically takes around 25 seconds.

The CPL system is supplied with function libraries that provide access to a range of bioinformatics sources of different types (e.g. databases, analysis tools [5]). TAMBIS uses these libraries, and a number that have been developed for TAMBIS, to provide a function based view of the sources. The public release of TAMBIS currently accesses 5 sources, and uses a total of around 300 CPL functions. Preliminary results have been obtained using the larger concept model with as many as 15 sources.

CPL can be seen as providing syntactically consistent, but not source transparent, access to the sources, and thus we view CPL as a wrapping mechanism tightly coupled with convenient language facilities for accumulating and transmitting results from different sources.

3. Sources and Services Model

The sources and services model (SSM) stores the relationships between the concepts and roles in the ontology and the functions used to wrap sources in CPL. In the SSM, the ontology is used to index the CPL functions used to evaluate queries written in terms of the ontology. The SSM contains descriptions of three broad categories of information: iterators that retrieve instances of concepts from sources, role evaluators that retrieve or compute values for the roles of instances, and filters that are used to discard instances that are not relevant to the query.

The query processor consults the SSM to identify what options are available for evaluating a query, and to generate the CPL program that answers a query.

3.1. Relating Concepts and Roles to Functions

The description of each CPL *Function* in the SSM has six attributes: $<$ *name: String* – the name of the function; *arguments: List of TypeName* – the names given to the CPL types of the arguments of the function; *resultType: TypeName* – the name given to the CPL result type for the function; *cardinality: Real* – the average number of results generated by a call to the function; *cost: Real* the average response time of the function in seconds; *source: String* – the name of the source with which the function is associated $>$.

The following categories of mapping information are supported within the SSM:

1. *Instantiable concepts:* An instantiable concept is one for which users may provide a value within a query. For example, an instance of `Species` is a value that could either be retrieved from a source or provided by the user. The SSM for each instantiable concept has two attributes: $<$ *name: ConceptName* – the name of the concept; prompt: String – the prompt used in the interface to request the value from the user $>$. For example: $< name$: `Species`; $prompt$: "Type in the latin name or common name of a species" $>$.

2. *Iteration:* Iteration allows the instances of a concept to be iterated over. For example, the instances of `Protein` can be obtained from SwissProt. The SSM for each iterator has two attributes: $<$ *name: Concept* – the name or description of the concept; *function: Function* – the function used to perform the iteration $>$. For example: $< name$: `Protein`; $function$: $< name$: "get-all-sp-entries"; $arguments$: []; $resultType$: "protein_record"; $cardinality$: 80000; $cost$: 8000.0; $source$: "SwissProt" $> >$ [2].

3. *Roles:* Roles allow the evaluation of a role on an instance to obtain a value for its filler. For example, it is possible to obtain the `AccessionNumber`

[2] Some of the times provided in this paper are quite large – this reflects, for example, the fact that SwissProt times are for accessing SwissProt in Switzerland from Manchester over the Internet. We are currently installing a range of heavily used sources locally, which will significantly improve response times. We note that these times are features of the sources being accessed, and not a feature of TAMBIS.

of a `Protein`, given the `Protein`. The SSM for role evaluation has three attributes: $<$ *conceptName: Concept* – the name or description of the concept on which the role is defined; *criterion: Criterion* – the criterion that can be evaluated; *function: Function* – the function used to compute the value for the role filler $>$. For example: $< name$: `Protein`; *rolename* : `hasAccessionNumber`; $function$: $< name$: "get-ac-from-sp-entry"; $arguments$: ["protein_record"]; $resultType$: "accession_number"; $cardinality$: 1; *cost* : 0.01; *source* : "" $> >$. The function used does not itself directly access a source, but rather accesses a data structure retrieved from a source by some other function.

