Seamless Cooperation of JAVA and PROLOG for Rule-Based Software Development

Ludwig Ostermayer

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

Abstract. Modern software often relies on the modular combination of several software subsystems, for instance rule-based subsystems for decision support, verification or knowledge management. Different software libraries in potentially different programming languages have to work together in just a single application. Even more complex is the case when different programming paradigms are combined. Such a diversification of programming languages and modules in just a single software application can only be mastered by smooth integration techniques that retain the power of different programming paradigms. Unfortunately, for the popular object-oriented programming language JAVA and the common logic programming language PROLOG occurring interoperability problems still are not solved sufficiently. To overcome various external and internal issues of former approaches we propose an intuitive and portable Connector Architecture for PROLOG and JAVA (CAPJA). A concise, extensible and independent communication layer with a pluggable interface system allows for a seamless integration of PROLOG programs in JAVA. Compared to former approaches we could reduce the code for the mere communication to a minimum. Several case studies with different PROLOG systems document not only the portability but the overall applicability of our approach.

Keywords. Multi-Paradigm Programming, Java, Prolog, Rule-Based Systems.

1 Introduction

Modern software systems often consist of dynamically linked subsystems handling different application relevant problem domains, for instance rule-based subsystems for decision support, verification or knowledge management. These subsystems evolve continually and often independently of one another. Not every problem can be solved elegantly in every programming language. To support multi-language systems we require smooth interaction mechanisms for the programming languages involved. In case of JAVA and PROLOG, this is an especially challenging task as they additionally follow two different programming paradigms.

The object-oriented programming paradigm is widely used in the field of industrial software engineering as well as in the academic sector. Currently, one of the most popular object oriented programming languages is JAVA. It has a rich set of libraries, especially for the development of refined graphical user interfaces (GUI), web development and embedded devices. In addition, there are several public tools and Integrated Development Environments (IDE) such as Eclipse [9] that substantially support the development with JAVA. Last but not least, JAVA has a very active community. But there are also complaints about JAVA such as being too verbose, full of boilerplate code, ill-suited for rapid prototyping and most notably being little declarative, just to mention a few. Although there are tools for JAVA, e.g. DROOLS [11], the development of rule-based systems in JAVA still comes with flaws [15].

Logic programming following an alternative, declarative programming paradigm excels at these points. Languages like the common logic programming language PROLOG are particularly well suited for an intuitive development of rule-based systems. Programs in PROLOG consist of a collection of rules and facts that describe Horn clauses and are evaluated top-down. PROLOG is widely used for knowledge engineering and expert systems [3], the development of business rule applications [16] and for natural language processing[20]. Backtracking, partial bindings, incomplete structures, negation and recursion result in elegant, concise and readable programs that improve maintainability and refactoring. For this reason PROLOG qualifies for rapid prototyping and agile software development. Features like custom operators, infix notation and definite clause grammars allow for an elegant and fast development of meta-interpreters for all purposes, for instance readable business rule systems [14] or domain specific languages.

Various approaches [1, 2, 4–6, 8, 12] of the last decade have attempted to solve the issues related to a cooperation of JAVA and PROLOG. We identified several external and internal problems that hinder or limit a smooth cooperation. Among the external ones are restrictions coming from architectural assumptions, dependencies on specific PROLOG systems or operating systems, the amount of necessary modifications to existing code bases, re-usability, performance or maintenance. Internal issues are, for instance, the amount of boilerplate code for the mere communication, mixtures of PROLOG and JAVA syntax and semantics, missing IDE support like syntax highlighting or auto-completion, options for customisation or a neat integration in the development cycle. Unfortunately, all former approaches come with more or less of these issues, especially with regard to a subsequent integration of PROLOG programs into existing JAVA applications.

