A PROLOG Framework for Integrating Business Rules into JAVA Applications

Ludwig Ostermayer, Dietmar Seipel

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

Abstract. Business specifications – that formerly only supported IT development – increasingly become business configurations in the form of rules that can be loaded directly into IT solutions. PROLOG is well–known for its qualities in the development of sophisticated rule systems. It is desirable to combine the advantages of PROLOG with JAVA, since JAVA has become one of the most used programming languages in industry. However, experts of both programming languages are rare.

To overcome the resulting interoperability problems, we have developed a framework which generates a JAVA archive that provides methods to query a given set of PROLOG rules; it ensures that valid knowledge bases are transmitted between JAVA and PROLOG. We use XML Schema for describing the format for exchanging a knowledge base between PROLOG and JAVA. From the XML Schema desciption, we scaffold JAVA classes; the JAVA programmer can use them and fill in the open slots by statements accessing other JAVA data structures. The data structure on the JAVA side reflects the complex structured knowledge base, with which the PROLOG rules work, in an object–oriented way.

We can to some extend verify the correctness of the data set / knowledge base sent from JAVA to PROLOG using standard methods for XML Schema. Moreover, we can add constraints that go beyond XML. For instance, we can specify standard integrity constraints known from relational databases, such as primary key, foreign key, and not–null constraints. Since we are dealing with complex structured XML data, however, there can be far more general integrity constraints. These can be expressed by standard PROLOG rules, which can be evaluated on the PROLOG side; they could also be compiled to JAVA by available PROLOG to JAVA converters such as Prolog Cafe – since they will usually be written in a supported subset of PROLOG.

We have used our framework for integrating PROLOG business rules into a commercial E–Commerce system written in JAVA.

Keywords. Business Rules, Logic Programming, PROLOG, JAVA.

1 Introduction

PROLOG is well–known for its qualities in rapid prototyping and agile software development, and for building expert systems. In this paper we present an approach that

allows to integrate PROLOG rules seamlessly into JAVA applications. We could largely automate the integration process with our framework PBR4J (PROLOG Business Rules for JAVA). PBR4J uses XML Schema documents, from which it generates (scaffolds) JAVA classes containing the information necessary for utilizing the business rules. The business rules are accessed from JAVA simply by invoking the generated JAVA methods. From the JAVA point of view, the fact that a set of PROLOG rules is requested is hidden. The derived facts can be accessed as a result set by JAVA getter methods. In terms of Domain Specific Languages (DSL) [8], we use PROLOG as an external DSL for expressing rules. Thus, our approach enables a clean separation between a JAVA application and the business logic, and applications can benefit from the strengths of both programming languages.

There exists the following *related work*. We have already discussed the usage of DROOLS [6], a popular JAVA tool for business rules development, and the advantages of knowledge engineering for business rules in PROLOG [11, 12]. There are several solutions for a communication between JAVA and PROLOG, for instance the bidirectional PROLOG/JAVA interface JPL [17] that combines certain C functions and JAVA classes. On the JAVA side, JPL uses the JAVA Native Interface (JNI), on the PROLOG side it uses the PROLOG Foreign Language Interface (FLI). When working with JPL, one has to create rather complex query strings and explicitly construct term structures prior to querying. Slower in performance than JPL, INTERPROLOG [5] provides a direct mapping from JAVA objects to PROLOG terms, and vice versa. 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 a core PROLOG functionality, but it lacks support for many PROLOG built–in predicates from the ISO standard.

However, the challenge of our work was not to develop another interface between JAVA and PROLOG, but to simplify the access to the PROLOG rules and data structures in JAVA. We did not mix PROLOG and JAVA syntax for querying the PROLOG rules in JAVA. Rules can be developed independently from JAVA, and our framwork ensures only valid calls from JAVA to the PROLOG rules. We just write the rules in PROLOG and use PBR4J to generate JAVA classes; request and result handling are encapsulated in standard JAVA objects. Therefore in JAVA, the flavour of programming is unchanged. On the other side, the easy–to–handle term structures and the powerful meta–predicates of PROLOG can be used to develop sophisticated rule systems. Furthermore, using PROLOG's parsing techniques (DCGs) and infix operators, the rule syntax can be largely adapted to a natural language level, which simplifies the rule creation process and improves the readability of the rules. In particular, this is important to bridge the gap between software engineers and business analysts without programming background.

