Prolobjectlink Projects

prolobjectlink-jpi-projog

v. 1.2-SNAPSHOT

User Guide

Prolobjectlink Project **2023-12-25**

Table of Contents

1 What Is

1.1 What is

1.1.1 Introduction

Java Prolog Interface (JPI) is an Application Provider Interface (API) for interaction between Java and Prolog programming languages. Is a bidirectional interface that communicate Java applications with Prolog program or database and Prolog procedures with Java class and methods.

JPI is an abstraction layer over concrete prolog drivers over Prolog Engines. This API define all mechanism to interact with any Prolog Engine and maintain the application independent to a specific underlying engine. JPI have several connectors to open source prolog engines like SWI, YAP, XSB native engines and tuProlog, jTrolog, jLog Java based prolog engines.

JPI study all related Java-Prolog integration libraries and take the betters features from each solution with the propose to achieve a common integration interface. The last feature allows switch the under laying Prolog Engine driver and the application code still be the same.

JPI run over any Java Virtual Machine that support Java SE 5 or above. The project was tested over HotSpot, Open J9 and JRockit Virtual Machines over Operating Systems like Windows (7,8,10), Linux (Debian, Ubuntu) and Mac OS X. Can be deployed on Servlets Containers like Jetty, Tomcat or Glassfish Application Server. JPI can be include in any Java Project using the commonest Java Integration Development Enviroment (IDE) like Eclipse, Netbeans, IntellijIDEA and so on.

JPI is developed and maintained by Prolobjectlink Project an open source initiative for build logic based applications using Prolog like fundamental Logic Programming Language in the persistence layer and application programming.

The selected license for JPI is Simplified BSD License a permissive license allowing to concrete implementations can use some possibilities like GPL, Apache 2.0 and others in the interface implementation. We suggest adopt the same license from prolog java driver if it is possible. In this way the java prolog driver and your JPI implementation share the same license and can be combined with JPI interface that is less restrictive licensed. Finally, license is the most restrictive licensed, being in many occasions the java prolog driver licenses the most restrictive.

1.1.2 Copyright and License Information

JPI is release under Simplified BSD License:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,

SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

1.1.3 Release Notes

Version 1.0.0: Initial release.

1.1.4 Acknowledgments

Thanks to Prolobjectlink Development Team, Contributors and Sponsors.

2 Getting Started

2.1 Getting Started

2.1.1 Install

Java Prolog Interface API is distributed with implementation adapter and concrete prolog driver library until it is possible according to related libraries licenses. The distributions are named normally such that **prolobjectlink-jpi-jpl7-swi7-x.y.z-dist.zip** meaning that this distribution is a JPI implementation over JPL version 7 or above and SWI-Prolog version 7 or above. The x.y.z is the distribution version. The distribution can be downloaded in zip or tar.gz compresses format. To install you need perform the following steps:

- Install Java Runtime Environment (JRE) 1.8 or above.
- Install Native Prolog Engine compatible to Operating System and your architecture. If the Prolog Engine to use is Java-based this step is omitted.
- Configure System Path with Prolog Engine routes. If the Prolog Engine to use is Java-based this step is omitted.
- Download Java Prolog Interface compatible to related prolog engine and unzip the distribution over Operating File System.
- Configure System Path with JPI unzip folder route.
- Open a new System console and type pllink –i to see the product information.

  For the JPI beginners we recommended start with a Pure Java-Prolog Engine because have less configuration aspects and native engine are more difficult to link.

2.1.2 Directories

After download and unzip JPI distribution in the final JPI folder you will see the following structure:

Folder/File	Description
bin	Binaries scripts
docs	Documentation
prt	Prolog programs files
lib	Library jars files
obj	Programs to link native engine procedures
src	Adapter source folder
CONTRIBUTING	Binaries scripts
LICENSE	Binaries scripts
NOTICE	Binaries scripts
README	Binaries scripts

2.1.3 Architecture

In general way and in bottom-up order the JPI architecture is composed by the guest Operating System at low level. Over this level we find compatible with guest Operating System and Native Prolog Engines implementations. Over this level we find Pure Java Prolog Engine implementations

and Java Driver libraries to Native Prolog Engine. Over this layer is the JPI interface adapter implementation for your correspondent Java Prolog Driver. In the top level we find a User Application that use the JPI interface.

2.1.4 Getting started Java to Prolog

After installation and architecture compression you can use the hello world sample for test the system integration. This hello world sample show how interacts with JPI from Java programming language with Abstracted Prolog Engine. For the first experience we suggesting use a Java-based Prolog engine like tuProlog because have less configuration aspects.

Create in your preferred development environment an empty project. Set in the project build path the JPI downloaded libraries located at lib folder. Create a Main Java class that look like below code:

```
public class Main {

	public static void main(String[] args) {
		PrologProvider provider = Prolog.
			getProvider(XsbProlog.class);
		PrologEngine engine = provider.newEngine();
		engine.asserta("sample('hello wolrd')");
		PrologQuery query=engine.query("sample(X)");
		System.out.println(query.one());
	}

}
```

2.1.5 Getting started Prolog to Java

Blah, Blah, …

3 Prolog Programming

3.1 Prolog Programming

3.1.1 Introduction

Prolog is a programming language that originated in the early 1970s and was initially developed for natural language processing. It became popular with the introduction of interpreters for various computer systems. Prolog evolved from being interpreted to a semi-interpreted language, thanks to the creation of a compiler. Its adoption for the fifth-generation computer project and the establishment of an ISO standard (ISO/IEC 13211-1) further contributed to its widespread use.

Prolog belongs to the paradigm of logic and declarative languages, which sets it apart significantly from more popular languages such as FORTRAN, Pascal, C, or Java. In the aforementioned programming languages, instructions are typically executed sequentially, one after another, in the same order as they are written. The order only changes when a control instruction is reached (such as a loop, conditional statement, or transfer).

