PPI- A Portable PROLOG Interface for JAVA

Ludwig Ostermayer, Frank Flederer, Dietmar Seipel

University of Würzburg, Department of Computer Science
Am Hubland, D – 97074 Würzburg, Germany
{ludwig.ostermayer,dietmar.seipel}@uni-wuerzburg.de

Abstract. As a first step to combine the two programming paradigms – object-oriented programming and logic programming – we have introduced a generic default mapping for JAVA objects and PROLOG terms. This mapping can be used without any modification to the JAVA classes that stand behind the objects. We also can generate automatically JAVA classes from predicates in PROLOG that map to each other. Apart from the default mapping it is further possible to customise the mapping by JAVA annotations. This allows for different Prolog-Views on a given class in JAVA. The data exchange format between JAVA and PROLOG is a simple textual representation of the JAVA objects and of the terms in PROLOG. Because this textual representation already conforms PROLOG's syntax, it can be directly used within PROLOG.

In a second step we have to develop the link between JAVA and PROLOG that executes the mapping and communication. We already have presented a connector architecture for PROLOG and JAVA and an example interface that successfully uses our object term mapping with high performance. However, this interface depends heavily on a single PROLOG implementation: SWI-PROLOG. But as we have a generic mapping between JAVA objects and PROLOG terms, we also strive for an interface that also is generic and can be used independently of the PROLOG implementation.

In this paper, we present the *Portable Prolog Interface* (`PPI`) for JAVA that uses the standard streams `stdin`, `stdout` and `stderr` to communicate with a PROLOG instance. Because these standard streams are available for all popular operating systems and are used by most of the PROLOG implementations for the user interaction, the `PPI` works for a broad range of PROLOG implementations. We evaluate our new generic interface `PPI` with different PROLOG engines and without changing the underlying JAVA or PROLOG source code of our tests.

Keywords. Multi-Paradigm Programming, Logic Programming, Prolog, Java.

1 Introduction

The object-oriented software engineering concept is one of the most used in the field of software development. However, there are other programming paradigms which are eminently suitable for particular problem domains. One of those programming concepts is the logic programming paradigm. The best known logical programming language is PROLOG. PROLOG programs consist of a collection of rules that describe horn clauses. In PROLOG programs, these rules are used by the inference mechanism to find solutions for which the rules are true. Due to the simple definition of those rules, PROLOG is eligible, for instance, for a simple definition of business rules.

It is desirable to combine different programming paradigms in order to use the strength of each individual concept for suitable parts in a piece of software. Several approaches for an interaction between JAVA and PROLOG have been proposed in the past. But most of those concepts are specialized to specific PROLOG implementations. The benefit of such a strong binding to a PROLOG implementation leads to a good performance, but it lacks of portability across several PROLOG implementations. A well known interface between JAVA and PROLOG is JPL. JPL, however, makes use of the foreign language interface (FLI) of SWI-PROLOG. Because of this strong binding to SWI-PROLOG it is not easily usable for other PROLOG implementations as XSB- or YAP-PROLOG. Another interface for PROLOG and JAVA is Interprolog. Also this interface is heavily dependent of a single PROLOG implementation: XSB-PROLOG. Although support for SWI- and YAP-PROLOG are announced, they have to put a lot of effort into porting Interprolog to those other PROLOG implementations. There are other implementations of interfaces between JAVA and PROLOG that attach importance to the portability regarding PROLOG implementations. One of those interfaces is JPC [5]. But also for this interface much porting effort has to be expended.

The main contribution of this paper is for our connector a new generic *Portable Prolog Interface* (`PPI`) which uses our object term mapping [14] for a smooth communication between JAVA and PROLOG. The `PPI` extends our connector architecture for PROLOG and JAVA as presented in [15] and allows the usage of the connector with several PROLOG implementations and operating systems. The `PPI` uses the standard streams `stdin`, `stdout` and `stderr` to communicate with a PROLOG instance. These standard streams are part of the operating systems and are also used by most of the PROLOG implementations for the interaction with users.

The rest of the paper is structured as follows: In Section 2 we discuss work that is related to the results presented in this paper. In Section 3 we recap our object term mapping and how it is realised for JAVA and PROLOG. This is followed by the presentation of the `PPI` in Section 4 which we evaluate in Section 5. Finally in Section 6, we conclude and discuss future work.

2 Related Work

The results in this paper are the continuation of our work with a portable connector architecture for JAVA and PROLOG that allows a smooth communication between both programming languages.

