A Customisable Mapping between JAVA Objects and PROLOG Terms

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. For combining the power of the programming languages JAVA and PROLOG a mapping between language artefacts is necessary. Several approaches for mappings have been discussed in the last decade. Especially the dynamic evolution of JAVA, and PROLOG, allows to develop ever refined concepts towards a smooth interaction.
However, previous approaches still have drawbacks such as additional program code of tedious helper structures of the mapping, a lack of control/flexibility in the mapping process, an unfavourable trade-off in execution time, and often a dependency on a particular PROLOG engine or PROLOG dialect. Extensions to the JAVA VM for an embedded version of PROLOG risk the portability of the JAVA program due to the modifications. Another problem with embedded versions of PROLOG is that they are cut off from the evolution of stand-alone versions.
In this paper, we propose a highly customisable object to term mapping that overcomes the mentioned disadvantages. We offer a default mapping for almost every class in JAVA. A controlled mapping multiplicity is given by annotations in JAVA that define different *Prolog Views* on a class in JAVA. We generate in PROLOG the annotations for such a *Prolog View* as well as classes in JAVA that map to a given term in PROLOG. For this purpose, we introduce the *Prolog-View-Notation* (PVN). Finally, we have identified and solved a mapping anomaly, the so-called *Reference Cycle*.

Keywords. Multi-Paradigm Programming, Logic Programming, PROLOG, JAVA.

1 Introduction

Modern software systems increasingly become more and more complex and dynamic. Such systems often consist of dynamically linked subsystems that operate on certain problem domains. The subsystems, as well as the programming languages they are implemented in, evolve continually and independently of each other.

Hence, a smooth interaction of subsystems can only be guaranteed by a smooth interaction of the programming languages involved. Language-interaction can be achieved for instance through embedding. However, embedding one language into another mostly sacrifices the dynamical and independent evolution of both languages. New features and enhancements of the original programming language have to be transferred somehow

to the embedded version. Changes of the host-language may also affect unfavourably the interaction with the embedded language.

Another option for language-interaction is through a communication layer, for instance an interface or a shared kind of buffer/memory. This way, the interacting languages can evolve independently. An integral part is the mapping of artefacts from one language to the other. In the case of JAVA and PROLOG this is especially a challenging task, because both programming languages are representatives of different programming paradigms. We already have discussed the advantages of knowledge engineering in PROLOG [11] and in contrast to it, the usage of DROOLS [12], a popular JAVA tool for the development of rules.

There are several approaches for mappings between JAVA and PROLOG. However, most approaches come with minor or major drawbacks. Often it is necessary to create complex object structures in JAVA to reflect terms in PROLOG. It leads to an increased programming effort due to this undesirable code for the helper structures in the mapping, a stretched program execution time and a clouded program purpose. In still other approaches, a more straight mapping to new data structures is proposed, for instance via the serialization mechanism in JAVA. These data then, will be analysed on the PROLOG side and represented as terms. Usually, there is no control over the mapping, i.e., the term representation of objects in PROLOG or of classes on the other side. To gain control over the mapping process new data structures are introduced that often lead to a tremendous trade-off in execution time for control. Another problem is the dependency on a certain PROLOG engine, a PROLOG dialect, or an extension of the JAVA VM. Such an extension can be used for embedded versions of PROLOG in JAVA, but puts the portability of the JAVA programs at risk. In addition, embedded versions of PROLOG often are cut off from the evolution and the rich libraries of stand alone implementations.
In this paper, we propose a highly customisable mapping between JAVA objects and PROLOG terms. Apart from a default mapping that is available for almost every class in JAVA, the user is able to define different *Prolog-Views* on a single class in JAVA. By a Prolog-View we understand a single object to term mapping that results in a textual term representation of a class' instance in PROLOG. To express in JAVA these different views we developed a new annotation layer. The equivalent of our annotations in JAVA are terms in PROLOG with a syntax following the in this paper introduced *Prolog-View-Notation* (PVN). The PVN can be used not only for generating the new annotations expressing a Prolog-View but also the source code of JAVA classes that map exactly to already defined terms in PROLOG.