The structure of this paper is as follows. Section 2 presents a set of business rules written in PROLOG, which will serve as a running example. In Section 3, we describe our framework: first, we represent a knowledge base in XML and generate a corresponding XML Schema. Then, we generate JAVA classes from the XML schema. In Section 4, we give an example of a JAVA call to the business rules in PROLOG. Finally, we summarize our work in Section 5.

2 Business Rules in PROLOG

In the following, we present a set of business rules in PROLOG, that is part of a real commercial Enterprise Resource Planning (ERP) system for online merchants. The purpose of the business rules is to infer financial key data and costs in a given E–Commerce scenario that is dealing with articles, online shopping platforms, shipping parameters, and various other parameters. The derived data support the business intelligence module of the application, which is implemented in JAVA.

Due to space restrictions, we present only a simplified version of the original set of business rules used in the application. We focus on a constellation consisting of order, platform and shipment charges. For every shipment, taxes have to be paid according to the country of dispatch. In our example, the inferred financial key data are gross margin, contribution margin and profit ratio. First, we describe the input data format necessary for a valid request, then we take a closer look at the business rules. Finally, we explain how to describe relationships between facts in a knowledge base and how to check them in PROLOG.

2.1 The Knowledge Base

The input knowledge base consists of global data and orders. A PROLOG fact of the form `tax(Country, Rate)` describes the purchase tax rate of a country. The PROLOG facts of the form `platform_charges(Category, Commission, Discount)` describe the different commissions that online shopping platforms charge according to article categories and merchants discount [7]. A PROLOG fact of the form `shipping_charges(Country, Logistician, Charges)` shows the price that a logistician charges for a shipment to a given country.

Listing 1.1: Global Data

```
tax('Germany', 0.190).
platform_charges('Books', 0.11, 0.05).
shipping_charges('Germany', 'Fast Logistics', 4.10).
```

An order is a complex data structure – represented by a PROLOG term – consisting of an article, the country of dispatch, and the used logistician. An article is a data structure relating a category and the prices (in Euro), i.e., base price and market price; every article has a unique identifier EAN (European Article Number; usually 13 digits, but we use only 5 digits in this paper).

Listing 1.2: An Order

```
order( article('98765', 'Books', prices(29.00, 59.99)),
   'Germany', 'Fast Logistics' ).
```

2.2 The Rule Base

The following business rule demonstrates the readability and compactness offered by PROLOG. Join conditions can be formulated easily by common variable symbols, and

the term notation offers a convenient access to objects and subobjects in PROLOG; more than one component can be accessed in a single line. Usually, If–Then–Else statements with many alternatives are hard to review in JAVA, but much easier to write and read in PROLOG. Due to the rules approach, multiple results are inferred implicitly; in a DATALOG style evaluation, there is no need to explicitly encode a loop. In a PROLOG style evaluation, all results can be derived using the meta–predicate `findall/3`.

Using the PROLOG package DATALOG* [14] from the DISLOG Developers' Kit (DDK), we can, e.g., support the development phase in PROLOG by visualizing the rule execution with proof trees [11]. DATALOG* allows for a larger set of connectives (including conjunction and disjunction), for function symbols, and for stratified PROLOG meta–predicates (including aggregation and default negation) in rule bodies.

The main predicate in the business rule base computes the financial key data for a single order. The facts of the input knowledge base will be provided by the JAVA application, as we will see later. Derived `financial_key_data/2` facts are collected in a PROLOG list, which will be presented as a result set to JAVA.

Listing 1.3: Business Rules for Financial Key Data

```
financial_key_data(Order, Profits) :-
   order_to_charges(Order, Charges),
   Order = order(article(_, _, prices(Base, Market)), _, _),
   Charges = charges(Shipping, Netto, Fees).
   Gross_Profit is Netto - Base,
   C_Margin is Gross_Profit - Fees - Shipping,
   Profit_Ratio is C_Margin / Market,
   Profits = profits(Gross_Profit, C_Margin, Profit_Ratio).

order_to_charges(Order, Charges) :-
   Order = order(Article, Country, Logistician),
   Article = article(_, Category, prices(_, Market)),
   call(Order),
   tax(Country, Tax_Rate),
   shipping_charges(Country, Logistician, Charges),
   Shipping is Charges / (1 + Tax_Rate),
   Netto is Market / (1 + Tax_Rate),
   platform_charges(Category, Commission, Discount),
   Fees is Market * Commission * (1 - Discount),
   Charges = charges(Shipping, Netto, Fees).
```

The predicate `order_to_charges/4` first computes the charges for the shipment, then an article's netto price using the tax rate of the country of dispatch, and finally the fees for selling an article on the online platform in a given category. We use the PROLOG terms `Order`, `Profits`, and `Charges` to keep the argument lists of the rule heads short. E.g., `order_to_charges/4` extracts the components of `Order` in line 2 and calls the term `Order` in line 4. Thus, we can avoid writing the term `Order` repeatedly – in the head and in the call. In the code, we can see nicely, which components of `Order` are used in which rule, since the other components are labeled by underscore variables.