Prolog programs consist of Horn clauses that represent "modus ponens" rules, meaning "If the antecedent is true, then the consequent is true." However, the way Horn clauses are written is the opposite of the usual convention. First, the consequent is written, followed by the antecedent. The antecedent can be a conjunction of conditions referred to as a sequence of goals. Each goal is separated by a comma and can be seen as similar to an instruction or procedure call in imperative languages. In Prolog, there are no control instructions. Execution is based on two concepts: unification and backtracking.

Thanks to unification, each goal determines a subset of clauses that can be executed. Each of these subsets is called a choice point. Prolog selects the first choice point and continues executing the program until determining whether the goal is true or false. If the goal is false, backtracking comes into play. Backtracking involves undoing everything that has been executed, placing the program in the same state it was in just before reaching the choice point. Then, the next pending choice point is taken, and the process is repeated. All goals conclude their execution either successfully ("true") or unsuccessfully ("false").

3.1.2 Data types

Prolog has a single data type called "term." Terms can be atoms, numbers, variables, or compound terms. Atoms are general-purpose names with no built-in meaning. Examples of atoms are x, red, 'Taco', and 'some atom'. Numbers can be floats or integers. Prolog systems compatible with the ISO standard can check the "bounded" flag. Most major Prolog systems support integers of arbitrary length. Variables are represented by strings consisting of letters, numbers, and underscores. They start with an uppercase letter or underscore. Variables closely resemble logic variables as they act as placeholders for any term.

A compound term consists of an atom called a "functor" and a number of "arguments," which are themselves terms. Compound terms are typically written as a functor followed by a list of comma-separated argument terms enclosed in parentheses. The number of arguments is referred to as the term's "arity." An atom can be seen as a compound term with arity zero. For example, person_friends(zelda, [tom, jim]) is a compound term.

Special cases of compound terms: - Lists: An ordered collection of terms denoted by square brackets. The terms are separated by commas. An empty list is represented by []. For instance, [1, 2, 3] or [red, green, blue].

- Strings: A sequence of characters surrounded by quotes. Depending on the value of the Prolog flag "double_quotes," a string can be treated as a list of character codes, a list of single-character atoms, or simply as an atom. For example, "to be, or not to be".

ISO Prolog provides predicates like atom/1, number/1, integer/1, and float/1 for type-checking.

3.1.3 Rules and Facts

Prolog programs define relationships using clauses. Pure Prolog is limited to Horn clauses. There are two types of clauses: facts and rules. A rule has the form:

Head :- Body.

This signifies that "Head is true if Body is true." The body of a rule consists of calls to predicates, which are the goals of the rule. The built-in logical operator ,/2 (denoting a binary operator named ",") represents the conjunction of goals, while ;/2 represents disjunction. Conjunctions and disjunctions can only appear in the body of a rule, not in the head.

Clauses with empty bodies are referred to as facts. An example of a fact is:

cat(tom).

This is equivalent to the rule:

cat(tom) :- true.

The built-in predicate true/0 is always true. Based on the given fact, we can ask: Is tom a cat?

?- cat(tom).

The answer is "Yes." We can also inquire about the things that are cats: What things are cats?

?- cat(X).

The answer is X = tom. Clauses with bodies are known as rules. An example of a rule is:

animal(X) :- cat(X).

If we include this rule and ask what things are animals:

?- animal(X).

The answer is X = tom.

Due to the relational nature of many built-in predicates, they can be used in multiple ways. For instance, length/2 can be used to find the length of a list (length(List, L)) given a list List, generate a list skeleton of a specific length (length(X, 5)), or generate both list skeletons and their lengths together (length(X, L)). Similarly, append/3 can be employed to append two lists (append(ListA, ListB, X)) given lists ListA and ListB, or split a given list into parts (append(X, Y, List)) given a list List. Consequently, a relatively small set of library predicates is sufficient for many Prolog programs.

As a general-purpose language, Prolog also offers various built-in predicates for common tasks such as input/output, graphics usage, and interaction with the operating system. These predicates do not have relational meanings and are only useful for their system-related effects. For example, the predicate write/1 displays a term on the screen.

3.1.4 Execution

To run a Prolog program, you start by entering a single goal called the query. The Prolog engine then attempts to find a resolution refutation of the negated query. Prolog uses a method called SLD resolution. If the negated query can be proven false, it means that the original query, with the appropriate variable assignments, is a logical consequence of the program. In this case, all the variable assignments are displayed, and the query is considered successful.

Operatively, Prolog's execution strategy can be seen as an extension of function calls in other programming languages. One difference is that multiple clause heads can match a particular call. When this happens, the system creates a choice point, where it matches the goal with the clause head of the first alternative and proceeds with that alternative's goals. If any goal fails during program execution, all variable assignments made since the most recent choice point was created are undone, and execution continues with the next alternative of that choice point. This strategy is known as chronological backtracking.

For example:

```
mother_child(trude, sally).
father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).
sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).
parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

Executing the following query will yield a true result:

```
?- sibling(sally, erica).
Yes
```

Here's how the result is obtained: Initially, the only clause head that matches the query `sibling(sally, erica)` is the first one. Therefore, proving the query is equivalent to proving the body of that clause with the appropriate variable assignments, which in this case is the conjunction `(parent_child(Z,sally), parent_child(Z,erica))`. The next goal to prove is the leftmost part of this conjunction: `parent_child(Z, sally)`. There are two clause heads that match this goal. The system creates a choice point and attempts the first alternative, which has the body `father_child(Z, sally)`. This goal can be proven with the fact `father_child(tom, sally)`, leading to the assignment `Z = tom`. The next goal to prove is the second part of the conjunction: `parent_child(tom, erica)`. This is also proven by the corresponding fact. Since all the goals have been proven, the query is considered successful. As the query doesn't contain any variables, no assignments are displayed to the user.

A query that includes variables, such as `?- father_child(Father, Child).`, will list all valid answers through backtracking. Note that with the given code, the query `?- sibling(sally, sally).` also succeeds. If there are specific restrictions, additional goals should be added to the code.

ISO Prolog is a technical standard developed by the International Organization for Standardization (ISO). It consists of two main parts. The first part, ISO/IEC 13211-1, was published in 1995 with the goal of standardizing the core elements of Prolog. This standard aims to bring clarity and remove ambiguities in the language, making it easier to write portable programs. Additionally, there have been three corrigenda issued: Cor.1:2007, Cor.2:2012, and Cor.3:2017.

The second part, ISO/IEC 13211-2, was published in 2000 and provides support for modules within the standard. The maintenance of this standard is overseen by the ISO/IEC JTC1/SC22/WG17 working group. In the United States, the US Technical Advisory Group for the standard is ANSI X3J17.

4 Bidirectional Interface

4.1 Bidirectional Interface

4.1.1 Getting started Java to Prolog

After installation and architecture compression you can use the hello world sample for test the system integration. This hello world sample show how interacts with JPI from Java programming language with Abstracted Prolog Engine. For the first experience we suggesting use a Java-based Prolog engine like tuProlog because have less configuration aspects.

Create in your preferred development environment an empty project. Set in the project build path the JPI downloaded libraries located at lib folder. Create a Main Java class that look like below code:

