CAPJA- A Connector Architecture for PROLOG and 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. Modern software often relies on the combination of several software modules that are developed independently. There are use cases where different software libraries from different programming languages are used, e.g., embedding DLL files in JAVA applications. Even more complex is the case when different programming paradigms are combined like within applications with database connections, for instance PHP and SQL.
Such a diversification of programming languages and modules in just one software application is becoming more and more important, as this leads to a combination of the strengths of different programming paradigms. But not always, the developers are experts in the different programming languages or even in different programming paradigms. So, it is desirable to provide easy to use interfaces that enable the integration of programs from different programming languages and offer access to different programming paradigms.
In this paper we introduce a connector architecture for two programming languages of different paradigms: JAVA as a representative of object oriented programming languages and PROLOG for logic programming. Our approach provides a fast, portable and easy to use communication layer between JAVA and PROLOG. The exchange of information is done via a textual term representation which can be used independently from a deployed PROLOG engine. The proposed connector architecture allows for *Object Unification* on the JAVA side.
We provide an exemplary connector for JAVA and SWI-PROLOG, a well-known PROLOG implementation.

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

1 Introduction

Business applications often are implemented with object oriented techniques. JAVA currently is one of the most used object oriented programming languages with rich libraries and a very active community. There are tools based on JAVA for writing complex rules, but these tools still come with flaws [10]. Logic programming languages like PROLOG are particular suitable to write rules more intuitively and declaratively, which helps in building, updating and testing complex structured sets of rules as we have successfully shown in the field of e-commerce in [9]. Because JAVA is the main programming language in most large-scale applications, it is desirable to connect JAVA with PROLOG for certain problem domains.

Many approaches have been proposed to make PROLOG available in JAVA, but in many cases there is no clear distinction between JAVA and PROLOG as they use PROLOG concepts like terms and atoms directly in JAVA. Our efforts are to keep PROLOG structures off from JAVA, but to enable in JAVA the use of existing PROLOG rules and facts. Therefore we propose a fast, portable and intuitive connector architecture between JAVA and PROLOG.

In our approach, objects can directly be used as PROLOG goals, without creating complex structures in JAVA that represent the terms in PROLOG. Member variables that are equal `null` in JAVA are translated into PROLOG variables. Those variables are unified by PROLOG when querying a JAVA object as a goal in PROLOG. The values, the variables are unified with, are set to the corresponding member variables of the JAVA objects. We call this mechanism *Object Unification*. Apart from using existing JAVA classes for Object Unification, we also provide in PROLOG a generator for JAVA classes. The instances of generated classes unify with terms initially passed to the generator.

The remainder of this paper is organized as follows. In Section 2 we look at related work and compare those concepts with our own approach. Section 3 introduces the components of the proposed connector. We show the mechanics of the object term mapping in Section 3.1 and in Section 3.2 the parsing of PROLOG terms in JAVA. An exemplary interface for JAVA and SWI PROLOG completes the connector architecture in Section 3.3. After that, the workflow with our connector architecture is shown in Section 4 from two viewpoints: from JAVA and from PROLOG. In Section 5 we evaluate our approach and finally discuss future work in Section 6.

2 Related Work

Providing a smooth interaction mechanism for JAVA and PROLOG is a challenging problem that has been studied in several research papers of the last decade.

A well known and mature interface between JAVA and PROLOG is JPL [13]. To enable a fast communication JPL provides JAVA classes that represent directly the structures in PROLOG. This leads to much code for complex PROLOG term structures. Also, it requires that either the JAVA developer knows how to program PROLOG or the PROLOG developer knows how to code JAVA in order to build the necessary structures in JAVA via classes like Compound. Furthermore, it is limited to the use with SWI-PROLOG, as it is shipped and created for just this single PROLOG implementation.

An interesting approach is INTERPROLOG [2] that uses the JAVA serialization mechanism in order to send serialized JAVA objects to PROLOG. These strings are analysed in PROLOG with definite clause grammars and a complex term structure is created which describes the serialized object. However, this generated object term structure is complex and contains a lot of class meta information that is not as natural for a PROLOG programmer as the textual term representations of objects in our approach.