In [13], we have presented the framework PBR4J (PROLOG Business Rules for JAVA) that allows to request a given set of PROLOG rules from a JAVA application. To overcome the interoperability problems, a JAVA archive (JAR) has been generated containing methods to query the set of PROLOG rules. PBR4J uses XML Schema to describe the data exchange format. From the XML Schema description, we have generated JAVA classes for the JAVA archive. For our connector the mapping information for JAVA objects and PROLOG terms is not saved to an intermediate, external layer. It is part of the JAVA class we want to map and though we can get rid of the XML Schema as used in

PBR4J. Either the mapping is given indirectly by the structure of the class or directly by annotations. While PBR4J just provides with every JAR only a single PROLOG query, we are now able to use every object as goal in PROLOG and depending on which variables are bound different queries are possible. PBR4J transmitted along with a request facts in form of a knowledge base. The result of the request was encapsulated in a result set. With our connector we do not need any more wrapper classes for a knowledge base and the result set as it was with PBR4J. That means that we have to write less code in JAVA. We either assert facts from a file or persist objects with JAVA methods directly to PROLOG's database.

The generic mapping mechanism as described in Section 3 was introduced first in [14]. Using the default mechanism, nearly every class in JAVA can be mapped to a PROLOG term without any modification to the class' source code. Additionally, if a customised mapping is needed, we provide JAVA annotations in order to realise the modification easily. These annotations do not affect the original program code in JAVA. Apart from the mapping, we have proposed the Prolog-View-Notation (PVN) to describe existing PROLOG terms and to create JAVA classes that map under the default mapping to the described terms. Because there are nested references of different JAVA objects, the mapping is done recursively. However, we observed a mapping anomaly that we call Reference Cycle. A Reference Cycle occurs if a JAVA object is referenced by itself. Using our mapping mechanism as proposed this leads to a cyclic term. We want to avoid cyclic terms because our mapping mechanism in this case leads to infinite long strings in JAVA. We have proposed a solution that replaces an attribute which is a member of the Reference Cycle by a list that contains all referenced objects. If any of those objects is referenced, a reference identifier is used instead of the nested term representations. Because the list is restricted to the referenced objects, we avoid the cyclic term problem and are able to map objects that have a self-reference.

In [15] we have introduced our general connector architecture for JAVA and PROLOG. The presented implementation of the connector is lightweight. The object term mapping is realised with only two classes. As communication interface we have presented the Prolog-Interface (`PI`) for SWI-PROLOG. The `PI` uses the Foreign Language Interface (FLI) of SWI-PROLOG and is therefore only applicable for this single PROLOG implementation. An evaluation of the performance has shown that the `PI` in combination with our object term mapping is as fast as JPL [16], a highly optimized JAVA interface for SWI-PROLOG. However, the implementations for the evaluation with our connector have proven simpler, clearer and shorter as with the reference JPL. We required with our connector 25% less lines of code than with JPL.

There are other approaches how to establish a communication between JAVA and PROLOG like [3,4,5,7,10] that we already have discussed in [14,15]. In contrast to our work, all these approaches are limited to single PROLOG implementations and none of these approaches allow the mapping of already existing classes to terms in PROLOG, especially without any modifications to the underlying source code. The implementation of our connector itself is much more lightweight and programs using our connector need less lines of code than with the other approaches.

3 The Object Term Mapping between JAVA and PROLOG

In [14] we have proposed a customisable mapping between JAVA objects and PROLOG terms. The mapping provides a default mapping of JAVA classes to PROLOG terms. This makes it possible to use almost any already existing JAVA class without any modification to the classes in JAVA. Thus, a JAVA developer can make use of PROLOG functionalities with minimal effort. If the default mapping of JAVA objects to PROLOG terms does not match already existing PROLOG predicates or any needed data structure on the PROLOG side, we also have proposed a customisable mapping of JAVA objects to PROLOG terms. This customisation allows the user to change the functor as well as the composition and the type of a term's arguments. In this section we want to recap the default and the customisable mapping of JAVA objects to PROLOG terms with some examples.

3.1 Default Mapping

To provide JAVA developers an easy way to use PROLOG, our approach implements a smart default mapping. For the default mapping JAVA classes can be used without any modifications. The JAVA programmer does not need to know the syntax and only little of the functionalities in PROLOG in order to establish a connection.

