PROVA: Rule-based Java-Scripting for a Bioinformatics Semantic Web

Alexander Kozlenkov[1] and Michael Schroeder[2]

[1] City University, London, UK
[2] Biotec/Dpet. of Computing, TU Dresden, Germany, ms@mpi-cbg.de

Abstract. Transparent information integration across distributed and heterogeneous data sources and computational tools is a prime concern for bioinformatics. Recently, there have been proposals for a semantic web addressing these requirements. A promising approach for such a semantic web are the integration of rules to specify and implement workflows and object-orientation to cater for computational aspects.
We present PROVA, a Java-based rule-engine, which realises this integration. It enables one to separate a declarative description of information workflows from any implementation details and thus easily create and maintain code. We show how PROVA is used to compose an information and computation workflow involving

- rules for specifying the workflow,
- rules for reasoning over the data,
- rules for accessing flat files, databases, and other services, and
- rules involving heavy-duty computations.

The resulting code is very compact and re-usable.
We give a detailed account of PROVA and document its use with a example of a system, PSIMAP, which derives domain-domain interactions from multi-domain structures in the PDB using the SCOP domain and superfamily definitions. PSIMAP is a typical bioinformatics application in that it integrates disparate information resources in different formats (flat files (PDB) and database (SCOP)) requiring additional computations.
PROVA is available at comas.soi.city.ac.uk/prova

Keywords: Rules, Reasoning, Declarative and Object oriented Programming, Workfbws.

1 Background

Systems integration is vital for progress in bioinformatics. Currently, users typically use systems via the web limiting their possibilities. While there are systems, which integrate a number of databases and tools, such as SRS, Expasy, NCBI, etc. these systems are inflexible and it is not possible to freely customize information workfbws. There are some efforts to tightly integrate data and computational resources such as Edit2Trembl [MLFA99,MSA01,MKA00], which integrates annotations of tools

for transmembrane prediction, GeneWeaver [BLJJ00], which integrates genome annotations, MyGrid [SRG03], which provides personalised access to bioinformatics resources focusing on service management and discovery as part of the workflows.

However, many approaches are hard-wired and cannot adapt to changes in the underlying information resources. They also rely on a hand-crafted mapping of schemata of the different information resources. To this end, the semantic web promises a solution to information integration through the use of shared ontologies, which serve as common reference point for the information resources. Essentially, an ontology is a hierarchically structured taxomomy, that maps out a domain. In bioinformatics, GeneOntology [Con00] is a very prominent effort that defines a vocabulary for molecular biology covering processes, functions, and location. Besides ontologies, a semantic web will contain web services, which extend traditional web pages in that they allow access to remote computations and resulting data.

Semantic web that consists of data specified with a hierarchical common ontology and web services for computations can be programmed by marrying concepts from declarative and object-oriented programming. Rules, reasoning, and logic are needed for dealing with ontologies and for specifying workflows and relationships between the data. Object-orientation is needed for reflecting the hierarchy of the ontology and for implementing services and computations. Rule-based programming has another advantage. It transparently extends the expressiveness and power of relational databases that are often underlying bioinformatics databases. Rules serve as virtual table generators that derive knowledge from elementary facts.

Rule-based programming and object-orientation serve complementary purposes. Declarative programming style is very good for flexible data integration, inference, and for specifying if-then logic. Object-orientation is optimal for modeling complex systems that are relatively stable and well understood. Declarative programming is good for logic, while object-orientation is good for computation.

PROVA, the system presented in this paper, offers a clean way of integrating object-oriented Java with rule-based programming, paving a way for intelligent knowledge-intensive data integration useful in knowledge discovery for biomedical research.

2 Motivation

Before we formulate requirements for a more flexible and fruitful bioinformatics web and how our language PROVA meets these requirements, let us analyse the available data and their formats in detail. Although many bioinformatics data resources are kept in flat files or more recently in XML, their nature is relational. End users typically download these files, define database schemes and populate the databases with the data.