4. *Mapped Roles:* Mapped roles are roles in which the concept that is provided as the role filler can be mapped to a scalar value that can be used as the argument to a CPL function that can in turn be used to generate instances of the original concept. For example, an instance of `Protein which hasFunction Receptor` can be retrieved from SwissProt by retrieving entries with *receptor* in their keyword field. This involves mapping the concept name `Receptor` to the corresponding keyword in the relevant source (in this case the mapping is trivial, but in practice a table lookup must be performed). The SSM for mapped roles has two attributes: $<$ *concept: Concept* – the description of the concept with the relevant criterion; *function: Function* – the function used to retrieve instances of the concept $>$. For example: $<$ *concept* : `Protein which hasFunction BiologicalFunction`; $function$: $< name$: "get-sp-entries-by-kwd"; $arguments$: ["string"]; $resultType$: "protein_record"; $cardinality$: 16000; *cost* : 640.0; *source* : "SwissProt" $> >$. This definition is relevant to the example `Protein which hasFunction Receptor` because `Receptor` is a kind of `BiologicalFunction`.

5. *Filters:* When instances of a concept have been retrieved, for example by iteration, criteria in the query may be used to discard some of the instances. For example, given an instance of `Protein` in the query `Protein which hasFunction Hydrolase`, the instance of `Protein` must be checked to see if it `hasFunction Hydrolase`. The SSM for filters has two attributes: $<$ *concept: Concept* – the description of the concept including the criterion; *function: Function* – the function used to carry out the filtering $>$. For example: $<$ *concept* : `Protein which hasFunction Hydrolase`; $function$: $< name$: "check-sp-entry-for-hydrolase"; $arguments$: ["protein_record"]; $resultType$: "boolean"; $cardinality$: 0.1; *cost* : 0.01; *source* : "" $> >$. The cardinality of a filter is always less than 1.0 – in essence the cardinality of a function that is used as a filter is an indication of the selectivity of the filter.

6. *Mapped Filters:* Mapped filters play the same role as standard filters, except that an argument value for the CPL function is obtained by mapping the name of the concept that is the role filler in the query onto a scalar value, as in *Mapped Roles*. For example, in the query `Protein which isAssociatedWithProcess Lactation`, the process name `Lactation` can be mapped (trivially in this case) to the string *lactation*, which can be checked for in a SwissProt record. The SSM for mapped filters has the same two attributes as *Filters*, but the CPL functions have two arguments. For example: $<$ *concept* : `Protein which isAssociatedWithProcess BiologicalProcess`; $function$: $< name$: "check-sp-entry-for-process"; $arguments$: ["protein_record", "String"]; $resultType$: "boolean"; $cardinality$: 0.2; *cost* : 0.01; *source* : "" $> >$. This definition is relevant to the example `Protein which isAssociatedWithProcess Lactation` because the concept `Lactation` is a kind of `BiologicalProcess`.

7. *Kind-of Filters:* Where the subsumption lattice has been asserted, it is sometimes the case that instances of a concept are retrieved using a mechanism that generates instances of a parent concept of the concept that appears in the query. For example, a function may have retrieved an instance of `Motif`, whereas the query specified a `SulphationSite`, where a `SulphationSite` is a kind if `Motif`. In this event, it is necessary to insert a filter that discards instances of `Motif` that are not `SulphationSites`. The SSM for kind-of filters has three attributes: $<$ *parentConcept: Concept* – the name of the parent concept; *childConcept: Concept* – the name of the child concept; *function: Function* – the function used to carry out the filtering $>$. For example: $< parentConcept$: `Motif`; $childConcept$: `SulphationSite`; $function$: $< name$: "check-ps-entry-in-sulphation-set"; $arguments$: ["motif_record"]; $resultType$: "boolean"; $cardinality$: 0.1; *cost* : 0.01; *source* : "" $> >$.

3.2. Other Features of the Sources and Services Model

The most important information in the SSM is that described in section 3.1. The two other aspects, which will not be described in detail because of limited space are:

Mappings: There are a number of different forms of mapping, for example, single valued mappings ($concept \times source \rightarrow String$), set valued mappings ($concept \times source \rightarrow \{String\}$) and computed single and set valued mappings. The first two of these are explicitly stored in lookup tables, and the third involves consulting the ontology with a general concept, e.g. `Metal`, and collecting its string mappable children (e.g., `Zinc`, `Iron`, ...). These mappings are referred to in section 5.2 as the function *map*.