The mapping target of an object is a term in PROLOG, also referred as target term in this paper. The default mapping from a JAVA object to a PROLOG term uses the object's class name as the target term's functor. Class names in JAVA are usually written in Upper Camel Case notation. But upper first characters in predicate names are not allowed in PROLOG because functors are atoms. Atoms in PROLOG usually begin with lowercase characters, otherwise the predicate's name must be escaped by surrounding single quotes, e.g. `'Book'`.

For the default mapping, we have decided to convert the in JAVA common Camel Case notation to the in PROLOG common Snake Case notation. This is done, by replacing uppercase characters by their lowercase equivalent and add an underscore prefix, if the character is not the first one. For instance, the class' name `MyBook` maps to the atom `my_book`.

Classes usually contain member variables. The default mapping just maps every member variable to an argument of the target term. This is done, by getting all these variables via Java Reflections. The order of the arguments in the target term is given by the `getDeclaredFields()` method in JAVA. According to JavaDoc, there is no assured order but Oracle's JVM (JAVA Virtual Machine) returns an array of fields that is sorted by the position of the variables' declarations in the JAVA class files.

Another aspect of our default mapping is the fix conversion of some types in JAVA to certain types/structures in PROLOG. A natural mapping of JAVA types (to PROLOG) is as follows: short (integer), int (integer), long (integer), float (float), double (float), String (atom), Array (PROLOG list), List (PROLOG list) and Object (compound PROLOG term). For other data types, only existing in certain PROLOG implementations like string in SWI-PROLOG [18], the default mapping can be further extended or changed. It is possible to save these changes to the default mapping to a configuration file.

A special part in logical programming are logical variables. The inference mechanism of PROLOG tries to assign valid values to them for which the rules of the PROLOG program are true. Because different values for a logical variable might be true, several solutions can be found for a single request made available through backtracking.

The inference mechanism and the unification of logical variables to valid values is a strong feature of PROLOG. In order to make this unification process available in JAVA we have introduced the concept of *Object Unification*.

When transforming JAVA objects to PROLOG terms, we transform specific variables of an object into a logical variable in PROLOG. We map `null` values in JAVA to variables in PROLOG. More precise, every time we map an object to PROLOG, all member variables with a `null` value are substituted in PROLOG with different variables. Then, these variables can be unified within the inference process of PROLOG. Finally, a in PROLOG unified term leads to a substitution of the initial `null` values in JAVA by the values of the in PROLOG unified variables.

To illustrate the default mapping mechanism, different implementations of a `Book` class follow. In each step, further details are implemented in order to show another detail of the default mapping mechanism. The first `Book` class does not implement any member variable at all:

```
class Book {
}
```

When we create a instance of the `Book` class and transform it via the default mapping to a term in PROLOG, any instance will result in the same term with an arity of zero:

```
book
```

The name of the class `Book` is transformed to a snake case notation which in this case just leads to the lowercase `book`.

We extend the `Book` class of the previous implementation by the `title` of the book:

```
class Book {
  private String title;
  // ... constructor\ getter\ setter
}
```

For lack of space, we omit the implementation details of a class' constructor, getter and setter methods. Now, we create again an instance:

```
Book b = new Book();
b.setTitle("Sophie's World");
```

The target term in PROLOG then looks like in the following listing:

```
book('Sophie\'s World')
```

The single member variable `title` is used for the single argument of the book term. Because the member variable is of the data type String in JAVA, it is transformed by the default mapping into an atom in PROLOG.

In the next step we extend the `Book` class by another member variable, the amount of pages. This time, we use the `int` data type in JAVA:

```
class Book {
  private String title;
  private int pages;
  // ... constructor\ getter\ setter
}
```

We create again an instance of the `Book` class:

```
Book b = new Book();
b.setTitle("Sophie's World");
b.setPages(518);
```

The instance `b` of `Book` then is transformed via the default mapping to:

```
book('Sophie\'s World', 518)
```

The resulting term in PROLOG contains the values of both member variables, because the default mapping just transforms all of the member variables of a JAVA class. The pages variable, however, is not set within quotes, as the default mapping transformes a JAVA int to a integer in PROLOG. The order of the two member variables within the PROLOG term is defined by the return of the `getDeclaredFields()` method of the JAVA reflection API. In this case, the sorting of the member variables is the declaration order of them within the `Book` class.