```
public class Main {
        public static void main(String[] args) {
        PrologProvider provider = Prolog.getProvider();
        PrologEngine engine = provider.newEngine();
        engine.asserta("sample('hello wolrd')");
        PrologQuery query=engine.query("sample(X)");
        System.out.println(query.one());
        }
 }
```

4.1.2 Architecture

JPI use a layered architecture pattern where every layer represents a component. The multi-engine Java Prolog connectors provide different levels of abstraction to simplify the implementations of common inter-operability task JPC. Java Prolog Connectors architectures describe three fundamentals layers, High-level API layer, Engine Adapter layer and Concrete Engine layer. High-level API layer define all services to be used by the users in the Java Prolog Application that is the final architecture layer on the architecture stack. High-level API provide the common implementation of Engine Abstraction, Data Type and Inter-Language conversion. The adapter layer adapts before mentioned features to communicate with the concrete Engine Layer, being the last responsible of execute the request services.

All existing Java Prolog Connectors implementation only bring support for Native Prolog Engines that have JVM bindings driver. JPI project is more inclusive and find connect all Prolog Engines Categories, Native and Java Based implementations. Some particular Java Based implementations in the future can be implement in strike forward mode the JPI interface. This particulars implementations reduce the impedance mismatch by remove the adapter layer. Therefore, JPI reference implementations will be faster than other that use adapter layer.

In JPI architecture stack in the bottom layer we have the Operating System. The Operating System can be Windows, Linux or Mac OS. Over Operating System, we have the native implementation of JVM and Prolog Engines like SWI, SWI7 and others. Over JVM and Prolog Engines we have Java Based Prolog Engines implementations and JVM bindings driver that share the runtime environment with JVM and native Prolog Engines. Over Java Based Prolog Engines implementations and JVM bindings drivers we have the JPI correspondent adapters. The adapters artifacts are the JPI implementations for each Prolog Engines. Over each adapter we have the JPI application provider interface and at the top stack we the final user application. The user application only interacts with the JPI providing single sourcing and transparency.

4.1.3 Prolog Provider

Prolog Provider is the mechanism to interact with all Prolog components. Provider classes implementations allow create Prolog Terms, Prolog Engine, Java Prolog Converter, Prolog Parsers and system logger. Using io.github.prolobjectlink.prolog.Prolog bootstrap class the Prolog Providers are created specifying the provider class in getProvider(Class ?) method. This is the workflow start for JPI. When the Prolog Provider is created the next workflow step is the Prolog Terms creation using Java primitive types or using string with Prolog syntax. Provider allow create/parsing all Prolog Terms (Atoms, Numbers, Variables and Compounds). After term creation/parsing the next step is create an engine instance with newEngine() method. Using previous term creation and engine instance Prolog Queries can be formulated. This is possible because the engine class have multiples queries creation methods like a query factory. After query creation the Query interface present many methods to retrieve the query results. The result methods are based on result quantities, result terms, result object types, etc… This is the final step in the workflow. In the table 10 is resumed all Prolog Provider Interface methods.

4.1.4 Prolog Terms

All Java Prolog connector libraries provide data type abstraction. Prolog data type abstraction have like ancestor the Term class. Prolog term is coding like abstract class and other Prolog terms are derived classes. In PrologTerm is defined the common term operation for all term hierarchy (functor, arity, compare, unify, arguments). The derived classes implement the correct behavior for each before mentioned operations. All Prolog data types PrologAtom, PrologNumber, PrologList, PrologStructure and PrologVariable are derived from this class. All before mentioned classes extends from this class the commons responsibilities. PrologTerm extends from Comparable interface to compare the current term with another term based on Standard Order.

PrologAtom represent the Prolog atom data type. Prolog atoms are can be of two kinds simple or complex. Simple atoms are defined like a single alpha numeric word that begin like initial lower case character. The complex atom is defining like any character sequence that begin and end with simple quotes. The string passed to build a simple atom should be match with {a-z}{A-Za-z0-9_}* regular expression. If the string passed to build an atom don't match with the before mentioned regular expression the atom constructor can be capable of create a complex atom automatically. For complex atom the string value can have the quotes or just can be absent. The printed string representation of the complex atom implementation set the quotes if they are needed.

```
    PrologTerm pam = provider.newAtom("pam");
         PrologTerm bob = provider.newAtom("bob");
```

PrologDouble represent a double precision floating point number. Extends from PrologNumber who contains an immutable Double instance. The Prolog Provider is the mechanism to create a new Prolog double invoking PrologProvider.newDouble(Number). PrologFloat represent a single precision floating point number. Extends from PrologNumber who contains an immutable Float instance. The Prolog Provider is the mechanism to create a new Prolog float invoking PrologProvider.newFloat(Number). PrologInteger represent an integer number. Extends from PrologNumber who contains an immutable Integer instance. The Prolog Provider is the mechanism to create a new Prolog integer invoking PrologProvider.newInteger(Number). Prolog term that represent a long integer number. Extends from PrologNumber who contains an immutable Long instance. The Prolog Provider is the mechanism to create a new Prolog long integer invoking PrologProvider.newLong(Number).