The concepts of linguistic symbiosis have been used in [3, 6, 7] to define a suitable mapping. Methods in JAVA are mapped to queries in PROLOG. This differs from our approach, as we use JAVA objects for terms as well as for queries in PROLOG.

A customisable transformations of JAVA objects to PROLOG terms was introduced with JPC [4]. Instead of using annotations, as it is done in our approach to customise the

mapping, in JPC custom converter classes can be defined. These converters implement methods which define the translation between objects and terms. This causes in a lot of extra code and files as the user has to define the converter classes instead of just writing annotations to existing classes.

In [5] tuProlog, a PROLOG engine entirely written in JAVA, was integrated into JAVA programs by using JAVA annotations and generics. But other than in our approach, PROLOG rules and facts are written directly into the JAVA code within annotations. Querying rules and facts is done again by JAVA methods. The mapping of input and return to arguments of a goal in PROLOG is defined with annotations. In contrast to our attempt, this approach is strongly dependent on tuProlog and therefore not compatible to other PROLOG engines.

In [11], 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 has been generated that contains 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. In our new approach 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 different goals depending on which variables are bound. 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 architecture we do not need any more wrapper classes for the knowledge base and the result set as it was with PBR4J. That means with our new connector we have to write less code in JAVA. We either assert facts from a file or persist objects with a JAVA method to the database of PROLOG.

3 A Connector for PROLOG and JAVA

The connector for PROLOG and JAVA is based on our work with mappings between objects in JAVA and terms in PROLOG. Before we discuss the individual parts of our connector, we recap briefly the highly customisable Object Term Mapping (`OTM`) which we have introduced in [12]. In addition to a simple, yet powerful default mapping for almost every class in JAVA, different mappings between objects and terms also easily can be defined. We call the mapping of an object to a PROLOG term *Prolog-View* on the given object. Multiple Prolog-Views for a single object can be defined. For this purpose, we only need three annotations in JAVA in a nested way as shown in Figure 1. Because JAVA does not support multiple annotations of the same type within a class until version 7, we use the annotation `@PlViews` to allow multiple `@PlView` annotations in a single given class. A `@PlView` is identified by `viewId` and consists of the following elements to customize the mapping of an object to a term. `functor` is used to change the target term's functor. The list `orderArgs` changes the arguments order and the list `ignoreArgs` prevents member variables to be mapped as arguments of the target

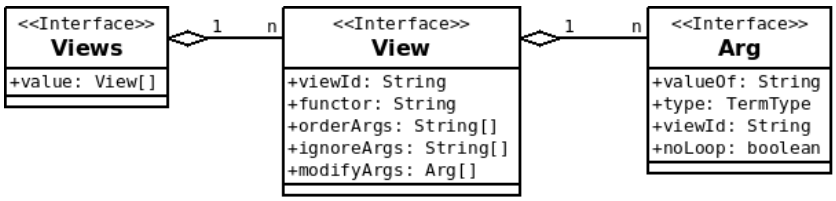

Fig. 1: The Interfaces for `@PlViews`, `@PlView` and `@Arg`

term. The list `modifyArgs` consists of `@Arg` annotations which are used to modify the mapping of a single member variable of the object. The member variable is referenced by `valueOf` and the type in PROLOG can be modified with `type`. If the member variable is a class type that has `@PlView` annotations, a particular Prolog-View can be selected via the appropriate `viewId`. All in all, arbitrary complex nested term structures can be created by the mapping. The following example shows a `Person` class and two different Prolog-Views on `Person`:

```
@PlViews({
  @PlView(viewId="personView1",
    ignoreArgs={"id"},
    modifyArgs=
       {@PlArg(valueOf="children", viewId="personView2")})
  @PlView(viewId="personView2", functor="child",
    orderArgs={"givenName")}
)
class Person {
  private int id;
  private String givenName;
  private String familyName;
  private Person[] children;
  // ... constructor/ getter / setter
}
```

In the listing below instances of `Person` are given followed by the textual term representation under the default mapping and under the Prolog-View `personView1`:

```
Person p1 = new person(1, 'Homer', 'Simpson');
Person p2 = new person(2, 'Bart', 'Simpson');
p1.setChildren(new Person[]{p2});

// default mapping of p1
"person(1,'Homer','Simpson',[person(2,'Bart','Simpson',[])])"
// mapping of p1 under the Prolog-View "personView1"
"person('Homer', 'Simpson', [child('Bart')])"
```

All the information needed for the creation of textual term representations are derived from the classes involved in the mapping. The default mapping uses the information of the classes structure itself. The customised mapping uses the information contained in the annotations `@PlView`.