Coercions: It is sometimes the case that the CPL function used to obtain an instance of a concept retrieves a CPL type that is unsuitable for use as an argument to a subsequent function call. For example, a function might retrieve the *accession_number* of a protein, when a subsequent function needs a *protein_record*. Coercions are CPL functions that can be used to obtain one representation of an instance from another. These functions may or may not require access to external sources.

4. Representing Queries Internally

GRAIL queries are declarative, in that the meaning of a query is not dependent on the order of evaluation of its components. As a result, the TAMBIS system, and not the user, must take responsibility for identifying an efficient evaluation order for the components of a GRAIL query. This section describes how GRAIL queries are represented internally for the purposes of optimisation, and how this internal representation is generated.

GRAIL queries are intrinsically nested structures. The query internal form (QIF) used in TAMBIS can be seen as an unnested representation of the original GRAIL query. This representation has been developed to allow easier reordering of the components of a query in the planner.

The QIF is a list of query components. Each query component has five attributes: $<$ *theConcept: Concept* – the base concept to which this component relates; *theCriteria: Set of QIFCriterion* – the criteria of *theConcept*; *theVariable: String* – the name of the variable to which instances of *theConcept* will be bound; *theTechnique: String* – the CPL function call to be used to retrieve instances of *theConcept*; *theFetchCriterion: QIFCriterion* – the criterion from *theCriteria* used to obtain instances of *theConcept* using *theTechnique* $>$.

```
< name : Protein
  theCriteria :
    < theCriterion : isHomologousTo Enzyme
      relatedComponent : component of enzyme-1
      userValue : “” >
    < theCriterion :
        hasOrganismClassification Species
      relatedComponent : null
      userValue : “human” >
  theVariable : protein-1
  theTechnique : “”
  theFetchCriterion : null >

< name : Enzyme
  theCriteria :
    < theCriterion : hasCofactor Zinc
      relatedComponent : null
      userValue : “” >
  theVariable : enzyme-1
  theTechnique : “”
  theFetchCriterion : null >
```

Figure 3. QIF for example query.

Criteria are represented by the type *QIFCriterion* in the QIF. Each *QIFCriterion* has three attributes: $<$ *theCriterion: Criterion* – the GRAIL criterion that the *QIFCriterion* represents; *relatedComponent: QueryComponent* – the (optional) query component of the role filler; *userValue: String* – the (optional) value provided by the user for the role filler $>$.

For example, the following query finds human protein homologues of zinc cofactor enzymes:

```
Protein which <
    hasOrganismClassification Species:human
    isHomologousTo
        (Enzyme which hasCofactor Zinc)>
```

The $<$ ... $>$ syntax is used to group the multiple criteria of the concept of which `Protein` is the base. The query is represented by two query components, as described in figure 3. The values for *theTechnique* and *theFetchCriterion* are identified during optimisation.

Generation of the QIF from a GRAIL query is straightforward, and is carried out in a single pass over the query. The translator starts from the query's base concept, generates a query component for this concept, and then loops through the criteria of the concept, calling itself recursively on all complex role fillers.

5. The Query Planner

This section describes the planner, which identifies how a query in the QIF can be evaluated given the resources

```
input: query: List of QueryComponent

finalPlan: List of QueryComponent
while query <> [] do
   bestQC := findBest(query)
   finalPlan := finalPlan ++ bestQC
   query := query -- bestQC
end
return finalPlan
```

Figure 4. The optimisation algorithm.

described in the SSM. As with other query optimisers, the TAMBIS query planner has two principal components – a search algorithm, described in section 5.1, and a collection of mappings that can be applied to query components, described in section 5.2.