We give now an example where one of the member variables is set to `null`. For this, we instantiate a `Book` object and do not set the pages member variable:

```
Book b = new Book();
b.setTitle("Sophie's World");
```

Not setting the pages amount leads to an uninitialized variable that is set to `null`. According to the default mapping, all member variables that have a value of `null` are transformed to variables in PROLOG:

```
book('Sophie\'s World', Book@123_pages)
```

The name of the logical variable is composed of the object's reference in JAVA (`Book@123`) and the name of the member variable (`pages`). Generating the name of logical variables this way we obtain an unique variable name. Its uniqueness results from the unique object reference within a JAVA program and the unique name of the member variable within a JAVA class. Even if the same JAVA object is transformed multiple times to PROLOG, it is sufficient. Because we need just one single unification for a member variable of an object, the multiple occurrence of a member variable of the same object, leads to a single unification on the PROLOG side.

But we are not limited by transforming a single JAVA object to PROLOG. Referenced objects are transformed recursively. To show this, we extend the book class again and create a new class named `Author`:

```
class Book {
  private String title;
  private Author author;
  private int pages;
  // ... constructor\ getter\ setter
}

class Author {
  private String name;
  // ... constructor\ getter\ setters
}
```

As one can see, we now have a reference from the `Book` class to the `Author` class.

```
Author a = new Author();
a.setName("Jostein Gaarder");

Book b = new Book();
b.setTitle("Sophie's World");
b.setAuthor(a);
```

The target term in PROLOG of the book `b` now is a complex compound term:

```
book('Sophie\'s World', author('Jostein Gaarder'),
   Book@123_pages)
```

The `Author` class just contains a single member variable, so does the resulting target term in PROLOG. The resulting term for the author object `a` is contained as argument within the target term for the book `b`.

3.2 Customised Mapping

If the default mapping does not map an object to a desired term structure, the user is able to modify the mapping with a special purpose annotation layer in JAVA. With JAVA annotations we can add the necessary meta-data of a desired mapping to the source code in JAVA. Note, that annotations are not part of a JAVA program, i.e. they do usually not affect the code itself they annotate. Annotations are parsed in JAVA with the methods of the Reflection API. To customise the mapping between objects and terms we only need three annotations in a nested way: `@PlView`, `@Arg` and `@PlViews`.

`@PlView` is used to describe a single *Prolog-View* on a given class in JAVA. With Prolog-View we mean single mapping of a given class in JAVA to term in PROLOG. It is possible to define different Prolog-Views on the same class. We achieve this with different `@PlView` annotations which are collected within a `@PlViews` annotation in the given class. A `@PlView` annotations consists of several elements:

`viewId` is a mandatory element and identifies the Prolog-View. The predicate name normally set by the default mapping can be written over by the element `functor`. There are three remaining elements of a `@PlView` annotation that are lists consisting of strings: `orderArgs`, `ignoreArgs` and `modifyArgs`. These lists are used to manipulate the structure of the target term corresponding to the desired Prolog-View.

`orderArgs` determines which member variable values, defined by their JAVA names, are used within the textual term representation. As the name `orderArgs` suggests the order of members in this list matters. The order of the resulting term arguments corresponds to the order of the member variable names in this list.

ignoreArgs removes one or a few member variables from the default mapping. The user simply can add the names of the ignored member variables in this list instead of writing all the other names into the orderArgs list. As ignoreArgs contains all the missing arguments, there is no order information that describes the order of the arguments left over to the mapping. Therefore, the relative order of the arguments within the default mapping is unaffected. The user is told not to use orderArgs and ignoreArgs together within the same @PlView annotation, as this could lead to anomalies like member variable names that are in both or in none of the two lists. To prevent an accidental wrong use, an exception is raised if both elements are used together within an @PlView annotation. The orderArgs and ignoreArgs are just to define which member variables are considered for the mapping and which not, as well as the order of those parameters.

modifyArgs modifies the mappings of single member variables to arguments of the target terms. It is an array consisting of @Arg annotations.

@Arg has three elements for the modifications: valueOf, type and viewId. As long as there is no @PlArg annotation in an @PlView annotation, the default mapping is applied for all mapped member variables.

valueOf references the name of the member variable whose mapping is going to be manipulated by the @Arg annotation. In a single @PlView annotation only one @Arg annotation is allowed for every member variable referenced by valueOf.

type defines the PROLOG type which will be used within the target term in PROLOG. Options for type are elements of an enumeration representing certain PROLOG types like atom, float, integer or structures like compound term, list. In case of compound term and list, it is possible that again several Prolog-Views in a referenced class exist. Though, the user again can select a Prolog-View for the referenced class via an according viewId. The type compound is always a reference to another object. Because an object can be mapped to a list, the type list can also be a reference.

viewId is used for member variables that reference a class for which different Prolog-Views are defined.

