

PROVA
Java Rule Language for
Information Integration and Semantic Agents

Version 2.0
User's Guide

Alex Kozlenkov
Betfair Ltd., London
Adrian Paschke
Technical University, Munich

November 2006

TABLE OF CONTENTS

1	INTRODUCTION.....	5
2	PROVA, DESIGN PRINCIPLES.....	7
2.1	DECLARATIVE VS. OBJECT-ORIENTED PROGRAMMING	7
2.2	PROVA SYNTAX.....	10
2.3	LOGIC AS RULES	11
2.4	DATA AND COMPUTATION WRAPPERS	13
2.5	IMPLEMENTATION WITHIN THE JAVA RUNTIME	14
2.6	COMPATIBILITY WITH JAVA-BASED SERVER AND AGENT ARCHITECTURES	14
3	LANGUAGE FEATURES.....	16
3.1	PROVA LISTS	16
3.2	JAVA-BASED TYPE SYSTEM.....	17
3.3	FACTBASE.....	18
3.4	ADVANCED RULEBASE MANIPULATION	20
3.5	JAVA INTEGRATION	20
3.6	EQUALITY IN PROVA.....	21
3.7	NON-DETERMINISTIC JAVA PREDICATES	21
3.8	EXCEPTION HANDLING.....	22
3.9	SQL INTEGRATION.....	24
3.10	CALLING PROVA FROM JAVA	27
4	BUILT-IN PREDICATES	33
4.1	FACT BASE UPDATES.....	33
4.2	ADVANCED RULEBASE MANIPULATION	34
4.3	VARIABLE TESTS.....	34
4.4	INPUT-OUTPUT PREDICATES	35
4.5	STRING MANIPULATION PREDICATES	37
5	USING PROVA FOR ONTOLOGY WRAPPING.....	40
6	PROVA AGENTS ARCHITECTURE PROVA-AA	42
6.1	MAIN FEATURES OF PROVA-AA	42
6.2	COMMUNICATOR PROVA-AA AGENT FOR JAVA APPLICATIONS	47
6.3	SPECIFYING THE BEHAVIOUR OF PROVA-AA AGENTS AS STATE MACHINES	49
6.4	ASYNCHRONOUS PROCESSING OF JAVA METHODS.....	52
7	REFERENCES.....	54

1 Introduction

Prova language is a derivative of a Java rule inference engine Mandarax. Historically, the project was initially focusing on bioinformatics applications with the following motivations:

- Practical applications in standard programming languages tend to *mix together logic, data access, and computation*. This makes them more difficult to write, maintain, and update.
- Fast prototyping and easy maintenance are not among the features of languages like Perl or Java.
- Implementing distributed workflows in a consistent and protocol independent manner is not easy in procedural languages like Java. The resulting code is not easily understandable, maintainable, adjustable, and upgradeable.
- There is no efficient and flexible wrapper mechanism for ontologies, especially needed because of their rapid rate of change.

To answer these needs we had either an option of making use of the existing technologies or proposing a new solution giving precise answers to these challenges. Our solution is a new programming language Prova combining the features of quintessential declarative (Prolog) and object-oriented (Java) languages.

We have set forth the following goals when designing a new language.

- The language should *combine* the benefits of declarative (rule-based), procedural (e.g., Java), and workflow languages (e.g., BPEL).
- The syntax and semantics of the standard rule-based component of the language should be as close to Prolog as possible.
- The language should include a very natural Java call mapping as close to Java syntax as possible.
- The type system compatible with Java should be available.
- The rule-based inference engine should be executed in the Java runtime and run either standalone or embedded in a Java application via an API.
- The language should include a flexible relational database mapping API.
- The language should be as efficient as possible.
- The language should be able to integrate Semantic Web technologies such as XQuery, RDF/RDFS, OWL, SPARQL.

We believe that the spirit and design of the Prova language is closely aligned with the recent Semantic Web initiative of W3C. According to W3C, rules, inference, and ontologies should become the cornerstone of the new Web. Importantly, we believe that this new Semantic Web will be much friendlier to technical and research applications than is currently possible.

The key advantage to using Prova is its elegant separation of logic, data access, and computation. To make applications more easily configurable, one step in the right direction was the introduction by the industry of various configuration files, for example, Windows registry or Java properties mechanism, specifying adjustable parameters making the applications more flexible. A further step was the use of declarative wiring of components in inversion-of-control frameworks like Spring.

We are arguing that in view of the proliferation of rapidly changing rich services, this approach to configuring applications is insufficient. The properties documents can only adjust parameters, not reconfigure the IF-THEN logic that is needed for achieving a truly flexible data, computation, and ontology integration.

Prova specifically targets this separation of concerns by allowing users to specify logic in the proposed language in a declarative fashion, and delegate computation-intensive tasks to Java code or services. To obviate the natural reluctance of the users and programmers to learn a new language, Prova follows very closely the syntax of two well-known languages, Prolog, for specifying logic in the form of rules in a declarative way, and Java calls invocation written directly in Java syntax. The current implementation of the language is freely available from our web site [1].

Prova includes numerous extensions including passive reaction rules based on Prova Agents Architecture *Prova-AA*, active reaction rules based on the ECA approach, test-driven development, transactional updates, constraints, OWL integration via types.

The Prova-AA framework includes syntactically simple and semantically clear extensions allowing for sending messages via either JADE-HTTP, JMS, or arbitrary Mule Enterprise Service Bus communication protocols and for specifying sophisticated reaction rules for processing inbound messages. Due to the natural integration of Prova with Java, Prova-AA offers a very clean and compact way of specifying agents' behaviour while allowing for efficient Java-based extensions to improve performance of critical operations.

The User's Guide explains in detail the design principles behind Prova in Section 2, discusses the main features of the language in Section 3, describes the collection of built-in predicates that simplify programming in Prova in Section 4, and presents some test case scenarios of the use of the system in Section 5. Section 6 describes Prova Agents Architecture *Prova-AA*. Section 7 presents conclusions and paves the way for the presentation of the GoPubMed and AuthorsPubMed systems.

2 Prova, Design Principles

This Section presents rationales and design principles behind the Prova language. We position this language as a platform for knowledge-intensive ontology-rich applications.

The Prova language aims to satisfy the following design goals:

- Combine the benefits of declarative and object-oriented programming;
- Syntax combines those of Prolog and Java—ultimate logic and object-oriented languages;
- Expose logic as rules;
- Run within the Java runtime;
- Make all Java API from available packages directly accessible from rules;
- Access data sources via wrappers written in Java or via services by using messaging;
- Be compatible with web- and agent-based software architectures;
- Provide functionality necessary for rapid application prototyping and low cost maintenance.

Prova is derived from Mandarax Java-based inference system developed by Jens Dietrich. Prova extends Mandarax by providing a proper language syntax and enhanced Java functionality. In the following, we will be using colours shown below to distinguish the language elements.

```
% Comments are in green

% Built-in predicates are brown
tokenize_list(Line,"\\t",[T|Ts])

% User-defined predicates are blue
member(X,[X|Xs])

% Java calls and constructors are red
Text1=P1.toString()

% Table names are magenta
sql_select(DB,cla,[px,PXA],[pdb_id,PDB_ID])
```

The remaining part of this section discusses some of the design principles behind Prova in more detail.

2.1 Declarative vs. object-oriented Programming

Declarative or rule-based programming (DP) is derivative of the predicate logic that is most directly and intuitively captured by the relational database design principles in which the data is modeled as a collection of multi-column tables. Rules in DP serve as virtual table generators that create knowledge from more elementary facts. In the database terminology virtual tables correspond to views, semi-persistent tables derivable from the tables stored in the database by SQL queries. Consider the following Prova code showing how knowledge can be inferred from the available facts.

The data in Prova can either be available locally as *facts*, for example, see the fact for a relation *is_a* above, or accessed via database queries as explained later. The particular relation fact *is_a* in the example above is read as “anticoagulant is a molecular_function”. It is often customary to use the first argument of the relation as the subject of the sentence, and use the following arguments as its objects. For example, *parent(X,Y)* is interpreted as “X is a parent of Y”. Importantly, following the Prolog convention, Prova syntax interprets strings starting with a capital letter or underscore as variables and those starting with a small letter or enclosed in double quotes as constant strings.

```

% Facts (what we know)

% "anticoagulant" is a "molecular_function"
is_a("anticoagulant","molecular_function").

% Rules (how to derive new knowledge)_
parent(X,Y) :- % X is parent of Y IF Y is a (kind of) X
               is_a(Y,X).

% Goal (what to do/what to derive)
% Who is Parent of "anticoagulant"?
:- solve(parent(Parent,"anticoagulant")).

> Parent="molecular_function"
    
```

Figure 1. Simple knowledge derivation in Prova

Following the declarative programming tradition, the Prova code complements facts with *rules*. The rules stipulate that the facts specified in the *head* (left-hand side) of the rule can be derived from the comma-separated list of facts specified in the *body* (right-hand side) of the rule. In the above example,

```

parent(X,Y) :- % X is parent of Y IF Y is a (kind of) X
               is_a(Y,X).
    
```

relation $parent(X,Y)$ on the left-hand side from the symbols “:-” read as “if” can be derived from (a single) relation $is_a(Y,X)$ specified on the right-hand side in the body of the rule. The rule can be interpreted as “X is a parent of Y is Y is a particular case of X”. Both X and Y above are *variables*, which means that any substitutions of strings to variables X and Y making the relation $is_a(Y,X)$ true make the relation $parent(X,Y)$ true also. If a fact in the body of a rule cannot be proved (fails) then backtracking occurs and other possible instantiations of the previous facts are explored resulting in the recursive depth-first search.

A more interesting rule could include more than one fact in the body of the rule:

```

uncle(X,Y) :- % X is an uncle of Y IF
               parent(Z,Y), % Z is a parent of Y
               sibling(X,Z). % AND X is a sibling of Z
    
```

Running the example in Figure 1 requires specifying a *goal*, a top level relation (predicate) for which facts matching the pattern specified in the goal have to be established. In particular,

```

% Goal (what to do/what to derive)
% Who is Parent of "anticoagulant"?
:- solve(parent(Parent,"anticoagulant")).
    
```

the goal for the relation $parent$ above seeks to discover (*solve* for) all values of variable $Parent$, for which the relation $parent(Parent,"anticoagulant")$ can be established. In other words, we are looking for all parents (represented by variable $Parent$) of “anticoagulant”. The output shown at the bottom of Figure 1 gives the only such possible value: $Parent="molecular_function"$. Indeed, although there are no facts for the relation $parent$, by virtue of the rule for the relation $parent$, the facts for this relation can be inferred from available facts for the relation is_a . Having established that the fact $is_a("anticoagulant","molecular_function")$ is true, the variables X and Y become “molecular_function” and “anticoagulant”, respectively, which leads to identification of one fact $parent("molecular_function","anticoagulant")$.

The mechanism of substituting variables in facts and rules is known as *unification* and intuitively, corresponds to finding the most general pattern that matches both unified terms. For example, a query

```

% Goal (what to do/what to derive)_
% What are the pairs in is_a relation?
    
```

```
:- solve(is_a(A,B)).
> A="anticoagulant", B="molecular_function"
```

complementing Figure 1 produces the values for variables *A* and *B* by unifying *A* and *B* with the respective constant strings in one stored fact for the *is_a* relation. It should be noted that variables are local (unique) to each rule, so no name clashes between the variables in the query and the rule can occur.

Object-oriented modeling and programming (OOP) has emerged as the leading paradigm for creating industrial and ad-hoc standalone and web applications. The main idea of the OOP is the modeling of the problem domain in terms of cooperating objects to which particular functionality is assigned. Objects exchange messages or method calls to collaborate in achieving the design goals of the application. Unified Modeling Language (UML) developed and promoted by Object Management Group (OMG) is a *de facto* standard for modeling applications using the OOP paradigm and achieving interoperability of designs. Java, C#, and C++ are perhaps the most popular and the most advanced object-oriented languages.

We show below equivalent Java and Prova program executing simple manipulations of a *HashMap* Java collection.

```
:- eval(test_collections()).

test_collections() :-
    Map=java.util.HashMap(), % Constructor of a Java HashMap
    Map.put(0,"false"), % Instance method calls
    Map.put(1,"true"),
    println(["Map=",Map]). % Built-in predicate for printing

> Map={1="true",0="false"}
```

```
class test_collections {
    int main() {
        Map map = new java.util.HashMap(), % HashMap Constructor
        map.put(0,"false"), % Instance method calls
        map.put(1,"true"),
        System.out.println("Map="+map). % Built-in printing
    }
}

> Map={1="true",0="false"}
```

The code shows that in the style of OOP, an object that will execute the required application function needs first to be constructed. This is done by a constructor call *new* in the Java code and by the special equality predicate in the Prova code where the left-hand side of the equality sign is the Java variable that will contain the constructed Java object, and the right-hand side contains the fully-qualified name of the class with optional arguments. Once constructed, the object *Map* in Prova is manipulated in exactly the same way as in Java. Finally, the contents of the *HashMap* are printed by using a built-in *println* predicate.

As the DP and OOP examples demonstrate, DP and OOP serve complementary purposes. The declarative programming style is very good for flexible data integration, inference, and for specifying IF-THEN logic. The OOP is very good for modeling complex systems that are relatively stable and

© Alex Kozlenkov and Adrian Paschke, 2002-2006 page 9

well understood, for example, the airlines ticket purchasing systems. Our conclusion is that DP is good for logic while OOP is good for computation.

Prova offers a very clean way of integrating Java into the DP paradigm, paving a way for intelligent knowledge-intensive data integration useful in many application areas such as information integration and knowledge discovery.

2.2 Prova syntax

The Prova syntax in EBNF is shown in the Figure 2 below. Version 1.5 has corrections shown in red.

```

prova ::= {statements}, end of file;
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 ::= qualified java class, ".", method name, "(", terms, ")";
instance java call ::= variable, ".", method name, "(", terms, ")";
constructor java call ::= left term, "=", qualified 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 ::= qualified 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 ::= "." newline;

```

Figure 2. The syntax of the Prova language.