Therefore, we propose an intuitive and portable Connector Architecture for PROLOG and JAVA (CAPJA). Using our approach existing objects in JAVA can be reused to query PROLOG without additional JAVA data structures representing terms in PROLOG. We reconcile both languages internally by using an extensible, bidirectional object-term mapping freeing up the programmer from bothering with boilerplate code to handle the mere communication. Lambdas, a feature recently introduced to JAVA, can be exploited with CAPJA to express complex queries to PROLOG in a natural, declarative and concise way. We use advanced meta-programming techniques in JAVA to support the development process with CAPJA and to improve the performance of resulting bytecode. All parts of CAPJA are implemented in JAVA and available as JAR (JAVA Archive) though portability and a simple integration are guaranteed. The mapping is combined with a pluggable interface system that enables various communication scenarios of JAVA with PROLOG, for instance via sockets or a foreign language interface. In addition, CAPJA provides a portable, built-in PROLOG interface that has been successfully tested with several open source systems like B-, CIAO-, GNU-, SWI-, TU-, XSB- and YAP-PROLOG. In PROLOG, we introduce the *Prolog-View-Notation* (PVN) to elegantly feed a generator producing JAVA classes that map to terms specified within PVN expressions.

2 Related Work

A well-known interface for JAVA and PROLOG is JPL [21] that is bundled with SWI-PROLOG [22]. To enable a fast communication JPL provides JAVA classes representing language artefacts of PROLOG like atoms or terms. JPL communicates directly with the core of SWI which is coded in C. Queries from JAVA do not have to be interpreted like native PROLOG code. This leads to fast execution times but requires much boilerplate code in JAVA. An integration 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. Complex queries in JAVA to PROLOG have to be coded manually by concatenating Strings containing the specific PROLOG code. But Strings usually do not benefit from auto-completion nor pre-compile time checking. Furthermore, JPL is limited to the use with SWI-PROLOG, as it is shipped and created for just this single PROLOG system although the concepts of its higher-level API can easily be adopted to interface other PROLOG systems.

Another interesting approach is INTERPROLOG [4] for XSB-PROLOG [19]. It uses JAVA's serialization mechanism in order to send serialized JAVA objects to PROLOG. Messages from JAVA are first interpreted as atoms and are further analysed in PROLOG by the use of definite clause grammars (DCG's). As a result there is a portion of PROLOG code that has to be ported in order to run INTERPROLOG with other PROLOG systems. On the JAVA side, the classes have to implement the `Serializable` interface and its members have to be serializable. If not, they have to be marked `transient` and their values are lost during serialization. As with JPL, queries in JAVA have to be created from Strings, too. On the PROLOG side, complex term structures have to be derived each time an object is transferred.

PROLOG CAFE [2] translates a PROLOG program into a JAVA program via the Warren Abstract Machine (WAM), and then compiles it using a standard JAVA compiler. PROLOG CAFE offers only a core PROLOG functionality, but lacks support for many PROLOG built–in predicates in the ISO-Standard. The creation of queries also is String based.

The concept of linguistic symbiosis has been used in [5, 8, 10] to define a suitable mapping of objects in JAVA to terms in PROLOG. Linguistic symbiosis targets on finding similar concepts of different paradigms in order to map them. Methods in JAVA are mapped to queries in PROLOG. The conversion is realised via meta programming techniques in JAVA, in particular by an extensive use of JAVA Annotation. The Annotations are processed during runtime using JAVA's Reflection API. The analysis via Reflections compared to static calls causes a not negligible overhead and with it a loss in performance. Castro et al. have their LOGICOBJECTS [5] compared to JPL. Their implementation has been slower than the corresponding JPL implementation by a factor of about 7. CAPJA, instead, competed well against JPL [17].

TUPROLOG [7], an engine entirely written in JAVA, has been integrated in [6] into JAVA programs using annotations and generics. Rules and facts are written directly into the JAVA code within annotations. The syntax for rules and facts is native PROLOG. This leads to a complex mixture of not only code in different programming languages but different programming paradigms, too. With manually customised methods PROLOG can be queried from JAVA. The mapping of input and return to arguments of a goal in PROLOG is defined by annotations. This approach is strongly dependent on TUPROLOG.