3.1 Creating Textual Term Representations

We only need two classes in JAVA to request PROLOG as shown in Figure 2. The conversion as well as the parsing is implemented within the wrapper class `OTT` (Object-Term-Transformer). The class `Query` is used to start a call to PROLOG. An example

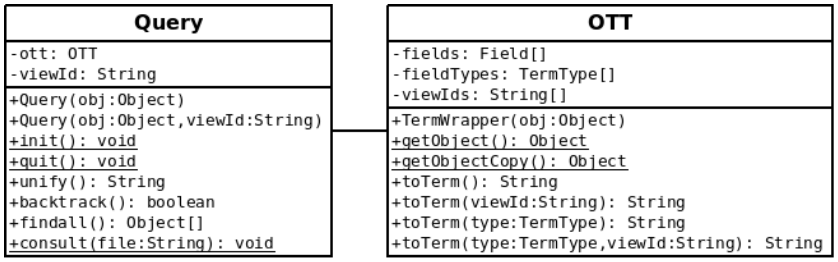

Fig. 2: Classes for CAPJA

for the usage of these classes is shown in Figure 3. The object `o1` is destined to be unified in PROLOG. It has references to two other objects `o2` and `o3` which will lead to a nested term structure in PROLOG. When the instance `query` gets `o1` passed to its constructor, `query` creates an instance of `OTT`, 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`.

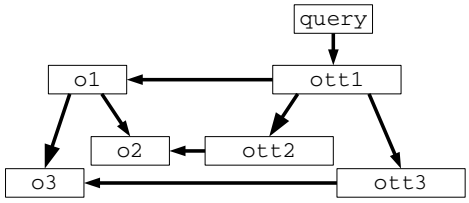

Fig. 3: A Dependency Tree of `OTT` Objects

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. Dependent on the `viewId` of the requested Prolog-View or on 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 about a view of 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.

3.2 Parsing Textual Term Representations

After `query` has received a textual representation of the unified term from PROLOG, it is parsed to set the unified values to the member variables of the appropriate objects in JAVA. The parsing uses again the structure of the nested `OTT` objects as shown in Figure 3. 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 on 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`.

3.3 The Interface for JAVA and SWI-PROLOG

Although the complete mapping process is located in JAVA, we still need an interface to the PROLOG implementation of choice in order to connect both programming languages. The open-source PROLOG implementation SWI-PROLOG [14] comes with the highly specialized, and for SWI optimized, JAVA interface JPL. We have implemented our own JAVA interface for SWI which is optimized for our mapping. Similar to JPL we use SWI's Foreign Language Interface (FLI) and the JAVA Native Interface (JNI). The FLI is bundled with SWI and provides a native interface which can be used to extend SWI by further (meta-)predicates. The FLI also provides an interface for C and is therefore accessible for all other programming languages which have access to C libraries.

We have JAVA on one side and the C interface FLI on the other, so we need the glue to enable the communication between these two worlds. This is done by the JAVA Native Interface (JNI), which enables the usage of in C defined functions in JAVA. With the help of the JNI, we implemented a bridge between JAVA and the SWI-PROLOG system. As mentioned, we focus on the simple transmission of strings that represent terms in PROLOG. This differs from the interface JPL, as our interface does not need complex class structures in JAVA to represent terms in PROLOG. We simply send strings to PROLOG and receives strings from it. The transmitted strings already satisfy PROLOG's syntax and thus can be converted directly into terms on the PROLOG side.