The language follows most of the features of standard Prolog with the exception of some simplifications. In particular, the equality arithmetic relation is represented by the equality sign "=".

Prova also provides important extensions to Prolog syntax. In particular, typed Java variables (and as we shall see later, OWL-typed variables) can be used instead of the generic untyped Prolog variables. This offers additional level of precision to queries and rules. This important feature is further discussed in Section 2.2.

The part of the Prova syntax corresponding to Java allows calling Java methods from the body of Prova rules. Instance and static calls are accessible with the usual Java syntax. For example:

```
Map.put(0, "false")           % Instance method call
Double.Y=java.lang.Math.cos(Double.X) % Static method call
```

Static method calls require fully qualified class names to appear before the name of the static method followed by arguments. Instance methods are mapped to concrete classes dynamically based on the type of the variable by using the Java reflection mechanism.

Both static and instance methods can be *functional* or *predicate*. Functional Java calls have a left-hand side with which the results of the call are unified. If the left-hand side is a free (unassigned) variable the latter stores the result of the invocation. If the left-hand side is a bound variable or a Prova list pattern the unification can succeed or fail and, consequently, the call itself can succeed or fail. Prova lists are used on the left-hand side to allow matching of the constructed objects to specified patterns. For this to work, the type returned by a Java call should be either an array of Object (or derivative) variables or a Java List implementation. Please, refer to Section 3.1 for an introduction to Prova lists and an example of matching constructed or returned objects to Prova lists. Predicate (non-functional) are assumed to be tests in such a way that the call succeeds only if a true *Boolean* or *boolean* variable is returned. To make a method call returning *Boolean* or *boolean* succeed even if *false* is returned a free left-hand variable should be used.

Java variables can be created by calling constructors:

```
List=java.util.ArrayList(),           % Construct a Java List
SubChain=psimap.SubChain(PX,C,B,E),    % Construct a SubChain
```

Constructors follow the Java syntax with the exception of the *new* keyword in front of the name of the class and the constructor arguments. If an exceptions occurs in either static, instance or constructor calls, the literal fails and backtracking occurs. Prova includes an exceptions handling mechanism described in Section 3.7.

The example below shows how a Java *List* returned by a Java instance method is unified against a Prova list that specifies the expected pattern. If the pattern did not match the returned list, a backtracking would have occurred.

```
demoJavaListReturn() :-
    IBMPortfolio = ws.prova.examples.IBMPortfolio(),
    % The method getJavaList returns a Java ArrayList containing ["A",3.14,4]
    [X,Double.D,I] = IBMPortfolio.getJavaList(),
    println([X,Double.D,I]).

> ["A",3.14,4]
```

An example [prova/prova-examples/ex033.prova](#) shows in detail how lists and Object arrays returned by Java are matched against various patterns supplied on the left-hand side of the Prova calls.

2.3 Logic as rules

As opposed to the procedural style of programming where precise computational steps should be specified to capture the logic, declarative rule-based style of programming offered by Prolog and

Prova is a more direct representation of *how the things are* as apposed to *how the things should be computed*. Consider the example below.

```

% Concatenation of list in argument 2 to the list in argument 1
%   is the list in argument 3
append([],L,L).           % L appended to an empty list [] is L
append([X|L1],L2,[X|L3]):- % Any list L2 appended to a list starting
                           % with X is a list starting with X
    append(L1,L2,L3).

% Goals
% What two lists when appended produce the list [1,2,3]?
:- solve(append(X,Y,[1,2,3])).

> X=[] Y=[1,2,3]
> X=[1] Y=[2,3]

> X=[1,2] Y=[3]
> X=[1,2,3] Y=[]

% What is the result of appending lists [1] and [2,3]?
:- solve(append([1],[2,3],Z)).

> Z=[1,2,3]

```

Figure 3. Logic as rules: list concatenation.

If a Java code was used to reproduce the functionality of the above Prova program, it would have been necessary to create four different programs representing four different cases of the lists concatenation. In the case of the rule-based programming, only two rules above are required with possible queries automatically resulting in correct answers. Two examples of queries in Figure 3 allow the user to query for possible constituent sublists of the result list, or to obtain the result of concatenating the two specified lists.

The Prova code captures directly the logic behind the definition of the concatenation of lists while an equivalent Java or Perl code would have to follow the specifics of the different computational procedures.

The code in Figure 4 shows how the rule-based 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* in Figure 4) and a unique ID of a particular data item (variable *ID* in Figure 4). The data is cached in the cache represented by variable *CacheData* above.

```

% 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).

```

Figure 4. Logic as rules: accessing data with caching.

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 *ID* and *Data* 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.

2.4 Data and computation wrappers

Given the general approach to separating the application workflow into the logic and computation parts, we capture the knowledge-intensive part of the application in Prova rules while using Java, SQL, synchronous or asynchronous messaging, 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 shown in Figure 5.

The code shows an example of a data integration rule used for computing the interaction of protein domains (subunits). 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 predicate *access_data* using the code in Figure 4. A pair of domain definitions is computed by the predicate *scop_dom_atoms* and returned in variables *DomainA* and *DomainB*, respectively. Finally, an instance method *interacts* returning *boolean* either succeeds or fails depending on whether the two domains interact.

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.

```
% 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).
```

Figure 5. Data wrappers: Protein domain interaction.

The example in Figure 6 shows how relational database queries can be wrapped by Prova predicates.

```
% Given the open database connection DB, a Table name, and a maximum
% recursion level, the relation id_recursive computes transitive closure
```

```
id_recursive(DB,Table,N,[F1,ID1],[F2,ID1]).
id_recursive(DB,Table,N,[F1,ID1],[F2,ID2]) :-
    N>0,
    N2=N-1,
    sql_select(DB,Table,[F1,ID1],[F2,ID3]),
    id_recursive(DB,Table,N2,[F1,ID3],[F2,ID2]).
```

Figure 6. Data wrappers: SQL wrappers.

Given the open database connection *DB*, a *Table* containing a directed graph structure with fields *F1* and *F2* representing names of the fields containing the start and end node of each edge, and a maximum recursion level *N*, the relation *id_recursive* associates the IDs of the nodes separated by no more than *N* edges. The *sql_select* Prova built-in wrapper predicate described in more detail in Section 2.2, can dynamically construct an appropriate SQL SELECT statement based on the values of field names and translating all assigned (bound) field values to WHERE clauses and non-deterministically enumerating all unassigned (free) variables corresponding to tuples in the result set.

Finally, an example in Figure 7 shows how external command-line utilities may be used for wrapping services that cannot be accessed via Java invocation.

```
exec(["gzip -d ",Remote]),
exec(["mv ",DecompressedRemote," ",QfileName]),
exec(["rm ",Remote]).
```

Figure 7. Data wrappers: command-line utilities.

2.5 Implementation within the Java runtime

The Prova system runs within the scope of the Java runtime either as a standalone application or via embedding a Prova “agent” inside Java code and calling an appropriate API synchronously or asynchronously. The standalone Prova system is normally invoked from command-line with the following string:

```
prova rules.prova
```

Figure 8. Prova invocation.

All goals in the invoked Prova file are executed in their respective order. We distinguish between *solve* goals and *eval* goals. For *solve* goals, for each successful inference, the engine prints to standard output the assignments for the variables in the goal predicate satisfying the query. For *eval* goals, the engine executes an exhaustive search of the rules until no more backtracking is possible.

The Prova files may consult other files with rules by using the following goal predicate.

```
:-eval(consult("/home/user/utills.prova")).
```

When a Prova agent is used from within the existing Java code, the agent can enumerate all “solutions” for the *solve* queries or choose to be called back by the rule base at any point in the rule execution. [Section 3.X discusses the Prova Java API in detail.](#)

2.6 Compatibility with Java-based server and agent architectures

In addition to defining goals in standalone Prova applications or submitting goals via the API to embedded Prova agents, the Prova system can also accept goals submitted as via messages via a large variety of messaging and networking protocols (including all protocols in the Mule Enterprise Service Bus framework). For this to work, an important design decision was made to provide a message-based interface to the Prova engine. A serial asynchronous queue can be used to submit

goals to the engine by directly calling a message posting method on the queue locally, or alternatively, JADE or ESB communication protocols can also be used for exchanging messages with remote hosts.

The local interfacing is useful for web-based architectures, where, for example, a JavaBean would create a queue and the engine object, submit a message with the consult goal as shown in Figure 8 to initialise the engine with a required set of facts and rules, and submit goals in response to incoming web requests.

The remote interfacing is useful for agent-based architectures where independent agents engage in conversations by exchanging messages and cooperating to achieve collaborative goals. In this case, in addition to usual derivation rules, the Prova code is extended with reaction rules specifying the sequences or branches of actions that an agent responds with upon detection of a matching pattern in the inbound message. As mentioned before, open source implementations of JADE and ESB communication protocols can be used for the actual message exchange.

3 Language features

This Section presents a more detailed description of the main features of the Prova together with some complete examples of its use. Some of the examples are based on scenarios in bioinformatics applications. In the following, we will be assuming basic knowledge of Prolog and Java while trying to keep the discussion as useful for a non-specialist as possible.

3.1 Prova lists

Although Prova was originally based on the Mandarax inference engine that does not offer list processing which is extensively used in Prolog-like languages, Prova contains a comprehensive implementation of lists. Internally, Prova lists are represented by Mandarax type *ComplexTerm*. The basic notation for lists is derived from the Prolog syntax. The formal Prova list syntax is part of the Prova syntax presented in Figure 2. The list elements in the *head* of the lists are separated by commas, and additionally, a *tail* of the list can be specified as a variable representing 0 or more elements following the head. The tail of the list is prefixed with a vertical bar character (`|`). The whole of the lists is surrounded by square brackets. An empty list is denoted as `[]`. Lists can be recursively embedded in other lists by introducing a (square bracket surrounded) list as an element in the head of the containing list. Variables starting with an underscore character are called *anonymous* and do not participate in unification and can be considered to have unique names for each instance of such variable. Consider an example in Figure 9.

```
% X is a member of a list if it is the first element
member(X,[X|Xs]).
% X is a member of a list if it is in the list tail
member(X,[_|Xs]) :-
    member(X,Xs).

:- solve(member(2,[1,2,3])).

> yes

:- solve(member(X,[1,2,3])).

> X=1
> X=2
> X=3

:- solve(member(X,[1,[2,3]])).

> X=1
> X=[2,3]

:- solve(member([X,X,3,4],[1,[2,2|Z],[2,3|Z]])).

> Z=[3,4] X=2
```

Figure 9. Use of lists in *member*.

The important property of lists in Prova (and Prolog) is that variables used in lists can be unified (matched against) variables occurring outside of the lists. Since the Prova variables are by default untyped, this allows matching variables to lists, and lists to lists. Note especially the last pattern-based query in Figure 9. The query looks for an element in the list provided in the second argument that matches the pattern `[X,X,3,4]`. Only one element `[2,2|Z]` matches this pattern, so the answer to the query is `X=2`. Note that the tail `Z` of the list is found to be `[3,4]` matching the two last elements in the

query list in the first argument. From this example, it is easy to see why declarative style of programming can be superior to procedural languages in situations when flexibility and code economy are required as is the case for knowledge-intensive information integration applications.

NB. It is very important to note that Prova lists can be represented in a Prolog list format and in a predicate notation with the first element in the list becoming a predicate symbol:

`[X,Y,...] <=> X(Y,...)`

Note that a list with one head element and a tail cannot be represented in a predicate notation:

`[X|Xs] <=> {no mapping}`

The equivalence of list and predicate notations is exploited in multiple examples and allows for passing the predicate calls to the built-in *derive* predicate for evaluation. This is how negation as failure is implemented in the supplied standard library *utils.prova*. The built-in *derive* predicate call allows to call a predicate dynamically with the predicate symbol unknown until run-time.

```

% Negation as failure
not([X|Args]) :-
    derive([X|Args]),
    !,
    fail().

```

3.2 Java-based type system

Because of the desired interoperability of Prova rules with Java code and extended use of types and classes in Java, we have decided to include a Java-based type system in Prova. Note that Prova 2.0 also includes an ability to type variables using imported OWL ontologies. **OWL-based typing is discussed in Section 6.X.** Typed variables are not commonly found in rule-based languages such as Prolog. The rationale behind the introduction of the type system was to offer the user an option to restrict the applicability of rules and to control the level of generality in queries and reaction rules. 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 in Figure 10 a variant of *member* predicate queries using the same rules for *member* as in Figure 9.

```

:- solve(member(X,[1,Double.D,"3"])).

> X=1
> X=Double.D
> X=3

:- solve(member(Integer.X,[1,Double.D,"3"])).

> Integer.X=1

```

Figure 10. Use of typed variables in *member*.

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, an *Integer* 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.

If variables are unified with constants (instantiated objects), the unification fails if a variable of a class is unified with an instantiated object of its superclass. Otherwise, the unification succeeds and the variable becomes instantiated with the constant object.

Figure 11 shows these rules at work. The built-in predicate *assert* is used to dynamically update the facts for a relation *mustbe*.

Assuming *IBMPortfolio* is a subclass of *Portfolio*, after a *mustbe* fact is asserted for *IBMPortfolio.IP*, the first query in Figure 11 results in a general query variable *Portfolio.IP* becoming a more specific variable *IBMPortfolio.IP*. After another fact for *mustbe* is asserted for a general variable *Portfolio.P2*, the second query to *mustbe* with specific variable *IBMPortfolio.IP2* is not changed by the query.

```

test024() :-
    % A relationship that holds for the subclass
    assert(mustbe(shares.IBMPortfolio.P,big)),
    % Query the relationship providing a superclass variable
    mustbe(shares.Portfolio.IP,big),

    % Since we query for something more general than the stored fact,
    % the result is the subclass for which the relationship holds.
    println(["Must print an IBMPortfolio variable=",shares.Portfolio.IP]),
    % A relationship that holds for the superclass
    assert(mustbe(shares.Portfolio.P2,balanced)),
    % Query the relationship providing a subclass variable
    mustbe(shares.IBMPortfolio.IP2,balanced),
    % The type of the variable stays the same, since
    % the relationships for subclasses hold for superclasses too
    println(["Must print an IBMPortfolio
variable=",shares.IBMPortfolio.IP2]).

> Must print an IBMPortfolio variable=shares.IBMPortfolio.@@11
> Must print an IBMPortfolio variable=shares.IBMPortfolio.@@9
    
```

Figure 11. Unification across the class hierarchy.