In [12], the JAVA virtual machine is extended 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 programs within JAVA. As the JAVA Virtual Machine (JVM) is changed, the portability of the JAVA program decreases. Every subsequent version of JAVA causes new porting effort. With regard to the integration into an already existing and running JAVA application a subsequent extension of the JVM proves often impossible. Another drawback is that only PROLOG libraries already in the extension included are supported.

3 Research Question and Goal

Bringing new software technology in an existing software project raises many questions that often result in restrictions and interoperability problems. Notably, there are: 1) external or environmental problems considering architectural assumptions, dependencies, extensive code modifications and maintenance and 2) internal or deployment problems that refer to the actual coding practice such as obscuration (boilerplate code), complexity and readability, IDE support and options for customisation and extension.

The main goal of our work is to introduce a Connector Architecture for PROLOG and JAVA (CAPJA) in order to realize a smooth cooperation of object oriented and logic programming. As an instantiation of our model we want to implement a framework that overcomes the mentioned interoperability problems and 1) has a concise, extensible and intuitive communication layer with a pluggable interface system, 2) is not limited to a certain PROLOG or operating system, 3) is well integrated in the development cycle, 4) allows for a seamless (subsequent) integration of PROLOG programs into JAVA applications and 5) operates with high performance during runtime.

4 Proposed Approach

To combine the two programming paradigms – object-oriented programming and logic programming – we propose an *Object-Term Mapping* (OTM). The default mapping is usable for JAVA classes without any modifications to their source code. The resulting data exchange format is a simple textual term representation. Because the textual term representation already conforms PROLOG's syntax, it can be directly called within PROLOG. In JAVA, the mapping is customisable by *Prolog-Views* that are defined as `@PlView` annotations. In PROLOG, we generate JAVA classes that correspond to given predicates. For this purpose, we introduce the *Prolog-View Notation* (PVN). Expressions in PVN are based on lists in PROLOG and describe predicates enriched by mapping information for JAVA. On the one hand, a PVN expression can be used to generate

`@PLView` annotations on a given class in JAVA. On the other hand, it is sufficient to generate a JAVA class that translate by the default to the PROLOG predicate that has been described in the PVN expression. In this way, we can control the mapping from the PROLOG side.

To decouple our connector architecture from a single PROLOG system we design a pluggable interface system based on a adapter pattern in JAVA. As built-in PROLOG interface, we provide 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 interaction with users, the `PPI` works for a broad range of PROLOG implementations.

A recently introduced new feature in JAVA are *Lambdas* [13]. Although intended only as syntactic sugar for anonymous classes containing only one method, we want to exploit Lambdas to express refined queries to Prolog.

We evaluate our approach with several case studies, in particular in the field of e-commerce, applied graph theory and logistics. In the case studies CAPJA helps to solve problems in commercial, real life scenarios. Knowledge about a problem domain is modelled declaratively in PROLOG reusing existing data structures of an associated JAVA application. The PROLOG knowledge base then is integrated in JAVA for queries.

5 Research Status and Preliminary Results

In [16], we have presented an early predecessor of CAPJA called PBR4J (PROLOG Business Rules for JAVA). PBR4J allows to request a given set of PROLOG rules from JAVA. To overcome the interoperability problems a JAVA Archive (JAR) has been generated containing the methods to query the rules. It uses XML Schema to describe the data exchange format. From the XML Schema description we have generated JAVA classes for the JAR. For CAPJA, we have subsequently improved the mapping mechanism in order to get rid of an intermediate, external layer for the data exchange format. While PBR4J enables with every generated JAR only a single, given PROLOG query, we are with CAPJA now much more flexible. The result of a request has been encapsulated in a result set that implements the `java.util.Iterator` interface with eager evaluation. Although very static, PBR4J has been used in a case study in [16] to successfully integrate a set of PROLOG business rules into a commercial JAVA e-commerce system.