Example 1. Relational Databases
All major data source such as e.g. PDB, SCOP, GeneOntology, DIP, SWISSPROT, etc. can be stored as relational tables. Let us consider two examples. PDB [Tea03], the protein databank, stores atomic coordinates of structures determined by X-Ray christallography and nuclear magnetic resonance. Part of a PDB database will have a table with the following schema

PDB: PDB_ID, Atom_ID, Atom type, x, y, z

SCOP [MBHC95], the structural classification of proteins, is based on PDB and hierarchically groups domains of proteins according to their structural composition. A SCOP database contains a table

Scop: PDB_ID, superfamily, family, domain

PSIMAP [DBG+03,BDH+03] is a database of structural domain-domain interactions derived from PDB and Scop. Such interactions can be stored in a table:

PSIMAP: superfamily, superfamily

Once data is available in a relational database format rather than the original flat files, it can be queried using SQL, the structured querying language.

Example 2. SQL queries
SQL could answer queries such as the following:

- PDB: What are the atomic coordinates of the structure with PDB_ID 1goj?
- Scop: How many domains does structure 1goj have?
- Scop: Does 1goj have a domain belonging to the P-loop superfamily?
- Interaction: What are the interaction partners of the P-loop?

However, sometimes SQL is not sufficiently expressive to answer queries.

Example 3. Expressiveness of SQL
SQL does not have the same computational expressiveness as a full programming language. In particular, it cannot express transitive closure. Hence, the queries

- Interaction: List all superfamilies the P-loop can directly or indirectly interact with

There is another problem. Most useful queries will involve more than one data source and thus there have to be joins across databases, which can be distributed.

Example 4. Joins across (distributed) databases

- SCOP/PDB: List all P-loop domains whose resolution is better than 1.5 Angstrom
- SCOP/PDB: For PDB entry 1goj list all atomic coordinates separated by their Scop domains

Finally, many queries will have a computational component to them involving additional tools and computations that process the integrated data.

Example 5. Computations
SCOP/PDB: For a given multi-domain structure in PDB, check for all its pairs of Scop domains whether these domains have at least 5 residue pairs within 5 Angstrom and can hence be said to interact.

Let us summarise the above observations.

1. Relational databases: Most information sources used in bioinformatics can be specified as relational databases.
2. Simple queries: For many purposes, SQL-type querying is useful and sufficient.
3. Expressive queries: Some queries require more expresiveness than SQL
4. Remote access: Resources are often distributed, but accessible through interfaces such as HTML, database server, or web services.
5. Computation: As a rule, information integration also has computation-intensive components.

These observations can serve as requirements, which need to be met to achieve flexible and transparent systems integration. Additionally, a prime concern in systems integration must be the separation of the workflow and the details of the data and computation resources to be integrated.

Rules are a natural extension of relational databases to achieve more expressiveness.

Example 6. Rules as workflows
PSIMAP [DBG+03,BDH+03] considers each multi-domain protein in PDB and establishes an interaction between a pair of domains if they have at least 5 residue pairs within 5 Angstrom. An implementation of PSIMAP will have some complexity, as PDB and Scop need to be wrapped and accessed. Most imperative approaches will mix the workflow with imlementation details such as wrappers and the actual computation. However, using rules the workflow itself can be specified as a single simple rule:

```
PSIMAP(SF1,SF2) if
  PDB(PDB_ID),
  Scop(PDB_ID,SF1,F1,D1),
  Scop(PDB_ID,SF2,F2,D2),
  D1<>D2,
  interact(D1,D2)
```

Superfamilies `SF1` and `SF2` interact if there is a PDB entry `PDB_ID` for which there is a Scop domain `D1` of superfamily `SF1` and a Scop domain `D2` of superfamily `SF2`, where `D1` and `D2` are different, and the two domains interact.

This is the pure workflow and it does not yet specify how the PDB entry is obtained, where the Scop data comes from and how the computation is carried out. These are all implementation details that are defined by separate rules, which in turn may require access to a database or system calls.

Besides using rules to specify workflows, rules can be used to infer relationships from the data. As shown in example 3, we might wish to know all superfamilies a given superfamily such as the P-loop is directly or indirectly connected to.

