Prolobjectlink Projects

prolobjectlink-jpi-jlog

v. 1.2-SNAPSHOT

User Guide

Prolobjectlink Project **2023-01-07**

Table of Contents

1. Table of Contents i
2. What Is 1
3. Getting Started 3
4. Prolog Programming 5
5. Bidirectional Interface 6
6. Development Tools 15
7. Contribution 17
8. Related Works 20
9. FAQ 22

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

Blah, Blah, …

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 Prolog Language

The Prolog belongs to the declarative programming languages paradigm. the Prolog programs are based on valid conclusion over facts. Prolog is a LP language specially indicated to modeling problems that imply objects and relations among objects. Prolog have simple syntax, is weakness typed, have simples data structures, support recursion and is extensible. One Prolog program have two basic components or clause, facts and rules.

All Prolog programs are a succession of clauses where every clause end with dot character. They are two kind of clauses, Facts and Rules. Every Prolog clause have two components, one head and one body. Facts are the case where the clause only have head component but don't have body and they are used to indicate things unconditionally true about some domain. Rule have head and body and both are separate by neck operator (:-). Rules are used to indicate things conditionally true about some domain. Neck operator semantic denote that one condition should be satisfy by the clause head to be true. The clause body specify the condition order to verify for the conclusion or head.

4 Bidirectional Interface

4.1 Bidirectional Interface

4.1.1 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.

4.1.2 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());
                }
        }
```

4.1.3 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.

4.1.4 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.

4.1.5 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);
```

4.1.6 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.

4.1.7 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.newSt
                 while (query.hasNext()) {
                        PrologTerm value =                                          que
                        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.quer
                         for (Collection<PrologTerm> col : query) {
                                for (PrologTerm prologTerm : col) {
                                        System.out.println(prologTerm);
                                }
                         }
                         query.dispose();
                         engine.dispose();
                }
        }
```

4.1.8 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();
```

4.1.9 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.

4.1.10 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).
```

4.1.11 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);
        }
}
```

4.1.12 Getting started Prolog to Java

This page describes a BSF (Bean Scripting Framework) engine for JLog, a Prolog-in-Java system. JLog is a full-featured Prolog interpreter that can be run as an applet, an application or embedded through an API. You can download the full package, which includes JLog-1.3.6, at JLog-1.3.6-ulf.zip. It's licensed under the GPL.

BSF enables a Java host program to call scripts or programs written in other languages in a language-neutral way. That means that a BSF-enabled application can a) call scripts and programs written in other languages without knowing in advance in which language they might be (embedding), and b) that any language for which a BSF engine is available can be used to script a BSF-enabled Java application (scripting). Currently, BSF integration is available for JavaScript?, XSLT, Jython, Python, Ruby, ObjectScript?, NetRexx?, TCL, Groovy and now for Prolog. BSF has been released under the Apache License and can be found at Apache Commons BSF. It's also included in the download above.

The JLog/BSF integration library (or BSF engine) was developed by myself. JLog was developed by Glendon Holst, and can be found at JLog. It's also licensed under the GPL.

4.1.12.1 Predicates

JLogBSFEngine? defines a number of Prolog predicates that can be used for interactions between both languages.

```
bsf_register (Name, Bean)
```

Stores the object in variable Bean under the given name in the registry. Name must be a string or a variable bound to a string. Bean must be a bound variable.

```
bsf_lookup (Name, ResultVar)
```

Looks up the object of this name in the registry, and binds it to the variable ResultVar. Name must be a string or a variable bound to a string.

```
bsf_unregister (Name)
```

Removes the object with this name from the registry. Name must be a string or a variable bound to a string.

```
bsf_import (Package)
```

Adds the given package name to the lists of imports, so that all Java objects in that package can be referenced just by their classname, instead of by their fully qualified name. java.lang is imported automatically. Package must be a string or a variable bound to a string.

```
bsf_static (Class, ResultVar)
```

Retrieves a class and stores it in variable ResultVar, so that static methods can be invoked on it (e.g. Integer.valueOf). Can also be used to retrieve static fields of classes (e.g. java.lang.System.out). Class must be a string or a variable bound to a string.

```
bsf_create (ResultVar, Class, Parameters [, Types])
```

Prolog wrapper for a Java constructor. The resulting object of class Class is put in the variable ResultVar. Parameters is the list of parameters. The Java equivalent would be ResultVar = new Class(Parameters). bsf_create/3 (w/o the Types parameter) tries to figure out the correct method signature based on the types of the given parameters. This works in most cases, but not always. If it can't decide which constructor to use, an exception is thrown that advises to use bsf_create/4 instead. That means that the list of parameters is needed to select the correct one.

```
bsf_invoke (ResultVar, Bean, Method, Parameters [, Types])
```

Prolog wrapper for a Java method invocation. Its Java equivalent would be ResultVar = Bean.Method(Parameters). bsf_invoke/4 (w/o the Types parameter) tries to figure out the correct method signature based on the types of the given parameters. This works in most cases, but not always. If it can#t decide which method to use, an exception is thrown that advises to use bsf_invoke/5 instead. That means that the list of parameters is needed to select the correct one.

```
bsf_addevent (Bean, Action, Script)
```

This causes the Prolog code in Script to be executed whenever the Java object Bean fires an Action event. The ScriptedUI example demonstrates this. Bean must be bound to a Java object. Action and Script must be strings or bound to a string.

```
bsf_j2p (Object, Term)
```

Converts Object -which must be bound to a Java object- to a Prolog term.

```
bsf_p2j (Term, Object)
```

Converts Term -which must be bound to a Prolog term- to a Java object.

4.1.12.2 Examples

This example is use to get the current time from 'java.util.Date' instance.

```
:- load_library('./bsf').

uptime(X) :-
        bsf_create(Date, 'java.util.Date', []),
        bsf_invoke(L, Date, 'getTime', []),
        bsf_create(Long, 'java.lang.Long', [L]),
        bsf_invoke(O, Long, 'intValue', []),
        bsf_j2p(O, X).
```

It prints a table of Fahrenheit and Celsius temperatures. This is not so much a practicable example, but rather a demonstration that wherever BSF is used, Prolog code can run.

```
:- load_library('./bsf').

f2c(Start, End) :-
        Start =< End,
        bsf_lookup('out', OUT),
        bsf_static('Math', MATH),
        bsf_invoke(_, OUT, 'print', [Start]),
        T is (Start-32) * 5/9,
        bsf_invoke(T1, MATH, 'round', [T]),
        bsf_invoke(_, OUT, 'print', [T1]),
        Start1 is Start + 10,
        f2c(Start1, End).
f2c(30, 100).
```

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-jlog 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]