```
PrologTerm pi = provider.newDouble(Math.PI);
PrologTerm euler = provider.newFloat(Math.E);
PrologTerm i = provider.newInteger(10);
PrologTerm l = provider.newLong(10);
```

PrologVariable is created using PrologProvider.newVariable(int) for anonymous variables and PrologProvider.newVariable(String, int) for named variables. The Prolog variables can be used and reused because they remain in java heap. You can instantiate a prolog variable and used it any times in the same clause because refer to same variable every time. The integer parameter represents the declaration variable order in the Prolog clause starting with zero.

```
PrologTerm x = provider.newVariable("X", 0);
PrologTerm y = provider.newVariable("Y", 1);
PrologTerm z = provider.newVariable("Z", 2);

engine.assertz(
       provider.newStructure(grandparent, x, z),
       provider.newStructure(parent, x, y),
              provider.newStructure(parent, y, z)
              );
```

PrologReference term is inspired on JPL JRef. This term is like a structure compound term that have like argument the object identification atom. The functor is the @ character and the arity is 1. An example of this prolog term is e.g. @(J#0000000000000425). To access to the referenced object, is necessary use PrologTerm.getObject().

PrologList are a special compound term that have like functor a dot (.) and arity equals 2. Prolog list are recursively defined. The first item in the list is referred like list head and the second item list tail. The list tail can be another list that contains head and tail. A special list case is the empty list denoted by no items brackets ([]). The arity for this empty list is zero. The Prolog Provider is the mechanism to create a new PrologList is invoking PrologProvider.newList() for empty list or PrologProvider.newList(PrologTerm) for one item list or PrologProvider.newList(PrologTerm[]) for many items.

```
PrologTerm empty = provider.newList();
PrologTerm one = provider.newInteger(1);
PrologTerm two = provider.newInteger(2);
PrologTerm three = provider.newInteger(3);
PrologTerm list = provider.newList(
                        new PrologTerm[] { one, two, three}
                );
for (PrologTerm prologTerm : list) {
       System.out.println(prologTerm);
}
```

PrologList implement Iterable interface to be used in for each sentence iterating over every element present in the list.

```
Iterator<PrologTerm> i = list.iterator();
 while (i.hasNext()) {
       PrologTerm prologTerm = i.next();
       System.out.println(prologTerm);
 }
```

```
for (Iterator<PrologTerm> i = list.iterator(); i.hasNext();) {
       PrologTerm prologTerm = i.next();
       System.out.println(prologTerm);
}
```

Prolog structures consist in a relation the functor (structure name) and arguments enclosed between parenthesis. The Prolog Provider is the mechanism to create a new Prolog structures invoking PrologProvider.newStructure(String, PrologTerm...). Two structures are equals if and only if are structure and have equals functor and arguments. Structures terms unify only with same functor and arguments structures, with free variable or with with structures where your arguments unify if they have the same functor and arity. Structures have a special property named arity that means the number of arguments present in the structure. There are two special structures term. They are expressions (Two arguments structure term with operator functor) and atoms (functor with zero arguments). For the first special case must be used PrologProvider.newStructure(PrologTerm, String, PrologTerm) specifying operands like arguments and operator like functor.

```
PrologTerm pam = provider.newAtom("pam");
PrologTerm bob = provider.newAtom("bob");
PrologTerm parent = provider.newStructure("parent", pam, bob);
```

4.1.5 Prolog Engine

Prolog Engine provide a general propose application interface to interact with Prolog Programing Language. Is a convenient abstraction for interacting with Prolog Virtual Machine from Java. In Java Prolog Engine connectors libraries, the abstract engine is able to answer queries using the abstract term representation before mentioned. There are several implementation engines and in this project we try connect from top level engine to more concrete or specific Prolog Engine. Based on JPC we have a top level engine that communicate with more concretes engines. Over this concretes engines we offer several services to interact with the concrete engines with low coupling and platform independency.

4.1.6 Prolog Query

Prolog query is the mechanism to query the prolog database loaded in prolog engine. The way to create a new prolog query is invoking query() method in the Prolog Engine. When this method is called the prolog query is open an only dispose() in PrologQuery object close the current query and release all internal resources. Prolog query have several methods to manipulate the result objects. The main difference is in return types and result quantities. The result types enough depending of desire data type. Maps of variables name key and Prolog terms as value, Maps of variables name key and Java objects as value, List of before mentioned maps, Prolog terms array, Prolog terms matrix, list of Java Objects and list of list of Java Objects. Respect to result quantities Prolog query offer one, n-th or all possible solutions. This is an important feature because the Prolog engine is forced to retrieve the necessary solution quantities. Prolog query implement Iterable and Iterator. This implementation helps to obtain successive solutions present in the query.

```
public class Main {
public static void main(String[] args) {
        PrologProvider provider = Prolog.getProvider();
        PrologEngine engine = provider.newEngine("zoo.pl");
         PrologVariable x = provider.newVariable("X", 0);
         PrologQuery query = engine.query(provider.newStructure("dark", x))
         while (query.hasNext()) {
                PrologTerm value = query.nextVariablesSolution().get("X");
                System.out.println(value);
         }
         query.dispose();
         engine.dispose();
        }
}
```

```
public class Main {
        public static void main(String[] args) {
                PrologProvider provider = Prolog.getProvider();
                 PrologEngine engine = provider.newEngine("zoo.pl");
                 PrologVariable x = provider.newVariable("X", 0);
                 PrologQuery query = engine.query(provider.newStructure("da
                 for (Collection<PrologTerm> col : query) {
                        for (PrologTerm prologTerm : col) {
                                System.out.println(prologTerm);
                        }
                 }
                 query.dispose();
                 engine.dispose();
        }
}
```

4.1.7 Prolog Query Builder

Prolog query builder to create prolog queries. The mechanism to create a new query builder is using PrologEngine.newQueryBuilder(). The query builder emulates the query creation process. After define all participant terms with the begin(PrologTerm) method, we specify the first term in the query. If the query has more terms, they are created using comma(PrologTerm) for everyone. Clause builder have a getQueryString() for string representation of the clause in progress. After clause definition this builder have query() method that create the final query instance ready to be used. The follow code show how create a Prolog query ?- big(X), dark(X). using PrologQueryBuilder interface.