To illustrate the meaning of the different annotations and their arguments, we give an example of a Book class with three different Prolog-Views defined on it by @PlView annotations:

```
@PlViews({
  @PlView(viewId="book1", orderArgs={"title"}),
  @PlView(viewId="book2", ignoreArgs={"author"}),
  @PlView(viewId="book3", functor="tome",
    modifyArgs={@PlArg(valueOf="pages", type=ATOM)}
  )}
)
class Book {
  private String title;
  private Author author;
  private int pages;

  // ... setters / getters
}
```

The first Prolog-View, identified by `book1`, just selects the member variable `title` to be part of the resulting PROLOG term. Therefore, the resulting term just contains the title of the book.

The second Prolog-View `book2` uses the selection list `ignoreArgs`. The only entry in this list is the member variable `author`. That means all the other member variables, namely `title` and `pages`, are still contained in the resulting target term in PROLOG.

The last Prolog-View `book3` has no restriction on the included member variables at all. Thus, all member variables are mapped to PROLOG. However, two modifications to the mapping are done within the annotation: the `functor` of the target term is changed from `book` to `tome`; the `type` in PROLOG of the mapping of the member variable `pages` is changed in PROLOG from integer as in the default mapping to atom. Therefore, the resulting term in PROLOG has a single quoted value for the pages of the book.

The resulting three target terms in PROLOG are summarised in the following listing:

```
book('Sophie\'s World').
book('Sophie\'s World', 518).
tome('Sophie\'s World', author('Jostein Gaarder'), '518').
```

3.3 Creating Textual Term Representations

All the information needed for the creation of textual term representations can be derived from the classes involved in the mapping. The default mapping uses the information of the class structure itself. The customised mapping uses the information contained in the JAVA annotations `@PlView` that are identified by the string `viewId`. As in [15] shown the object to term conversion as well as the parsing is implemented in a wrapper class called `OTT` (Object-Term-Transformer). An example for the usage of

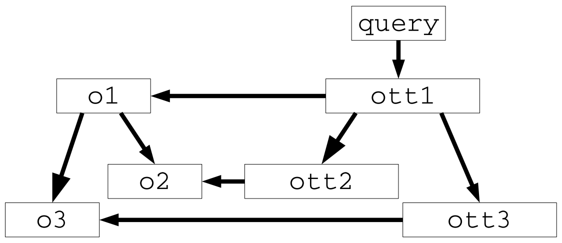

Fig. 1: A tree of `OTT` Objects

`OTT` instances is shown in Figure 1. The object `o1` is destined to be unified in PROLOG. It has references to two other objects `o2` and `o3` which lead to a nested term structure in PROLOG. The class `Query` is a wrapper for a call to PROLOG. To its constructor `o1` is passed and an instance of `OTT` is created, here `ott1`. For all the other references in `o1` instances of `OTT` are created in a nested way, namely `ott2` for `o2` and `ott3` for `o3`.

In order to create the textual term representation of `o1`, the instance `query` causes `ott1` to call its `toTerm()` method that triggers a recursive call of `toTerm()` in all involved instances of `OTT`. In doing so, the first operation is to determine which fields have to be mapped. Depending on a requested Prolog-View or the default mapping an array of Field references is created that contains all the needed member variables for the particular view in the corresponding order. The information about the Fields is retrieved with help of the Reflection API in JAVA. The same way, additional information like PROLOG types and `viewIds` for particular member variables are saved within such arrays. As the information of a Prolog-View on a class is solid and does not change with the instances, this field information is just created once and cached for further use. For the creation of the textual term representation, the functor is determined either from a customised `functor` element of an `@PlView` annotation or from the class name in the default case. After that, the Field array is iterated and the string representation for its elements are created. The pattern of those strings depend on the PROLOG type that is defined for a member. If a member is a reference to another object, the `toTerm()` method for the reference is called recursively.