5.1. The Algorithm

The search algorithm exploits the augmentation heuristic [17], which was selected as it is straightforward to implement, and provides a reasonable trade-off between cost of optimisation and quality of plan generated. The algorithm is given in figure 4. The basic strategy is to generate a plan as an ordered list of query components, where the first component in the list is predicted to be the least costly component to evaluate from scratch, and the subsequent components in turn are the least costly to evaluate given what has previously been evaluated.

The notion of "cost" here is derived from two components – the time *t* that it is predicted it will take to evaluate the component, and the number of instances *n* it is predicted will be retrieved. The "cost" of evaluating a component is computed as $t \times n^2$. This penalises query components that take a long time to respond and that generate large numbers of instances, but penalises most strongly components that generate large numbers of instances. The reason why large numbers of instances are considered particularly undesirable is that the rest of the query will be evaluated *n* times, so *n* is a multiplier in the predicted cost of the whole query. However, as the augmentation heuristic is a greedy algorithm, the complete cost of query plans is not used during the search.

5.2. The Mappings

The optimisation algorithm in figure 4 depends heavily on the definition of the *findBest* function. This function, given a query component, considers a variety of ways in which instances of the component can be retrieved from sources. Thus *findBest* considers a range of *mappings* from query components onto functions that generate concept instances from sources. The principal mappings are presented below. Example queries generated using these mappings are discussed in section 6.2. All assume that qc is the current query component.

The ontology is used within the mappings to index the functions in the SSM, and calls are made to the terminology server during rewriting to support the following: the predicate $isInverse$, which tests two criteria to see if they are inverses; and the operator $\sqsubseteq$, which performs a subsumption test – $C_1 \sqsubseteq C_2$ is true only if every instance of C_1 is also an instance of C_2.

1. *Retrieval by User Instantiated Filler:* Where the query component contains a criterion for which the user has supplied a value, a function that can be used to evaluate the inverse of the criterion can be used to fetch instances of the concept.

   The notation used throughout this section indicates that the function described in the first line can be used to obtain instances if the following condition holds. The function description retrieves values for the result of the function, the name of the function and the argument(s) of the function from variables bound in the condition. The condition associates features of the query component with data from the SSM, obtaining the latter from the SSM by accessing the data structures introduced in section 3.1.

   $qc.theVariable \leftarrow r.function.name(cr.userValue)$
   `if`
   $(\exists r \in Roles) \wedge$
   $(\exists cr \in qc.theCriteria :$
   $isInverse(r.criterion, cr.theCriterion) \wedge$
   $cr.userValue <>$ "")

   In the above, the *Roles* value r from the SSM represents a way of evaluating the inverse of the criterion *cr* on the query component *qc*. For example, if the criterion *cr* on a `Protein` represents `hasAccessionNumber`, then the role *r* would represent `isAccessionNumberOf`. Evaluating the function associated with `isAccessionNumberOf` yields a data structure representing an instance of `Protein`.

2. *Retrieval by Following a Function from a Known Value:* Where the query component contains a criterion *cr* for which the filler has already been evaluated, it may be possible to retrieve the concept by evaluating a function on the known filler. This function must be the function used to evaluate the criterion on the filler that is the inverse of cr.

$$
\begin{array}{l}
qc.theVariable \leftarrow r.function.name(cr.theVariable) \\
\texttt{if} \\
\quad (\exists cr \in qc.theCriteria : \\
\quad\quad knownVariable(\\
\quad\quad\quad cr.relatedComponent.theVariable)) \wedge \\
\quad (\exists r \in Roles : \\
\quad\quad isInverse(r.criterion, cr.theCriterion))
\end{array}
$$

This mapping is similar to the previous one – the principal difference is that the value used as an input to the generated function is a known variable, rather than a user supplied value.