Example 7. Inference and Rules
Two rules are needed to define if two superfamilies `X` and `Y` are connected, i.e. if they can interact directly or indirectly. First, `X` and `Y` are connected if they interact (directly). Second, `X` and `Y` are connected if `X` interacts directly with a third superfamily `Z`, which is connected to `Y`. In rule format, this looks as follows:

```
connected(X,Y) if interact(X,Y).
connected(X,Y) if interact(X,Z), connected(Z,Y).
```

3 PROVA

Our system PROVA, which is based itself on Mandarax [Die], addresses the above requirements by providing a rule-based Java scripting language. The use of rules allows one to declaratively specify the integration needs at a high-level without any 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 in Java. This section presents rationales and design principles behind the PROVA language. We position this language as a platform for knowledge-intensive ontology-rich applications in biomedical research. We are aiming to satisfy the following design goals with the proposed PROVA language:

- Combine the benefits of declarative and object-oriented programming;
- Merge the syntaxes 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 environement;
- Be compatible with web- and agent-based software architectures;
- Provide functionality necessary for rapid application prototyping and low cost maintenance.

Consider the following PROVA code showing how knowledge can be inferred from the available facts.

Example 8. (Declarative programming)
Consider a table of interacting proteins. We wish to infer all interactions, direct or indirect. In PROVA, this can be specified as follows (`:-` is read as "if"):

```
% Facts (what we know)
interactDirect(a,b).
interactDirect(b,c).
interactDirect(c,d).

% Rules (how to derive new knowledge)
interact(X,Y):-interactDirect(X,Y).
interact(X,Z):-interactDirect(X,Y),interact(Y,Z).
```

The query `:- solve(interact(a,X)).`, which can be read as "which proteins `X` interact with protein `a`?", will return the three answers `X=b`, `X=c`, and `X=d`.

Thus, PROVA follows classical Prolog closely by declaratively specifying relationships with facts and rules. Now let us consider two examples, where access to Java methods is directly integrated into rules.

Example 9. (Object-oriented programming)
The code below represents a rule whose body consists of three Java method calls: the first to construct a String object, the second to append something to the string, and the third to print the string to the screen.

```
hello(Name):-
  S = java.lang.String("Hello "),
  S.append(Name),
  java.lang.System.out.println(S).
```

Java method calls can be used to wrap up computations. As an example, consider the PSIMAP application, in which domain-domain interactions are computed by checking whether 5 residue pairs are within 5 Angstrom. The exact PROVA code to carry out this computation will be dicussed later. However, as an example for method call, consider the line below in which the Java method `interacts` of object `DomainA` is invoked with parameter `DomainB`.

```
scop_dom2dom(...):-
  ...
  DomainA.interacts(DomainB).
```

Let us consider the PROVA syntax in detail. The first example above shows how an object is constructed. The code is very similar to Java itself, only the `new` keyword is missing. Once an object is available in PROVA, methods can be invoked on it in the same way as in Java including both instance (`DomainA.interacts`) and static calls (`java.lang.System.out.println`). The latter needs fully qualified class names. An instance method can fail if an exception is raised or if the unification of the optional returned object with the supplied pattern fails. There are two important rules about the passing and returning of PROVA lists from Java methods. Essentially, all PROVA lists are automatically converted to the Java ArrayList implementing the List interface when passed to a Java method requiring a Collection for the corresponding parameter. Conversely, when a Java Object array (`Object[]`) is returned from a Java method, it is automatically converted into a PROVA list.

The full PROVA syntax in EBNF is shown in appendix A.

3.1 Programming with PROVA

Let us consider two example rules taken from the implementation of PSIMAP, the protein structure interaction map introduced earlier.