The remainder of this paper is organized as follows: Section 2 gives a detailed description of our mapping mechanism. We start in 2.1 with the default mapping for instances of classes in JAVA that are not annotated by a `@PlView` annotation. Then, we explain in 2.2 how to customise the default mapping using our special annotation layer to express Prolog-Views on classes in JAVA. In 3 we introduce the PVN, which is used in 3.1 to generate classes, and annotations in 3.2. Section 4 illuminates an interesting conversion anomaly and presents a solution to handle these kinds of anomalies with regard to our approach. We position our approach to related work in 5, and finally, give an overview in 6 what we have achieved so far and where to go in the future.

2 Prolog-Views on Objects in JAVA

In this section we describe the mapping mechanism between objects in JAVA and terms in PROLOG. The default mapping mechanism in 2.1 relies on meta programming concepts as provided by the Reflection API which we use to read the fields of an object that we want to map. In 2.2 the usage of the Reflections in JAVA can be controlled by an annotation layer which leads to a simple mechanism to customise the object term mapping.

2.1 Default Mapping between Objects and Terms

To provide JAVA developers an easy way to use PROLOG rules and facts, our approach implements a simple default mapping. For the default mapping JAVA classes can be used without any modifications to the class' source code. The JAVA programmer does not need to know the syntax and only little of the functionalities in PROLOG in order to use it.

In the default mapping of a JAVA object to a PROLOG term the object's class name is mapped to the 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. `'MyClass'`.

For the default mapping, we have decided to convert the Camel Case notation which is common in JAVA to the Snake Case notation which is popular in PROLOG. 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, a class' name `MyClass` maps to the atom `my_class`.

Classes may 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 method `getDeclaredFields()` in JAVA. According to JavaDoc [10], 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 (list), List (list term) and Object (compound term). For other data types, only existing in certain PROLOG implementations like string in SWI-PROLOG [17], the default mapping can be further extended or changed. It is possible to save the changes to the default mapping into a configuration file.

The mapping of member variables in JAVA with an assigned `null` value are handled particularly. In PROLOG a variable binding to `null` like in JAVA is not known. We consider `null` values in JAVA as (logical-)variables in PROLOG. That means every time we map an object to PROLOG, all `null` values are substituted with different variables in PROLOG. To illustrate the default mapping mechanism a simple example with a `Person` class follows:

```
class Person {
  private String givenName;
  private String familyName;
  private Person[] children;
  // ... constructor/ getter / setter
}
```

We omit the straightforward definitions of getter and setter methods and convenient constructors. The member variable `children` is a `Person` array. Now, we create an instance `homer` of the class `Person` that has a single child:

```
Person homer = new Person("Homer", "Simpson",
  new Person[]{new Person("Bart", "Simpson", null)}
)
```

Using the default mapping `homer` is mapped to the following `person/3` term. Note that `null` has been mapped to `X_1`, a variable in PROLOG:

```
person('Homer', 'Simpson', [person('Bart', 'Simpson', X_1)])
```

2.2 Customised Mapping between Objects and Terms

If the default mapping does not map an object to a desired term structure, the user is able to modify the mapping with special purpose annotations 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`, `@PlArg` and `@PlViews`.

`@PlView` is used to describe a single Prolog-View on a given class in JAVA. It is possible to define different Prolog-Views on the same class. We achieve this with different `@PlView` annotations within the class. An `@PlView` annotations consists of several elements: `viewId` is the only mandatory element which is used to set an identifying name for the Prolog-View. The element `functor` overwrites the predicate name normally set by the default mapping. The last three elements of an `@PlView` annotation are lists, namely: `orderArgs`, `ignoreArgs` and `modifyArgs`. These lists are used to manipulate the structure of the textual term representation corresponding to 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 arguments of the resulting term corresponds to the order of the member variables in the 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 sorting 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 conserved. 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 elements `orderArgs` and `ignoreArgs` define which member variables are considered for the mapping and which not, as well as the order of those parameters.