3.3 Factbase

Prova facts resemble relational tables kept in memory. The facts are normally asserted via a built-in predicate *assert*, and retracted with *retract* or *retractall*. The asserted facts are tested (matched) in the order in which they were added. The predicate *retract* removes the first fact matching the specified pattern, however, it fails if there is no matching fact. The predicate *retractall* removes all facts matching the specified pattern. This predicate always succeeds. The facts are tested by using the syntax:

predicate(args)

The example in Figure 12 shows how the first query to *symmetric* predicate returns two matches (*g* and *h*) printed by the *println* predicate in the next line. Once *retractall* removes all facts for predicate *symmetric*, the second *println* has no facts left to print.

Prova is different from standard Prolog in that it allows facts to contain variables. For example, Figure 13 shows how equality of *Integer* variables enforced by an asserted fact makes it possible during fact extraction to instantiate the query variables *I* and *J* to uninstantiated but equal *Integer* variables.

```

test_factbase() :-
    assert(symmetric(f)),
    assert(symmetric(g)),
    assert(symmetric(h)),
    retract(symmetric(f)),
    symmetric(X),
    println(["X=",X]),
    retractall(symmetric(_)),
    symmetric(Y),
    println(["Y=",Y]).

> X=g
> X=h

```

Figure 12. Fact base updates.

```

test_variables_factbase() :-
    assert(f(Integer.X,Integer.X)),
    f(I,J),
    println(["I=",I," J=",J]),
    retract(f(_,_)),
    f(I2,J2),
    println(["I2=",I2," J2=",J2]).

> I=<java.lang.Integer.@@13> J=<java.lang.Integer.@@13>

```

Figure 13. Variables in Prova facts.

A useful built-in predicate *fact* restricts the derivation of a literal to only facts as opposed to facts and rules.

```

% Demonstrate the use of 'fact'
:- eval(test020()).

symmetric(a).
symmetric(X) :-
    b=X.

test020() :-
    % Only 'symmetric' facts will be used
    fact(symmetric(X)),
    println(["Rule A: symmetric= ",X]).

test020() :-
    % Both 'symmetric' fact and a rule will be used
    symmetric(X),
    println(["Rule B: symmetric= ",X]).

```

An *update_fact* predicate combines *retractall* with *assert* for maintaining facts that are meant to be unique in the factbase. The first argument indicates the pattern to be removed from the factbase, while the second argument gives a new fact to be asserted. For example,

```

ex054() :-
    assert( f(2,'a') ),
    update_fact( f(2,_), f(2,'b') ),
    f(X,Y),
    println([f(X,Y)]).

> f(2,"b")

```

Section 4.1 describes these built-in predicates used for manipulating Prova facts in more detail.

3.4 Advanced rulebase manipulation

In addition to basic factbase manipulation introduced in the previous Section, Prova 2.0 includes the following advanced features:

- Dynamic updates of facts and rules using module ID's;
- Transactional updates using constraints.

3.5 Java Integration

The tight and natural Java integration of Prova makes the stated goal of combining logic with computation a reality. 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 implementation in Prova uses Java reflection to access Java classes. Due to the ongoing bugs in the Java reflection implementation there exist situations involving the use of embedded classes with public access that can not be called via reflection but only from compiled method invocation.

The example below shows how XML Document Object Model (DOM) is manipulated in the code. Because of the mentioned Java bug affecting the Java XML implementation, we have provided in Prova 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), % Constructor
    Document = XML(StringReader), % Constructor
    ResRoot = Document.getDocumentElement(), % Get root element
    ResRoot.setAttribute("ResId",ResId), % Set root attribute
    Paper = Doc.importNode(ResRoot,Boolean.TRUE), % Note the use of a Java
constant
    Root.appendChild(Paper),
    fail().
attachResults(Doc,Root,XMLList).
```

Figure 14. Manipulating XML DOM for PubMed papers.

The Java list *XMLPapers* contains papers returned from a query to PubMed [9] abstracts 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*.

The constructor calls are specified by equating the target object on the left-hand side with a qualified Java class name followed by the constructor attributes in brackets. A constructor can fail if an exception is raised. Static methods are formed by specifying a qualified class name followed by the method name and the method parameters in brackets. A static method can fail if an exception is raised or if the unification of the optional returned object with the supplied pattern fails. Finally, an instance Java method is formed by specifying a variable name (possibly with a prefix corresponding to its Java class) followed by the method name and the method parameters in brackets. 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[]*) or a Java *List* is returned from a Java method, it is automatically converted into a Prova list IF a variable explicitly typed as *RList* or a square-bracketed list is specified on the left-hand side. See more details in *prova/prova-example/ex033.prova*.

3.6 Equality in Prova

There are special rules regarding the use of equality '=' in Prova (see, e.g., an example *ex001.prova*).

- As of Prova 2.0, Prova lists can be used on both sides of an equal sign.
- Constant numbers (e.g., 2 or 3.14159) and Java constants corresponding to public static fields (e.g., *Boolean.TRUE*) can be used both on the left-hand and right-side side. The use of a Java constant on the left-hand side makes it possible to check the return of a Java call against this constant.
- Java method calls can only be used on the right-hand side.

3.7 Non-deterministic Java predicates

We provide a special mechanism allowing user to write non-deterministic predicates in Java. Non-deterministic predicates create a programming workflow in which all possible instantiations resulting from executing a non-deterministic predicate are explored depth-first. In this way no loops are required to enumerate result sets and the programming logic becomes more clearly understandable.

The example in Figure 15 shows what a programmer should do to create such method. A non-deterministic Java predicate should be a method with two arguments of types *Object[]* and *List*. The first argument correspond to an array (of possibly variable length) corresponding to the predicate parameters. The second argument is the result set containing a (possibly empty) Java *List* of *Object* arrays with the values of the parameters. The example shows how a built-in Prova predicate *read_enum* is defined.

```
public Boolean read_enum( Object[] r, List bindings ) throws IOException {
    BufferedReader in = (BufferedReader)r[0];
    String line = "";
    while( (line = in.readLine()) != null ) {
        Object[] retobj = new Object[2];
        retobj[0] = in;
        retobj[1] = line;
        bindings.add( retobj );
    }
    return Boolean.TRUE;
}

parse(DBName,Type,File) :-
    dbopen(DBName,DB),
    fopen(File,Reader),
    read_enum(Reader,Line),
    db_import(DB,DBName,Type,Line).
```

Figure 15. Programming non-deterministic Prova predicates in Java.

The predicate accepts a *BufferedReader* Java object and non-deterministically enumerates one-by-one all the lines returned by reading from the *BufferedReader*. In the Java code, the first argument of the *Object* array is converted to *BufferedReader*. If the conversion fails an exception is raised which is caught by the calling Prova code resulting in failed predicate call. If the first argument is indeed a *BufferedReader* a loop iterates over the lines read from the reader and for each line, an *Object* array of length 2 is created, filled, and added as an individual element to the *List* of bindings. The *Object* array contains the first element repeating the first argument of the predicate (*BufferedReader*) and the second element equal to a new line string. At the end, the bindings *List* contains pairs of the predicate arguments with the read lines.

If the returned bindings *List* is empty a non-deterministic predicate fails. In the Prova part of the example in Figure 15 it is shown how such predicate *read_enum* is used. After a file is open by a built-in predicate *fopen* each Line is enumerated non-deterministically by *read_enum* and sent to the local predicate *db_import* that processes the line and imports it into a database. In this way, neither loops nor IF_THEN logic are required, which creates a transparent and easily maintainable code. Note also that the recommended way to deal with exceptions in Prova callable Java predicates is to pass them to Prova for handling, so the method declares *IOException* as an exception it can throw. We discuss the Prova exception handling in the following Section.

3.8 Exception handling

Prova uses an exception handling mechanism that helps defining predictable correct behaviour in situations when something unexpected happens. Prova offers both exception handling that results in a failure at and backtracking from the offending literal or Java call, and compensation handling in which the control flow continues after the offending literal or Java call with appropriate corrective substitutions to variables that had to be instantiated by the offending call.

Exceptions can be thrown (raised) in several situations. The most direct way to raise an exception is by explicitly calling a built-in predicate *raise* supplying a constructed *java.lang.Throwable* object. The following fragment from an example *raise.prova* demonstrating how an exception is constructed, raised, and handled by an exception handler resulting in backtracking from the *raise* literal.

```
:- eval(raise_test()).

handle_exceptions(Msg) :-
    exception(Ex),
    print([Ex]),
    println([Msg]),
    fail().
raise_test() :-
    on_exception(java.lang.Exception,handle_exceptions(" in
raise_test()")),
    Ex = java.lang.Exception("A Prova exception"),
    raise(Ex).
```

There are a number of built-in predicates collaborating in the generation and processing of an exception above. The example starts with an activation of a local exception handler *handle_exceptions*. The activation is done by calling a built-in predicate *on_exception*. The activation defines the base class of exceptions that the handler will be intercepting, in this case, the basic *java.lang.Exception* class. Note that the handler call in the handler activation may be passed both bound and free variables. We shall see an example of the latter later. In the example above, a constant string is passed to the handler.

Once activated, an exception handler is active in all deeper predicate calls but disappears once the control returns from the body of the rule, in which the activation occurred. In the case when several exception handlers are active, the handler chosen to process an exception is the one that is closest in the inheritance hierarchy to the exception that has happened.

Once the handler is activated, we construct an exception object and call the raise predicate to throw the constructed expression explicitly. On exception, the handler *handle_exceptions* is called with the parameter(s) specified at the time of the corresponding activation. The body of *handle_exception* makes of the built-in *exception* predicate to access the corresponding exception object. In the example, we simply print the exception and the message passed to the handler and *fail*. This call to the built-in *fail* predicate results in the original raise predicate call to fail and the engine backtracks from this failure exploring the search space further.

Exception handling is particularly useful in cases when a Java or built-in predicate call results in an exception.

```
:- eval(ex021()).

handle_exceptions(Reader) :-
    exception(Ex),
    println([Ex]),
    % Corrective action: make Reader access a default String
    StringReader=java.io.StringReader('<?xml version="1.0"?>'),
    Reader = java.io.BufferedReader(StringReader).
ex021() :-
    on_exception(java.io.IOException,handle_exceptions(Reader)),
    fopen("nosuchfile.xml",Reader),
    % Enumerate and print one Line at a time
    read_enum(Reader,Line),
    println(["Line=",Line]).

> java.io.IOException
> Line=<?xml version="1.0"?>
```

In the example above, the built-in predicate *fopen* fails to open a non-existing file causing a *java.io.Exception*. The activated handler *handle_exceptions* is invoked and the exception is printed on standard output. This handler is different from the one in the previous fragment by not having a *fail* predicate call. This kind of handler takes a compensating correcting action and restores control at right after the *fopen* call as though no exception had happened. Note that the exception handler is passed a free variable *Reader* at the time the handler is activated by *on_exception*. The handler assigns a value to this variable and *fopen* is able to complete successfully with a properly instantiated variable as a result of its call. The following two lines in the *ex021* clause are executed successfully.

The following example shows the ways in which an exception handler becomes deactivated.

```
test() :-
    on_exception(java.io.IOException,handle_sql_exceptions(DB)),
    on_exception(java.io.IOException,clear),
    test2(),
    sql_select(DB,des1,options('limit 10')|Tail),
    ...

test2() :-
    on_exception(java.io.IOException,handle_sql_exceptions(DB)).
```

An exception handler may be deactivated by an explicit passing of a reserved handler name *clear* to an *on_exception* call for the type of exception for which the handler will be deactivated. Alternatively, any exception handler becomes deactivated once the control is returned to the caller of the clause in which the activation had occurred. In the example above, the first activation of the

handle_sql_exceptions gets explicitly deactivated in the next line, becomes reactivated again inside the body of the clause for the *test2* predicate, but becomes once again deactivated in the body of the *test* predicate immediately before the SQL query, which silently fails because the table *des1* does not exist. The SQL integration is presented in the next Section.

Prova exceptions can be conveniently used when dynamically importing (consulting) external code to Prova. The following fragment from *ex012.prova* shows how a call to the built-in *consult* predicate that causes parsing exceptions can be preceded by an activated exception handler for *ws.prova.parser.parsingException* exceptions.

```
on_exception(java.io.IOException,handle_io_exceptions(Filename)),
on_exception(java.io.IOException,handle_parsing_exceptions(Filename)),
consult(Filename),
```

IO and parsing exceptions are intercepted by different handlers that can either fail or compensate for the occurring exceptions. If the consulted files causes multiple exceptions, the handler for parsing exceptions gets called multiple times, once for each exception instance.

3.9 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.

Opening a database

The procedure for opening a database connection is shown in Figure 16.

```
: -eval(consult("utils.prova")).
location(database,"jdbc string","database_name","username","password").
...
dbopen("database_name",DB)
```

Figure 16. SQL Select integration with Prova predicates.

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 such as bioinformatics or 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)
```

Figure 17. Example of alternative mirror locations.

By default open database connections are cached. If the users wish to override the default on how many database connections are cached they should include the line like the one shown in Figure 18 after *utils.prova* is consulted. To switch off the connection caching, a value of 0 should be specified.

```
cache_capacity(database,10).
```

Figure 18. Specifying the cache capacity for database connections.