Via the FLI we provide backtracking if there are more solutions. This leads to a return that contains the next unified term in PROLOG. After sending from JAVA a string containing a goal with the logical variable `X`, our interface for SWI-PROLOG returns the

unified term as a string back to our JAVA application. The user on the JAVA side now can call `backtrack()` to send a backtrack command to SWI-PROLOG which returns the next solution.

4 Workflows

We start from two viewpoints: JAVA and PROLOG. Each viewpoint describes the development phase using our connector architecture.

From JAVA The default mapping enables the JAVA developer to use already existing JAVA classes in PROLOG as facts or as goals. If the default mapping of an object in JAVA does not provide a desired term structure in PROLOG, the textual term representation of the object can be altered by using the appropriate `@PlView` annotations. To unify an existing JAVA class the developer just has to wrap it within an instance of `Query` and call its method `unify` in order to call the class' textual term representation as goal in PROLOG:

```
Person p = new Person();
p.setId(1);
Query q = new Query(p, "personView1");
q.unify();
```

The example request to PROLOG above contains an instance `p` of the class `Person` from Section 3. Note, that the only value that is set for `p` is the `id` attribute. The other attributes are not initialized and therefore equal `null`. The class `Query` manages the call to PROLOG. The optional second parameter of the constructor of `Query` defines which Prolog-View is used for the object `p`. It is specified by the `viewId` element of a `@PlView` annotation, here `personView1`. When the method `unify()` is called the textual term representation is created. This is done either according to the default mapping or under the consideration of existing `@PlView` annotations that are defined for the class `Person` or any other referenced classes in `Person`. This string is already a valid term in PROLOG with arguments that represent the attributes of `p` and all referenced objects in `p`. The textual term representation has only arguments for attributes that are mapped as defined by the default mapping or by a referenced `@PlView` annotation. The textual term representation then is used as goal within the PROLOG query.

In the example above, most attributes of `p` are equal to `null` in JAVA. As `null` is a value that can not be transformed into an appropriate type in PROLOG it has to be handled in a particular way. We consider `null` to be in PROLOG a (logical-)variable that is supposed to be unified. After sending a call to PROLOG containing `null` values, the resulting variables are possibly unified in PROLOG. The unified term is sent back as string and parsed in JAVA. Changes to the initial string sent from JAVA to PROLOG will be detected and set to the initial object by JAVA reflections, in our example to the instance `p` of `Person`. This means, those attributes that formerly have been equal to `null` are set to the values of the variables unified in PROLOG. The original object `p` now has been updated and represents the solution of the unification process in PROLOG.

An important feature of PROLOG is unknown to JAVA: the backtracking mechanism. The method `unify` just returns the first solution PROLOG provides. But via backtracking PROLOG is able to provide other unifiable solutions. These solutions can be retrieved with another method of `Query` that is called `backtrack()`. It sends a backtrack request to PROLOG in order to retrieve the next solution, if there is one. The same way a the solution is set to the original object via `unify()`, the solution via `backtrack()` is set to the variables of the original object in JAVA. As it is not sure that there even are other solutions, `backtrack()` returns a boolean in JAVA whether a solution was found by PROLOG or not.

Similar to JPL, we have implemented a third request: get all solutions of a goal in one call. This is called `findall()`, named after the built-in predicate in SWI PROLOG. This method returns an array of the requested objects, e. g. Person. As the method returns multiple objects with different values in their variables, we have to create for each solution a new object. So, when using this method the original object is not touched at all. Creating new objects for every solution is the reason why we need the unifiable objects to have a default constructor in JAVA.

Beside these basic methods for Object Unification there is a method for asserting JAVA objects to the PROLOG database. This method is called `persist()` and just takes the generated string representation of the PROLOG term and asserts it by using the `assertz/1` predicate. After that method call the term representation of the appropriate object is available as fact in PROLOG.

From PROLOG Another viewpoint is the writing of PROLOG terms that are destined for the use in JAVA. In contrast to the previous viewpoint, there are no suitable JAVA classes yet within the current project. So, we show now how is easy it is to write PROLOG libraries that are accessible from JAVA by generated classes.