3.4 Parsing Textual Term Representations

After `query` has received the textual representation of the unified term from PROLOG, it is parsed to set the unified values to the appropriate member variables of the JAVA objects involved. The parsing uses again the structure of nested `OTT` objects as shown in Figure 1. The class `OTT` has the method `fromTerm(String term)`. This method splits the passed string into functor and arguments. The string that contains all the arguments is split into single arguments. This is done under consideration of nested term structures. According to the previously generated Field array the arguments are parsed. This parsing happens in dependence of the defined PROLOG type of an argument. For instance, an atom either has single quotes around its value or, if the first character is lowercase, there are no quotes at all. If there is a quote detected, it is removed from the string before assigning it as a value for the appropriate member variable. Assignments for referenced objects in `o1` are derived recursively by calling the `fromTerm(String term)` method of the appropriate instances of `OTT`, in our example `ott2` and `ott3`.

4 PPI - The Portable PROLOG Interface

In a previous paper [15] we already have presented our connector architecture for PROLOG and JAVA. Figure 2 gives an overview of the components and how the connector works. An object is transformed to a PROLOG term in string format via the object term mapping (`OTM`) as described in Section 3. This string is transmitted via a Prolog-Interface (`PI`). Then the string is parsed in PROLOG. Because the string already conforms to PROLOG's syntax, this is an easy task. The resulting term is unified and sent back again in string format which is finally processed by a parser in JAVA. The used `PI`

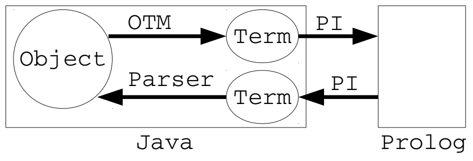

Fig. 2: Components of the Connector

can be any Prolog Interface for JAVA that is available for the considered PROLOG implementation. We already have implemented a high performance `PI` for SWI-PROLOG based on its *Foreign Language Interface*. The combination of our mapping and the `PI` for SWI has been optimized to the point where our connector works as fast as an implementation with JPL [16] which is the standard JAVA interface bundled with SWI. It is more complex to operate with JPL than with our connector. We have this quantified by the amount of lines of code necessary to call PROLOG from JAVA: an implementation with our connector needs 25% less lines of code. In addition, a user developing with JPL must have a deeper understanding for PROLOG and its structures.

However, the `PI` is only applicable with SWI-PROLOG. In order to conserve an independence regarding the PROLOG implementations we introduce in this paper a generic interface suitable for almost every PROLOG implementation and operating systems, the *Portable Prolog Interface* (`PPI`). Instead of a specialized interface between JAVA and a certain PROLOG implementation, we use standard streams of every operating system to connect to a PROLOG process: the standard input (`stdin`), the standard output (`stdout`) and the standard error (`stderr`). Every PROLOG implementation usually provides user interaction via these streams. To write as user a request directly to PROLOG the `stdin` stream is used. The output is channelled via the `stdout` or `stderr` stream to a user interface. The output contains the resulting bindings of the variables that have been unified by PROLOG's inference engine.

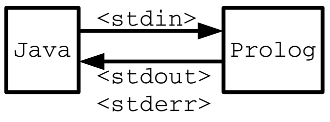

Fig. 3: Structure of the Pipe Interface

Our object term mapping can be combined with the `PPI` in a native way because the textual term representation that we transmit already conforms to PROLOG's syntax. Our connector using the `PPI` now is deployable for a broad range of operating systems and PROLOG implementations. Normally, these streams are used for writing and calling goals as well as for getting the variable bindings of a solution. In addition, the user is able to kick off features in most PROLOG systems like backtracking by typing the character semicolon. This is just an input for `stdin` and therefore our interface is also able to use such meta commands.

Another difference of the PPI and the interface PI in [15] is that the PI returned the unified term as a whole. Using the standard streams PROLOG returns only the binding of the variables, e.g. X=4. In order to make this separate bindings usable for our mapping process, the variables in the initial term are replaced by the appropriate bindings. Fig. 4 shows the schematic flow between the individual pipes that process the information flow. When opening a connection from a JAVA program to a PROLOG engine two classes are initialised in JAVA: OutPipe and InPipe. The class OutPipe shares the main thread of the underlying JAVA program and has the job to write text to PROLOG's stdin. When the unify method is called within the JAVA program, OutPipe writes the the textual term representation, which should be unified, to PROLOG's stdin. The class InPipe runs a separate thread and receives the results from PROLOG by reading the stdout or stderr stream.