`modifyArgs` is an array consisting of `@PlArg` annotations that modify the mapping of a single member variable to an argument of the target term.

`@PlArg` has three elements for the modifications: `valueOf`, `ViewType` and `viewId`. As long as there is no `@PlArg` annotation in an `@PlView` annotation, the default mapping as described in 2.1 is applied for all mapped member variables.

`valueOf` references the name of the member variable whose mapping is going to be manipulated by the `@PlArg` annotation. A unique identifier for the `valueOf` element is mandatory for an `@PlArg` annotation.

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

`ViewType` defines the PROLOG type which will be used within the target term in PROLOG. Options for `ViewType` are elements of an enumeration representing certain PROLOG types like atom, float, integer or structures like compound, list. In case of compound and list, it is possible that again several Prolog-Views for a referenced class exist. Though, the user can select a Prolog-View on the referenced class via an appropriate `viewId`. If the member variable is a reference to another object, `type` must either be set to compound or list. In case of list we map the member variable to a PROLOG list containing only the arguments of the mapping target.

A modified version of the `Person` class as defined in 2.1 demonstrates the usage of the just mentioned annotations:

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

The new Prolog-View on the class `Person` is called `personView1`. It maps the object `homer` as defined above to another term with functor `person` that only has two arguments with an order as defined in the `orderArgs` array. Within `modifyArgs` the reference to the `Person` array `children` modifies the resulting PROLOG list. The list consists of `person` terms that again result from the Prolog-View `personView1` on

the class `Person`. All together, the Prolog-View `personView1` on the JAVA object `homer` has the following textual term representation in PROLOG:

```
person('Homer', [person('Bart', X_1)])
```

In this example the member variable `familyName` is missing because it is not listed within `orderArgs`. The `givenName` and `children` are sorted within the PROLOG term as defined in `orderArgs`. In addition, the member variable `children` was modified via the `@PlArg` annotation. The inner `person` term also has only two arguments, `'Bart'` as `givenName` and an empty PROLOG list representing the empty array `children`.

`@PlViews` finally, is the third annotation. JAVA does not support multiple annotations of the same type within a class until version 7, we need to introduce the annotation `@PlViews` just to allow multiple Prolog-Views on a single given class.

```
@PlViews({
  @PlView(viewId="personView1", functor="person",
    orderArgs={"givenName", "children"},
    modifyArgs=
       {@PlArg(valueOf="children", viewId="personView1")})
  @PlView(viewId="personView2",
    orderArgs={"familyName", "givenName", "children"})})

class Person {
  private String givenName;
  private String familyName;
  private Person[] children;
  // ... constructor/ getter / setter
}
```

3 Prolog-View-Notation

This Section introduces the *Prolog-View-Notation* (PVN). With the PVN, we generate in 3.1 classes in JAVA that map exactly to a given term in PROLOG. In 3.2 we show in PROLOG a simple way to generate the annotations that describe a Prolog-View on a given class in JAVA.

3.1 Generating JAVA Classes

In Section 2 we have described how to define in JAVA the mapping of objects to terms in PROLOG. However, this section focuses on how to control the mapping on the PROLOG side. Therefore, we exploit the information contained in a PVN expression. This notation is used to describe the structure of terms that already exist in PROLOG. The structural information is used to generate JAVA classes, if necessary with annotations. These classes map exactly to the terms that are described by the PVN expression in PROLOG. This is possible because apart from the structural information of terms, an