2.3 Constraints in PROLOG

In knowledge bases, facts often reference each other. E.g., in our business rules application, we have the following foreign key constraints: for every `order/3` fact, there must exist corresponding facts for `tax/2` and `shipping_charges/4`, whose attribute values for `Country` match the attribute value for `Country` in `order/3`. The same holds for `category` in `platform_charges/3` and `category` in `order/3`. Another frequently occuring type of constraints are restrictions on argument values; e.g., the values for `Country` could be limited to countries of the European Union.

This meta information between facts in a knowledge base usually remains hidden; the developer of the rule set knows these constraints, and only sometimes they are easy to identify within the set of business rules. For validation purposes of knowledge bases, however, this information is crucial, in particular when a knowledge base for a request is arranged by a programmer other than the creator of the set of rules.

Constraints, such as the foreign key constraints from above, can simply be specified and tested in PROLOG. The execution of the PROLOG predicate `constraint/1` is controlled using meta–predicates for exception handling from SWI PROLOG. With `print_message/2`, a meaningsful error message can be generated, and exceptions can be caught with `catch/3`. In Listing 1.4, the foreign key constraints on `Country` and `Category` are checked.

We can also represent standard relational constraints in XML. XML representations for `create table` statements have been developed and used in [3, 16]. Thus the knowledge base – including the constraints – can be represented in XML.

Listing 1.4: Foreign Key Constraints

```
constraint(fk(shipping_charges)) :-
   forall( shipping_charges(Country, _, _),
      tax(Country, _) ).

constraint(fk(article_charges)) :-
   forall( article(_, Category, _),
      platform_charges(Category, _, _) ).
```

3 Integration of PROLOG Business Rules into JAVA

The workflow of PBR4J follows three steps, cf. Figure 1. First, PBR4J extracts an XML Schema description for the knowledge base and the result set of a given set of PROLOG rules. Then, the user must extend the extracted XML Schema by names for atomic arguments, numbers and strings from PROLOG and review the type description. Finally, PBR4J uses the XML Schema to generate JAVA classes and packs the generated classes into a JAVA Archieve (JAR). After embedding the JAR into the JAVA application, the set of PROLOG rules can be called from JAVA. The facts derived in PROLOG are sent back to JAVA, where they are parsed; then, they can be accessed by the generated classes.

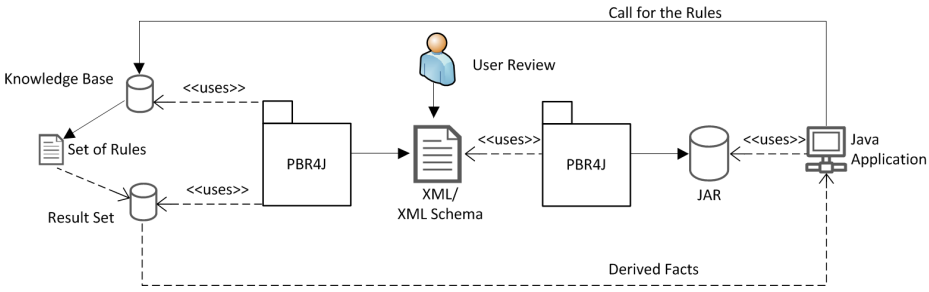

Fig. 1: Workflow of PBR4J

In the following, we describe the transformation of the knowledge base to an XML representation, from which we subsequently extract the XML Schema. Then we show that the JAVA classes generated from the XML Schema reflect the complex structured knowledge base in an object–oriented way. The result set is handled in a similar manner; thus, we describe only the transformation of the knowledge base and omit further processing details for the result set.

3.1 An XML Schema for the Knowledge Base

XML is a well–known standard for representing and exchanging complex structured data. It allows for representing PROLOG terms and improves the interoperability between PROLOG and JAVA programs, since XML is easy to read. We extract an XML Schema from the XML representation of the knowledge base, and we generate JAVA classes from the extracted XML Schema.