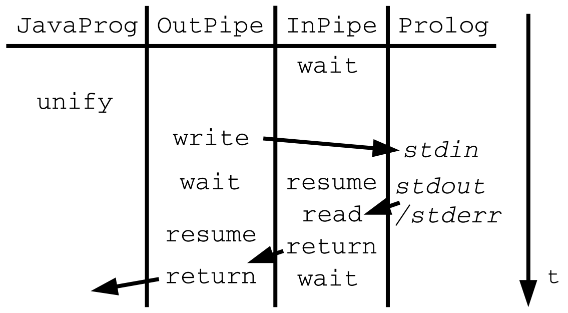

Fig. 4: Information Flow via the PPI

To avoid unnecessary overhead the two threads are paused if they are not needed. Because we want the result as return of the method unify, we pause the calling thread in order to wait for the result from PROLOG. We have sent a request to PROLOG and await the result to be written to PROLOG's stdout or stderr. Therefore, after writing the text to stdin, the InPipe thread is resumed in order to collect the unification result. As soon as the resulting data has arrived via stdout or stderr, the InPipe returns it to OutPipe which resumes its computations and thread of InPipe is paused. After the result, in form of variable bindings, is converted back to the textual representation of the in PROLOG unified term, the resulting string is returned by unify.

5 Evaluation

In this section we evaluate the combination of the generic PPI with our connector architecture for PROLOG and JAVA. We have implemented three tests for the evaluation. For the computations, we have successfully used the following freely available PROLOG implementations: B-, CIAO-, GNU-, SWI-, YAP- and XSB-PROLOG. In doing so, no modifications to the original program files in JAVA and PROLOG have been necessary. We have tested[1] consecutively 50000 calls. The resulting average execution time for the different PROLOG implementations are presented in the tables that follow the short descriptions of the tests. Note, that the resulting execution times in tables always include the time necessary to process the goal on the different PROLOG engines.

In the first test the goal, that we send from JAVA to PROLOG, is just the atom `true` which is always true and needs no unification. We do this, to better estimate the time that only is spent for establishing a connection and the transmission of the data.

B-PROLOG	CIAO	GNU	SWI	XSB	YAP
3.0 sec	15.5 sec	3.0 sec	7.4 sec	3.7 sec	2.9 sec

The second test has two different implementations. The first implementation sends as goal to PROLOG a variable assignment to an atom consisting of 100 characters. The second implementation has an increased character count of 1000 for the assigned atom. The purpose of this test is to analyse the influence of the length of a goal, measured in characters, on the execution time.

Characters	B-PROLOG	CIAO	GNU	SWI	XSB	YAP
100	4.4 sec	15.5 sec	4.2 sec	12.5 sec	5.5 sec	8.6 sec
1000	18.4 sec	33.4 sec	18.5 sec	40.5 sec	18.6 sec	61.7 sec

Third test considers underground railway networks, as in [15] the London Underground. These networks are represented as undirected graphs with stations as nodes and lines as edges connecting the individual stations. In PROLOG this is simply realised via the facts `connected`. The first and the second argument of a `connected` fact is a station. The third argument is the line connecting the two stations. The next listing gives some examples for `connected` facts for the London Underground:

```
connected(station(green_park), station(charing_cross),
  line(jubilee)).
connected(station(bond_street), station(green_park),
  line(jubilee)).
...
```

In this third test we request for a station adjacent to a given station and line. We process the graphs for the London Underground, Sydney and Vienna. The number of edges in these graphs decreases from London with 412 edges over Sydney with 284 edges to Vienna with only 90 edges.

	B-PROLOG	CIAO	GNU	SWI	XSB	YAP
London	7.8 sec	25.5 sec	4.6 sec	15.3 sec	8.5 sec	5.0 sec
Sydney	6.3 sec	22.8 sec	4.4 sec	15.2 sec	7.7 sec	4.9 sec
Vienna	5.2 sec	22.0 sec	3.8 sec	13.7 sec	7.6 sec	4.7 sec

[1] on Core i5 2x2.4 GHz, 6 GB RAM, Ubuntu 14.04

As one can see in the second table the amount of data which is transmitted to and from PROLOG has a huge influence on the execution time. All the times of the execution with 1000 characters are up to 7 times slower than the execution with 100 characters.

In the last table, one can see that the decrement of execution time for the PROLOG inference mechanism has only a slight effect on the complete execution time including the input and output operations. The slowest execution in this evaluation is the 1000 character benchmark for YAP. But 61.7 seconds for 50000 executions means an execution time of 1.234 milliseconds for a single execution. Because in real world applications 50000 executions in a row are unusual, this delay of about one millisecond is still a good value.