PVN expression also contains information about the mapping targets in JAVA, such as class names or member variable names. The PVN provides two nested predicates:

pl_view(ViewName, ViewType, ViewSpec). The Argument `ViewName` is the functor of a mapping target, a term in PROLOG. As the default mapping implies, `ViewName` also defines a JAVA class name. In the class generation process, beginning lowercase characters are replaced by their uppercase equivalent and a `ViewName` in Snake Case notation is converted to Upper Camel Case notation. The Argument `ViewType` of a `pl_view` is always compound, as a `pl_view` defines a term with arity greater zero. The Argument `ViewSpec` is a PROLOG list of all the arguments of the term. All members of the list `ViewSpec` are `pl_arg` terms that describe a single argument more precisely.

pl_arg(ArgName, ArgType, ArgSpec). The Argument `ArgName` defines the JAVA name of a class' member variable and `ArgType` defines the PROLOG type/structure of the term's corresponding argument. There are different types/structures which can be used: atom, compound, float, integer and list. Differently used is `ArgSpec`:

- *Reference*: If another term should be used here, a reference to a `pl_view` term is created. In this case, `ArgSpec` just contains the `ArgName` of the referenced `pl_view` term. The `ArgType` of a reference can be either a compound term or a list in PROLOG, according to the desired representation for this argument.
- *Array*: A JAVA array is represented in PROLOG as a list containing only terms with the same functor and arity. Therefore, `ArgSpec` consists of a PROLOG list with a single `pl_arg` argument that specifies the type of the generated JAVA array. For this purpose, `ArgType` must be set to `list`. If the list contains more than one `pl_arg` term, the generator creates a new class with appropriate member variables according to the mixed list. This is done, because arrays in JAVA can not contain elements of different JAVA types, with exception to `Object` arrays. However, `Object` arrays in JAVA are commonly avoided because of the missing type safety.
- *Primitives*: Some data types for JAVA member variables do not need any further specification. The `ArgSpec` argument in a `pl_arg` term can be omitted in cases where `ArgType` is set to atom, float and integer.

The following PVN example describes a new `person/3` term in PROLOG and a new `Person` class in JAVA:

```
pl_view(person, compound, [pl_arg(addresses, list,
   [pl_arg(address, compound, address)]),
   pl_arg(names, list,
      [pl_arg(givenName, atom), pl_arg(familyName, atom)])])

pl_view(address, compound,
   [pl_arg(street, atom), pl_arg(postalCode, integer)])
```

The first `pl_arg` term in the first PVN expression has a `ArgType` equal to list and the `ArgSpec` is a PROLOG list that contains only a single `pl_arg` term. This leads in the class generation process to the `Address` array in JAVA. The list `names` contains two `pl_arg` elements which leads to the creation of a distinct class `Names` that has the two member variables `givenName` and `familyName`. With the given PVN expression our generator creates the following `Person` class in JAVA:

```
class Person {
  private Address[] addresses;
  private Names names;
  // ... constructor/ getter / setter
}

class Address {
  private String street;
  private Integer postalCode;
  // ... constructor/ getter / setter
}

class Names {
  private String givenName;
  private String familyName;
  // ... constructor/ getter / setter
}
```

Primitive data types in JAVA always have a value assigned and it is not possible to set `null` to a primitive type like `int` or `double`. However, we use `null` for the mapping to variables in PROLOG. Therefore, the generator implements `postalCode` using the wrapper class `Integer` instead of using the primitive type `int`.

3.2 Generating JAVA Annotations

Up to this point, we generated JAVA classes from PVN expressions in PROLOG. But beside classes, we also can create `@PlView` annotations from PVN expressions. Because a PVN expression describes a term representation of a class in JAVA it contains all the necessary information to generate an appropriate `@PlView` annotation. With help of appropriate PVN expressions that describe the term `person/3` as defined in 2.1, we generate the following `@PlView` annotation:

```
@PlView(viewId="personView3", functor="person",
   orderArgs={"givenName", "familyName", "children"},
   modifiedArgs={
      @PlArg(valueOf="givenName", type=ATOM),
      @PlArg(valueOf="familyName", type=ATOM),
      @PlArg(valueOf="children", type=LIST)
  } )
```