3. *Retrieval by evaluating a Mapped Role:* Where the query component contains a criterion for which the filler can be mapped to a scalar, the mapping of this filler may be usable as an argument to a function enabling retrieval of the base concept.

$$
\begin{array}{l}
qc.theVariable \leftarrow \\
\quad mr.function.name(map(cr.theCriterion.filler, \\
\quad\quad mr.function.source)) \\
\texttt{if} \\
\quad (\exists cr \in qc.theCriteria) \wedge \\
\quad (\exists mr \in MappedRoles : \\
\quad\quad qc.base \; \texttt{which} \; cr.theCriterion \sqsubseteq mr.concept)
\end{array}
$$

4. *Retrieval by Iteration:* Where no more direct way of obtaining instances can be identified, it is necessary to iterate over all available values.

$$
\begin{array}{l}
qc.theVariable \leftarrow it.function \\
\texttt{if} \\
\quad (\exists it \in Iteration : qc.name \sqsubseteq it.name)
\end{array}
$$

These mappings are used by *findBest* from figure 4 to identify possible ways of obtaining instances for query components. There are a total of 8 mappings for instance retrieval in the current version of TAMBIS; those not provided here for reasons of limited space are mostly variants of those given.

Overall, *findBest*, given a collection of query components, applies the mappings to identify promising ways of evaluating each of the components, scans the criteria to identify the extent to which they may be able to filter the instances retrieved, and computes a cost factor for each component. Where *theTechnique* generates a more general value than is required for the query (i.e. an instance of a parent of the concept named in the query), *findBest* also identifies an appropriate filter from kind-of filters, and plants this as part of *theTechnique*.

6. Generating CPL

6.1. The Code Generator

The code generator is fairly straightforward, and constructs a query in a single pass through the QIF. For each QIF component, the code generator writes out *theTechnique* identified by the planner, and iterates over the component's other criteria, writing out function calls associated with roles, filters, mapped filters and mapped roles, as required. To handle criteria, the code generator currently uses an additional 8 mappings similar in nature to those in section 5.2. The code generator also places a projection expression at the start of the CPL query, and plants coercions where necessary.

6.2. Examples

This section provides some examples of queries currently supported within TAMBIS, showing the original query and the generated CPL, and providing some comments on how the query was processed.

1. *Which motifs are components of guppy proteins?* This is the example query used in section 2.2. The CPL generated for this query is:

```
{m1 |
  \p1<-get-sp-entry-by-os("guppy"),
  \m1<-do-prosite-scan-by-entry-rec(p1)}
```

This query contains two query components, one for `Protein` and the other for `Motif`. The query component for `Protein` is chosen for evaluation first, and the mapping rule for *retrieval by instantiated filler* (mapping number 1 in section 5.2) is used to generate the function that retrieves proteins from SwissProt given the name of the organism. The `Motifs` in the `Protein` are then computed using a function generated by the mapping rule for *retrieval by following a function from a known value* (mapping number 2 in section 5.2).

2. *Which human proteins are homologous to zinc cofactor enzymes?* This is the example query used in section 4. The CPL generated for this query is:

```
{p1 |
  \e1<-get-enz-entry-by-cf("zinc"),
  \p1<-do-blastp-by-sq-in-enz-entry(e1),
  check-sp-species-entry("human",p1)}
```

The planner has thus chosen to evaluate the query component for the concept `Enzyme` in figure 3 before the query component for `Protein`. Evaluating the query component for Enzyme involves accessing the database called *enzyme* using a CPL function generated by the mapping for *retrieval by evaluating a mapped role* (mapping number 3 in section 5.2). The query component for protein has been compiled into two CPL functions. The first is generated by the mapping for *retrieval by following a function from a known value* (mapping number 2 in section 5.2). The second function is a filter that retains only the proteins from humans – this function is written out by the code generator, which consults the *Filters* information from the SSM.

3. *Which proteins have function kinase?* The GRAIL query for this is `Protein which hasFunction Kinase`. The CPL for this query is:

```
{p1 | \p1<-
  get-enzyme-entry-by-node("2.7.-.-")}
```

This query contains a single query component. The planner uses the mapping for *retrieval by evaluating a mapped role* (mapping number 3 in section 5.2). The distinctive feature of the function selection process is that there are several *Mapped Roles* in the SSM that match the query component. These are indexed by the concepts `Protein which hasFunction BiologicalFunction` and `Protein which hasFunction EnzymicFunction` (as `Kinase` $\sqsubseteq$ `EnzymicFunction` $\sqsubseteq$ `BiologicalFunction`). As `Protein which hasFunction EnzymicFunction` is more specialised than `Protein which hasFunction BiologicalFunction`, the former is used to provide the CPL function. The string `"2.7.-.-"` is the node for `Kinase` in the functional classification used by the source *enzyme*.