Querying a database

The main format for Prova predicates dynamically mapped to SQL Select statements is shown in Figure 19. Both single table queries and arbitrary joins are supported starting with version 1.4.

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

Figure 19. SQL Select integration with Prova predicates.

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, e.g., *'term as t1,synonym as s,term as t2'*. *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 uninstantiated (free) variable and will be returned by the query in each record in the result set. In addition to simple field

names, N_1, \dots, N_k 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. 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, $[where, "pdb_id \text{ like } '%%gs'"]$. 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 $[order, Order]$ pair, for example, $[options, "order \text{ by } count(px) \text{ desc } limit \ 10"]$. A number of examples of SQL Select mapped predicates are shown in Figure 20.

```
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"])
```

Figure 20. Examples of SQL Select mapped Prova predicates.

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 $table \text{ as } alias$ is used in the $From$ clause. The example in Figure 21 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
%==== The same query as one statement that is executed much faster =====
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'])
```

Figure 21. Examples of a join with separate sql_select (s) and of its single clause equivalent.

```
sql_select(DB, 'term, synonym',
  ['term.name', 'ZYX_MOUSE'], ['synonym.id2', ID2], [where, 'term.id=synonym.id1'])
```

Figure 21A. Another example of a multi-join.

A more modern SQL syntax can also be used with sql_select . The following fragment from a new example $outer_join.prova$ shows how a left join can be specified.

```
% select t.name, t.type, s.id2 from term t left join synonym s on t.id=s.id1 limit 10
sql_select(DB, 'term t left join synonym s on t.id=s.id1',
  ['t.name', N], ['t.type', T], ['s.id2', ID2], [options, 'limit 10'])
```

Inserting records into a database

Prova provides a built-in predicate sql_insert providing a flexible mapping to SQL Insert statements. The format is shown in Figure 22.

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

Figure 22. SQL Insert integration with Prova predicates.

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 in Figure 23 shows a complete rule that parses a text-based database file with descriptions of protein domains from the SCOP database [7].

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

Figure 23. Example of SQL Insert mapped Prova predicate.

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 be 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.

Updating records in a database

The functionality of SQL Update can be achieved by using a utility predicate *sql_update* that can be consulted from the standard *utils.prova* library. The following fragment from an example *sql_update_001.prova* shows the simplest type of *sql_update* in which the WHERE clause represented by a string that is not parsed by Prova and passed as it is to the SQL engine.

```
% update gtd t set t.scop_id=5,t.prim_dbname='gi' where t.id=1
% Note that we use single quotes in double quotes for "'gi'"
%   for it to be passed to the statement.
% If we used something like "t.scop_id = t.scop_id+1",
%   it would have to be encoded as ["t.scop_id","t.scop_id+1"].
sql_update(DB,"gtd t",["t.scop_id",5],["t.prim_dbname","'gi'"],[where,"t.id=1"])
```

Figure 23A. Example of SQL Update mapped Prova predicate.

There is a second form of *sql_update*, in which the fields and values to be passed to the WHERE clause of the SQL Update. This allows the field values to be directly passed to the *sql_update* predicate that assembles it into a valid SQL Update statement. The fragment below shows an example of this usage.

```
% update gtd t set t.scop_id=0,t.prim_dbname='gi' where t.id=1 and
%   t.sec_dbname='ref'
ID=1,
Sec_dbname="'ref'",
sql_update(DB,"gtd t",["t.scop_id",0],["t.prim_dbname","'gi'"],
    [where,["t.id",ID],["t.sec_dbname",Sec_dbname]]),
```

Figure 23B. Example of SQL Update mapped Prova predicate with variables used for the WHERE clause values.

3.10 Calling Prova from Java

There are two main scenarios of using Prova in applications.

1. *Prova controls the workflow.* The Prova engine is started from the command line specifying an initial rulebase file as an input.

2. *Java controls the workflow.* An instance of the Prova engine is run from a Java application or server-side component.

In this Section, we discuss the second option. The Java code can choose to instantiate several instances of the Prova engine that can co-exist in the Java runtime. The actual rulebases in each instance comprise the initial rulebases passed to the engine instance at the time of its instantiation and any additional code that can be dynamically added (consulted) after the instantiation.

Instances of the Prova engine are instantiated by constructing an object of class *ws.prova.Communicator*. The Prova engine is inherently message-based, so calling Prova from Java can be done both synchronously and asynchronously. If the Prova engine instance requires only synchronous querying, the instance should be created in a synchronous mode, in which case the message queue is not created at all. If any asynchronous processing is required, an asynchronous instance should be created, and the processing of goals and external messages uses a message queue. In the synchronous mode, the Prova engine does not create any additional threads but in the asynchronous mode, each engine instance runs in its own thread and thread-safety is ensured. In the asynchronous mode, the engine instances are also known as Prova agents because they can communicate with their peers by local, JADE, or JMS protocols.

The following JavaDoc fragment shows the parameters required by the constructor of *Communicator* objects.

```
public Communicator(java.lang.String agent,
                    java.lang.String port,
                    java.lang.Object rules,
                    int timeout,
                    boolean async)
```

A preferred constructor for a *Communicator*.

Parameters:

agent - A logical name of the Prova agent (must be unique on each machine)

port - An external port used for JADE communications. If *port* is `null`, JADE communication is not used

rules - There are three options to use here:

if the *Object* is a *String*, it must be a filename of the file to be consulted.

otherwise, it must be either a *StringBuffer* or a *BufferedReader*, from which the initial rulebase will be consulted.

timeout - The Prova agent terminates once this timeout (in seconds) expires after the last message. The value of `-1` lets the agent run indefinitely.

async - If it is *Communicator.ASYNC*, the *Communicator* is started in a separate thread. If it is *Communicator.SYNC*, it is started in the current thread.

Calling Prova synchronously.

The following fragment shows a typical constructor call initialising *Communicator* in a synchronous mode. Please, refer to *ws.prova.test.ProvaRunner.java* for examples of the synchronous mode.

```
Communicator comm =
    new Communicator("prova", null, "init.prova", 0,
Communicator.SYNC);
```

The synchronous mode is chosen by using the constant *Communicator.SYNC* as the last argument. In this mode, the port number in the second parameter is normally *null*, corresponding to the Prova engine that is not really capable of communicating directly with its peers via the JADE protocol. The engine instance lifetime is limited by the lifetime of the *Communicator* object, so the timeout parameter value in the fourth parameter is not important. The engine starts by consulting the initial rulebase from either an external file or a *StringBuffer* or *BufferedReader* provided as the third argument.

During the initialization, the initial rulebase is consulted together with any other files consulted explicitly from that rulebase. All goals in the initial rulebase are run synchronously in their order. The parsing exceptions occurring while consulting this code are printed on standard output and all other exceptions occurring while executing the code are assumed to be processed in the code itself. If this handling of exceptions is not acceptable because more control is required, an empty string should be passed as the third parameter to the *Communicator* constructor, and consulting the Prova code should be done explicitly by calling appropriate *Communicator* methods.

The first concern once the engine instance is initialised is to make sure that the output from the Prova code gets redirected appropriately as required. The following call makes all output from the Prova code disappear. Alternatively, and Java *PrintWriter* can be used to capture the output from Prova.

```
comm.setPrintWriter(NullWriter.getPrintWriter());
```

Consulting the code synchronously is performed by calling the method *Communicator.consultSync*.

```
public java.util.List consultSync(java.lang.String input,
                                java.lang.String key,
                                java.lang.Object[] objects)
    throws java.lang.Exception
```

A wrapper for a synchronous query to the rulebase

Parameters:

input - A fragment of Prova code that will be consulted into the Prova rulebase. Special constructs *_N* (e.g., *_0*) may be embedded in the code to represent Java objects used in goals, facts, and rules.

key - Unique key identifying the consulted code (useful in interactive environments)

objects - Java objects embedded into the *input* parameter.

Returns:

A list of *ws.prova.kernel.ProvaResultSet* objects, where each result set corresponds to one goal in the consulted code. For one goal, each resultset contains zero or more solutions with instantiations of all free variables in that goal.

Throws:

java.lang.Exception

Note that there is no functionality for consulting the code from an external file, *StringBuffer*, or *BufferedReader* as in the *Communicator* constructor. However, the recommended way to consult from these sources by calling *consultSync* with the following input.

```
comm.consultSync(":-eval(consult(_0)).", key, new Object[]{rules});
```

The placeholders *_N* indicate the places in the code where Java objects should be embedded. In the above example, the only object passed to the consulted code is the string *rules* with the filename of the external Prova code. In this way, the consulted code from the string contains a goal whose execution results in consulting from this external file.

Once the Prova rulebase is initialised, the goals can be run by consulting a code containing Prova goals as in the following example.

```
String input = ":- solve(holds_at(X,T)).";
```

```
List resultSets =
    comm.consultSync(input, Integer.toString(key++), new Object[]{});
```

In the example, the consulted code contains a goal with two free variables *X* and *T*, and on return from the call, the collection *resultSets* contains solutions assigning values to these variables. Each element of the list is an object implementing a *ws.prova.kernel.ProvaResultSet* interface. Each resultset corresponds to one goal in the consulted code, in our example, a single goal.

There are several useful methods available for a *ProvaResultSet*. Calling the method *getException* returns a *null* or an *Exception* object that contains either parsing or run-time exception not caught in the rulebase. The parsing errors are returned in a *ws.prova.parser.ParsingException*, with the actual parsing messages accessible by calling a method *ParsingException.errorsToString()*. The methods *first* and *next* allow iterating over the solutions contained in a *ProvaResultSet*. Once a new solution is accessed, the method *getResult* of *ProvaResultSet* returns each instantiation of a variable in the query given the class and name of this variable.

The following fragment from *ws.prova.test.ProvaRunner.java* groups together the above functionality.

```
for( Iterator rsit = resultSets.iterator(); rsit.hasNext(); ) {
    ProvaResultSet rs = (ProvaResultSet) rsit.next();
    final Exception e = rs.getException();
    if( e != null ) {
        System.out.println("Test fails with " + e);
        if( e instanceof ParsingException )
            System.out.println(((ParsingException) e).errorsToString());
        continue;
    }
    while( rs.next() ) {
        // The type of untyped variables is java.lang.Object
        final Object o = rs.getResult(Object.class, "X");
        System.out.println("result=" + o);
    }
    rs.close();
}
```

Calling Prova asynchronously.

A *Communicator* object created in an asynchronous mode is a fully-functional embedded Prova agent that can participate in message-based communication with other peer agents accessible on the local machine or via the Intra- or Internet by using JADE or JMS protocols. In version 1.8, communication between several instances of asynchronous *Communicator* is only possible via a loopback JADE protocol. In this case, JADE protocol should be enabled when creating each *Communicator*.

The following fragment from *ws.prova.test.ProvaRunner2.java* shows how a *Communicator* object is created in asynchronous mode.

```
Communicator comm =
    new Communicator("prova", null, "init.prova", -1, Communicator.ASYNC);
```

The second parameter can be given a non-null value, corresponding to a port on which a created JADE agent will be listening using the JADE HTTP protocol. We suggest the value 7778 that is also used in all examples of JADE agents in the directory *prova/prova-aa*. The fourth parameter equal to -1 corresponds to the agent existing indefinitely, i.e., terminating together with the embedding application.

An asynchronous *Communicator* is run in a separate thread, so you should weigh up the benefits of having a fully-featured embedded agent vs. the required resources. On the other hand, having one or only a few embedded agents should not normally present any problem.

Consulting the code asynchronously is performed by calling the method *Communicator.consultAsync*. The following example shows how you can use this method.

```
comm.consultAsync(":- eval(happy_for_eval(_0)).", new Object[]{this});
```

The method returns immediately while the code is placed in the message queue of the agent and is processed once it is dequeued by the agent. The actual execution of the code may take arbitrary long time or involve further communication with other agents, starting new threads, communicating with the user, etc. The Java objects can be embedded inside the consulted code by using the placeholders in the format *_N* exactly as in the synchronous mode discussed previously. In this example, we are passing the reference *this* to the object *ProvaRunner2* that contains the *Communicator* object to the consulted code. The reference will be used by the Prova code to call back the method *ProvaRunner2.results_ready* once the Prova code is ready to communicate the results back.

The following Prova code shows a clause for the predicate *happy_for_eval* that is consulted in the previous fragment.

```
happy_for_eval(Caller) :-  
    on_exception(java.lang.Exception,handle_exception()),  
    findall(X,happy(X),ResultList),  
    println(["Called from: ",Caller]),  
    Caller.results_ready(ResultList).
```

There exists also an alternative way for the Java code to communicate with the embedded agent. It is achieved by calling the method *Communicator.sendMessage*, the JavaDoc documentation for which is presented below.

```
public java.lang.Object sendMessage(java.lang.Object xid,  
                                   java.lang.String protocol,  
                                   java.lang.Object obj_receiver,  
                                   java.lang.String perf,  
                                   java.lang.String term,  
                                   java.lang.Object[] objs)
```

Send a message to an asynchronous Prova agent (including a local *Communicator* created in asynchronous mode)

Parameters:

- xid* - Conversation-id for the message. Specifying *null* starts a new conversation. Otherwise, it is assumed that the message is a follow-up to the conversation with the specified conversation-id.
- protocol* - Currently, "self", "jade", "jms", or "esb". For sending messages to the Prova engine in this *Communicator*, specify "self".
- obj_receiver* - The logical name of the target agent. Specifying "0" here results in sending this message to the Prova agent in this *Communicator* (it is) equivalent to specifying the *protocol* as "self".
- perf* - Performative, i.e., speech act communicated by the message. Broadly speaking, it is the "type" of the message. Examples include standard FIPA performatives such as *query-ref*, *ask-if*, or *inform*.
- term* - The message contents as a *String*. It is assumed to be a comma-separated list with various tags and parameters to the message. Java objects may be embedded by using placeholders in the format *%N* with corresponding to an index into the array with objects in *objs*.
- objs* - Array with Java objects to be embedded in the locations specified by placeholders in the message contents *term*.

Returns:

Conversation-id of the sent message. This id may be useful when a new conversation is

initiated. In this case, possible follow-up messages must be sent with the conversation-id returned here.

The following fragment from *ws.prova.test.ProvaRunner2.java* shows how message can be sent to the embedded Prova agent.