6 Conclusions and Future Work

In this paper we have presented the portable PROLOG interface (`PPI`) for our connector architecture. The `PPI` is based on standard streams that are part of nearly every operating system and are used by most PROLOG implementations for the user interaction. In a first evaluation we could verify the applicability of the `PPI` within our connector for several PROLOG implementations, and that with a decent performance. This way, we have improved the portability of our connector architecture for PROLOG and JAVA.

In a future work, we want to extend our tests with the `PPI` to other PROLOG implementations, maybe to commercial PROLOG systems, too. In addition, we currently work on the integration of existing high performance interfaces into our connector. In doing this, we expect to get better execution times for our mapping technique between JAVA and PROLOG.

References

1. A. Amandi, M. Campo, A. Zunino. *JavaLog: a framework-based integration of Java and Prolog for agent-oriented programming.* Computer Languages, Systems & Structures 31.1, 2005. 17-33.
2. M. Banbara, N. Tamura, K. Inoue. *Prolog Cafe: A Prolog to Java Translator.* Proc. Intl. Conference on Applications of Knowledge Management, INAP 2005, Lecture Notes in Artificial Intelligence, Vol. 4369, Springer, 2006. 1-11.
3. M. Calejo. *InterProlog: Towards a Declarative Embedding of Logic Programming in Java.* Proc. Conference on Logics in Artificial Intelligence, 9th European Conference, JELIA, Lisbon, Portugal, 2004.
4. S. Castro, K. Mens, P. Moura. *LogicObjects: Enabling Logic Programming in Java through Linguistic Symbiosis.* Practical Aspects of Declarative Languages. Springer Berlin Heidelberg, 2013. 26-42.
5. S. Castro, K. Mens, P. Moura. *JPC: A Library for Modularising Inter-Language Conversion Concerns between Java and Prolog.* International Workshop on Advanced Software Development Tools and Techniques (WASDeTT), 2013.
6. S. Castro, K. Mens, P. Moura. *Customisable Handling of Java References in Prolog Programs.* arXiv preprint arXiv:1405.2693, 2014.
7. M. Cimadamore, M. Viroli. *A Prolog-oriented extension of Java programming based on generics and annotations.* Proc. 5th international symposium on Principles and practice of programming in Java. ACM, 2007. 197-202.

8. K. Gybels. *SOUL and Smalltalk - Just Married: Evolution of the Interaction Between a Logic and an Object-Oriented Language Towards Symbiosis.* Proc. of the Workshop on Declarative Programming in the Context of Object-Oriented Languages, 2003.
9. M. D'Hondt, K. Gybels, J. Viviane *Seamless Integration of Rule-based Knowledge and Object-oriented Functionality with Linguistic Symbiosis.* Proc. of the 2004 ACM symposium on Applied computing. ACM, 2004.
10. T. Majchrzak, H. Kuchen. *Logic java: combining object-oriented and logic programming.* Functional and Constraint Logic Programming. Springer Berlin Heidelberg, 2011. 122-137.
11. L. Ostermayer, D. Seipel. *Knowledge Engineering for Business Rules in Prolog.* Proc. Workshop on Logic Programming (WLP), 2012.
12. L. Ostermayer, D. Seipel. *Simplifying the Development of Rules Using Domain Specific Languages in Drools.* Proc. Intl. Conf. on Applications of Declarative Programming and Knowledge Management (INAP), 2013.
13. L. Ostermayer, D. Seipel. *A Prolog Framework for Integrating Business Rules into Java Applications.* Proc. 9th Workshop on Knowledge Engineering and Software Engineering (KESE), 2013.
14. L. Ostermayer, F. Flederer, D. Seipel. *A Customisable Mapping between Java Objects and Prolog Terms.* `http://www1.informatik.uni-wuerzburg.de/database/papers/otm_2014.pdf`
15. L. Ostermayer, F. Flederer, D. Seipel. *CAPJa - A Connector Architecture for Prolog and Java.* `http://www1.informatik.uni-wuerzburg.de/pub/ostermayer/paper/capja_2014.html`
16. P. Singleton, F. Dushin, J. Wielemaker. *JPL 3.0: A Bidirectional Prolog/Java Interface.* `http://www.swi-prolog.org/packages/jpl/`
17. J. Wielemaker, T. Schrijvers, T. Markus, L. Torbjörn. *SWI-Prolog.* Theory and Practice of Logic Programming. Cambridge University Press, 2012. 67-96.
18. J. Wielemaker. *SWI Prolog.* `http://www.swi-prolog.org`