Inductive Logic Programming for Bioinformatics in Prova

Adrian Paschke
RuleML Inc., Canada
adrian.paschke AT gmx.de

Michael Schröder
Biotec/Dept. of Computing, TU Dresden, Germany
ms AT biotec.tu-dresden.de

ABSTRACT

This paper describes the inductive logic programming (ILP) features of Prova, a state-of-art distributed Semantic Web and Life Science inference service system and architecture for multi-relational data mining of complex Life Science phenomena such as complex biological relationships. The proposed novel design artifact implements typical ILP inference formalisms for rule-based generalization and specialization and combines them with expressive logic-based formalisms such as scoped meta-data based reasoning and typed logic in order to constrain the search space and the level of generality of relevant background knowledge. The tight integration of declarative rule-based programming with object-oriented programming (Java) allows outsourcing of computation intensive functionalities such as aggregations and data selections to highly optimized procedural code and query languages such as SQL, XQuery, OWL2Prova RDF, SPARQL. Parallel processing of ILP tasks is supported by a distributed service-oriented and event-driven middleware where several Prova rule engine instances are deployed on the Web as distributed inference services having access to modular data sources and distributed web-based resources. As a result our approach preserves the high expressiveness and flexibility of ILP for multi-relational data mining and attempts to overcome well-known computational and logical problems of ILP when facing very large and scattered heterogenous amounts of data with complex relationships published on the (Semantic) Web.

1. INTRODUCTION

Typical propositional data mining approaches use a simplified assumption that all data is stored in a single relation and that each object of interest is represented by one row. However, mining biological data, such as in molecular biology, requires expressive, efficient and scalable multi-relational data mining algorithms to find highly complex structural elements in multiple and possibly distributed data relations. In data mining there exists two main approaches for handling relational data: Inductive Logic Programming (ILP) and Propositionalization. [13]

Propositionalization converts the relational complex data into a flat propositional representation and generates one single relation out of multiple relations such that typical propositional learners can be applied. This can be achieved by using e.g. typical aggregation functions as provided by relational database languages such as SQL or by generating features (attributes) by applying logic-oriented propositionalization.

In contrast, ILP systems directly operate on multiple relations where the relational patterns are represented as subsets of first-order logic as logic programs (LPs) consisting of rules and facts. They search for regularities by inductively generalizing the specialized individual instances to more general rules which describe new relations.

Both approaches have pros and cons. Database oriented propositionalization approaches allow using highly-optimized queries and aggregations to reduce the number of relations and apply efficient propositional learning techniques. However, beside the computational costs of joins, they typically produce one huge propositional relation with a large number of possible redundant features which might negatively impact the performance of learning algorithms. ILP systems directly operate on multi-relational models, provide expressive declarative representation languages (logic programming languages) and can handle additional (user-defined) background knowledge to substantially improve the results of learning in terms of accuracy and efficiency. On the other hand, large background knowledge bases (KBs) with many irrelevant information for the problem might have the opposite effect since the induction algorithm has to search over all the relations and rules and generalized model construction might take very long or even be infinite (depending on the logic class).

In this paper we introduce Prova, a distributed web-based rule inference system, which combines expressive declarative logic programming techniques with procedural object-oriented programming and distributed web technologies. In particular, we describe the ILP meta program implementations of Prova which beside the inductive logical inference algorithms allows utilizing expressive logical formalisms for, e.g., building constructive scopes on modular (distributed) KBs, object-oriented (OO) description languages with external OO and Semantic-Web type systems (e.g. meta data vocabularies, ontologies), integration of multiple external tools and data from e.g. relational databases, and parallel computation by distributing inference tasks to multiple web-based Prova inference services deployed on a stable and highly scalable service-oriented communication middleware. This novel integrated approach preserves the expressiveness benefits of ILP and adopts the aggregation and constructive view approach of relational database systems to logic programming. Moreover, it addresses the heterogeneousness of complex data and data types in the Life Science domain by integrating Semantic Web domain ontologies and meta data and considers computational complexity due to large and and increasing amounts of data via distribution of computational tasks to multiple Prova inference services (akin to service grids) for parallel computation.

The further paper is structured as follows: Section 2 describes the relevant background in ILP. Section 3 implements the ILP formalisms of Prova and elaborates on several expressive formalisms in Prova which can be used to access and query external data sources using existing highly optimized query languages and construct modular scopes on the possible distributed knowledge base in order to constrain the search space on relevant background knowledge. Section 4 extends Prova with a highly scalable and efficient service-oriented middleware for deploying several Prova rule engines as distributed inference services on the Web. The middleware features complex event processing and conversation-based messaging for seamless integration of external tools and resources and for distributing ILP tasks in the Prova service grids. Finally, section 5 summarizes the novel design artifact for distributed rule-based ILP proposed in this paper.