The listing above shows the Prolog-View `personView3` identified by its `viewId`. It maps a `Person` object to a term with functor `person`. The term `person` has

the three arguments `givenName`, `familyName` and `children` in the given order. The arguments' types in PROLOG are defined as follows: atom for `givenName` and `familyName`, and a PROLOG list for `children`. The generation of `@PlView` annotations is also used in the creation of multiple Prolog-Views on a single given class in JAVA. We define multiple Prolog-Views on the same class by referencing the same class name in the first argument of multiple `pl_view` terms. However, following the process as described so far, the generation of a JAVA class from multiple `pl_view` terms referencing the same class would lead to different source codes for the same class. Instead, we intend to generate a single JAVA class with multiple `@PlView` annotations. The class generator achieves this by internally merging all `pl_view` terms together. The resulting set of member variables is a union of all the member variables that are defined in the different `pl_view` terms. For every `pl_view` term an `@PlView` annotation is generated. The following example shows a generated class `Person` that has been created from multiple `pl_view` terms. We start with two different `pl_view` terms that describe two different `person/3` terms in PROLOG:

```
pl_view(person, compound,
   [pl_arg(givenName, atom),
    pl_arg(familyName, atom),
    pl_arg(children, list, person)])

pl_view(person, compound,
   [pl_arg(familyName, atom),
    pl_arg(givenName, atom),
    pl_arg(mother, compound, person)])
```

Both term representations contain the arguments `givenName` and `familyName`, but with different order. The first `pl_view` term describes a list named `children`, whereas the second `pl_view` term keeps only a reference to the `mother`. Merging the informations from both `pl_view` terms in the generation process leads to the following class `Person` with two different `@PlView` annotations:

```
@PlViews({
  @PlView(viewId=Person.VIEW_1
    orderArgs={"givenName", "familyName", "children"})
  @PlView(viewId=Person.VIEW_2
    orderArgs={"familyName", "givenName", "mother"})})

class Person {
  public static final String VIEW_1 = "personView1";
  public static final String VIEW_2 = "personView2";

  private String givenName;
  private String familyName;
  private Person[] children;
  private Person mother;
  // ... constructor/ getter / setter
}
```

Note, in the example the union of the PVN expressions yields under the default mapping to a `person/4` term in PROLOG whereas the two Prolog-Views lead to different `person/3` terms as previously defined in the PVN expressions. The `viewId` of generated `@PlView` annotations is numbered in the order of the appearance of the `pl_view` terms in PROLOG.

When generating Prolog-Views on JAVA classes the `viewId` is saved as a `static final` String. This way, we support auto-completion functionalities of development environments where only the listed static values are suggested as `viewId`.

4 Breaking Reference Cycles

Using our approach, we could observe an anomaly which we call *Reference Cycle*. The reason for the anomaly is that we can not create cyclic term structures in PROLOG with our mapping mechanism as presented so far. Incidentally, there are simple cases that are not uncommon for ordinary programs in JAVA and already lead to infinite nested terms in PROLOG if we use the mapping as introduced up to this point. In the following, we want to discuss such a simple case where a Reference Cycle in JAVA occurs:

```
Person bart = new Person("Bart", "Simpson");
Person lisa = new Person("Lisa", "Simpson");

bart.setSibling(lisa);
lisa.setSibling(bart);
```

This implementation of a `Person` class in JAVA has the attributes `givenName`, `familyName` and `sibling`. The attribute `sibling` is a reference to another object of type `Person`. In the code snippet above, we instantiate the two `Person` objects `bart` and `lisa` and reference each other with the method `setSibling`. Then, `lisa` and `bart` are siblings. This leads to a Reference Cycle as shown in Figure 1.