```
comm.sendMsg(null,"self","0","eval","consult,%0",new
Object[]{"discount.prova"});
Order order = new Order("cust A");
order.addItem("item A");
order.addItem("item B");
order.addItem("item C");
comm.sendMsg(null,"self","0","process","order,%0",new Object[]{order});
```

The following Prova fragment from *discount.prova* shows how the sent message can be intercepted and a discount assigned to the *Order* object.

```
rcvMsg(XID,Protocol,From,process,order(Order)) :-
    Customer=Order.getCustomer(),
    Items=Order.getItems(),
    assign_discount(Order,Customer,Items).
assign_discount(Order,Customer,Items) :-
    element(Item,Items),
    not(orders_history(Customer,Item)),
    Order.setDiscount(Item,0.25).
```

The net result of sending the message is that the *Order* is assigned a discount. This processing is done asynchronously, so in a real-world application the caller should be notified by a call back from the Prova code when the results become ready. In Section 6, we discuss in detail the Prova agent messaging including reaction rules defined as clauses for the reserved *rcvMsg* predicate.

4 Built-in predicates

The following section lists important built-in predicates that make programming in Prova easier. Since complete Java library and innumerable free and commercial Java packages are easily accessible from Prova, only minimal convenience functionality is provided in built-in Prova predicates. Non-deterministic predicates can be especially useful as are the predicates more closely related to internal Prova features. In the following, non-deterministic predicates are annotated with the prefix **ND**, and predicate arguments are shown with annotations *[in]* or *[out]* denoting the usual patterns of usage. All *[in]* parameters should be provided, otherwise, the predicate fails. It is very important to note that all *[out]* parameters can be also supplied on input, in which case the unification is attempted between the value to be returned from the predicate and the supplied value. This, for example, offers a very useful way of filtering the results immediately with the pre-defined constraint on some parameters.

4.1 Fact base updates

assert([in]Fact)

Assert a fact specified in *Arg1*. *Arg1* can contain variables.

Example:

```
IDs=java.util.HashSet(),% construct a HashSet
assert ids("user",IDs),% store a fact ids with IDs
IDs.add("Andrew"),      % store some data
IDs.add("Andy"),
ids("user",X),          % retrieve the fact with embedded user-ids
element(ID,X),          % non-deterministically enumerate the user-ids
println(["User: ",ID]). % print user-ids one at a time
```

retract([in]Pattern)

Retract the first fact matching the supplied pattern. *Arg1* contains a pattern that a fact should match to be retracted. *Arg1* can contain variables. This predicate fails if there is no matching fact and, therefore, nothing is deleted.

Example:

```
assert(symmetric(f)),
assert(symmetric(g)),
retract(symmetric(_)).
```

retractall([in]Pattern)

Retract all facts matching the supplied pattern. *Arg1* contains a pattern that the facts should match to be retracted. *Arg1* can contain variables. This predicate never fails.

Example:

```
assert(symmetric(f)),
assert(symmetric(g)),
retractall(symmetric(_)).
```

update_fact([in]PatternToUpdate)

update_fact([in]PatternToRemove, [in] PatternToAdd)

Update a fact that is supposed to have be in some sense unique in the factbase. In the first version, all facts matching the specified pattern are retracted and the fact is asserted to the factbase. In the second version, all facts matching the first supplied pattern are retracted and then the fact specified in *Arg2* is added. This predicate never fails.

Example:

```
assert (symmetric(a)),
update fact(symmetric( )),
% only symmetric( ) is in the factbase now
update fact(symmetric( ),symmetric(b)),
% only symmetric(b) is in the factbase now
```

4.2 Advanced rulebase manipulation

add([in]ModuleID, [in] ProvaCode)

add([in]ModuleID, [in] ProvaCode, [in] SubstitutionVariables)

Dynamically add new knowledge (Prova code) specified in the second argument to the Prova rule base. The actual code may contain both facts and rules (clauses). The added code can be referred to by the *ModuleID* specified in the first argument. The built-in *remove* can be used to delete all facts and rules for a given module. In the second version, a list of (often, Java) variables can be passed to the predicate. The *ProvaCode* will then have a number of matching variable substitution patterns (*_0* ... *_N*) for which the supplied variables will be substituted prior to asserting new knowledge. This predicate can fail with *ParsingException*. An instance of fact *module(ModuleID)* is also added to the rulebase if such instance does not already exist.

The following example taken from *prova/prova-examples/ex060.prova* demonstrates this feature:

```
ex060() :-
    add(id,"d(x):-x=_0.",[d]),
    d(X),
    println([X]),
    % Any facts or rules with consult-id=XID are removed
    retractall(id),
    % The following will fail and nothing will be printed the second time
    d(Y),
    println([Y]).

> d
```

remove([in]ModuleID)

Dynamically remove knowledge (Prova code) referenced by the specified *ModuleID*. In addition to all corresponding facts and rules, the *module(ModuleID)* instance is also removed.

4.3 Variable tests

bound([in]Variable)

Given an input Prova variable, test whether it is bound (instantiated). The supplied *Arg1* may contain (a) a free variable, (b) a constant, (c) or a Prova list. The predicate succeeds in the case (b). It also succeeds in the case (c) if the supplied Prova list only contains constants. Otherwise, the predicate fails.

Example:

```
bound(ID),
% Access a hashmap CacheData only if the ID is constant
Data=CacheData.get(ID).
```

free([in]Variable)

Given an input Prova variable, test whether it is free (uninstantiated). The supplied *Arg1* may contain (a) a free variable, (b) a constant, (c) or a Prova list. The predicate succeeds in the case (a). It also succeeds in the case (c) if the supplied Prova list contains at least one variable. Otherwise, the predicate fails.

Example:

```
treemembers(_,T):-
  free(T),      % check if T is free
  !,           % It is free so cut to avoid backtracking
  fail().      % fail this rule with no further backtracking possible
```

type([in]Variable, [out]Variable)

Given an input Prova variable, the predicate returns in the second parameter its Java type name as a string. If the first argument is a variable or if it is a Prova list containing at least one variable the predicate fails. Otherwise, the type of the constant object is returned, and in the case of a Prova list, "ComplexTerm" is returned.

Example:

```
test type() :-
  Dir=java.io.File("."), % Access the current directory
  Files=Dir.list(),     % Return the Object array of file names.
  type(Files,TypeOfFiles), % Find out the type of variable Files
  println([TypeOfFiles]). % Print "ComplexTerm"
```

4.4 Input-output predicates

We provide a limited number of such predicates that help in simple chores related to reading files. In particular, we provide non-deterministic predicates for parsing the file that are useful for iterating.

fopen([in]FileName,[out]BufferedReader)

Open a file for reading based on its path name given in *Arg1*. On success, *Arg2* contains a *java.io.BufferedReader* object corresponding to the opened file. This predicate fails if the file could not be opened. The file should be available either on the PATH or CLASSPATH.

Example:

```
test fopen() :-
  fopen(File,Reader),
  % Non-deterministically enumerate lines in the file
  read_enum(Reader,Line),
  println([Line]). % Print one line at a time
```

ND read_enum([in] BufferedReader, [out] Line)

Non-deterministically enumerate the lines given an opened *java.io.BufferedReader* supplied as the first argument. On each success, *Arg2* contains a *java.lang.String* containing a line from the reader.

Example:

```
fopen(File,Reader),
% Non-deterministically enumerate lines in the file
read_enum(Reader,Line),
println([Line]).           % Print one line at a time
```

copy([in] Reader, [out] Writer)

Copy a character stream from an instance of a class implementing the interface *java.io.Reader* to an instance of a class implementing the interface *java.io.Writer*. This built-in predicate can accommodate a large number of scenarios due to the availability of many classes implementing *Reader* and *Writer* in Java, for example, *java.io.BufferedReader* or *java.io.StringWriter* as in the example below.

Example:

```
% Upload a rule base read from File to the host at address Remote via Prova-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(jms,Remote,eval,consult(SB)).
```

copy_stream([in] InputStream, [out] OutputStream)

Copy a byte stream from an instance of class *java.io.InputStream* to an instance of class *java.io.OutputStream*. This built-in predicate can accommodate a large number of scenarios due to the availability of many classes inheriting from *InputStream* and *OutputStream* in Java, for example, *java.io.FileOutputStream* as in the example below.

Example:

```
% Copy full data from URL to a local FileName
read_URL(URL,FileName) :-
% Open a byte stream from a URL
urlstream(URL,IS),
% Open a byte stream to FileName
OS=java.io.FileOutputStream(FileName),
% Go on copying data
copy_stream(IS,OS).
```

byte_stream([in] string, [in] encoding, [out] ByteArrayInputStream)

byte_stream([out] string, [in] encoding, [in] ByteArrayOutputStream)

Depending on whether *Arg1* is bound and *Arg3* is free, or vice versa, either create a *ByteArrayInputStream* for reading from a *String*, or create a *String* in *Arg1* from a given *ByteArrayOutputStream* in *Arg3*. The example below taken from 'test038.prova' shows how one can store a *String* into a file compressed with built-in Java compression and then decompress the contents into a *String*.

Example:

```
test038() :-
```

```

byte stream("toto", "UTF-8",BAIS) ,
FO=java.io.FileOutputStream("toto.gz"),
ZFO=java.util.zip.GZIPOutputStream(FO),
copy stream(BAIS,ZFO),
println(["Compressed file created."]).
test038() :-
FI=java.io.FileInputStream("toto.gz"),
ZFI=java.util.zip.GZIPInputStream(FI),
BAOS=java.io.ByteArrayOutputStream(),
copy stream(ZFI,BAOS),
byte stream(Result,"UTF-8",BAOS),
println(["The original string read from the compressed file: ",Result]).

```

4.5 String manipulation predicates

Prova includes a number of very useful predicates for dealing with strings. Of particular interest are non-deterministic predicates making iteration tasks very straightforward.

concat([in]List, [out]Result)

Given a Java Collection or a Prova list in *Arg1*, concatenate the strings representing its elements. On return, *Arg2* contains the concatenated strings. If an element of the collection is not a string, *toString()* is called on it before it is appended to the result string in *Arg2*.

Example:

```

Dir=java.io.File("."), % access the current directory
Files=Dir.list(), % create a Prova list containing the
 % file names
implode(",", " ,Files,Result), % embed commas between the file names
concat(["{",Result,"}"],Final), % prepend "{" and append "}"
println([Final1]). % print the result

```

implode([in]StringToEmbed, [in]List, [out]Result)

Given a Java Collection or a Prova list in *Arg2*, embed the string in *Arg1* between the strings in *Arg2*. The result string is stored in *Arg3*.

Example:

```

Dir=java.io.File("."), % access the current directory
Files=Dir.list(), % create a Prova list containing the file names
implode(",", " ,Files,Result), % embed commas between the file names
concat(["{",Result,"}"],Final), % prepend "{" and append "}"
println([Final1]). % print the result

```

unescape([in]String, [out]String)

Use a separator `\` to allow introducing special characters in string in *Arg1* with the result returned in *Arg2*. `\n`, `\r`, `\t`, `\'`, `\"` are supported with the backslash character `\` that can also be unescaped as `\\`.

Example:

```

% Note embedded double quotes in the string to be unescaped
unescape("\tline 1'\n\t\t"line 2\"'\n\tline 3",Unescaped),
println([Unescaped]).

```

```
> line 1'
> "line 2"
> line 3
```

It should be noted that the current policy of dealing with single and double quotes allows for alternating embedded single and double quotes in strings.

```
println(["AAA'BBB"CCC'DDD'EEE"FFF'GGG"])
> AAA'BBB"CCC'DDD'EEE"FFF'GGG
```

parse list([in]String, [in]RegularExpression, [out]ResultList)

Parse a string in *Arg1* using a regular expression in *Arg2* producing a Prova list in *Arg3* with elements corresponding to groups in the regular expression. The syntax of the regular expression follows the standard Java syntax.

Example:

```
db import (DB, scop, cla, Line) :-
  tokenize list (Line, "\t", [Sid, Pid, DDefs, Sccs, PX, Trace]),
  tokenize enum (DDefs, "", DDef),
  parse list (DDef, "(?: (\w*): |) (-?\d*\w?) -? (\d*)", [T|Ts]),
  sql_insert (DB, subchain, [px, chain_id, begin, end], [PX, T|Ts]).
```

tokenize list([in]String, [in]Separator, [out]ResultList)

Tokenize a string returning tokens as a list. *Arg1* contains an input string and *Arg2* contains a separator used for breaking the input string into parts. On return, *Arg3* contains the list of individual tokens.

Example:

```
% Tokenize by using tabs
tokenize list (Line, "\t", [T|Ts]),
% Insert a new record with field values
% corresponding to the identified tokens
sql_insert (DB, des, [id, type, sccs, sid, description], [T|Ts]).
```

ND tokenize enum([in]String, [in]Separator, [out]Result)

Tokenize a string non-deterministically returning one token at a time. *Arg1* contains an input string and *Arg2* contains a separator used for breaking the input string into parts. On each success, *Arg3* contains a new individual token.

Example:

```
% Tokenize non-deterministically by using tabs
tokenize enum (Line, "\t", T),
% Verify that the token T is in ArrayList
ArrayList.contains (T),
% Print the token
println ([T]).
```

ND capture enum([in]String, [in]RegularExpression, [out]ResultList)

Given an input string, non-deterministically enumerate the collections of groups captured according to a supplied regular expression pattern. *Arg1* is an input string. *Arg2* is a regular expression pattern containing possibly more than one group enclosed in brackets. *Arg3* returns *multiple* Prova lists containing substrings of *Arg1* corresponding to the captured groups. Note that there can be many

groups in the pattern, and there can be multiple occurrences of the same pattern. The former are stored in the returned Prova list. The latter are enumerated over non-deterministically.

Example:

```
Text="A doodle is a doddle",
% Non-deterministically enumerate regular expression groups
capture_enum(Text, "(d?) (dl)", Groups),
% Prints ["", "dl"] the first time, and ["d", "dl"], the second
println([Groups]).
```

parse_nv([in]String, [in]RegularExpression, [out]ResultListA, [out]ResultListB)

Given an input string in *Arg1* containing multiple embedded pairs of tokens and a regular expression pattern *Arg2* containing two capture groups, create two lists in *Arg3* and *Arg4* corresponding to, respectively, the first and second elements of the pairs. This predicate is useful for parsing data containing name-value pairs in various formats. This sort of data is, for example, found in text-based representations of biomedical data.

In the following example, a line from a textual representation of the SCOP protein classification database is first parsed into tokens and then name-value pairs of data separated by equality signs are separated into two lists with names and values respectively. Finally, the data is inserted into a database record with names corresponding to field names and values stored as data.