```
PrologVariable x = provider.newVariable("X", 0);
PrologStructure big = provider.newStructure("big", x);
PrologStructure dark = provider.newStructure("dark", x);
PrologQueryBuilder builder = engine.newQueryBuilder();
PrologQuery query = builder.begin(dark).comma(big).query();
```

4.1.8 Prolog Clause

Prolog clause is composed by two prolog terms that define a prolog clause, the head and the body. This representation considers the prolog clause body like a single term. If the body is a conjunctive set of terms, the body is a structure with functor/arity (, /2) and the first argument is the first element in the conjunction and the rest is a recursive functor/arity (, /2). The functor and arity for the clause

is given from head term functor and arity. This class define some properties for commons prolog clause implementations. They are boolean flags that indicate if the prolog clause is dynamic multi-file and discontiguos. This class have several methods to access to the clause components and retrieve some clause properties and information about it. Additionally, this class contains a prolog provider reference for build terms in some operations.

4.1.9 Prolog Clause Builder

Prolog clause builder to create prolog clauses. The mechanism to create a new clause builder is using PrologEngine.newClauseBuilder(). The clause builder emulates the clause creation process. After define all participant terms with the begin(PrologTerm) method, we specify the head of the clause. If the clause is a rule, after head definition, the clause body is created with neck(PrologTerm) for the first term in the clause body. If the clause body have more terms, they are created using comma(PrologTerm) for everyone. Clause builder have a getClauseString() for string representation of the clause in progress. After clause definition this builder have asserta(), assertz(),clause(),retract() that use the wrapped engine invoking the correspondent methods for check, insert or remove clause respectively.

```
PrologTerm z = provider.newVariable("Z", 0);
PrologTerm darkZ = provider.newStructure("dark", z);
PrologTerm blackZ = provider.newStructure("black", z);
PrologTerm brownZ = provider.newStructure("brown", z);
PrologClauseBuilder builder = engine.newClauseBuilder();
builder.begin(darkZ).neck(blackZ).assertz();
builder.begin(darkZ).neck(brownZ).assertz();
```

The Prolog result in database is showed in the follow code. The table 19 show the Prolog clause builder interface methods.

```
dark(Z): -
        black(Z).
 dark(Z): -
        brown(Z).
```

4.1.10 Prolog Scripting in Java

Java 6 added scripting support to the Java platform that lets a Java application execute scripts written in scripting languages such as Rhino JavaScript, Groovy, Jython, JRuby, Nashorn JavaScript, etc. All classes and interfaces in the Java Scripting API are in the javax.script package. Using a scripting language in a Java application provides several advantages, dynamic type, simple way to write programs, user customization, easy way to develop and provide domain-specific features that are not available in Java. For achieve this propose Java Scripting API introduce a scripting engine component. A script engine is a software component that executes programs written in a particular scripting language. Typically, but not necessarily, a script engine is an implementation of an interpreter for a scripting language. To run a script in Java is necessary perform the following three steps, create a script engine manager, get an instance of a script engine from the script engine manager and Call the eval() method of the script engine to execute a script.

```
public class Main {
        public static void main(String[] args) {
                ScriptEngineManager manager = new ScriptEngineManager();
                ScriptEngine engine = manager.getEngineByName("prolog");
                Boolean result = engine.eval("?- X is 5+3.");
                Integer solution = engine.get("X");
                System.out.println(solution);
        }
}
```

Using script engine, it possible read Prolog source file. Read Prolog source file allow coding all prolog source in separate mode respect to Java program.

```
public class Main {
        public static void main(String[] args) {
                ScriptEngineManager manager = new ScriptEngineManager();
                ScriptEngine engine = manager.getEngineByName("prolog");
                Boolean read = engine.eval(new FileReader("family.pl"));
                Boolean eval = engine.eval("?- parent( Parent, Child)");
                Object parent = engine.get("Parent");
                Object child = engine.get("Child");
                System.out.println(parent);
                System.out.println(child);
        }
}
```

4.1.11 Getting started Prolog to Java

One of the main advantages of tuProlog open architecture is that any Java component can be directly accessed and used from Prolog, in a simple and effective way, by means of the JavaLibrary library: this delivers all the power of existing Java components and packages to tuProlog sources. In this way, all Java packages involving interaction (such as Swing, JDBC, the socket package, RMI) are immediately available to increase the interaction abilities of tuProlog: “one library for all libraries” is the basic motto.

4.1.11.1 Mapping data structures

Complete bi-directional mapping is provided between Java primitive types and tuProlog data types. By default, tuProlog integers are mapped into Java int or long as appropriate, while byte and short types are mapped into tuProlog’s Int instances. Only Java double numbers are used to map tuProlog reals, but float values returned as result of method invocations or field accesses are handled properly anyway, without any loss of informa- tion. Boolean Java values are mapped into specific tuProlog Term constants. Java chars are mapped into Prolog atoms, but atoms are mapped into Java Strings by default. The any variable ( ) can be used to specify the Java null value.

4.1.11.2 General predicates description

The library offers the following predicates: (i) the java object/3 predicate is used to create a new Java object of the specified class, according to the syntax:

```
java_object(ClassName, ArgumentList, ObjectRef )
```

ClassName is a Prolog atom bound to the name of the proper Java class (e.g. ‘Counter’, ‘java.io.FileInputStream’), while the parameter ArgumentList is a Prolog list used to supply the

required arguments to the class constructor: the empty list matches the default constructor. Also Java arrays can be instantiated, by appending [] at the end of the ClassName string. The reference to the newly-created object is bound to ObjectRef , which is typically a ground Prolog term; alternatively, an unbound term may be used, in which case the term is bound to an automatically-generated Prolog atom '$obj-N ', where N is a progressive integer. In both cases, these atoms are in- terpreted as object references – and therefore used to operate on the Java object from Prolog – only in the context of JavaLibrary's pred- icates. The predicate fails whenever ClassName does not identify a valid Java class, or the constructor does not exists, or arguments in ArgumentList are not ground, or ObjectRef already identifies an object in the system.

According to the default behaviour of java object, when a ground term is bound to a Java object by means of the predicate, the binding is kept for the full time of the demonstration (even in the case of backtracking). This behaviour can be changed, getting the bindings created by the java object undone by backtracking, by changing the value of the flag java object backtrackable to true (the default is false).

(ii) the &lt-/2 predicate is used to invoke a method on a Java object according to a send-message pattern:

```
ObjectRef <- MethodName (Arguments )
ObjectRef <- MethodName (Arguments ) returns Term
```

ObjectRef is an atom interpreted as a Java object reference as explained above, while MethodName is the Java name of the method to be invoked, along with its Arguments . The returns keyword is used to retrieve the value returned from non-void Java methods and bind it to a Prolog term: if the type of the returned value can be mapped onto a primitive Prolog data type (a number or a string), Term is unified with the corresponding Prolog value; if, instead, it is a Java object other than the ones above, Term is handled as ObjectRef in the case of java object/3. Static methods can be invoked using the compound term class(ClassName) in the place of ObjectRef . If MethodName does not identify a valid method for the object (class), or arguments in ArgumentList are not valid (because of a wrong signature or not ground values) the predicate fails.

(iii) the . infix operator is used, in conjunction with the set / get pseudo- method pair, to access the public fields of a Java object. The syntax is based on the following constructs:

```
ObjectRef . Field &lt- set(GroundTerm )
ObjectRef . Field &lt- get(Term )
```

As usual, ObjectRef is the Prolog identifier for a Java object. The first construct set the public field Field to the specified GroundTerm , which may be either a value of a primitive data type, or a reference to an existing object: if GroundTerm is not ground, the infix predi- cate fails. The second construct retrieves the value of the public field Field , where Term is handled once again as ObjectRef in the case of java object/3. As for methods, static fields of classes can be accessed using the compound term class(ClassName) in the place of ObjectRef . Some helper predicates are provided to access Java arrayelements:

```
java_array_set(ArrayRef,Index,Object )
java_array_set_Basic_Type(ArrayRef,Index,Value )
```

to set elements,

```
java_array_get(ArrayRef,Index,Object )
java_array_get_Basic_Type(ArrayRef,Index,Value )
```

to get elements,

```
java_array_length(ArrayObject,Size )
```

to get the array length.

It is worth to point out that the set and get formal pseudo-methods above are not methods of some class, but just part of the construct of the . infix operator, according to a JavaBeans-like approach.

(iv) the as infix operator is used to explicitly specify (i.e., cast) method argument types:

```
ObjectRef as ClassName
```

By writing so, the object represented by ObjectRef is considered to belong to class Classname : both ObjectRef and Classname have the usual meaning explained above. The operator works also with primitive Java types, specified as Classname (for instance, myNumber as int). The purpose of this predicate is both to provide methods with the exact Java types required, and to solve possible overloading conflicts a-priori.

(v) The java class/4 predicate makes it possible to create and load a new Java class from a source text provided as an argument, thus supporting dynamic compilation of Java classes:

```
java_class(SourceText, FullClassName, ClassPathList, ObjectRef )
```

SourceText is a string representing the text source of the Java class, FullClassName is the full Java class name, and ClassPathList is a (possibly empty) Prolog list of class paths that may be required for a successful dynamic compilation of this class. ObjectRef is a reference to an instance of the class java.lang.Class that represents the newly-created class. The predicate fails whenever SourceText contains errors, or the class cannot be located in the package hierarchy as specified, or ObjectRef already identifies an object in the system.

4.1.11.3 Predicates

Here follows a list of predicates implemented by this library, grouped in categories corresponding to the functionalities they provide.

4.Method Invocation, Object and Class Creation

• java_object/3 java_object(ClassName, ArgList, ObjId) is true iff ClassName is the full class name of a Java class available on the local file system, ArgList is a list of arguments that can be meaningfully used to in- stantiate an object of the class, and ObjId can be used to reference such an object; as a side effect, the Java object is created and the reference to it is unified with ObjId. It is worth noting that ObjId can be a Prolog variable (that will be bound to a ground term) as well as a ground term (not a number). According to the value of the flag java object retractable, the binding that is established between the ObjId term and the Java object is not destroyed with backtrack- ing (false value, default case) or destroyed (true value).

```
Template: java_object(+full class name,+list,?obj id)
```

• java_object_bt/3 java_object_bt(ClassName, ArgList, ObjId) has the same behaviour of java_object/3 when the value of java object backtrackable is true.

```
Template: java object bt(+full class name,+list,?obj id)
```

• java_object_nb/3 java_object_nb(ClassName, ArgList, ObjId) has the same behaviour of java object/3 when the value of java object backtrackable is false.

```
Template: java_object_nb(+full class name,+list,?obj id)
```

• destroy_object/1 destroy_object(ObjId) is true and as a side effect the binding be- tween ObjId and a Java object, possibly established, by previous pred- icates is destroyed.

```
Template: destroy_object(@obj id)
```

• java_class/4 java_class(ClassSourceText, FullClassName, ClassPathList, ObjId) is true iff ClassSouceText is a source string describing a valid Java class declaration, a class whose full name is FullClassName, accord- ing to the classes found in paths listed in ClassPathList, and ObjId can be used as a meaningful reference for a java.lang.Class object representing that class; as a side effect the described class is (possibly created and) loaded and made available to the system.