We use the predicate `prolog_term_to_xml(+Term, -Xml)` for the transformation of a PROLOG term to XML. Listing 1.5 shows the XML representation for the PROLOG term with the predicate symbol `order/3`. Notice the XML attribute `type` and the names of elements representing arguments of complex terms on the PROLOG side.

Listing 1.5: An Order in XML Format

```
<order type="class">
   <country type="string">Germany</country>
   <logistician type="string">Fast Logistics</logistician>
   <article type="class">
      <ean type="integer">98765</ean>
      <category type="string">Books</category>
      <prices type="class">
         <base type="decimal">29.00</base>
         <market type="decimal">59.99</market>
      </prices>
   </article>
</order>
```

These are necessary, because JAVA is a typed language, whereas PROLOG builds data structures from a few basic data types. The representation for class attributes in JAVA is a typed `Name="Value"` pair. In order to map the knowledge base from PROLOG to JAVA, we must give names to arguments of PROLOG facts, if they are atomic, numbers, or strings, and we must add a type information. The functor of a complex PROLOG term is mapped to the tag of an element with `type="class"`. The structure of the XML representation easily can be generated from the PROLOG term structure, and some of the type information can be inferred automatically from the basic PROLOG data types. But, type preferences and meaningful names for atoms, numbers, and strings must be inserted manually.

From the XML representation of the knowledge base and the result set, we can extract a describing XML Schema using PROLOG. The XML Schema is a natural way to describe and to define the complex data structure. Known techniques are available for validating the XML representation of the knowledge base w.r.t. the XML Schema. Listing 1.6 shows the description of an `order/3` term in XML Schema. The XML Schema of the knowledge base can contain further information in attributes like `minOccurs` and `maxOccurs`.

Listing 1.6: Fragment of the XML Schema describing `order/3`

```
<xsd:element name="order" type="order_Type"
   minOccurs="1" maxOccurs="unbounded" />

<xsd:complexType name="order_Type">
   <xsd:sequence>
      <xsd:element name="article" type="article_Type" />
      <xsd:element name="country" type="xsd:string" />
      <xsd:element name="logistician" type="xsd:string" />
   </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="article_Type">
   <xsd:sequence>
      <xsd:element name="ean" type="xsd:integer" />
      <xsd:element name="category" type="xsd:string" />
      <xsd:element name="prices" type="prices_Type" />
   </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="prices_Type">
   <xsd:sequence>
      <xsd:element name="base" type="xsd:decimal" />
      <xsd:element name="market" type="xsd:decimal" />
   </xsd:sequence>
</xsd:complexType>
```

3.2 Scaffolding of JAVA Code

From the XML Schema, we generate JAVA classes using the PROLOG–based XML transformation language FNTRANSFORM [13]. FNTRANSFORM offers recursive transformations of XML elements using a rule formalism similar to – but more powerful than – XSLT. Every `xsd:element` in the schema with a complex type will be mapped to a JAVA class. Child elements with simple content are mapped to attributes. Figure 2 shows a fragment of the UML diagram for the generated classes.