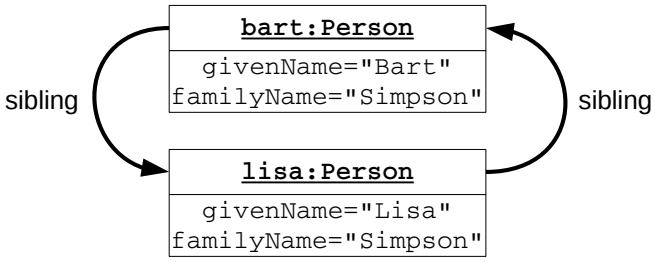

Fig. 1: Example of a Reference Cycle

Suppose, we want to map the object `bart` to PROLOG using our default mapping this leads to the following nested term structure in PROLOG:

```
person('Bart', 'Simpson', person('Lisa', 'Simpson',
   person('Bart', 'Simpson', person('Lisa', 'Simpson',... )))).
```

As one can see, the resulting term in PROLOG is an infinite alternation of the same two nested `person` terms, q.v. Figure 2.

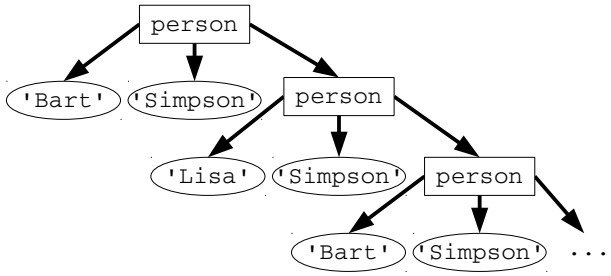

Fig. 2: Infinite Alternation of `person/3` in PROLOG

We can detect Reference Cycles in JAVA. If one incurs, an exception with an error report can be raised. The consequence of such an exception is that on the JAVA side as well as on the PROLOG side changes to the source code are necessary to solve the Reference Cycle.

In JAVA, we provide an extra element for the `@PlArg` annotation. It is called `noCycle` and is of type boolean. If its value is `false`, it can be omitted, otherwise the referenced member variable in the `@PlArg` annotation is always mapped to a special list in PROLOG. We call this list the *Object Reference List* which has the following structure:

```
[this_oid(This_OID),
   [oid_to_term(Ref_OID, Ref_Term), ...]].
```

When transforming JAVA objects to PROLOG terms, each reference with a `noCycle` element equal to `true` is replaced by an Object Reference List. This is done the first time the reference is processed. Instead of generating a nested term structure, the Object Reference List is created as shown for our example in the listing below:

```
person('Bart', 'Simpson',
   [this_oid('Person@123'),
      [oid_to_term('Person@456',
          person('Lisa', 'Simpson', 'Person@123'),
       oid_to_term('Person@123',
          person('Bart', 'Simpson', 'Person@456')])]).
```

The first element `this_oid` in the list contains the Reference ID of the object in JAVA that we want to map to PROLOG. In the example, this is `Person@123` for the `Person` object `bart` which has the object `lisa` as a sibling.

The second element is again a list that contains terms called `oid_to_term/2`. The purpose of `oid_to_term` is to save the relationship between the Reference ID of an object and its corresponding term representation in PROLOG. A Reference Cycle can be broken by a modification of the mapping of a single member variable that is part

of the Reference Cycle. The `oid_to_term` list has members for all objects that are a reference target of the member variable that is used to break the Reference Cycle.

If an object represented in this list references via the member variable used to break the Reference Cycle another object that is also represented in the list, then the object's Reference ID is used for the mapping instead of the usual term representation.