2. INDUCTIVE LOGIC PROGRAMMING

In the following, we assume that the reader is familiar with logic programming techniques [3]. We use the standard LP notation with an ISO Prolog related scripting syntax, i.e. variables start with upper-case letters, constants/individuals with lower-case letters.

ILP is a research area at the intersection of machine learning and logic programming. [13] It allows inductively deriving general information from specific knowledge. Traditionally, ILP has been concerned with finding patterns expressed as logic programs. In recent years, however, the scope of ILP has broadened to cover the whole spectrum of data mining tasks (classification, regression, clustering, association analysis). There are two main directions in ILP: learning from entailment and learning from interpretations. Learning from entailment is also called explanatory ILP. Most of ILP systems are learning from entailments (e.g. RDT, Progol, FOIL, SRT, Fors). Learning from interpretations is also called descriptive ILP. Examples of the ILP systems which are based on this setting are Claudien, ICL, and Tilde. The differences between the two ILP approaches are in the way they represent the examples, the background knowledge and the way the final hypothesis is induced. The entailment paradigm represents all the data examples together with the background knowledge as one LP. Background knowledge is a prior knowledge, provided by the user to be used in the construction of rules. In ILP background knowledge is expressed in the form of clauses (facts or rules) and is used in the construction of relations. ILP generalizes from individual instances or observations in the presence of background knowledge, finding regularities or hypotheses about yet unseen instances. It learns from examples, usually positive ground clauses as positive examples (+ negative examples) with additionally taking background knowledge into account. To test the coverage of the learned hypothesis, a function $covers(H, E)$ returns the value true if E (the examples) is covered by H (the hypothesis), and false otherwise. ILP systems can be differentiated into systems which only learn one hypothesis or several, systems which know all examples from the beginning (batch learner, e.g. empirical ILPs such as FOIL, MARKUS, GOLEM, LINUS) or incrementally learn them (incremental learner, e.g. MIS, MARVIN, CLINT, CIGOL), and systems which might ask an additional oracle (interactive) or not (non-interactive).

To enable a direct and efficient search the search space for hypothesis needs to be structured in a certain way. θ-subsumption ordering introduces a syntactic notion of generality: A rule (clause) r (resp. a term t) θ-subsumes another rule r', if there exists a substitution θ, such that $r \subseteq r'$, i.e. a rule r is *as least as general as* the rule r' ($r \leq r'$), if r θ-subsumes r' resp. *is more general than* r' ($r < r'$) if $r \leq r'$ and $r' \nleq r$. (see e.g. [12]). Specialization techniques search the hypothesis space in a top-down manner, from general to specific hypotheses, using a θ-subsumption-based specialization operator, called refinement operator. Generalization techniques search the hypothesis space in a bottom-up manner. Bottom-up learners start from the most specific clause that covers a given example and then generalize the clause until it cannot further be generalized without covering examples. Two basic generalization techniques are: *relative least general generalization* (rlgg) and *inverse resolution*. A lgg is the generalization that keeps an generalized term t (or clause) as special as possible so that every other generalization would increase the number of possible instances of t in comparison to the possible instances of the lgg. The extension of lgg builds the relative least general generalization (rlgg), which takes into consideration available background knowledge. Inverse resolution faces the following "inverse" problem: given a clause R and a parent clause C_1, find a second parent clause C_2 such that R is an instance of a resolvent of C_1 and c_2. θ-subsumption and rlgg has some nice computational properties and it works for simple terms as well as for complex terms, e.g. $p() : -q(f(a))$ is a specialization of $p : -q(X)$. θ-subsumption and lgg are purely syntactic notions. Their computation is therefore simple, as compared to inverse resolution or inverse implication, which are both computationally intractable. Thus, *theta*-subsumption and rlgg qualify to be the right framework of generality in the application of ILP in the domain of bioinformatics data mining.

3. INDUCTIVE LOGIC PROGRAMMING IN PROVA

Among other application domains the Prova project [1] is addressing Semantic Web Life Science applications [2]. It follows the spirit and design of the recent W3C Semantic Web initiative and combines declarative rules, ontologies and inference with dynamic object-oriented Java API calls and access to external data sources such as relational databases or enterprise applications and IT services. One of the key advantages of Prova is its elegant separation of logic, data access, and computation and its tight integration of Java and Semantic Web technologies. In the following we first describe the ILP support of Prova and then elaborate on several expressive extensions of Prova in this context.