Given the general approach of decomposing an application workflow into the logic and computational parts, we capture the knowledge-intensive part of the application in PROVA rules directly while using Java, SQL, or command-line utilities based services for accessing or generating data for which the highest level of performance is needed

and the logical component of the query is minimal. Consider the code fragment below, which implements the following definition: "If there is a PDB entry and according to SCOP this entry has two domains and according to the computation `interacts` at least 5 residue pairs are within 5 Angstrom, then these two domains interact."

```
% Given the open database connection DB
% and a unique protein identifier in Protein
% Data Bank PDB_ID, test whether the provided
% domains with IDs PXA and PXB interact
% (have at least 5 atoms within 5 angstroms)
scop_dom2dom(DB,PDB_ID,PXA,PXB) :-
  access_data(pdb,PDB_ID,Protein),
  scop_dom_atoms(DB,Protein,PXA,DomainA),
  scop_dom_atoms(DB,Protein,PXB,DomainB),
  DomainA.interacts(DomainB).
```

The code shows an example of a data integration rule used for computing the interaction of protein domains. The `scop_dom2dom`-predicate in the head of the rule represents a virtual table (view) composed from computational wrappers and lower-level predicates. The body of the rules includes a generic cache-based access to PDB data via the predicate `access_data`, which is discussed below and which implements an active cache. Given a PDB entry, a pair of domains is returned by the predicate `scop_dom_atoms` and returned in the variables `DomainA` and `DomainB`, respectively. The predicate requires a variable `DB`, which contains a database connection from where the data is read. The use of databases and SQL is discussed below. Finally, an instance method `interacts` returning boolean either succeeds or fails depending on whether the two domains interact or not.

The example shows how external Java based data wrappers fit into the rule-based data integration. Java variables can be constructed and returned from predicates associated with Java calls while boolean methods can be used to test conditions.

Now let us consider how the caching of PDB is implemented. The code below shows how logic representation helps dealing with the complexities of defining the caching mechanism for retrieving any type of data. We represent the data items by the data type (variable `Type`) and a unique ID of a particular data item (variable `ID`). The data is stored in the cache represented by variable `CacheData`.

```
% Top-level rule for accessing Data
% given data Type and particular ID.
access_data(Type,ID,Data) :-
  % Retrieve or create the cache for Type
  cache(Type,CacheData),
  access_data(Type,ID,Data,CacheData).

% Two alternative rules for either retrieving data
% from the cache or accessing the data from its
% original location and caching it.
```

```
access_data(Type,ID,Data,CacheData) :-
  % Attempt to retrieve the data
  Data=CacheData.get(ID),
  % Success, Data (whatever object it is) is returned
  !.

access_data(Type,ID,Data,CacheData) :-
  % Retrieve the data from its location and update the cache
  retrieve_data_general(Type,ID,Data),
  update_cache(Type,ID,Data,CacheData).
```

The code above demonstrates a very high level of abstraction and flexibility offered by rule-based systems. In particular, the cache variable is untyped which means that if implementation for storing a cache changes the code will remain valid. Also, both the ID and the Data itself are also untyped which allows one to use for instance complex IDs represented as lists for unique IDs of the data items. The last two rules show how IF-THEN logic is realised as alternative rules. If `Data=CacheData.get(ID)` succeeds, a cut operator ('!') prevents further backtracking to another rule. Otherwise, the `retrieve_data_general` predicate is used for fetching the data from its original location. If there were several mirrors available for the original data, they are automatically explored until the working mirror is identified.

The procedural code representing this type of logic would have been much more cumbersome to write and maintain. Enhanced level of generality and flexibility is exactly what is required for real-world data integration and manipulation projects involving access to multiple rapidly changing data sources.

3.2 Typing in PROVA

Because of the desired tight integration of PROVA rules with Java code and extended use of types and classes in Java, we have decided to include a type system in PROVA. This language feature is not commonly found in rule-based languages such as Prolog. The rationale behind the introduction of the type system was to offer the user the option to restrict the applicability of rules and to control the level of generality in queries. The notation for typed variables in PROVA normally involves prefixing a variable name with a fully qualified name of the class to which the variable should belong. All classes in the java.lang package can be used without their full prefix specifying their package. Consider a `member` predicate returning the members of a list. The query `member(X,[1,Double.D,"3"]))`, will return `X=1`, `X=java.lang.Double.D`, and `X=3`. The query variable `X` could be further qualified as an integer. Then the query `member(Integer.X,[1,Double.D,"3"]))` will return only `java.lang.Integer.X = 1`, as the other assignments fail as they are double and string, respectively.