In the following, we explain for our example the modifications to the source code in JAVA and PROLOG due to the incidence of a Reference Cycle. In JAVA, we just extend the class `Person` from our example by a `@PlView` annotation. The mapping of the attribute `sibling` is modified with a `noCycle` element in order to break the Reference Cycle.

```
@PlView(viewId="person",
  modifyArgs={@PlArg(valueOf="sibling", noCycle=true)})
```

On the PROLOG side, we now have to handle a different `person` term. Note, that the arity the term `person/3` is unaffected by the modifications resulting from the element `noCycle` in JAVA. However, the second argument of `person` now is a PROLOG list as shown above. We have to decompose in PROLOG the Object Reference List in order to get desired information.

```
sibling_to_givenName(Person, SibGivenName) :-
   Person = person(GivenName, FamilyName, ObjRefList),
   ObjRefList = [this_oid(ThisOid), OidTerms],
   member(oid_to_term(ThisOid, person(_, _, SibOid)),
      OidTerms),
   member(oid_to_term(SibOid, person(SibGivenName, _, _)),
      OidTerms).
```

The listing above shows an easy rule that has as an input parameter a `person/3` term. If `ObjRefList` matches an Object Reference List as introduced above, the rule retrieves the `GivenName` of a person's sibling. With `member/2` we select from the Object Reference List the given person and the referenced sibling.

In our example, there is just a single other object involved in the Reference Cycle, so there is only one class where the Reference Cycle can be broken with a `@PlView` annotation containing a `noCycle` element. In other cases, there might be a cycle that concerns several JAVA classes and therefore multiple Object References. Because just one break in the whole cycle is sufficient to get rid of the infinite nesting problem, the user is free to select any class within a Reference Cycle.

Other approaches like [4] also provide a recursive mapping of JAVA objects and PROLOG terms. Although these approaches also face the problem with Reference Cycles, neither a solution has been proposed nor the problem itself has been reported so far.

5 Related Work

There are several approaches that have proposed a transformation of JAVA data structures into PROLOG terms, and vice versa.

The interface JPL [15] between SWI PROLOG and JAVA uses a symbolic term representation in Java in order to map objects to terms in PROLOG. Such an approach uses lots of tedious program code wrapping the term structure in JAVA. A JAVA developer has to recreate the structure of the terms in PROLOG with classes like `Compound` or `Atom`, every time he wants to call PROLOG.

In [13] we have presented an approach that integrates PROLOG rules into JAVA. This is done by the generation of JAVA source code thats purpose is to call a single goal in PROLOG. For all terms, corresponding classes are generated to build up the goal in PROLOG. All generated classes are saved to a JAVA archive (JAR) for the integration to existing JAVA projects. An XML Schema had been introduced that describes the data exchange format. The XML Schema then had been used to generate the classes in JAVA. This former proposal requires a lot of generated code and every generated JAR is limited to call a single goal in PROLOG. There had no mapping been provided for already existing classes in JAVA. However, with the proposed default mapping our new approach is usable without any source code generation. We need no external data format like the XML Schema to store the information that is crucial to the mapping. Our annotation layer allows a far more flexible mapping.

In [9], the JAVA virtual machine is extended for the purpose to embed logic programming directly into JAVA source code. Of course, this means that the programming language itself is extended in order to be able to write logic programmes within JAVA. But there are disadvantages: As the JAVA virtual machine is changed, the portability of the JAVA program decreases because the virtual machine has to be ported to other platformas, too. A second drawback is that established libraries for PROLOG or other logic programming languages can not be used without a certain effort.

The proposals in [4,8] have used linguistic symbiosis to connect JAVA and PROLOG via frameworks. Linguistic symbiosis connects different programming languages by identifying and linking similar language artefacts. In the case of JAVA and PROLOG, they use methods in JAVA to call goals in PROLOG. JAVA objects, as value holders, are transformed to terms in PROLOG. This is similar to our approach but we do not map JAVA methods. Instead, we map JAVA objects to goals in PROLOG.

In [3], the serialisation mechanism in JAVA is exploited. A serialised JAVA object is represented as a complex structured term in PROLOG. This term, first, has to be analysed on the PROLOG side and transformed into a proper, more natural form. The PROLOG developer has to filter for the information he needs. In our approach the resulting structure of an object to term mapping is predefined by the default mapping or customised by the `@PlView` annotations. The PROLOG developer gets exactly what he needs without any further transformations.