3.1 ILP Meta Program

The ContractLog KR [8, 5, 7] of Prova implements a meta inference engine which allows

- computing the substitution sets of terms and clauses,
- apply the substitutions to compute specializations of clauses (instantiations of rules),
- generalize clauses/terms and compute the (r)lgg
- compute the coverage

The ILP inference engine is implemented as a meta program (as a Prova LP script). Meta-programming and meta-interpreters have their roots in the original von Neumann computer architecture where program and data treated in a uniform way and are a popular technique in logic programming for representing knowledge, in particular, knowledge in the domains containing logic programs as objects of discourse. LPs representing such knowledge are called meta-programs (a.k.a. meta interpreters) and their design is referred to as meta-programming. The core inference functions implemented in the meta program are:

Specialization

- $substitution(Term1, Term2, Subst)$ - Compute and return the substitution S of two terms $t1$ and $t2$.
- $substitute(Clause, ClauseInstance, Subst)$ $substitute(Term, TermInstance, Subst)$ - Apply the substitutions to a clause/term and compute the specialized instance
- $specializations(Goal, Clause, Instances)$ - Unify (i.e. specialize) a clause (rule) with a goal (set of subgoals)and compute the specialized top level instances (specializations)
- $specialize(Goal, InputLP, OutputLP)$ - Specialize an input LP (set of clauses) with a goal and return the specialization of the LP (i.e. set of top level clause instances).

Generalization

- $lgg(Clause1, Clause2, LGG)$ - compute (r)lgg of two clauses
- $lgg(Term1, Term2, LGG)$ - compute (r)lgg of two terms
- $lggs(Clause, LP, LGGs)$ - compute all (r)lggs of a clause and a LP (set of clauses)
- $generalize(InputLP, OutputLP)$ - Generalize an input LP (set of clauses) and returns the generalized and minimalized (compacted) output LP using relative least general generalization with the given background knowledge in the input LP.

Cover / Coverage

- $cover(LP1, LP2, CoveredClause)$ - Return the covered clause from both LPs, i.e. the clauses which are variants
- $coverage(Goal, LP, CoveredClauses, NotCoveredClauses, CoverageLevel)$ - Computes the test coverage for a given hypothesis and a given LP.

The specialization and generalization functions can be used to define meta reasoning rules for reasoning on top of the LP/knowledge base and the contained rules, where a logic program is viewed as a single logical formula. For example, recursively computing the substitution sets, the substitutions and continuing this process with the body literals (sub goals) of the computed substitution leads to a standard top-down derivation. In order to enable processing of clauses and their terms in the ILP meta inference engine, queries, rules and facts are internally represented in a list format, e.g. a term $p(X)$ can be equivalently represented as $[''p'', X]$. That is, rules a represented as a list starting with the head literal and then the body literals, e.g. $p(X) : -q(X)$ is written as $[[p(X)], [q(X)]]$. A fact is a rule consisting only of the rules' head, e.g. the fact $q(a)$ is written as $[[p, a]]$ or equivalently $[p(a)]$.

Here are some examples to illustrate the use of the inductive logic / meta inference functions and the list representation:

```
% compute the substitution set for the two complex terms
:-solve(substitution(f(g(A),B),f(g(h(a)),i(b)),Subst)).

% substitute a complex term with the substitution set
% {(A / h(a)),(B / h(b))}
:-solve(substitute(f(g(A),A),Instance,
                   [[A,["h","a"]],[B,["h","b"]]])).

% compute the lgg = f(X, g(Y,Z), c).
:-solve(lgg(f(a, g(b, h(X)), c), f(d, g(j(X), a), c),LGG)).

% Generalize a LP with the rules p(a):-q(a). p(a):-q(a),r(a).
% ... and the facts r(a). q(a).  ...
% and return the generalized LP (set of general rules)
:-solve(generalize([
        [p(a),q(a)],
        [p(a),q(a),r(a)],
        [p(b),q(b)],
        [p(c),q(c)],
        [r(a)],[q(a)],[q(b)],[q(c)]],
    Generalization)).
```

The special built-in predicate $metaLP(LP)$ (coming with the ContractLog distribution or the Prova distribution since 2.0) automatically translates the internal rules/facts of the knowledge base into the list representation format and binds it to the variable LP. The "meta" LP can then be used in further meta reasoning rules, e.g. the rule $clauses(Clause) : -metaLP(LP), member(Clause, LP)$. returns all clauses of the logic program using the member function on the list of clauses bound the the variable LP. The combination of generalization and specializations allows implementing typical top-down and bottom-up ILP learning algorithms as well as combinations of both (akin to Muggletons' unifying framework of generalization which combines rlgg and inverse resolution.

As it is well-known several problems in pure $theta$- subsumption and rlgg arise due to the combinatorics of the search-space (the space is infinite in multi-relational models) and the determinancy problem. To make the search space tractable and efficient, it is thus necessary to constrain the search space in some way. In the following subsections we will elaborate on several formalisms in Prova which can be used to limit the number of clauses in an useful way.