In the first query, an untyped variable `X` is used to query for list members. The first variable in the target list is an Integer constant, while the second one is a Double variable, and the last one is a string constant. Three solutions of this query demonstrate this. The second query asks specifically for an Integer list members specifying an Integer variable as the first argument. As a result, only the first element, a constant 1, is returned.

When more complex Java classes are used in PROVA, the following rules apply to variable-variable unification. If the query and target variable are not assignable to each other, the unification fails. Otherwise, the unification succeeds. If the query variable belongs to a subclass of the class of the target variable, the query variable assumes the type of the target variable. If the query variable belongs to a class that is a superclass of the class of the target variable or is of the same class as the target variable, the query variable retains its class.

3.3 PROVA SQL Integration

PROVA SQL integration has a crucial role in providing an efficient and flexible mechanism for data and ontology integration. PROVA offers a seamless integration of predicates with most common SQL queries and updates. The language goes beyond providing embedded SQL calls and attempts to achieve a more flexible and natural integration of queries with PROVA predicates.

```
:-eval(consult("utils.prova")).
:-eval(test()).

location(database,"jdbc string", "database_name",
                              "username","password").
test():-dbopen("database_name",DB).
```

Opening a database only requires calling the `dbopen` predicate and providing the database name. The rules for the `dbopen` predicate together with accompanying rules for caching and mirroring are provided with the supplied external module `utils.prova` that needs to be included (consulted) at the beginning of a user file. Opening database requires one or more location records to be present in the fact base. The location records help organising the information about possible alternative locations of data sources. The data sources are organised according to their type, name, and any required keys for a particular dataset to be retrieved. Caching for datasets is automatically provided.

In the case of opening a database, the data source type is `database`, the data source name is the database name provided as the second argument to location, a JDBC connection string is provided as the third argument, and optional username and password can be provided as strings. If more than one record for a particular database is provided, the system attempts to establish connection with the each one in turn until a successful connection is established or all locations are exhausted and opening the database fails. This mirroring technique is especially useful for web-intensive applications where configuration flexibility is needed. For example, if an application is developed on one computer and then deployed on the web server, the rule system above can be used to achieve complete code independence. The example below illustrates this situation.

```
:-eval(consult("utils.prova")).

location(database,scop,"jdbc:mysql://comas.soi.city.ac.uk","u","p").
```

```
location(database,scop,"jdbc:mysql://localhost","u","p").
dbopen(scop,DB)
```

By default open database connections are cached. If the users wish to override the default on how many database connections are cached they can override the corresponding fact for the predicate `cache_capacity`.

The main format for PROVA predicates dynamically mapped to SQL Select statements is shown below:

```
sql_select(DB,Table,[N1,V1],...,[Nk,Vk],
     [where,Where],[having,Having],[options,Options])
```

The built-in `sql_select` predicate non-deterministically enumerates over all possible records in the result set corresponding to the query. The predicate fails if the result set is empty or an exception occurs. It accepts a variable number of parameters of which only the first two are required. DB corresponds to an open database connection and Table is the name of the table to be queried. Note that the table name can be a variable that only becomes instantiated during the execution of the code. Not only the table name can be determined dynamically, but also all the remaining parameters can be either variables or constants or even the whole list of parameters can be dynamically constructed.

The most important part of the syntax of `sql_select` is 0 or more field name-value pairs `[N1,V1],...,[Nk,Vk]`. `N1,...,Nk` correspond to field names included in the query. As opposed to ordinary SQL Select statements, this list of fields includes both the fields to be returned from the query and those that can be supplied in the SQL Where clause. Whether a particular field `Ni` will be returned or used as a constraint depends on the values `Vi` corresponding to these field. If `Vi` is a constant at the time of the invocation, it becomes a constraint in the automatically constructed Where clause. Otherwise, `Vi` is an uninstantiated (free) variable and will be returned by the query in each record in the result set. In addition to simple field names, `N1,...,Nk` can be strings containing special SQL modifiers such as Distinct (for example, "distinct name") or group functions such as Count (for example, "count(px)").