In [17] the customisable mapping mechanism of CAPJA has been introduced. The following listing shows a `Person` class containing two different Prolog-Views.

```java
@PlView(viewId="V1", dropArgs={"familyName"},
  modifyArgs={@PlArg(valueOf="children",viewId="V2")})
@PlView(viewId="V2", functor="child",
  orderArgs={"firstName"})
class Person {
  private String firstName;
  private String familyName;
  private int age;
  private ArrayList<Person> children; /*...constr. & methods*/ }
```

First, we instantiate the persons `homer` and `bart`, and add `bart` as child of `homer`.

```
Person homer = new Person("Homer", null, 40);
Person bart = new Person("Bart", "Stimpson", 10);
homer.getChildren().add(bart);
```

Then, the we show in the listing below the textual term representations of `homer` as a result of the default mapping and the Prolog-View `"V1"` of the `Person` class.

```
% default mapping of homer
person('Homer', X, 40, [person('Bart',10,[])]).
% the Prolog-View "V1" of homer
person('Homer', 40, [child('Bart')]).
```

The next listing demonstrates the PVN expression that describes the `person/4` predicate. It is enriched by name and type information for the mapping to JAVA. Therefore, it is sufficient to generate the skeleton of the corresponding `Person` class without annotations, constructors and methods as these can be subsequently added in JAVA on demand. PVN expressions can also be used to generate `@PlView` annotations in order to control the mapping behaviour of given classes from PROLOG.

```
pl_view(person, compound, [pl_arg(firstName, string),
   pl_arg(familyName, string), pl_arg(age, int),
   pl_arg(children, arrayList, person)]).
```

As a first implementation we have combined in [17] JAVA with SWI-PROLOG [22] using it's Foreign Language Interface. To enable the mapping and the communication with PROLOG only a small JAR has to be integrated in JAVA. An evaluation of the performance has shown that it is as fast as JPL [21]. However, the implementation with CAPJA has proven simpler, cleaner and shorter as the reference with JPL. With CAPJA, we have saved 25% lines of code.

In [18] we have uncoupled CAPJA from any specific PROLOG system by introducing the Portable PROLOG Interface (`PPI`). It is based on standard streams that are part of every operating system and used by most PROLOG systems for the user interface. A case study in [18] verifies the applicability of the `PPI` with CAPJA for several PROLOG systems with decent performance. Without changing the underlying JAVA or PROLOG source code we have successfully tested with B-, CIAO-, GNU-, SWI-, TU-, XSB- and YAP-PROLOG.

Constructors of the generic `Unifier<T>` class in CAPJA offer an elegant way to specify a query from JAVA to PROLOG. A simple Prolog goal can be specified by a sole class instance as parameter for the constructor. Members of such an instance with value `null` are interpreted by the OTM as logical variables for PROLOG.

More complex goals can be encoded declaratively into JAVA Lambdas [13]. In order to decode the Lambdas correctly they have to be processed before being passed to the JAVA compiler. The developer marks them in the source file with an `@JPLambda` annotation. An annotation processor activates every time a Lambda has potentially changed. We use it to register the changed files in a build script which in turn triggers a component of CAPJA. As the following procedure changes an existing source file, an annotation processor alone proves to be inadequate. The initial source file is parsed yielding all contained class imports and Lambda expressions. A customisable transcoder decodes the Lambdas before translating the necessary classes describing terms with the OTM. Every part of a Lambda that cannot be translated to PROLOG is retained in their JAVA representation. The resulting PROLOG term strings and their corresponding untranslated JAVA code pieces form a method call containing the PROLOG goal. These are wrapped into custom, auto-generated `Unifier` subclasses that provide specific iterators for the result set. Finally, `Unifier` constructor calls in the original source file are replaced with references to the generated `Unifier` classes.