3.2 Aggregations and Constructive Scopes

A common technique in logic-based propositionalization to reduce the number of relations to be considered for feature generation is aggregation by using aggregation functions as provided by SQL. Aggregation replaces a set of values by a suitable single value that summarizes properties of the set. The data in Prova can either be available locally as facts in the KB, or dynamically accessed via database queries on arbitrary external data sources such as relational databases, XML documents, Semantic Web RDF or RDFS/OWL ontologies which can be queried by several built-in query languages (e.g. SQL, RDF, XQuery, SPARQL) or wrapped via

Java APIs (e.g. local enterprise java beans or distributed web services):

Prova Java Integration: The tight and natural Java integration of Prova [2] allows dynamically calling external procedural Java methods during runtime. That is, efficient procedural code can be integrated into the rule executions and used for dynamically accessing external data sources and tools using their programming interfaces (APIs). Methods of classes in arbitrary Java packages can be dynamically invoked from Prova rules. The method invocations include calls to Java constructors creating Java variables and calls to instance and static methods for Java classes. The example below shows how XML Document Object Model (DOM) is manipulated in the code. Prova provides a special wrapper object for XML DOM with a built-in class XML. The objects of this class can be constructed from *StringReader* objects and can be manipulated with ordinary methods of the standard Java *org.w3c.dom.Document* class.

```
 attachResults(Doc,Root,XMLPapers) :-
    element(XMLPaper,XMLPapers),
    ResId = XMLPapers.indexOf(XMLPaper),
    StringReader = java.io.StringReader(XMLPaper),
    Document = XML(StringReader),
    ResRoot = Document.getDocumentElement(),
    ResRoot.setAttribute("ResId",ResId),
    Paper = Doc.importNode(ResRoot,Boolean.TRUE),
    Root.appendChild(Paper),
    fail().
attachResults(Doc,Root,XMLList).
```

The Java list $XMLPapers$ contains papers returned from a query to an external database. The built-in predicate *element* non-deterministically enumerates each paper in the list. The method $indexOf$ invoked on the list $XMLPapers$ returns $ResId$ as the sequential number of the current paper. An XML DOM document is imported from the text based XML representation contained in $XMLPaper$ by first creating a *StringReader* object from it and then constructing an XML DOM object. The root attribute is set in the next two lines and then standard Java XML *importNode* and *appendChild* methods are used to append the *Paper* node to the XML DOM in *Doc*.

By calling external Java methods, computation intensive functions can be implemented by highly optimized procedural code and external data sources can be accessed via calling Java wrappers and Java (web) service APIs. For typical data sources such as relational databases, Semantic Web and XML documents, Prova provides specialized query and update built-ins.

Prova SQL Integration: Provas' SQL integration has a crucial role in providing an efficient and flexible mechanism for relational data 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.

The main format for Prova predicates dynamically mapped to SQL Select statements is as follows:

```
sql_select(DB,From,[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 $From$ is either the name of a table to be queried or a valid $From$ clause in SQL syntax enclosed in single or double quotes. $From$ can be a variable but it must become instantiated before the execution of the query. Not only the $From$ clause 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 (with possible modifiers) 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 automatically constructed part of a 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 un-instantiated (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, *distinctname*) 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. An automatically constructed $Where$ clause part is concatenated via AND with the explicit $Where$ clause specified in this parameter. This syntax is useful in situations requiring the use of such constraints as *Like* or *Rlike*, for example, $[where, "pdb_i dlike'\%\%gs'"]$. 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 $[order, Order]$ pair. Queries with joined tables can either be constructed by combining several single table queries or by using a composite $From$ clause and making sure each field name is prefixed with either the corresponding table name or an alias variable if a syntax table as alias is used in the $From$ clause.

```
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"])

sql_select(DB,'cla as c1,cla as c2',['c1.px',PXA],['c2.px',PXB],
['c1.pdb_id',PDB_ID],[where,'c1.pdb_id=c2.pdb_id and c1.px<c2.px'])
```

The *where* clause can be used to define a view on the relational data base and constrain the number of considered instances. The last rules in the example 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. Beside querying a database Prova also supports built-ins for inserting knowledge and updating databases.

Prova RDF / Ontology Integration: As for SQL, Prova provides a special RDF query predicate which can be used in the body of rules to interact with Semantic Web data sources and explicitly express queries, such as concept membership, role membership or concept inclusion on the ontologies. [4, 6] The special query predicate *rdf* is used to query external ontologies written in RDF(S) or OWL (OWL Lite or OWL DL).