The remaining parameters are entirely optional. In the pair `[where,Where]`, "where" is a reserved word and `Where` is a variable or constant containing an explicit SQL Where clause. This syntax is useful in situations requiring the use of such constraints as `Like` or `Rlike`, for example, `[where, "pdb_id like '%%gs'"]`. The pair `[having,Having]` allows specifying a post processing filter on the results returned by the query, for example, `[having,"count(px)>1"]`. A large variety of other modifiers for the query can be included with the `[options,Optioons]` pair, for example, `[options,"order by count(px) desc limit 10"]`. A number of examples of SQL Select mapped predicates working with SCOP are shown below.

```
sql_select(DB,cla,[pdb_id,"1alx"],[px,Domain])
sql_select(DB,cla,[pdb_id,PDB_ID],[count(px),2])
sql_select(DB,cla,[pdb_id,PDB_ID],[count(px),Count])
sql_select DB,cla,[pdb_id,PDB_ID],[count(px),Count],
   [where,"pdb_id like '%%gs'"]
sql_select(DB,cla,["distinct pdb_id",PDB_ID],[options,"limit 10"])
```

Although only queries corresponding to a single table are supported at this moment, queries with joined tables can be constructed by combining several single table queries. A future optimisation will provide an automatic construction of joins from `sql_select` predicates that have one or more common variables. The example below shows how two `sql_select` calls can be used to compute an inner join for table `cla` finding two different domains `PXA` and `PXB` belonging to the same PDB file.

```
sql_select(DB,cla,[px,PXA],[pdb_id,PDB_ID]),
sql_select(DB,cla,[px,PXB],[pdb_id,PDB_ID]),
PXA<PXB
```

PROVA provides a built-in predicate `sql_insert` providing a flexible mapping to SQL Insert statements.

```
sql_insert(DB,Table,[N1,...,Nk],[V1,...,Vk])
```

The `sql_insert` predicate is structured differently from `sql_select` in that it accepts the field names as a separate sublist in the third argument and the fourth argument contains a sublist with values corresponding to these fields. The example below shows a complete rule that parses a text-based database file with descriptions of protein domains from the SCOP database.

```
db_import(DB,scop,des,Line) :-
tokenize_list(Line,"\t",[T|Ts]),
sql_insert(DB,des,[id,type,sccs,sid,description],[T|Ts]).
```

The predicate `db_import` receives an open database connection `DB`, the name `scop` of the database to be imported, the name of the table to insert records into, and a `Line` from the text file. The built-in predicate `tokenize_list` builds `Line` tokens separated by tab characters and outputs them in the list `[T|Ts]`. Finally, `sql_insert` inserts a new record into the table `des` with the specified fields and the values copied from the list of tokens.

4 Comparison and Conclusion

Rules play an important role in bioinformatics. They come in many guises such as database views, logic programs, constraints, active rules, description logics, ontologies, etc. They have been used in many different contexts such as constraints for structure prediction [BWBB99,KB02,PKWM00], model checking of biochemical networks [CF03], logic programming for consistent annotation of proteins [MLFA99,MSA01,MKA00], ontologies for transparent access to bioinformatics databases [LJ03,LE03], ontologies for health applications [GE01,GES01], and integration of an ontology with gene expression data [Bad03,BT03]. All but the first two applications involve systems integration and therefore showcase the usefulness of rules, logic and ontologies for this task.

In this paper, we have argued that rules alone are not sufficient to easily facilitate systems integration. We argued that a rule-based approach also needs to cater for computation, database access, communication, web services, etc. All of these requirements

can be met by integrating a rule-based approach with an object-oriented approach such as Java. The challenge is to make this integration transparent and get the best out of two worlds.

There are a number of approaches that integrate object-orientation and deduction such as Rock&Roll [FWPM94], ConceptBase [JGJ+95], and Flora [YK00]. These approaches aim for a conceptually coherent integration of the two different programming paradigmns. In contrast to this, PROVA aims to provide a rule-engine for Java, which caters for rule-based processing in Java. At first glance, this objective seems to be addressed also by catering for native method calls in Prolog engines. as exemplified by systems such as JIProlog. There is however a difference to PROVA: PROVA emphasises flexible integration as opposed to achieving simple interfacing of Java with Prolog. In the following, we provide some comments for features listed in the table. This different perspective leads to a host of features absent from JIProlog, but provided by PROVA such as native syntax Java call, Java type system within the rules, access to static java variables from the rules, automatic conversion of returned Java object lists into Prolog lists, and interpretation instead of compilation. Furthermore, PROVA provides features specifically useful for systems integration, namely variable arity and argument tails, flexible database access, predicate names as variables, message-passing and reaction rules for implementing active behaviour.