Another form of customisation of the mapping between objects and terms is described in [5,6]. Instead of annotations, special converter classes are used that must have methods which take care of the mapping. Although this approach is flexible, it requires knowledge in both programming languages, JAVA and PROLOG, to implement theses converters in JAVA. In our approach the customisation of the mapping can be developed not only in JAVA, but with the PVN in PROLOG, too.

6 Conclusion and Future Work

In this paper, we have proposed different concepts which are shown in Figure 3. We use the PVN to describe the structure of a PROLOG term as the mapping target of an instance of a class in JAVA. This class can be created by a generator in PROLOG, that uses PVN expressions as input parameters.

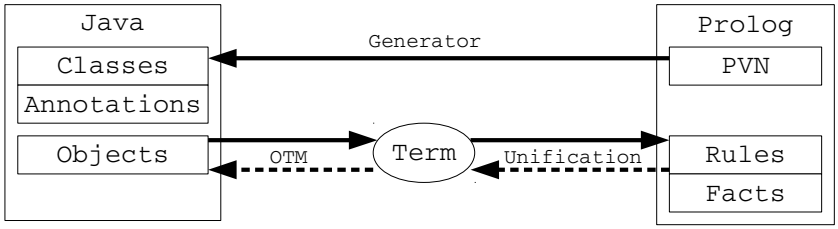

Fig. 3: Overview of our proposed Concepts

The `@PlView` annotations can be used to modify the mapping of an object to a term. These annotations also can be generated in PROLOG in order to annotate existing JAVA classes. Objects and terms are mapped via the proposed object term mapping (`OTM` in Figure 3).

For lack of space, we postpone to another paper [14] the implementation and further evaluation of the mapping as presented in this work. We use our mapping for calls from JAVA to PROLOG and realize this way an *Object Unification*. As Figure 3 indicates, the target term of the mapping can be unified within PROLOG and causes a variable binding. The unified term, then, is mapped back from PROLOG to the original object in JAVA.

In the future, we want to evaluate our approach with several case studies and technically refine implementations of our concepts. For instance, a technical improvement comes with JAVA 8 where repeating annotations are introduced. This feature enables the usage of the same annotation for a single target multiple times. That means, we can get rid of the `@PlViews` annotation and just write multiple `@PlView` annotations into a single class in JAVA as follows:

```
@PlView(viewId="personView1",
  orderArgs={"givenName", "children"},
  modifyArgs={@PlArg(valueOf="children", viewId="personView1")})

@PlView(viewId="personView2",
  orderArgs={"familyName", "givenName", "children"})

class Person {
  private String givenName;
  private String familyName;
  private Person[] children;
  // ... constructor/ getter / setter
}
```

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.* In 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.* Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages, 2003.
9. T. Majchrzak, H. Kuchen. *Logic java: combining object-oriented and logic programming.* Functional and Constraint Logic Programming. Springer Berlin Heidelberg, 2011. 122-137.
10. Oracle Corporation. *Java Platform, Standard Edition 7 API Specification.* `http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html#getDeclaredFields()`
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. *CAPJa - A Connector Architecture for Prolog and Java.* `http://www1.informatik.uni-wuerzburg.de/pub/ostermayer/paper/capja_2014.html`
15. P. Singleton, F. Dushin, J. Wielemaker. *JPL 3.0: A Bidirectional Prolog/Java Interface.* `http://www.swi-prolog.org/packages/jpl`
16. J. Wielemaker, T. Schrijvers, T. Markus, L. Torbjörn. *SWI-Prolog.* Theory and Practice of Logic Programming. Cambridge University Press, 2012. 67-96.
17. J. Wielemaker. *SWI Prolog.* `http://www.swi-prolog.org`