To appear in Logics in Artificial Intelligence, 9th European Conference, JELIA'04 ©Springer Verlag 2004

InterProlog: towards a declarative embedding of logic programming in Java

Miguel Calejo

Declarativa
Rua da Cerca 88, Porto, Portugal
mc@declarativa.com http://www.declarativa.com/interprolog

Abstract. InterProlog is the first Prolog-Java interface to support multiple Prolog systems through the same API; currently XSB and SWI Prolog, with GNU Prolog and YAP under development – on Windows, Linux and Mac OS X. It promotes coarse-grained integration between logic and object-oriented layers, by providing the ability to bidirectionally map any class data structure to a Prolog term; integration is done either through the Java Native Interface or TCP/IP sockets. It is proposed as a first step towards a common standard Java + Prolog API, gifting the Java developer with the best inference engines, and the logic programmer with simple access to *the* mainstream object-oriented platform.

1 Introduction

InterProlog (http://www.declarativa.com/interprolog) is an open source library for developing Java + Prolog applications. It's been introduced elsewhere [1,2,3], and in addition to academic use it supported the development of a substancial Java GUI system for Prolog tools [7]. Whereas some Java - Prolog interfaces taste like objectified versions of the underlying Prolog/C interfaces, requiring explicit building of term structures prior to querying, InterProlog provides a higher-level API directly mapping Java objects to Prolog terms, inducing a more concise and declarative programming style.

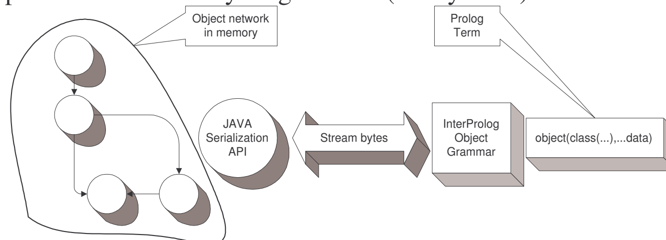
This short paper introduces its new multiple Prolog implementation support, intended to provide a common API for bridging the most relevant representatives of the object-oriented and logic programming paradigms. As of writing, InterProlog supports XSB and SWI Prolog. When you read this it may already support also GNU Prolog and YAP. InterProlog is the only Java-Prolog interface API supporting more than one Prolog implementation. Lack of space prevents a comparison with other systems, but a list (with some comments) can be found in [5].

Linking Java and Prolog is relevant both for the industry and academia fields: to the first because "real-world" applications demand full-blown real logic engines, such as those produced by the logic programming community over the last decades, instead of toy engines or inferior technology; and to the second too, because reusing Java's GUI infrastructure and other functionality liberates the logic programming commu-

nity from wasting resources into condemned "Prolog driven" ecosystems. Prolog's survival is in large part dependent on the simplicity of its embedding into Java and other "real world" language environments.
We'll next review the overall functionality of InterProlog with some examples, and conclude with future plans and room for collaboration.

2 The InterProlog System

InterProlog is middleware for Java and Prolog, providing method/predicate calling between both, either through the Java Native Interface or sockets; the functionality is basically the same in both cases. InterProlog's innovation to this problem is its mapping between (serialized) Java objects and their Prolog specifications, propelled by the Java Serialization API which does most of the work on the Java side; the Prolog side is built upon a DCG that analyses/generates (the bytes of) serialized objects:



In order to support multiple Prologs, two things were done recently:

- The Prolog layer was revised to be compatible with "de facto" ISO Prolog; it now has a small part dedicated to each Prolog system (XSB and SWI; GNU and YAP under development).
- The Java class hierarchy was restructured; each Prolog system has a specific PrologEngine subclass, as well as a subclass of (an abstract class) PrologImplementationPeer, where most system-dependent knowledge is.

To understand InterProlog we'll start from two viewpoints: Java and Prolog.

2.1 Java programming perspective

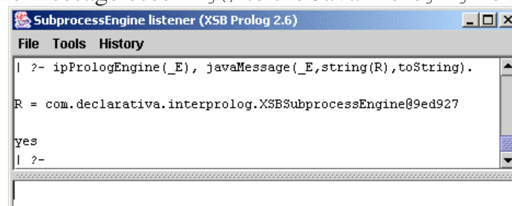
InterProlog brings to the Java developer a simple API to access the power of full blown logic engines. The next fragment allows a Java programmer to use a Prolog file bundled into a jar file, and perform a simple query:

```
PrologEngine engine = new SWISubprocessEngine();
engine.consultFromJar("test.pl");
// or consultRelative (to the class location), or consultAbsolute(File),...
Object[] bindings =
   engine.deterministicGoal("descendent_of( X, someAncestor )","[string(X)]");
if(bindings!=null){// succeeded
  String X = (String)bindings[0];
  System.out.println( "X = " + X);
}
```

The only SWI Prolog dependence is the first line, so by changing it (e.g. `XSBSubprocessEngine`) a different Prolog will be used. Complex structures can be passed in both directions with customized class objects, understoodable on the Prolog side by their InterProlog term specifiers, see [4].