In [12] we have described a default and a customised mapping between JAVA objects and PROLOG terms. As long as no customisation is defined for a JAVA class via special annotations, a default mapping is applied which links a class to a certain term in PROLOG. With annotations in JAVA the user is able to customise the mapping. These annotations determine which of the member variables will be part of the term representation and which PROLOG type they will be (e. g. ATOM, COMPOUND, LIST). It is possible to define several different views on a class.

We also have introduced in [12] the PVN (*Prolog-View-Notation)* that can be used to define in PROLOG the mapping between JAVA objects and PROLOG terms. Expressions in PVN consist of two predicates: `pl_view` and `pl_arg`. The term `pl_view` describes a textual term representation of a JAVA class. The term `pl_arg` term in a PVN expression is used to define the mapping of the member variables.

A Rule in PROLOG can be made accessible from JAVA using the PVN to describe a rule's head. From this PVN expression we generate JAVA classes with the appropriate `@PlView` annotations. For this purpose we have developed two predicates in PROLOG:

```
create_class_from_pvn(?Pvn, ?Class)
create_annotation_from_pvn(?Pvn, ?Annotation)
```

Typically for PROLOG, both predicates can have the first or the second argument as input. The first predicate generates from a PVN expression source code for JAVA containing all necessary classes. These classes map directly to the terms in PROLOG from which we started from. The second predicate is used to generate the `@PlView` annotations.

5 Evaluation

To evaluate our approach we reimplemented the London Underground example as in [3]. We made two implementations, one with JPL and one with our connector. The structure of the London Underground is defined by `connected/3` facts in PROLOG. Speaking of the undirected graph, representing the London Underground with stations as nodes and lines as edges, a `connected` fact describes in this context adjacent stations. The first and the second argument of a `connected` fact is a station. The third argument is the connecting line. We give some examples for `connected` facts:

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

In our first implementation we use JPL in order to retrieve a station connected to a given station:

```
1 public class Line {
2   public String name;
3   public Term asTerm() {
4     return new Compound("line", new Term[]{new Atom(name)});
    }
  }
5 public class Station {
6   public String name;
7   public Station(String name) { this.name = name; }
8   public Term asTerm() {
9     return new Compound( "station", new Term[]{
        new Atom(name)});
    }
10  public static Station create(Term stationTerm) {
11    String name = ((Compound)stationTerm).arg(1).name();
12    return new Station(name);
    }
13  public Station connected(Line line) {
14    String stationVarName = "Station";
15    Term[] arguments = new Term[]{asTerm(),
      new Variable(stationVarName), line.asTerm() };
16    Term goal = new Compound("connected", arguments);
17    Query query = new Query(goal);
18    Hashtable<String, Term> solution = query.oneSolution();
```

```
19      Station connectedStation = null;
20      if(solution != null) {
21        Term connectedStationTerm = solution.get(stationVarName);
22        connectedStation = create(connectedStationTerm);}
23     return connectedStation;}}
```

As one can see, the implementation with JPL leads to a lot of lines of code. In the method `connected` the complex term structure is created in order to query the predicate `connected`. The result handling is tedious again. With our approach we do not have to create any term structures in JAVA. Instead, we need to implement an extra class `Connected` representing the goal in PROLOG with the predicate `connected/3`:

```
1 public class Connected {
2   public Station stat1;
3   public Station stat2;
4   public Line line;
5   public Connected() { };
6   public Connected(
        Station stat1, Station stat2, Line line) {
7     this.stat1 = stat1;
8     this.stat2 = stat2;
9     this.line = line;}
  }
10 public class Line {
11   public String name;
  }
12 public class Station {
13   private String name;
14   public Station connected(Line line) {
15     Connected connected = new Connected(this, null, line);
16     Query query = new Query(connected);
17     query.unify();
18     return connected.stat2;}
  }
```

As the following table shows, our approach needs less lines of code to implement the London Underground example than the implementation with JPL.

	Line	Station (w/o connected())	Connected	connected()	sum
JPL loc	4	8	0	11	23
CAPJA loc	2	2	9	5	18

However, lines of code do not say anything about the code's complexity and information density. Our class `Connected` is very simple. It contains only member variables and two simple constructors whereas in JPL already the method `connected()` of the class `Station` is fairly complex.