```
Template: java_class(@java source,@full class name,@list,?obj id)
```

• java_call/3 java_call(ObjId, MethodInfo, ObjIdResult) is true iff ObjId is a ground term currently referencing a Java object, which provides a method whose name is the functor name of the term MethodInfo and possible arguments the arguments of MethodInfo as a compound, and ObjIdResult can be used as a meaningful reference for the Java object that the method possibly returns. As a side effect the method is called on the Java object referenced by the ObjId and the object possibly returned by the method invocation is referenced by the ObjIdResult term. The anonymous variable used as argument in the MethodInfo structure is interpreted as the Java null value.

```
Template: java_call(@obj id,@method signature,?obj id)
```

• ’&lt-’/2 ’&lt-’(ObjId, MethodInfo) is true iff ObjId is a ground term currently referencing a Java object, which provides a method whose name is the functor name of the term MethodInfo and possible arguments the arguments of MethodInfo as a compound. As a side effect the method is called on the Java object referenced by the ObjId. The anonymous variable used as argument in the MethodInfo structure is interpreted as the Java null value.

```
Template: ’<-’(@obj id,@method signature)
```

• return/2 return(’&lt-’(ObjId,MethodInfo),ObjIdResult) is true iff ObjId is a ground term currently referencing a Java object, which provides a method whose name is the functor name of the term MethodInfo and possible arguments the arguments of MethodInfo as a compound, and ObjIdResult can be used as a meaningful reference for the Java object that the method possibly returns. As a side effect the method is called on the Java object referenced by the ObjId and the object possibly returned by the method invocation is referenced by the ObjIdResult term. The anonymous variable used as argument in the MethodInfo structure is interpreted as the Java null value. It is worth noting that this predicate is equivalent to the java_call predicate.

```
Template: return(’&lt-’(@obj id,@method signature),?obj id)
```

4.Java Array Management

• java_array_set/3 java_arrayset(ObjArrayId,Index,ObjId) is true iff ObjArrayId is a ground term currently referencing a Java array object, Index is a valid index for the array and ObjId is a ground term currently refer- encing a Java object that could inserted as an element of the array (according to Java type rules). As side effect, the object referenced by ObjId is set in the array referenced by ObjArrayId in the po- sition (starting from 0, following the Java convention) specified by Index. The anonymous variable used as ObjId is interpreted as the Java null value. This predicate can be used for arrays of Java ob- jects: for arrays whose elements are Java primitive types (such as int, float, etc.) the following predicates can be used, with the same se- mantics of java_array_set but specifying directly the term to be set as a tuProlog term (according to the mapping described previously):

```
java_array_set_int(ObjArrayId,Index,Integer)
java_array_set_short(ObjArrayId,Index,ShortInteger)
java_array_set_long(ObjArrayId,Index,LongInteger)
java_array_set_float(ObjArrayId,Index,Float)
java_array_set_double(ObjArrayId,Index,Double)
java_array_set_char(ObjArrayId,Index,Char)
java_array_set_byte(ObjArrayId,Index,Byte)
java_array_set_boolean(ObjArrayId,Index,Boolean)
```

```
Template: java_array_set(@obj id,@positive integer,+obj id)
```

• java_array_get/3 java_array_get(ObjArrayId, Index, ObjIdResult) is true iff ObjArrayId is a ground term currently referencing a Java array object, Index is a valid index for the array, and ObjIdResult can be used as a meaning- ful reference for a Java object contained in the array. As a side effect, ObjIdResult is unified with the reference to the Java object of the array referenced by ObjArrayId in the Index position. This predicate can be used for arrays of Java objects: for arrays whose elements are Java primitive types (such as int, float, etc.) the following predi- cates can be used, with the same semantics of java_array_get but binding directly the array element to a tuProlog term (according to the mapping described previously):

```
java_array_get_int(ObjArrayId,Index,Integer)
java_array_get_short(ObjArrayId,Index,ShortInteger)
java_array_get_long(ObjArrayId,Index,LongInteger)
java_array_get_float(ObjArrayId,Index,Float)
java_array_get_double(ObjArrayId,Index,Double)
java_array_get_char(ObjArrayId,Index,Char)
java_array_get_byte(ObjArrayId,Index,Byte)
java_array_get_boolean(ObjArrayId,Index,Boolean)
```

```
Template: java_array_get(@obj id,@positive integer,?obj id)
```

• java_array_length/2 java array length(ObjArrayId, ArrayLength) is true iff ArrayLength is the length of the Java array referenced by the term ObjArrayId.

```
Template: java_array_length(@term,?integer)
```

4.Helper Predicates

• java object string/2 java object string(ObjId,String) is true iff ObjId is a term refer- encing a Java object and PrologString is the string representation of the object (according to the semantics of the toString method pro- vided by the Java object).

```
Template: java object string(@obj id,?string)
```

4.Java Swing GUI from tuProlog

What about creating Java GUI components from the tuProlog environment? Here is a little example, where a standard Java Swing open file dialog win- dows is popped up:

```
open_file_dialog( FileName ):-
        java_object( 'javax.swing.JFileChooser', [], Dialog ),
        Dialog <- showOpenDialog(_) returns Result,
        write(Result),
        Dialog <- getSelectedFile returns File,
        File <- getName returns FileName,
        class('java.lang.System') . out <- get(Out),
        Out <- println('you want to open file '),
        Out <- println(FileName).
```

5 Development Tools

Paragraph 1, line 1. Paragraph 1, line 2.

Paragraph 2, line 1. Paragraph 2, line 2.

5.1 Section title

5.1.1 Sub-section title

5.1.1.1 Sub-sub-section title

5.Sub-sub-sub-section title

5.Sub-sub-sub-sub-section title

- List item 1.
- List item 2.
  Paragraph contained in list item 2.
  - Sub-list item 1.
  - Sub-list item 2.
- List item 3. Force end of list:

```
Verbatim text not contained in list item 3
```

1. Numbered item 1.
   A. Numbered item A.
   B. Numbered item B.
2. Numbered item 2.

List numbering schemes: [[1]], [[a]], [[A]], [[i]], [[I]].

Defined term 1

of definition list.

Defined term 2

of definition list.