```
db import(DB, scop, cla, Line) :-
  tokenize_list(Line, "\t", [Sid, Pid, DDefs, Sccs, PX, Trace]),
  parse_nv(Trace, "(?: (\w+) = (\w+), ?)", [N|Ns], [V|Vs]),
  sql_insert(DB, cla, [sid, pdb_id, sccs, N|Ns], [Sid, Pid, Sccs, V|Vs]).
```

5 Using Prova for Ontology Wrapping

Ontologies play an increasingly more important role in practical applications to provide knowledge-based meta-level descriptions that help making the data self-describing. Ontologies can range from fairly simple ones (for example, simple hierarchies) in which the semantics is not very detailed to more complex ones (for example, DAML+OIL or OWL). Although semantically rich ontologies may become more important in biomedical applications in future, such applications can already profit from ontologies that are semantically thinner. In particular, Gene Ontology [4] and MESH [5] are two such examples of thin ontologies that play a very important role in annotating biomedical data including gene or protein data.

In order to make access to such ontologies more efficient, the designers of these ontologies normally use relational databases such as MySQL to store relational representation of these ontologies. The Prova language is ideally suited for capturing the logic behind more complex, especially recursive, queries to such relational representations.

As Prova has matured, we were able to use it for real-world efficient querying of the Gene Ontology. In particular, the code in Figure 24 shows the complete Prova code to check that Gene Ontology does not contain inheritance (*is_a*) links forming a triangle in such a way that a term *A is_a B* and at the same time *A is_a C* and *C is_a B*. The code runs in 16 seconds on a 1.6GHz PC.

The code in Figure 24 uses the tables *term* and *term2term* of Gene Ontology to find inheritance triangles. The rule for predicate *neighbours* uses the built-in *permute_bound* predicate to non-deterministically either go up or down the term hierarchy. The result of running this code confirms that the current version the Gene Ontology does not contain redundant inheritances.

```
% GeneStream: Transitive Redundancy query
:- solve(term2term transitive([term1_id,T1],
    [term2_id,T2],[relationship_type_id,5])).

% Rule to establish Transitive Redundancy
term2term transitive([F1,T1],[F2,T2]|R):-
  dbopen(go,DB),
  term2term transitive(DB,[F1,T1],[F2,T2]|R).
term2term transitive(DB,[F1,T1],[F2,T2]|R):-
  sql_select(DB,term2term,[F1,IDX],[F2,IDX]|R),
  sql_select(DB,term2term,[F1,IDX],[F2,IDX]|R),
  ID1<IDX,
  neighbours(DB,term2term,[F1,IDX],[F2,IDX]|R),
  sql_select(DB,term,[name,T1],[id,IDX]),
  sql_select(DB,term,[name,T2],[id,IDX]).

% Rule A1
neighbours(DB,Table,[F1,IDX],[F2,IDX]|R):-
  permute_bound([IDX,IDX],[IDA,IDB]),
  sql_select(DB,Table,[F1,IDA],[F2,IDB]|R).
```

Figure 24. Testing the Gene Ontology redundancy for length 1.

```
% GeneStream: Transitive Redundancy query
:- solve(term2term transitive(1,[term1_id,T1],
    [term2_id,T2],[relationship_type_id,5])).

% Rule to establish Transitive Redundancy
term2term transitive(N,[F1,T1],[F2,T2]|R):-
  dbopen(go,DB),
  term2term transitive(DB,N,[F1,T1],[F2,T2]|R).
term2term transitive(DB,N,[F1,T1],[F2,T2]|R):-
  N<2,
  !,
  sql_select(DB,term2term,[F1,IDX],[F2,IDX]|R),
  sql_select(DB,term2term,[F1,IDX],[F2,IDX]|R),
  ID1<IDX,
  neighbours(DB,term2term,[F1,IDX],[F2,IDX]|R),
```

```

    sql select (DB, term, [name, T1], [id, ID1]),
    sql select (DB, term, [name, T2], [id, ID2]).
term2term transitive (DB, N, [F1, T1], [F2, T2] | R) :-
sql select (DB, term2term, [F1, ID1], [F2, ID2] | R),
sql select (DB, term2term, [F1, ID2], [F2, ID1] | R),
ID1<>ID2,
neighbours recursive (DB, term2term, N, [F1, ID1], [F2, ID2] | R),
sql select (DB, term, [name, T1], [id, ID1]),
sql_select (DB, term, [name, T2], [id, ID2]).

% Rule A1
neighbours (DB, Table, [F1, ID1], [F2, ID2] | R) :-
permute bound ([ID1, ID2], [IDA, IDB]),
sql_select (DB, Table, [F1, IDA], [F2, IDB] | R).

% Rule A2
neighbours recursive (DB, Table, N, [F1, ID1], [F2, ID2] | R) :-
permute bound ([ID1, ID2], [IDA, IDB]),
id_recursive (DB, Table, N, [F1, IDA], [F2, IDB] | R).

% Rule B
id_recursive (DB, Table, N, [F1, ID1], [F2, ID1] | R).

% Rule C
id_recursive (DB, Table, N, [F1, ID1], [F2, ID2] | R) :-
N>0,
N2= N-1,
sql_select (DB, Table, [F1, ID1], [F2, ID2] | R),
id_recursive (DB, Table, Integer.N2, [F1, ID2], [F2, ID2] | R).

```

Figure 25. Testing the Gene Ontology redundancy for length 2.

The code in Figure 25 is slightly more complicated and allows searching for redundant indirect inheritances of length 2 or more. Two rules for the predicate *id_recursive* allow a completely general recursive navigation up to level *N* of relational representation of directed acyclic graphs, while the predicate *neighbours_recursive* makes it possible to go either up or down the hierarchy to find the term neighbours.

6 Prova Agents Architecture *Prova-AA*

The newly released version 1.2 of Prova introduces major extensions called Prova Agents Architecture *Prova-AA*. The language includes syntactically simple constructs allowing for sending messages via either JADE-HTTP or JMS communication protocols and for specifying sophisticated reaction rules for processing incoming messages. Due to the natural integration of Prova with Java, Prova-AA offers a syntactically economic and compact way of specifying agents' behaviour while allowing for efficient Java-based extensions to improve performance of critical operations.

To access Prova-AA features from Prova, additional *jar* files should be available on the *CLASSPATH* and special command-line arguments should be included to launch the relevant communication platforms. A subdirectory *prova-aa/examples* of the main *prova* distribution directory contains two sets of examples one each working with JADE-HTTP and JMS protocols. The provided *README* files explain the procedures needed to run the examples.

One example of the distributed systems that can be built with Prova-AA is a distributed version of the GoPubMed system [2] that allows users to *map* the results of querying PubMed medical abstracts to Gene Ontology. We have extended the web server functionality of GoPubMed by embedding a Prova agent capable of communicating via JADE into the server. The *examples/jade* subdirectory includes working prototypes used for some BioGrid applications, in particular, the versions of worker and requestor agents for GoPubMed system coded in Prova-AA.

Java Message Service JMS is an industry standard message oriented middleware platform that is part of Java 2 Enterprise Edition J2EE. Prova-AA uses a particular implementation of JMS called Joram [3]. This version has the advantage to be open source and mature, and furthermore, it is now part of a broad ObjectWeb initiative bringing together various middleware and J2EE technologies that will be used by major Linux vendors like RedHat and Suse. JMS in general has the advantage of being a guaranteed delivery messaging platform. Intuitively it means that when computer *A* sends a message to computer *B* the latter is not required to be operational. Once *B* goes online the messages will be delivered.

The JADE-HTTP has only minimal configuration requirements compared with the JMS administration, allowing for a creation of ad-hoc networks of agents.

6.1 Main features of Prova-AA

The main design philosophy behind Prova-AA is the minimalism and simplicity of the introduced syntax extensions. Prova-AA provides three main constructs for enabling agent communication: *sendMsg* predicates, reaction *rcvMsg* rules, and *rcvMsg* or *rcvMult* inline reactions.

sendMsg predicate

The *sendMsg* predicate can be embedded into the body of an arbitrary derivation or reaction rule. It can fail only if the parameters are incorrect and the message could not be sent due to various other reasons including the dropped connection (note that in the JMS case, the message may be sent anyway even if the network is down).

The format of the predicate is:

sendMsg(XID,Protocol,Agent,Performative,[Predicate|Args])Context)

or

sendMsg(XID,Protocol,Agent,Performative,Predicate(Args))Context)

where

XID is the conversation identifier (conversation-id) of the conversation to which the message will belong. If *XID* is bound (instantiated) prior to the *rcvMsg* invocation, the message will be a follow-up to a previously existing conversation with the specified conversation-id. If *XID* is free (a variable), the message will belong to a new conversation and *XID* will be instantiated after *rcvMsg* returns to the unique conversation-id generated for this new conversation. The *XID* instantiated in this latter case will have a format:

agent@machineN

where *agent* is the name of the local agent, *machine* is the machine name as recognized by *InetAddress.getLocalHost().getHostName().toLowerCase()*, and *N* is the sequential number of the conversation initiated by this agent.

Protocol can currently be either *self*, *jade*, or *jms*. It is either a message to *self*, or a message sent via *jms* or *jade* communication protocols.

Agent denotes the target of the message. For *self*, *jade*, and *jms* methods, *Agent* is the name of the target agent. For *jade* messages, the agent name takes the form *agent@machine* while for *jms* messages the agent locations are read from configuration files and are not specified in the *Agent* parameter.

Performative corresponds to the semantic instruction—the broad characterization of the message. A standard nomenclature of performatives is FIPA Agents Communication Language ACL [4].

[Predicate|Args] corresponds to the bracketed form and *Predicate(Args)* corresponds to functional form of the message content sent in the message envelope. The first form can be useful to match any literal including arity-0 predicates (in which case, *query()* is represented as *[query]*) or arity-1 predicates (in which case, *query(arg1)* is represented as *[query,arg1]*). The problem with the functional form is that it is impossible to specify a general pattern accommodating predicates of arbitrary arity while the bracketed version is compatible with any arity.

Context includes an arbitrary length list of comma-separated parameters that can be used to identify the message or to distinguish the replies to this particular message from other messages. In particular, it can be useful to include the protocol as part of context for the recipient of the message to be able to reply by using the same protocol.

The following code shows a complete rule that sends a code base (a fragment of Prova code) from an external *File* to the agent *Remote* that will then assimilate the rules being sent. The rules are encapsulated in a serializable Java *StringBuffer* object and sent with the literal for the built-in predicate *consult*. The particular version of *consult* will then read on the *Remote* machine the Prova statements from a *StringBuffer* (in contrast to the standard version of *consult* that reads statements from the specified file provided as an input string).

```
% Upload a rule base read from File to the host at address Remote via Prova-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,jms,Remote,eval,consult(SB)).
```

Before the relevant discussion below about the reaction rules, it should be mentioned that the mechanism of conversation-id(s) is designed to work in tandem with reaction rules and inline reactions to ensure that the message flows corresponding to different conversations can run in parallel.

Reaction *rcvMsg* rules

The target agent reacts to the message based on its pattern including the conversation-id, protocol, sender, performative, message content, and context. Reaction rules look exactly like normal derivation rules for a reserved predicate *rcvMsg*. The syntax of the *rcvMsg* parameters is exactly the same as that for the *sendMsg* parameters except for the *Agent* parameter corresponding in the *rcvMsg* case to the agent *from which* the message has arrived.

The following code shows general purpose but simplified reaction rules for *queryref* messages. The protocol allows for multiple replies and therefore is different from a standard FIPA query interaction protocol. The first rule triggers a non-deterministic derivation of the literal *[X|Xs]* sent as the message content by using the local rules and replies to the *queryref* originator with as many messages with the performative *inform* as there are possible instantiations to *[X|Xs]*. The second rule sends a special *end_of_transmission* message that the originator of the *queryref* message can use to determine that the sequence of replies is complete. The *end_of_transmission* message has special semantics: it is only sent when all waiting subordinate inline reactions corresponding to the same incoming message are discharged. The *Protocol* parameter available as the first parameter allows the recipient of *queryref* to learn the protocol (*jade*, *jms* etc.) that should be used for replies. Note how the conversation-id *XID* of the incoming message is reproduced in the *sendMsg* call to ensure that the new message is a follow-up (a reply) to the incoming query-ref request. The sender is then able to match replies to its requests (for example, if it has issued many similar requests) based on specific conversation-id(s).