With the data of the complete London Underground with 297 stations, 13 lines and 412 connections, we made 50,000 executions[1] with both implementations. The result of the performance test is presented in the following table:

[1] Core i5 2 x 2.4 GHz, 6 GB RAM, Ubuntu 14.04

	ø execution time of 50.000 calls
JPL	$\sim 1.2s$
CAPJA	$\sim 2.6s$

Castro et al. did a similar comparison between JPL and their LOGICOBJECTS [3]. Their implementation with LOGICOBJECTS is slower than the corresponding JPL implementation by a factor of about 7 whereas our connector implementation is just about 2.13 times slower.

Aside from a performance improvement in form of field structure caching, as mentioned in Section 4, we identified that getting and setting the values of the member variables via Reflections is slow. In the future we want to use static calls as often as possible instead of using the Reflection API. In order to make use of direct calls, we need to generate *Specialized* `OTT` classes (`SOTT`) for all classes that we want to map. These generated classes contain highly specialized `toTerm()` and `fromTerm(String)` methods that call their member variables directly if public or with their getter and setter methods. This attempt picks up concepts from our prior work in [11]. But this time, we want to make use of so called *Annotation Processors* that extend the JAVA compiler in order to generate additional code at compile time. Those generated `SOTT` classes are only optional. The `OTT` class as presented in this work, still will be used in the case that no `SOTT` class exists. We have implemented an early prototype in order to test these ideas for feasibility. In an early test using `SOTT` classes, we have measured an average time for 50.000 executions of about 1.5 seconds for the London Underground example. This is a huge performance gain and is just about 25% slower than JPL, the highly optimized interface for SWI-PROLOG.

6 Future Work

The presented interface in Section 3.3 has proven to be well applicable for SWI-PROLOG. However, our approach is not limited to this PROLOG implementation. We currently develop a standard interface based on pipes that is suitable for most PROLOG implementations and completes our generic approach. This way, we want to accomplish a portable solution that is independent from any PROLOG implementation.

In addition, we further want to reduce the necessary lines of code. In our current approach we use a wrapper class called `Query` for calling a PROLOG goal. Instead, we could have used an abstract superclass that is extended by the class of an object that is going to be mapped. This superclass manages the `OTT` objects that contain the logic behind the creation of the textual term representations and the parsing. Even the request control for `unify()`, `backtrack()` and `findall()` then is part of this superclass. Using the abstract superclass the request for PROLOG from JAVA in the Underground example in the lines 16, 17 can be reduced to a single line containing just the method call `connected.unify()` which additionally saves the initialisation of a `Query` object.

However, the approach with a superclass has a big drawback: we want to be able to use almost every class in JAVA for the Object Unification. This will not work for classes that already extend a class because JAVA does not support multiple inheritance

yet. In JAVA 8 there is a new feature called *Default Methods* that allows to implement a method directly within a JAVA interface. Using this new feature we can implement all the needed functions as Default Methods in an interface. Because multiple JAVA interfaces can be implemented by a single class, we achieve with this new interface the same reduction in lines of code as with an abstract superclass. This way, we can avoid the multiple inheritance problem for classes.

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. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. T. Majchrzak, H. Kuchen. *Logic java: combining object-oriented and logic programming.* Functional and Constraint Logic Programming. Springer Berlin Heidelberg, 2011. 122-137.
9. L. Ostermayer, D. Seipel. *Knowledge Engineering for Business Rules in Prolog.* Proc. Workshop on Logic Programming (WLP), 2012.
10. 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.
11. 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.
12. L. Ostermayer, F. Flederer, D. Seipel. *A Customisable Mapping between Java Objects and Prolog Terms.* `http://www1.informatik.uni-wuerzburg.de/pub/ostermayer/paper/otm_2014.html`
13. P. Singleton, F. Dushin, J. Wielemaker. *JPL 3.0: A Bidirectional Prolog/Java Interface.* `http://www.swi-prolog.org/packages/jpl`
14. J. Wielemaker. *SWI Prolog.* `http://www.swi-prolog.org`