2.2 Prolog programming perspective

The main InterProlog contribution for Prolog programming is the `javaMessage` predicate shown below, but it also provides a simple "Prolog listener" window: a traditional "console" front-end, where it is easy to experiment access to Java. The following invokes the message `toString()` to the Java `PrologEngine` in use:



The next goal causes a window to appear:

```
javaMessage('javax.swing.JFrame',W,'JFrame'(string(myTitle))),
javaMessage(W,C,getContentPane),
javaMessage('javax.swing.JLabel',L,
  'JLabel'(string('Hello Prolog, greetings from Swing:-)'))),
javaMessage(C,add(string('Center'),L)),
javaMessage(W,pack), javaMessage(W,show).
```

The above example illustrates how easy it is to message Java objects (and classes), but is a bit too procedural. Depending on the project at hand, rather than "writing Java constructors in Prolog" as above, it may be best to specify visual hierarchies with Prolog terms that are "interpreted" on the Java side, as in the XJ system [7]; parts of the interface may be populated later by lazily calling Prolog goals, e.g. the term specifies a lazy data structure / GUI fragment (such as when visually browsing a large Prolog structure).

A simplified variant of this principle can be experienced with the `browseTerm` term visualizer bundled in InterProlog; a Prolog term acts as a complete (eager) specification for a tree of `TermModel` objects on the Java side, which constitute a (Swing) model for a `JTree` (tree visualization) widget, see [5].

These approaches encourage a more coarse-grained approach to Java+Prolog system development (as opposed to "redoing Java constructors in Prolog"), which is good for performance, debugging and code maintenance.

On to another subject: the next clause allows a Prolog system to call another through Java, by using the `PrologEngine` method `deterministicGoal(TermModel g)`:

```
callAnotherProlog(Engine,G) :- buildTermModel(G,GM),
    javaMessage(Engine,SM,deterministicGoal(GM)), recoverTermModel(SM,G).
```

The goal is transformed in a `TermModel` object tree specification `GM`; on invoking the Java method it materializes as a Java tree, which is them converted to a TermModel specification on the other engine, from which the solution term is recovered to a Java `TermModel` tree, etc. The next XSB goal finds operators defined in XSB and not in SWI:

```
javaMessage('com.declarativa.interprolog.SWISubprocessEngine',
  SWI,'SWISubprocessEngine'),
callAnotherProlog(SWI, findall(op(P,T,Name), current_op(P,T,Name), SWIops)),
findall(op(XSBP,XSBT,XSBO),
  ( current_op(XSBP,XSBT,XSBO), not(member(op(XSBP,XSBT,XSBO),SWIops) ),
  XSBonly).
```

3. Conclusion

We've reviewed the first Java interface API to support multiple Prolog implementations, and thus a candidate to evolve into a standard Prolog/Java API. Future work:

- Support for more engines; work has started on GNU and YAP.
- Provide javax.rules [6] compliance as added value for supported Prologs; a preliminary analysis suggests that javax.rules concepts map largely into InterProlog concepts.
- Use Prolog threads; currently InterProlog supports Java multiple threads for deterministic goals. Prolog threads will allow (a) multiple solution support and (b) light (multiple) engine creation, e.g. for server applications.
- Optimization of some call patterns. The serialization-based primitives provide maximum flexibility. But it may be the case that, with more applications being developed, a need arises to speed beyond the 3 mS/call currently measurable on a typical PC; thus being the case, specialized treatment of some call patterns can be tuned to avoid (generic) serialization.

References

1. Calejo, M.: InterProlog, a declarative Java-Prolog interface, in Procs. Logic Programming for Artificial Intelligence and Information Systems (thematic Workshop of the 10th Portuguese Conference on Artificial Intelligence), Porto, December 2001
2. Calejo, M.: InterProlog: a simple yet powerful Java/Prolog interface, Computational Logic Magazine, Dec 1998, http://www.cs.ucy.ac.cy/compulog/dec98update/projects/interprolog.htm
3. Calejo, M.: Java+Prolog: A land of opportunities, in Procs. The First International Conference on The Practical Application of Constraint Technologies and Logic Programming, ISBN 1 902426 01 0, London 1999
4. Declarativa: Java+Prolog Systems, http://www.declarativa.com/interprolog/systems.htm
5. Declarativa: Prolog API, http://www.declarativa.com/interprolog/systems.htm
6. Toussaint, A. et. al.: JSR 94: JavaTM Rule Engine API, http://www.jcp.org/en/jsr/detail?id=094, June 6 2004
7. XSB, Inc.: XJ Platform, http://www.xsb.com/techPlatforms.html, June 6 2004