All in all, PROVA is a first step towards realising a semantic web for bioinformatics by declaratively and transparently integrating data and rules including database access and computations captured in procedural (Java) code. PROVA facilitates the separation of declarative workflows from implementation details and thus leads to more compact and maintainable code. We have shown how all of the above features are used to implement the PSIMAP system, which computes domain-domain interactions from SCOP and PDB.

References

[Bad03] Liviu Badea. Functional discrimination of gene expression patterns in terms of the Gene Ontology. In *Proceedings of the Pacific Symposium on Biocomputing - PSB03*, 2003.

[BDH+03] Dan Bolser, Panos Dafas, Richard Harrington, Jong Park, and Michael Schroeder. Visualisation and graph-theoretic analysis of the large-scale protein structural interactome network psimap. *BMC Bioinformatics*, 4(45), 2003.

[BLJJ00] K. Bryson, Michael Luck, Mike Joy, and D. T. Jones. Applying agents to bioinformatics in geneweaver. In *Cooperative Information Agents*, pages 60–71, 2000.

[BT03] Liviu Badea and Doina Tilivea. Integrating biological process modelling with gene expression data and ontologies for functional genomics (position paper). In *Proceedings of the International Workshop on Computational Methods in Systems Biology*, University of Trento, 2003. Springer-Verlag.

[BWBB99] Rolf Backofen, Sebastian Will, and Erich Bornberg-Bauer. Application of Constraint Programming Techniques for Structure Prediction of Lattice Proteins with Extended Alphabets. *Journal of Bioinformatics*, 15(3):234–242, 1999.

[CF03] Nathalie Chabrier and François Fages. Symbolic model checking of biochemical networks. In *Proceedings of the First International Workshop on Computational*

Methods in Systems Biology CMSB'03, LNCS, Riverto, Italy, March 2003. Springer-Verlag.

[Con00] The Gene Ontology Consortium. Gene ontology: tool for the unification of biology. *Nat Genet*, 25:25–29, 2000.

[DBG+03] Panos Dafas, Dan Bolser, Jacek Gomoluch, Jong Park, and Michael Schroeder. Fast and efficient computation of domain-domain interactions from known protein structures in the PDB. In H.W. Frisch, D. Frishman, V. Heun, and S. Kramer, editors, *Proceedings of German Conference on Bioinformatics*, pages 27–32, 2003.

[Die] J. Dietrich. *Mandarax*. http://www.mandarax.org.

[FWPM94] A.A.A. Fernandes, M.H. Williams, N. Paton, and M.L. Maria. Object-Oriented Database Programming Languages Founded on an Axiomatic Theory of Objects. In *Workshop on Logical Foundations of Object-oriented Programming*, 1994.

[GE01] Rolf Grütter and Claus Eikemeier. Development of a Simple Ontology Definition Language (SOntoDL) and its Application to a Medical Information Service on the World Wide Web. In *Proceedings of the First Semantic Web Working Symposium (SWWS '01)*, pages 587–597, Stanford University, California, July/August 2001.

[GES01] Rolf Grütter, Claus Eikemeier, and Johann Steurer. Towards a Simple Ontology Definition Language (SontoDL) for a Semantic Web of Evidence-Based Medical Information. In S. Quaglini, P. Barahona, and S. Andreassen, editors, *Artificial Intelligence in Medicine. 8th Conference on Artificial Intelligence in Medicine in Europe, AIME2001*, Cascais, Portugal, July 2001. Springer-Verlag.

[JGJ+95] M. Jarke, R. Gallersdörfer, M.A. Jeusfeld, m. Staudt, and Stefan Eberer. CConceptBase - a deductive object base for meta data management. *J. of Intelligent Information Systems*, February 1995.