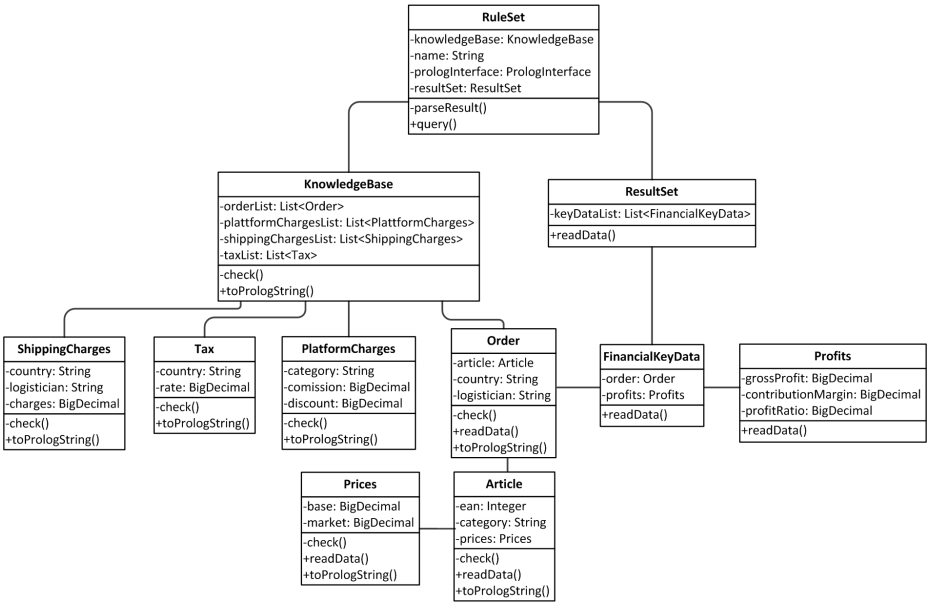

Fig. 2: Generated Classes

All classes associated with the class `KnowledgeBase` implement the methods `check` and `toPrologString`. An example of the method `toPrologString` of the generated class `Order` is shown in Listing 1.7. A recursive call of `check` controls that all necessary input data are set before the method `toPrologString` is called to build a knowledge base in a string format, which can be parsed easily by PROLOG using the predicate `string_to_atom/2`. The transformation to a PROLOG term can be achieved by `atom_to_term/3`.

Parts of the generated class `RuleSet` are shown in Listing 1.8. The method `query` sends a PROLOG goal together with a knowledge base in string format from JAVA to PROLOG. As a default interface between JAVA and PROLOG, we have implemented a simple connection with a communication layer based on standard TCP/IP sockets. Other interfaces can be implemented and set as the value for the attribute `prolog-Interface` of the class `RuleSet`. The default interface is represented by the class `PrologInterface`, which is fix and not generated every time a set of PROLOG rules

is integrated into a given JAVA application via PBR4J. The class `PrologInterface` must be integrated into the JAVA application once, and it must be accessible for all generated classes of the type `RuleSet`.

Listing 1.7: `toPrologString` in `Order`

```
public String toPrologString() {
   this.check();
   StringBuilder sb = new StringBuilder();
   sb.append( "order" + "(" +
      this.article.toPrologString() + ", "
      "'" + this.getCountry() + "'" + ", "
      "'" + this.getLogistician() + "'" + ")" );
   return sb.toString();
}
```

The result set that is sent back from PROLOG to JAVA is parsed by the method `parseResult` of the class `RuleSet`. As for the class `PrologInterface`, the class `PrologParser` is not generated and must be integrated into the JAVA application once and be accessible for all generated classes of the type `RuleSet`. The method `parseProlog` of `PrologParser` saves the content of the string returned from PROLOG in a structured way to a hashmap. The hashmap than can be further processed efficiently by the method `readData` that all classes associated with the class `ResultSet` must implement. The method `readData` analyses the hashmap and fills the data list of the class `ResultSet`.

Listing 1.8: The Class `RuleSet`

```
package pbr4j.financial_key_data;

public class RuleSet {
   private PrologInterface prologInterface = null;
   private String name = "financial_key_data";
   private KnowledgeBase knowledgeBase = null;
   private ResultSet resultSet = null;
   // ... code that we omit...
   private void parseResponse(String prologString) {
      DataList data = PrologParser.parseProlog(prologString);
      this.resultSet = new ResultSet();
      this.resultSet.readData(data); }
   // ... code that we omit...
   private void query(KnowledgeBase kb) {
      if (prologInterface == null) {
         this.setDefaultInterface(); }
      String response = prologInterface.callProlog(
         this.name, kb.toPrologString());
      this.parseResponse(response); }
   // ... code that we omit...
}
```

All generated classes are organised in a namespace via a JAVA package. The package access protection ensures that the class `RuleSet` can only contain a `KnowledgeBase` from the same package. The package can be stored in a JAVA Archive (JAR) – a compressed file that can not be changed manually. This creates an intentional generation gap, cf. Fowler [8]. The JAR file can be embedded into any JAVA application easily, and all classes in the JAR become fully available to the JAVA developers.

4 A JAVA Call to the Business Rules

In the following, we will give a short example of a request to a set of PROLOG rules using the classes generated with PBR4J. Listing 1.9 shows a test call from JAVA to the set of business rules described in Section 2. We omit object initialisation details, but we assume that the necessary objects for a successful call are provided. For improving the readability of the result of the call, we assume that all classes associated with the class `ResultSet` implement the method `toPrologString`.

Listing 1.9: A JAVA Call to the Business Rules

```
import pbr4j.financial_key_data.*;

public class TestCall {

   public static void main(String[] args) {
      RuleSet rules = new RuleSet();
      KnowledgeBase kb = new KnowledgeBase();
      // ... filling the knowledge base with data ...
      rules.query(kb);
      ListIterator<Object> it =
         rules.getResultSet().listIterator();
      while (it.hasNext()) {
         System.out.println(it.next().toPrologString() + ".");
      } } }
```

It is not visible in JAVA that a request is made to a rule set written in PROLOG. Running the JAVA code from above creates the system output shown in Listing 1.10; we have added some newlines to improve readability. The first fact is derived from the data described in Subsection 2.1. The second fact is derived from another order of the same article, that is shipped to France. Charges for the shipment to a foreign country are higher, and the tax rate of France is 0.196, which explains the slightly lower argument values of `profits/3`.