7. Related Work

This section briefly compares TAMBIS with other knowledge based source integration systems. A more comprehensive survey is provided in [15]. The closest relatives to TAMBIS are other proposals that make use of DLs for query processing in distributed systems. The two proposals that are closest to TAMBIS are OBSERVER [14] and SIMS [1].

In OBSERVER there is no global schema, but rather the emphasis is on peer-to-peer querying among sources, each of which is described using an ontology. DL queries expressed over the ontology of one source are translated by a collection of source-specific rewrite rules into queries over another source. We conducted an experiment in the use of source specific rewrite rules early in the design of TAMBIS, but were concerned that collections of such rules would be difficult both to write and to maintain. OBSERVER does not address the question of how the DL queries over a local ontology are mapped onto the local source. How to do this for sources with relational query interfaces is understood [8], but is an open problem for sources that are not databases, which is the case for many bioinformatics sources.

In SIMS there is a global schema, described in the description logic LOOM. In SIMS, sources are described in the ontology along with the domain concepts, with source-specific concepts subsumed by their source-independent counterparts. This is quite an elegant approach to linking sources with source-independent concepts, but may provide a less flexible integration framework than that supported by source specific (OBSERVER) or source-independent (TAMBIS) mappings. Once a source-independent DL query has been converted into a source dependent DL query, an access plan is generated for optimisation. Our reading of the SIMS papers is that the current planner assumes some measure of query processing capabilities from sources, which is not the case with TAMBIS, as many bioinformatics sources lack such ability.

Other work on querying heterogeneous sources using knowledge based techniques includes the following. Information Manifold [13] uses deductive view definition languages to characterise the information in different sources. Architecturally, this is quite different from TAMBIS, as there is no central ontology (early work on Information Manifold mentioned DL models, but this is absent in later papers). The DWQ project [6] uses a similar approach to Information Manifold for relating sources to a global model. In DWQ, the global model is described using a description logic, but the focus is on data warehousing rather than on the execution of queries at the sources. TSIMMIS [10] is a wrapper-mediator system, with the focus more on effective wrapper construction than on providing a consistent, high-level view of heterogeneous sources. As there is no global schema or ontology in TSIMMIS, it can be seen as addressing different kinds of problems from those that are central to TAMBIS, and in different ways.

8. Conclusions

This paper has described the TAMBIS query processor. The query processor has been fully implemented in Java. The query processing system, including the sources and services model but excluding the user interface, consists of around 4000 lines of Java code.

TAMBIS is distinctive in a number of respects. It is the first source integration system in the important area of bioinformatics to use an ontology as a global schema. It provides a novel ontology-driven user interface, in which users express queries by navigating through the ontology. It provides users with a conceptual level view of unusually diverse and heterogeneous information sources.

The query processor described in this paper is a crucial component in the TAMBIS architecture. Key features include the following:

1. The scope of the query processor – the use of wrappers to provide syntactic consistency to the highly heterogeneous sources allows the sources and services model and query processor to remain manageable in scale and complexity. Wrappers are also helpful in localising changes needed to parallel the rather frequent changes to the call interfaces of bioinformatics sources.

2. The fact that TAMBIS interfaces with sources that generally lack query interfaces, and that queries routinely access multiple sources, means that optimisation must be carried out at some point in the integration system. Although we could have extended the CPL optimiser to optimise at the function level, carrying out planning in the context of the model has been both natural and straightforward.

3. Services already available as a consequence of using a DL, and its associated terminology server, facilitate the query processing at several stages, by: ensuring that only valid, sensible queries are submitted from the query interface; indexing the CPL functions in the sources and services model; computing some collections of mappings used in retrieval and filtering; identifying inverses of roles; and managing the identification of the most specialised CPL functions available for use with a query component.