```
Verbatim text
                        in a box
```

--- instead of +-- suppresses the box around verbatim text.

Figure caption

Centered cell 1,1	Left-aligned cell 1,2	Right-aligned cell 1,3
cell 2,1	cell 2,2	cell 2,3

Table caption

No grid, no caption:

cell	cell
cell	cell

Horizontal line:

5.2 ^L New page.

Italic font. **Bold** font. `Monospaced` font.

Anchor. Link to anchor. Link to http://www.pixware.fr. Link to showing alternate text. Link to Pixware home page.

Force line
break.

Non breaking space.

Escaped special characters: ~, =, -, +, *, [,], <, >, {, }, \.

Copyright symbol: ©, ©, ©.

6 Contribution

6.1 Contribution

6.1.1 Issues

See the issue tracker at https://github.com/prolobjectlink/prolobjectlink-jpi-projog to create a new issue or take an existing one.

6.1.2 Changes and Build

Fork the repository in GitHub.

Clone your forked repository in your preferred IDE

Prolobjectlink development requires.

- Java 1.8 - Maven 3.1.0 or above

Make changes in your cloned repository

Run all test to see if the system still consistent after your changes

Create unit-tests and make sure that the include changes are covered to 100%

Run the benchmark to see if the system performance still consistent after your changes

Add a description of your changes in CHANGELOG.txt and src/changes/changes.xml

Commit the changes.

Run an integration test on Travis-CI

Submit a pull request.

6.1.3 New Implementations

The project start with some adapters implementations over most used open source prolog engines.

We accept any new adapter implementation of another prolog engine not covered at this moment.

For this propose create a new GitHub source code repository naming this follow the project convesion:

prolobjectlink-jpi- *new engine implementation name*

Create an new maven project in your preferred IDE named like repository.

Copy the src/assembly/dist.xml descriptor

Copy the src/build/filters folder and change by your console main entry point

Copy and clean src/changes/changes.xml to go reporting every change

Copy src/site folder to generate a similar project site.

Copy the pom.xml properties, build, report, etc... from another implementation

Change the project information.

Add your dependencies including Java Prolog Interface API

```
<repositories>
        <repository>
                <id>ossrh</id>
                <name>Sonatype Nexus Snapshots</name>
                <url>https://oss.sonatype.org/content/repositories/snapshots</url>
                <releases>
                        <enabled>false</enabled>
                </releases>
                <snapshots>
                        <enabled>true</enabled>
                </snapshots>
        </repository>
</repositories>
...
<dependencies>
        ...
        <dependency>
                <groupId>org.prolobjectlink</groupId>
                <artifactId>prolobjectlink-jpi</artifactId>
                <version>[1.0.0, )</version>
        </dependency>
        ...
        <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>[4.10, )</version>
                <scope>test</scope>
        </dependency>
        ...
</dependencies>
```

In test package copy the unit-tests cases from another implementation to develop in test driven mode.

We suggest like adapter implementation order begin with data types, parsers, engine and finally query.

Run all test to see if the system to see if your implementation pass all.

Create unit-tests and make sure that the include changes are covered to 100%

Create the benchmark to see if the system performance.

Add a description of your changes in CHANGELOG.txt and src/changes/changes.xml

Commit the changes.

Run an integration test on Travis-CI or another CI system

6.1.4 Version Numbering

Prolobjectlink version signature is Major.Minor.Micro.

Major version is change when the API compatibility is broken. Minor version is change when a new feature is include in the release. Micro version is change when some bug is fixed or some maintenance take place

Prolobjectlink suggest work over the started 1.Y.Z version to preserve compatibility all the time. You are free of make any change adding new features, fixing bugs or code maintenance.

6.1.5 Contact us

Please contact us at our project mailing list https://groups.google.com/group/prolobjectlink to debat over project evolution

Thanks for contributing to Prolobjectlink!

7 Related Works

Paragraph 1, line 1. Paragraph 1, line 2.

Paragraph 2, line 1. Paragraph 2, line 2.

7.1 Section title

7.1.1 Sub-section title

7.1.1.1 Sub-sub-section title

7.Sub-sub-sub-section title

7.Sub-sub-sub-sub-section title

- List item 1.
- List item 2.
  Paragraph contained in list item 2.
  - Sub-list item 1.
  - Sub-list item 2.
- List item 3. Force end of list:

```
Verbatim text not contained in list item 3
```

1. Numbered item 1.
   A. Numbered item A.
   B. Numbered item B.
2. Numbered item 2.

List numbering schemes: [[1]], [[a]], [[A]], [[i]], [[I]].

Defined term 1

of definition list.

Defined term 2

of definition list.

```
Verbatim text
                        in a box
```

--- instead of +-- suppresses the box around verbatim text.

Figure caption

Centered cell 1,1	Left-aligned cell 1,2	Right-aligned cell 1,3
cell 2,1	cell 2,2	cell 2,3

Table caption

No grid, no caption:

cell	cell
cell	cell

Horizontal line:

7.2 ^L New page.

Italic font. **Bold** font. `Monospaced` font.

Anchor. Link to anchor. Link to http://www.pixware.fr. Link to showing alternate text. Link to Pixware home page.

Force line
break.

Non breaking space.

Escaped special characters: ~, =, -, +, *, [,], <, >, {, }, \.

Copyright symbol: ©, ©, ©.

8 FAQ

8.1 Frequently Asked Questions

General

1. Why Java Prolog Interface?
2. How can use Java Prolog Interface?
3. How include Java Prolog Interface into Maven project?

8.2 General

Why Java Prolog Interface?

Blah, Blah, …

[top]

How can use Java Prolog Interface?

You can use Java Prolog Interface following these steps:

- Step One
- Step Two
- Step Three

[top]

How include Java Prolog Interface into Maven project?

Blah, Blah, …

```
...
    <dependencies>
       <dependency>
         <groupId>io.github.prolobjectlink</groupId>
         <artifactId>prolobjectlink-jpi</artifactId>
         <version>1.0</version>
       </dependency>
    </dependencies>
     ...
```

Blah, Blah, …

[top]