Listing 1.10: order Result Set

```
financial_key_data(
   order( article('98765', 'Books', prices(29.00, 59.99)),
      'Germany', 'Fast Logistics' ),
   profits(21.41, 11.70, 0.195) ).

financial_key_data(
```

```
order( article('98765', 'Books', prices(29.00, 59.99)),
   'France', 'Fast Logistics' ),
profits(21.16, 7.70, 0.128) ).
```

5 Conclusions

We have presented a largely automatic approach for integrating a set of PROLOG rules seamlessly into JAVA applications. XML Schema is used for specifying the XML format for exchanging a knowledge base and a result set, respectively, between PROLOG and JAVA.

On the PROLOG side, we use a generic mapping from the PROLOG representation of the knowledge base and the result set to an XML representation enriched by data type information and names for atoms or numbers, and we extract a describing XML Schema from the XML representation. On the JAVA side, we scaffold JAVA classes from the XML Schema, that reflect the complex structured PROLOG terms in an object–oriented way. Accessing a set of rules from JAVA is simply done by invoking the JAVA methods of the generated classes without programming PROLOG or creating complex query strings.

We have illustrated our approach using a set of business rules that we have already integrated with our framework into a commercial E–Commerce system written in JAVA.

Acknowledgement. We acknowledge the support of the Trinodis GmbH.

References

1. S. Abiteboul, P. Bunemann, D. Suciu: *Data on the Web – From Relations to Semi–Structured Data and* XML*,* Morgan Kaufmann, 2000.
2. M. Banbara, N. Tamura, K. Inoue.: *Prolog Cafe: A Prolog to Java Translator,* Proc. Intl. Conf. on Applications of Declarative Programming and Knowledge Management (INAP) 2005, Springer, LNAI 4369.
3. A. Böhm, D. Seipel, A. Sickmann, M. Wetzka: *Squash: A Tool for Designing, Analyzing and Refactoring Relational Database Applications.* Proc. Intl. Conf. on Applications of Declarative Programming and Knowledge Management (INAP) 2007, Springer, LNAI 5437.
4. H. Boley: *The Rule Markup Language:* RDF–XML *Data Model,* XML *Schema Hierarchy, and* XSL *Transformations.* Proc. Intl. Conf. on Applications of Declarative Programming and Knowledge Management (INAP) 2001, Springer, LNAI 2543.
5. M. Calejo: *InterProlog: Towards a Declarative Embedding of Logic Programming in Java,* Proc. 9th European Conference on Logics in Artificial Intelligence, JELIA, 2004.
6. *Drools – The Business Logic Integration Platform.* `http://www.jboss.org/drools/`.
7. *Ebay Seller Fees.* `http://pages.ebay.de/help/sell/seller-fees.html`.
8. M. Fowler. *Domain–Specific Languages.* Addison–Wesley, 2011.
9. E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns. Elements of Reusable Object–Oriented Software.* Addison–Wesley Longman, 2010.
10. B. v. Halle. *Business Rules Applied.* Wiley, 2002.
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. D. Seipel. *Processing* XML*–Documents in Prolog.* Proc. 17th Workshop on Logic Programming (WLP), 2002.
14. D. Seipel. *Practical Applications of Extended Deductive Databases in* DATALOG*. Proc. Workshop on Logic Programming (WLP) 2009.
15. D. Seipel. *The* DISLOG *Developers' Kit (*DDK*).* `http://www1.informatik.uni-wuerzburg.de/database/DisLog/`
16. D. Seipel, A. Boehm, M. Fröhlich: *Jsquash: Source Code Analysis of Embedded Database Applications for Determining* SQL *Statements.* Proc. Intl. Conf. on Applications of Declarative Programming and Knowledge Management (INAP) 2009, Springer, LNAI 6547.
17. P. Singleton, F. Dushin, J. Wielemaker: *JPL: A Bidirectional Prolog/Java Interface* `http://www.swi-prolog.org/packages/jpl/`, 2004.
18. J. Wielemaker. SWI PROLOG *Reference Manual* `http://www.swi-prolog.org/pldoc/refman/`