4. The fact that a challenging domain, and a substantial ontology used for querying that domain, can be addressed using natural extensions of existing query processing techniques. This demonstrates that ontology driven query interfaces to highly heterogeneous sources are a practical proposition.

Acknowledgements: This work is funded by Zeneca Pharmaceuticals and the BBSRC/EPSRC Bioinformatics programme, whose support we are pleased to acknowledge. Significant input to the development of the user interface was received from Alex Jacoby and Gary Ng.

References

[1] Y. Arens, C. Knoblock, and W.-M. Shen. Query Reformulation for Dynamic Information Integration. *J. Intelligent Information Systems*, 6(2/3):99–130, 1996.

[2] P. Baker, A. Brass, S. Bechhofer, C. Goble, N. Paton, and R. Stevens. TAMBIS - Transparent Access to Multiple Biological Information Sources. In *Proc. Int. Conf. on Intelligent Systems for Molecular Biology*, pages 25–34. AAAI Press, 1998.

[3] P. Baker, C. Goble, S. Bechhofer, N. Paton, R. Stevens, and A. Brass. An Ontology for Bioinformatics Applications. *Bioinformatics*, 1999. to be published.

[4] A. Borgida. Description logics in data management. *IEEE Trans. Knowledge and Data Engineering*, 7(5):785–798, 1995.

[5] P. Buneman, S. Davidson, K. Hart, C. Overton, and L. Wong. A Data Transformation System for Biological Data Sources. In *Proc. 21st VLDB*. Morgan Kaufmann, 1995.

[6] D. Calvanese, G. D. Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. Information Integration: Conceptual Modelling and Reasoning Support. In *Proc. COOPIS*, pages 280–291, 1998.

[7] I.-M. A. Chen, A. Kosky, V. Markowitz, and E. Szeto. Constructing and Maintaining Scientific Database Views in the Framework of the Object Protocol Model. In *Proc. SSDBM*. IEEE Press, 1997.

[8] P. Devanbu. Translating Description Logics to Information Server Queries. In *Proc. Conference on Information and Knowledge Management*, pages 256–263. ACM Press, 1993.

[9] T. Etzold, A. Ulyanov, D. Nardi, and W. Nutt. SRS: information retrieval system for molecular biology data banks. *Methods Enzymol.*, 266:114–128, 1996.

[10] H. Garcia-Molina et al. The TSIMMIS Approach to Mediation: Data Models and Languages. *J. Intelligent Information Systems*, 8(2):117–132, 1997.

[11] C. Goble, P. Crowther, and D. Solomon. A medical terminology server. In D. Karagiannis, editor, *Proc. 5th DEXA*, volume LNCS Vol 856, pages 661–670. Springer-Verlag, 1994.

[12] G. Kemp and P. Gray. Using the Functional Data Model to Integrate Distributed Biological Data Sources. In P. Svensson and J. French, editors, *Proc. SSDBM*, pages 176–185. IEEE Press, 1996.

[13] A. Levy, D. Srivastava, and T. Kirk. Data Model and Query Evaluation in Global Information Systems. *J. Intelligent Information Systems*, 5:121–143, 1995.

[14] E. Mena, V. Kashyup, A. Seth, and A. Illarramendi. OBSERVER: An Approach for Query Processing in Global Information Systems based on Interoperation across Preexisting Ontologies. In *Proc. COOPIS*, pages 14–25, 1996.

[15] N. Paton, C. Goble, and S. Bechhofer. Knowledge Based Information Integration Systems. *Information and Software Technology*, 1999. to be published.

[16] A. Rector, S. Bechhofer, C. Goble, I. Horrocks, W. Nowlan, and W. Solomon. The GRAIL Concept Modelling Language for Medical Terminology. *Artificial Intelligence in Medicine*, 9:139–171, 1997.

[17] A. Swami. Optimization of Large Join Queries. In *ACM SIGMOD*, pages 367–376. ACM Press, 1989.