```
% Bind all individuals of type "Gene" to the variable "Subject"
%using the owl ontology "gene1.owl" and the "rdfs" reasoner
rdf(
    "http://www.gene.com/gene1.owl",
    "rdfs",
    Subject,"rdf_type","gene1_Gene")
```

The first argument specifies the URL of the external ontology. The second argument specifies the external reasoner which is used to infer the ontology model and answer the query. The hybrid approach provides a technical separation between the inferences in the ontology (Description Logic) part which is solved by an optimized external DL reasoner and the Logic Programming components which is solved by the rule engine. As a result the combined heterogenous integration approach is robustly decidable, even in case where the rule language is far more expressive than Datalog. Moreover, the triple-based query language also supports queries to plain RDF data sources. The following predefined reasoner are supported:

- "" — "empty" — null = no reasoner
- default = OWL reasoner
- transitive = transitive reasoner
- rdfs = RDFS rule reasoner
- owl = OWL reasoner
- daml = DAML reasoner
- dl = OWL-DL reasoner
- swrl = SWRL reasoner
- rdfs_full = rdfs full reasoner
- rdfs_simple = rdfs simple reasoner
- owl_mini = owl mini reasoner
- owl_micro = owl micro reasoner

User-defined reasoners can be easily configured and used. By default the specified reasoners are used to query the external models on the fly, i.e. to dynamically answer the queries using the external reasoner. But, a pre-processing mode is also supported. Here the reasoners are used to pre-infer the ontology model, i.e. build an inferred RDF triple model where the logical DL entailments such as transitive subclasses are already resolved at compilation time. Queries then operate on the inferred model and are hence much fast to answer, however with the drawback that updates of the ontology model require a complete recompilation of the inferred model.

Prova Meta-Data LPs and Scoped Reasoning: To capture the often distributed and open structure of multi-relational knowledge/data bases which are deployed on the Web Prova implements *expressive updates and imports* of Prova scripts (knowledge modules) from web URIs, *meta-data annotated labelled logic programs (LLPs)* and *scoped reasoning*. [8] Arbitrary meta-data such as rule labels, module labels or Dublin Core annotations (e.g., author, date, topic) can be attached to rules and facts. These additional meta-data annotations become in particular interesting when the knowledge base consists of several (possibly distributed) rule sets, so called modules, which might be dynamically imported from different external sources accessible by their Web-based URIs. The meta-data might be used to to create constructive (explicitly closed) views on the distributed KB via *scoped reasoning* by scoped queries, e.g., "all rules/facts to a particular topic" or "all facts with time stamps after a certain date/time". Hence, scoping leads to much smaller search spaces and allows an explicit management of the level of generality of queries/goals.

To explicitly annotate clauses in a labelled logic program (LLP) P with an additional set of meta-data labels Prova introduces a general n-ary *metadata* function into the LP language. The function *metadata* is a partial injective labelling function that assigns a set of meta data annotations m (property-value pairs) to a clause cl in the program P, i.e., $m : cl$. It is syntactically defined separated from a clause (rule/fact/query) by "::":

$metadata(L_1, .., L_n) :: H \leftarrow B$

where L_i are a finite set of unary positive literals (positive meta data literals) which denote an arbitrary meta data $property(value)$ pair, e.g., $label(rule1)$. The explicit $metadata()$ annotation is optional, i.e., a program P without meta data annotated clauses coincides with a standard unlabelled LP.

```
metadata(label(rule1), topic("mutagenesis"), dc_date(2006-11-12))::
    p(X):-q(X).
metadata(label(fact1))::q(1).
% scoped query using topic as scope
:-solve(scope(p(X),topic("mutagenesis"))).
```

The example shows a rule with rule label $rule1$, a topic *mutagenesis*, an additional Dublin Core annotation $dc\_date(2006-11-12)$ and a fact with fact label $fact1$. The meta annotation of rules and rule sets (modules) enables (meta) reasoning with the semantic annotations. The meta data can act as an explicit scope for constructive queries (creating a view) on the knowledge base. For instance, the meta data annotations might be used to constrain the level of generality of a scoped goal literal to a particular module (defined by the meta data constraints), i.e., to consider only the set of rules and facts which belong to the specified module. A *scoped literal* is of the form $L : \overline{C}$ where L is a positive or negative atom and $\overline{C}$ is the scope definition which is a set of one or more meta data constraints. Scoped literals are only allowed in the body of a rule. Scoped literals might be default negated $\sim L : \overline{C}$. Syntactically, the following built-in predicates are used to query the meta data annotations and define the scope of literals for metadata-based *scoped reasoning* on explicitly specified parts of the KB:

```
% scoped literal
scope(<literal>,<meta data value>)
% query meta data value
metadata(<literal>,<Variable>,<meta data property>)
% constrain scoped goal literal
metadata(<literal>,<meta data value>,<meta data property>)
```

Scoped reasoning is crucial to explicitly close open and possibly distributed KBs on the Web. Comparable to database views created by Where-SQL clauses scoped goals can be used to create constructive views on the KB and reduce the number of relations and background knowledge which needs to be considered in ILP. Moreover, more meaningful and relevant information can be selected from the KB by the additional meta data of the rules/facts and rule sets (modules).

3.3 Types and Modes

Type and mode declarations are a common way in many ILP systems, like in PROGOL, TILDE or WARMR, to constrain the search space and state how clauses can be refined. Prova provides rich support for modes and external order-sorted type systems, in particular Semantic Web ontologies and Java class hierarchies by a polymorphic order-sorted typed unification [6].

In order to type a variable with a Java type the fully qualified name of the Java class to which the variable should belong must be specified as a prefix separated from the variable by a dot ".".

```
java.lang.Integer.X          variable X is of type Integer
java.util.Calendar.T         variable T is of type Calendar
java.sql.Types.STRUCT.S      variable S is of SQL type Struct
```

Java objects, as instances of Java classes, can be dynamically constructed by calling their constructors or static methods using highly-expressive procedural attachments. The returned objects, might then be used as individuals / constants that are bound by an equality relation (denoting typed unification equality) to appropriate variables, i.e., the variables must be of the same type or of a super type of the Java object. Ad-hoc polymorphic specialized functions can be implemented based on the type declarations, as can be seen in the following example showing two variants of the *add* function.

```
 add(java.lang.Integer.In1,java.lang.Integer.In2,Result):-
    Result = java.lang.Integer.In1 + java.lang.Integer.In2.
 add(In1, In2,Result):-
    I1 = java.lang.Integer(In1),
    I2 = java.lang.Integer(In2),
    X = I1+I2,
    Result = X.toString().
```

Beside Java class hierarchies Semantic Web taxonomies and ontologies (e.g. RDFS taxonomies or OWL ontologies) can be used as external order-sorted type systems in the multi-sorted Prova rule language. The implementation follows a prescriptive hybrid DL-typing approach with an polymorphic order-sorted unification and incorporates ontology type information directly into the names of symbols in the rule language. [8, 4, 6]

```
sameTranscriptionDirection(patika_P53Protein:A,
patika_MacroMolecule:B) :-
  orfDirection(patika_P53Protein:A,patika_MacroMolecule:D),
  orfDirection(patika_MacroMolecule:B,patika_MacroMolecule:D),
  patika_P53ProteinA<>patika_MacroMolecule:B.
```

The example annotates variables ($Type : Term$) with conceptual types such as $P53Protein$ or $MacroMolecule$ from an ontology *patika*, which denotes the namespace.

In the rlgg computation in ILP types are used to select relevant facts and rules from the background knowledge.

4. DISTRIBUTED INDUCTIVE LOGIC PROGRAMMING IN PROVA

Biological data mining systems typically accesses large and distributed web-based data sources and integrates multiple services, tools and resources during runtime. In this section we will implement Prova as a highly efficient and scalable inference service architecture with a communication middleware which supports parallel computation and resource allocation, seamless integration of external tools and communication of data, tasks and results between the distributed Prova inference services and external data sources / tools using an enterprise service bus (ESB) as communication middleware. [9] Figure 1 exemplifies the technical design of our approach.

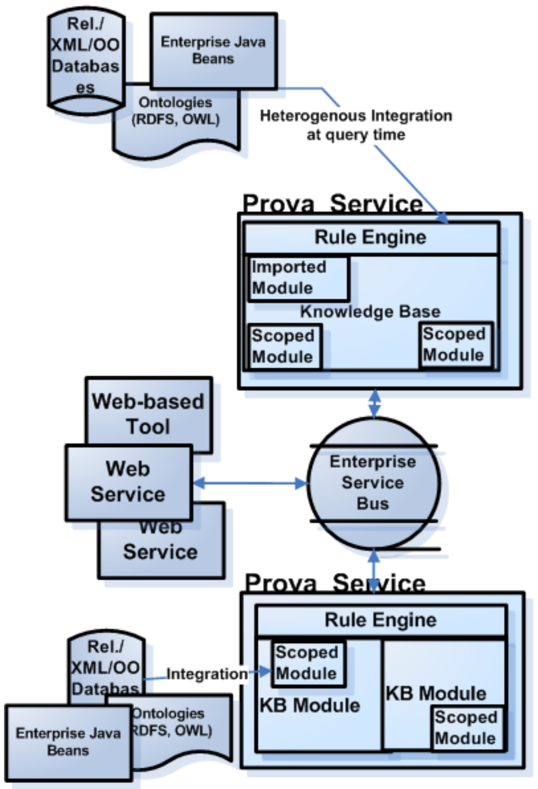

Figure 1: Distributed Prova Web Services

The three core design artifacts in our architecture are several instances of Prova rule engines deployed as web-based inference services (web-based execution environments), a scalable and efficient service-broker and communication middleware (an ESB) and, a common platform-independent rule interchange format to interchange rules, data and events between arbitrary Prova inference services and with external tools and data sources.

Several Prova rule engines might be deployed as distributed web-based services. Each service might dynamically import or pre-compile and load distributed rule bases which implement ILP theories and background knowledge. External data from data sources such as Web resources or relational databases and external application tools, web services and object representations can be directly integrated during runtime or by translation during compile time by the expressive homogenous and heterogenous integration interfaces of Prova. Furthermore, the ESB can be used to communicate with external components such as web services via asynchronous publish-subscribe message conversations. The ESB is used as object broker for the Prova inference services and as stable and efficient messaging middleware between the services [10]. Different transport protocols such as JMS, HTTP or SOAP (or Rest) can be selected to transport rule sets, data, queries and answers as payload of Reaction RuleML event messages between the internal Prova inference services deployed on the ESB. RuleML/Reaction RuleML [11, 10] is used as common platform-independent rule interchange format in which the Prova platform-specific execution language is translated and vice versa.

The main Prova language constructs for rule interchange are: *sendMsg* predicates, reaction *rcvMsg* rules, and *rcvMsg* or *rcvMult* inline reactions:

```
sendMsg(XID,Protocol,Agent,Performative,Payload |Context)
rcvMsg(XID,Protocol,From,queryref,Paylod|Context)
rcvMult(XID,Protocol,From,queryref,Paylod|Context)
```

where *XID* is the conversation identifier (conversation-id) of the conversation to which the message will belong. *Protocol* defines the communication protocol. More than 30 protocols such as JMS, HTTP, SOAP, Jade are supported by the underlying ESB as efficient and scalable object-broker and communication middleware. *Agent* denotes the target (an agent or service wrapping an instance of a Prova rule engine) of the message. *Performative* describes the pragmatic context in which the message is send (e.g. a multi-request in a contract net protocol). *Payload* represents the message content sent in the message envelope. It can be a specific query or answer or a complex interchanged rule base (set of rules and facts).

```
% Upload a rule base read from File to the host
% at address Remote via JMS
upload_mobile_code(Remote,File) :-
    % Opening a file returns an instance
    % of java.io.BufferedReader in Reader
    fopen(File,Reader),
    Writer = java.io.StringWriter(),
    copy(Reader,Writer),
    Text = Writer.toString(),
    % SB will encapsulate the whole content of File
    SB = StringBuffer(Text),
    sendMsg(XID,esb,Remote,eval,consult(SB)).
```

The example shows a rule that sends a rule base from an external *File* to the inference service *Remote* using the ESB. The inline sendMsg reaction rules is locally used within a derivation rule, i.e. only applies in the context of the derivation rule. The corresponding global receiving rule on the inference service side could be:

```
rcvMsg(XID,esb,Sender,eval,[Predicate|Args]):-
    derive([Predicate|Args]).
```

This rule receives all incoming messages from the ESB send to the inference service with the pragmatic context *eval* and derives the message content. The list notation $[Predicate \mid Args]$ will match with arbitrary n-ary predicate functions, i.e., it denotes a kind of restricted second order notation since the variable *Predicate* is always bound, but matches to all predicates in the signature of the language with an arbitrary number of arguments *Args*).

Rules and data are translated and interchange as inbound and outbound Reaction RuleML messages $< Message >$ over the ESB:

```
<Message mode="outbound" directive="ACL:inform">
  <oid> <!-- conversation ID--> </oid>
  <protocol> <!-- transport protocol --> </protocol>
  <sender> <!-- sender agent/service --> </sender>
  <content> <!-- message payload -->  </content>
</Message>
```

- @*mode* = *inbound*|*outbound* – attribute defining the type of a message
- @*directive* – attribute defining the pragmatic context of the message, e.g. a FIPA ACL performative
- $< oid >$ – the conversation id used to distinguish multiple conversations and conversation states
- $< protocol >$ – a transport protocol such as HTTP, JMS, SOAP, Jade, Enterprise Service Bus (ESB) ...
- $< sender >< receiver >$ – the sender/receiver agent/service of the message
- $< content >$ – message payload transporting a RuleML / Reaction RuleML query, answer or rule base

By distributing mobile code to several inference services parallel computation of ILP tasks becomes possible. Relevant parts of the background knowledge for learning particular hypotheses are bundled to modules using constructive scopes and distributed to several client inference services in parallel. The learned rlggs from each client are send back to the manager node and integrated and aggregate in the background knowledge removing redundant and irrelevant clauses. The manager node then constructs new modules from the updated background knowledge and again sends out the ILP tasks to the clients for parallel processing. This process is repeated until a certain fixpoint in the incremental learner is reached for the inductively derived hypotheses such that no further generalizations can be found. Typical verification and validation tasks such as *coverage* proving the learned hypotheses with negative examples by specialization, or finding and removing failures in the background knowledge can be also solved in parallel by "outsourcing" this processes to the client services.

In summary, the described middleware addresses the needs for a seamless integration of distributed external data sources, tools and resources and provides the technical infrastructure to develop new distributed and service-oriented ILP algorithms which share Web resources and data.

5. CONCLUSION

While previous work in data mining has focused on extracting useful information from large database and on implementing scalable, robust algorithms for propositional and flat relations, multi-relation data mining operating on heterogenous and distributed data sources on the Web is a relatively young field. In this paper we have introduced Prova as a state-of-art distributed Semantic Web inference service which supports distributed multi-relational inductive logic programming based on a rule and event-based middleware. Prova combines technologies from declarative rule-based programming with enterprise application technologies for object-oriented programming, relational and semi-structured heterogenous data access and novel techniques for service oriented computing and complex event processing as basis for inference service grids, resource sharing networks and parallel computation. The resulting design artifact addresses real-world requirements in ILP-based mining of biological data such as: highly complex structural elements with diverse and unusual relational, semi-structured or object-centered data types, e.g. using Semantic Web ontology languages as semantically rich concept description languages; large amounts of data stored in distributed heterogenous data sources; seamless integration and combinations of tools and services demanding for efficient interchange of data and events; high computational complexity of the ILP tasks due to the complex combinatorics of multi-relational search space and the open-world assumption of the open distributed Web knowledge bases.

Our distributed rule-based Prova approach, which is akin to grid service networks, has the potential to overcome these problems in standard ILP and establish ILP as a potential approach to analyze biological data in multi-relational Life Science data bases published on the Web.

The implementations described in this paper are part of the Prova / ContractLog open-source distribution [1] and we have successfully demonstrated the usability and adequacy of our ILP and enterprise service middleware approach in various domains of research and industry use cases such as for test-driven verification and validation of correctness and quality of rule bases (see RBSLA project [7, 5, 8]), and Semantic Web-based virtual organizations and web service collaborations (see Rule Responder project [9]).

6. REFERENCES

[1] A. Kozlenkov, A. Paschke, and M. Schroeder. Prova, http://prova.ws, accessed jan. 2006. 2006.

[2] A. Kozlenkov and M. Schroeder. Prova: Rule-based java-scripting for a bioinformatics semantic web. *Proceedings International Workshop on Data Integration in the Life Sciences*, 2004.

[3] J. W. Lloyd. *Foundations of logic programming; (2nd extended ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[4] A. Paschke. Owl2prova: Homogeneous and heterogeneous integration of description logics into logic programming, http://prova.ws/forum/viewtopic.php?t=152, accessed dec. 2005, 2005.

[5] A. Paschke. Rule based service level agreements, http://ibis.in.tum.de/projects/rbsla/index.php, 2006.

[6] A. Paschke. A typed hybrid description logic programming language with polymorphic order-sorted dl-typed unification for semantic web type systems. In *OWL-2006 (OWLED'06)*, Athens, Georgia, USA, 2006.

[7] A. Paschke. Verification, validation and integrity of distributed and interchanged rule based policies and contracts in the semantic web. In *Int. Semantic Web and Policy Workshop (SWPW' 06)*, Athens, Georgia, USA, 2006.

[8] A. Paschke. *Rule-Based Service Level Agreements - Knowledge Representation for Automated e-Contract, SLA and Policy Management*. IDEA, Munich, 2007.

[9] A. Paschke, H. Boley, A. Kozlenkov, and B. Craig. Rule responder: A ruleml-based pragmatic agent web, www.responder.ruleml.org, 2007.

[10] A. Paschke, A. Kozlenkov, and H. Boley. A homogenous reaction rules language for complex event processing. In *International Workshop on Event Drive Architecture for Complex Event Process (EDA-PS 2007)*, Vienna, Austria, 2007.

[11] A. Paschke, A. Kozlenkov, H. Boley, M. Kifer, S. Tabet, M. Dean, and K. Barrett. Reaction ruleml, http://ibis.in.tum.de/research/reactionruleml/, 2006.

[12] G. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5, 1970.

[13] S. Wrobel. *Inductive Logic Programming for Knowledge Discovery in Databases*. Relational Data Mining. Springer, Berlin, 2001.