CAPJA encapsulates the complex process outlined above and offers a convenient, semantic oriented coding pattern that results in readable source.

```
Unifier<Person> personUnifier = new Unifier$3141<>(plEngine,
   protocol, person -> person.getFamilyName() == "Stimpson"
                    && person.getAge() < 18);
Iterator<Person> personIt1 = personUnifier.unify();
Iterator<Person> personIt2 = personUnifier.findall();
```

The example above asks a given PROLOG system `plEngine` via a given `protocol`, e.g. plain text or XML, for the persons in the family `Stimpson` with age less than 18. CAPJA has replaced a call for the constructor of `Unifier` in the original source file by the call for the constructor of the auto-generated class `Unifier$3141`. The methods `unify` and `findall` return specialized `Person` Iterators with lazy or eager evaluation of the result set, respectively.

6 Conclusion

In this paper we have presented CAPJA, an intuitive and portable Connector Architecture for PROLOG and JAVA. The most recent instantiation already complies with most of the goals as stated in Section 3. Many external and internal issues have been eliminated by design. CAPJA is applicable for several open-source PROLOG systems with at least decent performance. The pluggable interface system has been successfully tested with various types of interfaces. A significant reduction in lines of code in comparison to other approaches reflects the clarity and conciseness of our approach. A novel approach using Lambdas allows to construct elegantly readable PROLOG queries in JAVA.

However, tests for a cooperation with commercial PROLOG systems are still missing and a public release of CAPJA under an open-source license requires additional preparations. Furthermore, our main attention, so far, has been the direction of JAVA calling PROLOG. Currently, CAPJA is limited to support only queries from JAVA to PROLOG. Because the OTM is bidirectional, we suppose that the opposite direction can be handled with the help of PVN expressions and a JAVA message predicate in PROLOG, an approach similar to INTERPROLOG.

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. I. Bratko. *Prolog Programming for Artificial Intelligence* International Computer Science Series, 4th edition, Addison Wesley, 2011.
4. 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.
5. 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.
6. 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.
7. E. Denti, A. Omicini, A. Ricci. *tuProlog: A light-weight Prolog for Internet applications and infrastructures.* Practical Aspects of Declarative Languages. Springer Berlin Heidelberg, 2001. S. 184-198.
8. 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.
9. Eclipse Foundation. *Desktop IDEs.* `http://eclipse.org/ide`
10. 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.
11. JBoss Community. *Drools – The Business Logic Integration Platform.* `http://www.jboss.org/drools`
12. T. Majchrzak, H. Kuchen. *Logic java: combining object-oriented and logic programming.* Functional and Constraint Logic Programming. Springer Berlin Heidelberg, 2011. 122-137.
13. Oracle Corporation. *The Java Tutorials - Lambda Expressions.* `https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`
14. L. Ostermayer, D. Seipel. *Knowledge Engineering for Business Rules in Prolog.* Proc. Workshop on Logic Programming (WLP), 2012.
15. 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.
16. 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.
17. L. Ostermayer, F. Flederer, D. Seipel. *CAPJA - A Connector Architecture for Prolog and Java.* Proc. 10th Workshop on Knowledge Engineering and Software Engineering (KESE), 2014.
18. L. Ostermayer, F. Flederer, D. Seipel. *PPI - A Portable Prolog Interface for Java.* Proc. 28th Workshop on Logic Programming (WLP), 2014.
19. K. Sagonas, T. Swift, D. S. Warren. *XSB as an efficient deductive database engine.* ACM SIGMOD Record. ACM, 1994. S. 442-453.
20. D. Seipel, W. Wegstein *metaDictionary - Towards a Generic e-Infrastructure for Detecting Variance in Language by Exploiting Dictionary Information.* Proc. International Symposium on Grids and Clouds (ISGC), 2011.
21. P. Singleton, F. Dushin, J. Wielemaker. *JPL 3.0: A Bidirectional Prolog/Java Interface.* `http://www.swi-prolog.org/packages/jpl`
22. J. Wielemaker. *SWI-Prolog.* `http://www.swi-prolog.org`