[KB02] L. Krippahl and P. Barahona. PSICO: Solving Protein Structures with Constraint Programming and Optimisation. *Constraints*, 7(3/4):317–331, July/October 2002.

[LE03] P. Lambrix and A. Edberg. Evaluation of ontology merging tools in bioinformatics. In *Proceedings of the Pacific Symposium on Biocomputing - PSB03*, pages 589–600, 2003.

[LJ03] P. Lambrix and V. Jakoniene. Towards transparent access to multiple biological databanks. In *Proceedings of the First Asia-Pacific Bioinformatics Conference*, pages 53–60, Adelaide, Australia, 2003.

[MBHC95] A.G. Murzin, S.E. Brenner, T. Hubbard, and C. Chothia. SCOP: a structural classification of proteins database for the investigation of sequences and structures. *J. Mol. Biol.*, 247:536–540, 1995.

[MKA00] S. Möller, E. V. Kriventseva, and R. Apweiler. A collection of well characterised integral membrane proteins. *Bioinformatics*, 16(12):1159–1160, 2000.

[MLFA99] S. Möller, U. Leser, W. Fleischmann, and R. Apweiler. EDITtoTrEMBL: a distributed approach to high-quality automated protein sequence annotation. *Bioinformatics*, 15(3):219–227, 1999.

[MSA01] S. Möller, M. Schroeder, and R. Apweiler. Conflict-resolution for the automated annotation of transmembrane proteins. *Comput. Chem.*, 26(1):41–46, 2001.

[PKWM00] P.N. Palma, L. Krippahl, J.E. Wampler, and J.J.G. Moura. BiGGER: A new (soft) docking algorithm for predicting protein interactions. *Proteins: Structure, Function, and Genetics*, 39:372–384, 2000.

[SRG03] Robert D. Stevens, Alan J. Robinson, and Carole A. Goble. mygrid: personalised bioinformatics on the information grid. In *Eleventh International Conference on Intelligent Systems for Molecular Biology*, volume 19, 2003.

[Tea03] The PDB Team. The protein data bank. In *Structural Bioinformatics*, pages 181–198. Wiley, 2003.

[YK00] G. Yang and M. Kifer. FLORA: Implementing an efficient DOOD system using a tabling logic engine. In *LNAi 1861*. Springer, 2000.

A Syntax of PROVA

```
                  prova ::= statements, end of fi le;
             statements ::= statement, statements;
              statement ::= (fact — rule — query), end of statement;
                   fact ::= relation;
                   rule ::= relation, ":-", atoms;
                  query ::= ("eval" — "solve"), "(", relation, ")";
                  atoms ::= atom, ",", atoms;
                   atom ::= relation — arithmetic relation — java call — cut;
               relation ::= predicate symbol, "(", terms, "—", argument tail, ")";
        argument tail ::= variable;
     predicate symbol ::= lowercase word — uppercase word;
              java call ::= functional java call — predicate java call — constructor java call;
 functional java call ::= left term, "=", predicate java call;
  predicate java call ::= static java call — instance java call;
       static java call ::= qualifi ed java class,".",method name,"(", terms,")";
   instance java call ::= variable, ".", method name, "(", terms, ")";
constructor java call ::= left term,"=",qualifi ed java class,"(",terms,")";
                 terms ::= term, ",", terms;
                  term ::= left term — (func, "(", terms, ")");
             left term ::= variable — constant — prova list;
                  func ::= variable — constant;
              variable ::= uppercase word — typed variable;
              constant ::= lowercase word — ('"', string, '"') ;
        typed variable ::= qualifi ed java class, uppercase word;
            prova list ::= "[]" — ("[", head, "—", tail, "]");
  arithmetic relation ::= left term, binary operator, term;
      binary operator ::= "=" — "!=" — "¿" — "¡" — "¿=" — "¡=";
                  head ::= term;
                  tail ::= variable;
     uppercase word ::= ["A"-"Z","_"], lowercase word;
     lowercase word ::= ["a"-"z"], word;
                  word ::= ["a"-"Z",0-9]+;
                   cut ::= "!";
   end of statement ::= ".";
```