```
% Reaction rule to general queryref
rcvMsg(XID,Protocol,From,queryref,[X|Xs]|Context) :-
```

```

derive ([X|Xs] ) ,
sendMsg (XID, Protocol, From, inform, [X|Xs] | Context) .
rcvMsg (XID, Protocol, From, queryref, [X|Xs] , Protocol) :-
sendMsg (XID, Protocol, From, end_of_transmission, [X|Xs] | Context) .

```

Inline reaction rules

If one considers the problem of an agent communicating asynchronously with several agents at the same time, one must appreciate the difficulty in recognizing the incoming messages that can look exactly the same but belong to different communication protocols or different stages of the same communication protocol. It is also important when specifying the code in the form of rules to maintain the context (or state) of the particular conversation.

Inline reactions simplify the programming style of specifying communication protocols by offering the user an opportunity to insert the literals for the predicate *rcvMsg* directly into the *body* of the rules. The syntax of the inline *rcvMsg* parameters is the same as when *rcvMsg* occurs in the head of the reaction rules.

The following example contrasts two possible ways of invoking a query for the predicate *gopubmed* locally and remotely. In the case of a local query, the *gopubmed* predicate is invoked directly from the body of the rule. This results in the instantiation of the result variable *SB*, which is communicated back to the invoking object *BioGrid* passed as the first parameter in the head of the rule. In the case of a remote query, a pair of *sendMsg/rcvMsg* predicates replaces this local call with the asynchronous remote call to the target agent *prova@rocket*. Note that the conversation-ids *XID* of *sendMsg* and *rcvMsg* are the same for the reaction to intercept only the matching reply.

```

manager ("prova@rocket") .

local query gopubmed (BioGrid, X, MaxResults, Flags) :-
gopubmed (X, MaxResults, Flags, SB) ,
BioGrid.results_ready (SB) .

remote query gopubmed (BioGrid, X, MaxResults, Flags) :-
manager (Manager) ,
sendMsg (XID, jade, Manager, queryref, gopubmed (X, MaxResults, Flags, SB) , id0) ,
rcvMsg (XID, jade, Manager, inform, gopubmed (X, MaxResults, Flags, SB) , id0) ,
BioGrid.results_ready (SB) .

```

It should be noted that the inference engine does not wait on the *rcvMsg* call. Instead, the execution flow is stored in a transparently constructed *temporal reaction rule* that is activated once the message matching the specified pattern in *rcvMsg* has arrived. The engine then continues processing other incoming messages and goals. The temporal rule includes the pattern of the message specified in *rcvMsg* (or *rcvMult*, see below) in the *head* of the rule and all remaining goals at the current execution point in its *body*. This behaviour is different from suspending the current thread and waiting for the replies to arrive as implemented by some programming languages such as "Go!". The Prova-AA execution engine maintains one main thread while allowing an unlimited number of conversations to proceed at the same time without incurring the penalty and limitations of multiple threads otherwise used solely for the purpose of maintaining several conversation states at the same time.

The *rcvMsg* inline reaction should normally be used only in a situation when zero or one replies are expected. According to the semantics shown for the (outline) *queryref* reaction rules above, a special *end_of_transmission* message serves as an indicator that no more replies will be arriving. If the *queryref* returns no replies because the queried predicate fails, *end_of_transmission* will still be sent to the originator of the query. In the case of a waiting *rcvMsg* inline reaction in the form of its corresponding temporal rule, the *end_of_transmission* results in discharging of the temporal rule and failure of the literals that follow *rcvMsg*. For a single result, the temporal rule corresponding to *rcvMsg* is deleted upon the receipt of the reply, and *end_of_transmission* is simply ignored.

If more than one reply is sent back to the query originator, *rcvMsg* will be discharged after the first reply and all subsequent replies and *end_of_transmission* will be ignored unless there exist (outline) reaction rules that match their pattern.

The last Prova-AA construct *rcvMult* is introduced in order to deal with the multiple replies situation. The arguments of *rcvMult* are exactly the same as for *rcvMsg* but the semantics is different. The temporal rule corresponding to *rcvMult* continues to wait for incoming messages until the *end_of_transmission* arrives. For each incoming reply, the trail of outstanding goals in the search tree will be independently explored until exhaustion or until a new *rcvMsg* or *rcvMult* reaction is encountered.

The following fragment shows how a remote query to predicate *parent* possibly returning multiple replies results in non-deterministic exhaustive printing of *all* the replies until finally, *end_of_transmission* arrives and *rcvMult* is discharged.

```
test() :-
% Send a queryref message with replies to be sent back to the agent
sendMsg(XID,jade,Me,queryref,parent(X,Y),id0),
rcvMult(XID,jade,Me,inform,parent(X,Y),id0),
println(["Inline reaction ",rcvMult(XID,Me,reply,parent(X,Y),id0)]).
```

Returning to the issue of the processing of conversation-id(s), note that in the above example, because *XID* is not initialized prior the call of *sendMsg*, a new conversation and conversation-id are created and *rcvMult* (that uses the same *XID*) will match only the reply to this particular *sendMsg*. There could be other messages sent between these two messages (thereby initiating another conversations or following up on other conversations) but the *rcvMult* would still be assigned to the correct conversation.

When a *rcvMsg* or *rcvMult* inline reaction is used with a bound conversation-id *XID* as the first parameter, the reaction waits for a matching message with the specified conversation-id. Otherwise, when *XID* is free before the invocation, the resulting pattern will include a variable conversation-id that will match any incoming message matching the remaining part of the pattern.

Consider the following two examples.

```
:- eval(test047()).

test047() :-
rcvMsg(XID,Protocol,From,run,[Y|Ys],id0),
println(["Printed last: ",XID]).
test047() :-
iam(Me),
sendMsg(XID,self,Me,run,t(),id0),
println(["Printed first: ",XID]).

> Printed first: mediator3
> Printed last: mediator3
```

In the first example shown above, the first clause for the predicate *test047* results in waiting for an incoming message with arbitrary conversation-id *XID*. It should be clear that the printout with the words "Printed last" is NOT printed at this stage as the rule never progresses beyond the first *rcvMsg* before the execution of the second *test047* clause. When the second clause is executed, a message initiating a new conversation is sent to the same agent *Me* via *self* protocol. The conversation-id *XID* of this conversation is initially free (which is exactly why a NEW conversation is initiated) but becomes initialized (after the *sendMsg* returns) to the newly generated conversation-id which is then printed. The message is then intercepted by the inline reaction *rcvMsg* in the first clause of *test047* resulting in the second printout to be printed. The fact that the conversation-ids in the *test047* clauses are both called *XID* is of course of no consequence since they are different variables in their bodies. They

become instantiated to the same value only because of *XID* in the first clause becoming instantiated with the conversation-id of the incoming message sent from the second clause.

The second example below is more complex and shows more subtle details of the Prova messaging.

```
:- eval(test048()).

% This is the caller that sends a queryref
% then waits in a rcvMult inline reaction,
% and also sends a message that is expected by the queryref callee
% before the latter sends back a reply.
test048() :-
  iam(Me),
  sendMsg(XID1,self,Me,queryref,test(X),id0),
  sendMsg(XID2,self,Me,data,result("toto"),id0),
  % end_of_transmission sent when rcvMsg in test(X) is discharged
  % will discharge the following inline reaction.
  rcvMult(XID1,self,Me,inform,test(X),id0),
  println(["Result2=",X]).

test(R) :-
  rcvMsg(XID,Protocol,From,data,result(R),ID),
  % When the waiting for this message is over, the end of transmission
  % for the queryref will be sent back to be intercepted in test048().
  println(["Result1=",R]).

% Reaction rules to queryref
rcvMsg(XID,Protocol,From,queryref,[X|Xs]|LocalContext) :-
  derive([X|Xs]),
  sendMsg(XID,Protocol,From,inform,[X|Xs]|LocalContext).
rcvMsg(XID,Protocol,From,queryref,[X|Xs]|LocalContext) :-
  sendMsg(XID,Protocol,From,end_of_transmission,[X|Xs]|LocalContext).

➤ Result1=toto
➤ Result2=toto
```

The body of the *test048* clause is responsible for sending a *queryref* request for *test(R)* as well as for sending a message in an independent new conversation *XID2* that will be used by *test()* called in the reaction to this *queryref*. After this message is sent, the rule waits for possibly multiple *queryref* results to be sent back. As explained above, *rcvMult* will be discharged only after an *end_of_transmission* is sent back from the *queryref*. The reaction rules to *queryref* includes one rule to derive the query *[X|Xs]* that was sent in the incoming message—in this case, matching *test(X)*. The derivation of *test(X)* results in the evaluation of the clause *test(R)*. The latter cannot complete immediately and starts waiting for an incoming message matching the pattern in the *rcvMsg* in its body. Note that since *XID* is a free variable, arbitrary conversation-id will be accepted.

The second reaction rule for *queryref* seems to be immediately sending an *end_of_transmission* even though the first rule is still waiting for an incoming message. However, due to the special semantics built into the sending of the *end_of_transmission* messages, the sending of this message is postponed until all waiting reactions corresponding to the current incoming message are discharged. As mentioned before, there is a reaction that is still pending, so the sending of *end_of_transmission* is indeed postponed. After the required message arrives, “Result1” is printed in the body of *test(R)*, and the reaction to *queryref* has finally all subordinate reactions discharged, resulting in the release of the *end_of_transmission* message. The inline reaction *rcvMult* in the body of *test048* receives one reply resulting in the printout “Result2” but is not immediately discharged due to the semantics of *rcvMult*. However, once *end_of_transmission* arrives, *rcvMult* is finally discharged and the body of *test048* is fully complete.

6.2 Communicator Prova-AA agent for Java applications

We complete this section with a description of embeddable Communicator Agent. In view of the need to integrate fairly disparate information spaces and to ensure the interoperability of various platforms we have developed and included with the Prova-AA distribution a special class *reagent.Communicator* that allows arbitrary Java applications to embed a Prova-AA agent and become equal participants in the network of such agents.

The following listing shows a fragment of a class that instantiates an embedded version Prova-AA *Communicator* that can exchange messages via JADE-HTTP on port 7778. The *Communicator* is constructed by calling its constructor with two arguments: the port number and the file with Prova script with initial goals, facts and rules constituting its initial rule base. The variable *queueAgent* inside a communication represents a message queue for the Prova engine. The queue insertions and deletions are synchronized by the class *RMessageQueue* inside Prova.

```

final private static String rules = "requestor.prova";
protected Communicator comm;
protected RMessageQueue queueAgent;

public void results_ready( StringBuffer sb ) {
    System.out.println( sb );
}

public void submit_gopubmed( String s, int imaxpapers, int imode ) {
    RMessage r = new RMessage("");
    Integer maxpapers = new Integer(imaxpapers);
    Integer mode = new Integer(imode);
    r.append_string("[0,\"eval\",[query_gopubmed,")
        .append(this)
        .append_string(",")
        .append(s)
        .append_string(",")
        .append(maxpapers)
        .append_string(",")
        .append(mode)
        .append_string("]]");
    queueAgent.add( r );
}

public static void main(String[] args) {
    // ...
    comm = new Communicator( "7778", rules );
    queueAgent = comm.queueAgent;
}

```

The Java application interfaces with the Prova rule base by using this message queue for submitting goals and using methods similar to the method *results_ready* shown above to receive results of various asynchronous queries inside the Java or Web application. The distributed GoPubMed-D web application embeds a Communicator agent in this way.

The listing below shows how easy a java application can programmatically issue queries to the running GoPubMed-D system. The listing is the full text of the *requestor.prova* file used for initializing the *Communicator* agent in the *BioGridAgent* class as shown above.

```

gopubmed server("prova@comas.soi.city.ac.uk").

query_gopubmed(BioGrid,X,MaxResults,Flags) :-
    gopubmed_server(GoPubMed),
    sendMsg(XID,jade,GoPubMed,queryref,gopubmed(X,MaxResults,Flags,SB)),
    rcvMsg(XID,jade,GoPubMed,inform,gopubmed(X,MaxResults,Flags,SB)),
    BioGrid.results_ready(SB).

```

The fact *gopubmed_server* stores the agent address of the Prova-AA agent embedded in the GoPubMed-D web application. The query for the predicate *query_gopubmed* is called from the Java application represented by the class *BioGrid*. In the body of the corresponding rule the server address is retrieved and a pair *sendMsg/rcvMsg* is used to query the predicate *gopubmed* inside the

GoPubMed-D server agent. Upon the receipt of the reply the method *results_ready* in BioGrid is called returning a *StringBuffer SB* with the HTML code for the produced page.

6.3 Specifying the behaviour of Prova-AA agents as state machines

In this section, we consider a more rigorous, state machines based implementation of the roles the Prova-AA agents play in conversations. Consider two agent roles *Seller* and *Buyer* engaging in a Direct Buy protocol [10]. The following UML statecharts and Prova-AA listings show the specifications and implementations of the Seller and Buyer roles. The code is included with the Prova distribution as test049.prova. The general pattern for encoding states and transitions in Prova-AA is:

```
state_j(...conversation_paramaters...) :-
    [!,],
    [event_j+1(...),] OR [condition_j_j+1(...), !,]
    [actions_j+1(...),]
    [state_j+1(...)].
state_j(...) :-
    ...
state_j+1(...) :-
    ...
```

For each state *j* in the state machine of an agent role, possibly parameterized with additional conversation parameters passed from the preceding states, we specify a set of rules for predicate *state_j*. If there is a family of states that are chosen dynamically based on their parameters, we specify one rule for *state_j* with the particular parameters instantiation. Moreover, if there are *N* transitions from *state_j* to subsequent states, there is one rule for *state_j* per transition. Each rule for *state_j* contains in its body:

- a possible Cut (!) used when parameters of *state_j* select this particular parameterized state;
- [EITHER] the event for a transition *j*→*j+1* using *rcvMsg* or *rcvMult* inline reactions;
- [OR] the conditions for this transition;
- the actions that should be executed on entry to state *j+1*,
- and the call to next state *j+1*.

If both event and condition are required and there could be multiple different events leading to different states, we would normally introduce an intermediate state before taking a further transition guarded by the condition. If there is no multiple choice involved in the event, event and condition(s) can be combined in one rule.

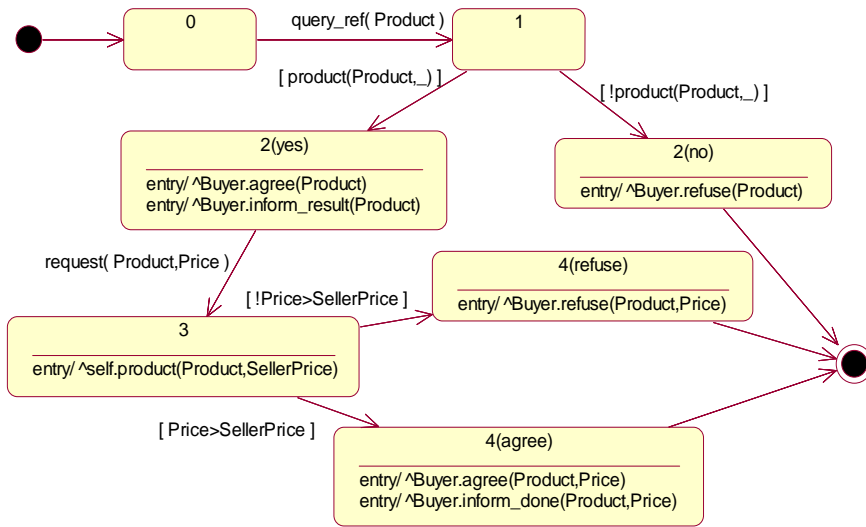
Consider for example, the two rules for the *directbuy_seller_1* corresponding to state 1 in *Seller*. There are two rules because there are two transitions from state 1. The first rule checks if the product with name *Product* exists in the database. If the product is there, as is the case for *car(volvo,s60)*, the *Cut* follows and the two messages corresponding to the entry actions for state *directbuy_seller_2(yes)* are executed. Otherwise, a transition to *directbuy_seller_2(no)* with the corresponding message sending follows.

Consider the transitions from state 1 in *Buyer* to states 2(*agree*) and 2(*refuse*). The following line

```
rcvMsg(XID, Protocol, From, Perf, Product, seller)
```

allows the agent to wait for a message for an unspecified performative *Perf*. The family of states 2 is parameterized by the particular incoming message (*agree* or *refuse*) that are chosen by unification of the predicate parameters in the heads of the corresponding rules.

Seller:



```

:- eval(directbuy seller 0() ).

product(car(volvo,s60),11000) .
product(car(ford,fiesta),5000) .

directbuy seller 0() :-
    rcvMult(XID,Protocol,From,query_ref,Product,buyer) ,
    directbuy seller 1(XID,Protocol,From,Product) .

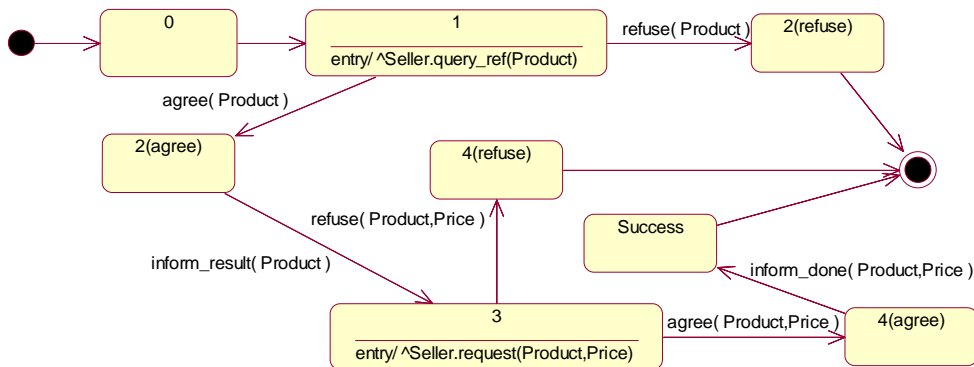
directbuy seller 1(XID,Protocol,From,Product) :-
    product(Product| ),
    !,
    sendMsg(XID,Protocol,From,agree,Product,seller) ,
    sendMsg(XID,Protocol,From,inform_result,Product,seller) ,
    directbuy seller 2(yes,XID,Protocol,From,Product) .
directbuy seller 1(XID,Protocol,From,Product) :-
    sendMsg(XID,Protocol,From,refuse,Product,seller) ,
    directbuy seller 2(no,XID,Protocol,From,Product) .

directbuy seller 2(yes,XID,Protocol,From,Product) :-
    !,
    rcvMsg(XID,Protocol,From,request,[Product,Price],buyer) ,
    product(Product,SellerPrice) ,
    directbuy seller 3(XID,Protocol,From,Product,Price,SellerPrice) .
directbuy seller 2(no,XID,Protocol,From,Product) .

directbuy seller 3(XID,Protocol,From,Product,Price,SellerPrice) :-
    Price>SellerPrice,
    !,
    sendMsg(XID,Protocol,From,agree,[Product,Price],seller) ,
    sendMsg(XID,Protocol,From,inform_done,[Product,Price],seller) ,
    directbuy seller 4(agree,XID,Protocol,From,Product,Price) .
directbuy seller 3(XID,Protocol,From,Product,Price,SellerPrice) :-
    sendMsg(XID,Protocol,From,refuse,[Product,Price],seller) ,
    directbuy seller 4(refuse,XID,Protocol,From,Product,Price) .

directbuy seller_4(agree,XID,Protocol,From,Product,Price) .
directbuy seller_4(refuse,XID,Protocol,From,Product,Price) .
    
```

Buyer: state machine and Prova-AA code



```

:- eval(directbuy buyer 0()).

directbuy buyer 0() :-
    iam(Me),
    sendMsg(XID,self,Me,query_ref,car(ford,fiesta),buyer),
    directbuy buyer 1(XID,self,Me,Product,4500).
directbuy buyer 0() :-
    iam(Me),
    sendMsg(XID,self,Me,query_ref,car(volvo,s60),buyer),
    directbuy buyer 1(XID,self,Me,Product,12000).
directbuy buyer 0() :-
    iam(Me),
    sendMsg(XID,self,Me,query_ref,car(ford,focus),buyer),
    directbuy buyer 1(XID,self,Me,Product,12000).
directbuy buyer 0() :-
    % This rule will terminate the seller's behaviour (non-standard)
    iam(Me),
    sendMsg(XID,self,Me,end of transmission,[X|Xs],buyer).

directbuy buyer 1(XID,Protocol,From,Product,Price) :-
    % Wait for a generic reply only knowing the conversation-id XID
    rcvMsg(XID,Protocol,From,Perf,Product,seller),
    % Proceed based on their first reply Perf [agree OR refuse]
    directbuy buyer 2(Perf,XID,Protocol,From,Product,Price).

directbuy buyer 2(agree,XID,Protocol,From,Product,Price) :-
    !,
    rcvMsg(XID,Protocol,From,inform result,Product,seller),
    println(["Agreed-1:"|Product]," "),
    % Send our offer (Price for Product)
    sendMsg(XID,Protocol,From,request,[Product,Price],buyer),
    directbuy buyer 3(XID,Protocol,From,Product,Price).
directbuy buyer 2(refuse,XID,Protocol,From,Product,Price) :-
    println(["Refused-1"|Product]," "),
    fail().

directbuy buyer 3(XID,Protocol,From,Product,Price) :-
    % Wait for seller's decision on our offer
    rcvMsg(XID,Protocol,From,Perf,[Product,Price],seller),
    % Proceed based on their first reply [agree OR refuse]
    directbuy buyer 4(Perf,XID,Protocol,From,Product,Price).

directbuy buyer 4(agree,XID,Protocol,From,Product,Price) :-
    !,
    println(["Agreed-2:",Product,"for",Price]," "),
    rcvMsg(XID,Protocol,From,inform done,[Product,Price],seller),
    println(["Transaction complete:",Product,"for",Price]," ").
directbuy buyer 4(refuse,XID,Protocol,From,Product,Price) :-
    println(["Refused-2",Product,"for",Price]," "),
    fail().
    
```

The output from running the above example is shown below.

```
> Agreed-1: car ford fiesta
> Agreed-1: car volvo s60
> Refused-1 car ford focus
> Refused-2 ["car","ford","fiesta"]for 4500
> Agreed-2: ["car","volvo","s60"]for 12000
> Transaction complete: ["car","volvo","s60"]for 12000
```

The product *car(ford,focus)* is refused because there is no matching product in the *Seller's* directory, *car(ford,fiesta)* is refused at the second stage of the negotiation due to the insufficient bid, and *car(volvo,s60)* is fully accepted by the *Seller* state machine resulting in the completed sale. The printouts demonstrate that runs of the state machine progress in parallel for all three conversations initiated by *Buyer*. The agent processes one incoming message at a time suspending the current processing on an inline reaction *rcvMsg* or *rcvMult*. If a longer activity was required it could have been asynchronously spawned in a separate thread.

In summary, it should be easy to encode fairly sophisticated agents conversations in Prova-AA. Inline reactions are especially useful for directly encoding the transition events. Patterns for incoming messages can contain variables (including a variable containing the performative) which allows the code to choose the next state based on the incoming event simply by unifying the relevant message parameters with the parameters of the next state. The example used the *buyer* and *seller* keywords as the context part of each message pattern to distinguish the two agent roles running in the same agent. However, the same protocol can be easily distributed between two agents on different machines using Prova-JADE or Prova-JMS for communication without any change at all in the rules except for the initial state *directbuy_state_0* where the messages to the *Seller* should be sent with *jade* or *jms* specified as communication protocols.

6.4 Asynchronous Processing of Java Methods

Inline reactions simplify greatly the process of spawning parallel computations corresponding to Java methods. Consider the following two fragments that include a call to a method 'executeTask' in the class stored in the fact for predicate 'processor'. In the first fragment, if the call takes a considerable time to finish, the whole message queue of the agent will be frozen until the method returns. In the second alternative fragment, the method is called asynchronously in a separate thread. An inline reaction *rcvMsg* enables the context of the call, including the variable 'Result' that will be returned from the clause 'worker', to be available when the method returns. The "bracket" *spawn/rcvMsg* directly replaces the synchronous call in the first fragment, transparently preserving the call context including all variables in the clause.

```
%===== Query that includes a synchronous method call =====
worker(Result,Task) :-
    processor(Psimap),
    Psimap.executeTask(Task),
    Result=Psimap.getTaskResult().

%===== The same query that calls the method asynchronously =====
worker(Result,Task) :-
    processor(Psimap),
    spawn(XID,Psimap,executeTask,Task),
    rcvMsg(XID,self,Me,return,[Ret]),
    Result=Psimap.getTaskResult().
```

The built-in predicate *spawn* requires the following three parameters:

Arg1 [inout] the conversation ID of the asynchronous call (should be a free variable);

Arg2 [in] the object whose method will be called;

Arg3 [in] the name of the method to be called.

The list of parameters may also continue with [in] parameters to the method as required by its signature.

The *rcvMsg* call that follows *spawn* uses the same variable *XID* for conversation-id. This allows the call to accept only results from this particular computation. The protocol *self* specifies that the incoming message with the results will be sent by the local agent. The performative *return* is the required type of the message returned by the Prova runtime in response to the spawned computation. Finally, the object returned from the method call is the only element in the contents list (*Ret* in the above example). If the method has a *void* return or returns null, a string constant 'null' is returned in this element.

In Prova versions later than 1.5, the results of the call can be directly returned to the clause. In the example above, the computation results are preserved in the called object itself, so an additional method call 'getTaskResult' retrieves the computation *Result* that is returned back to the caller of the 'worker' clause. However, if *Psimap.executeTask()* had a necessary return object, the latter could have been intercepted by *rcvMsg*.

The fragments below are taken from the example in */prova-aa/examples/jade/exchange* demonstrating how easy it is to run a (long) remote computation. The subdirectory *machine_a* contains a client code in *aa.prova*, while *machine_b* contains a server code in *aa_server.prova*. The reaction rule in the server code accepts messages with performative *rexec* in conversation *XID*, starting a new conversation *XID1* with an asynchronous computation for the requested *Object*, *Method*, and *Args* as explained above. The inline reaction for *XID1* intercepts the completion and the results in *Ret*. Finally, the reaction for *rexec* responds to the client with a message with performative *return* copying the returned object as well as all the input data (it is needed because the method can have side effects on its target or parameters).

On the client side in *aa.prova*, the inline reaction for the message *return* intercepts the returned results and prints a message.

```
% From aa_server.prova
rcvMsg(XID,Protocol,From,rexec,[Object,Method|Args]|LocalContext) :-
    spawn(XID1,Object,Method|Args),
    rcvMsg(XID1,self,Me,return,[Ret]),
    sendMsg(XID,Protocol,From,return,[Ret,Object,Method|Args]|LocalContext).

% From aa.prova
sendMsg(XID4,jade,Remote,rexec,[java.lang.Thread,sleep,10000],"timer",0),
% Inline reaction to the timer
rcvMsg(XID4,jade,Remote,return,[Ret,java.lang.Thread,sleep,Millisec],"timer",Count0),
println(["Timer expired:",Count0]," ").
```

In summary, both the remote invocation on the client and local invocation on the server are asynchronous, and the mechanism of inline reactions makes the programming exceptionally easy.

7 References

1. The Prova Language, City University, London, <http://www.prova.ws>.
2. DAML+OIL The DARPA Agent Markup Language, <http://daml.org>.
3. OWL W3C Web Ontology Language, <http://www.w3.org/TR/2003/CR-owl-guide-20030818>.
4. Gene Ontology, <http://www.geneontology.org>.
5. MESH Medical Subject Headings, <http://www.nlm.nih.gov/mesh/meshhome.html>.
6. The GoPubMed system, City University, London, <http://gopubmed.net>.
7. Structural Classification of Proteins SCOP, <http://scop.mrc-lmb.cam.ac.uk/scop>.
8. Mandarax Java inference system. <http://sourceforge.net/projects/mandarax>.
9. Biomedical citations system PubMed, <http://www.ncbi.nlm.nih.gov/PubMed>.
10. Direct Buy protocol, <http://taga.umbc.edu/taga2/doc/taga004>.