Knowledge Engineering for Business Rules in PROLOG

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. To bridge the gap between business analysts and developers, most companies nowadays rely on *business process management* (BPM). When controlling progressively more complex business processes, the connection between *business rules* and BPM gains a tremendous importance.
Currently, the most popular BPM tools are JBPM, an extensible workflow engine written in pure JAVA, and DROOLS, an expert system framework, that uses rules for knowledge representation. It is possible to combine JBPM and DROOLS. However, there are still flaws with the verfication and the update of knowledge bases, and in particular with building more komplex knowledge structures.
In this paper, we want to propose an alternative *knowledge engineering* approach for building, updating and testing complex, structured sets of business rules using SWI PROLOG, and we illustrate it by an example in the field of e–commerce.

Keywords. Knowledge Engineering, Logic Programming, Business Rules, Business Process Management.

1 Introduction

Business rules are statements that define or restrict certain aspects of a business process [6]. They model business structures, and they control the behavior of business processes. Business rules can be applied to persons, processes, business behavior and computer systems in companies. While business rules can occur purely informally, the careful, unambiguous, and consistent formulation of the rules is particularly important. Thus, cost–intensive misconceptions can be avoided, the communication can be improved, the legal regulations are obeyed, and the customer satisfaction is improved. Due to the inreasing complexity of system development, a communication problem arises between developers and business analysts. Both need a common language for the problem domain. In fact, the business analysts should have the possibility to create process logic specifications on a formal, abstract level, which can then be processed by the developers (software engineers).

Currently the most popular tool for *business process management* [1] is JBPM, an extensible workflow engine written purely in JAVA, and the most popular tool for business rules is DROOLS, an expert system framework, that uses rules for knowledge representation. It is possible to combine JBPM and DROOLS. However, there are still flaws with the verification and the update of knowledge bases, and in particular with building more complex knowledge structures.

Knowledge engineering for business rules has some specialties coming from the fact that the business analyst and the customer should be able to understand and refine – some of – the declarative specification. In this paper, we propose using logic programming in SWI PROLOG for implementing business rules. For illustration, we use a simple example out of a large e–commerce system, which has originally been developed in JAVA, and which we are reimplementing currently using logic programming. Especially the implementation of the business logic was overly complicated, irreproducible, and difficult to maintain; changing a rule in the original system was a very difficult task. The first step was breaking down the original program logic into clear, formal and understandable rules. As a result, we present the aforementioned example in Section 3.

There already exist *program development tools* for PROLOG– or DATALOG–based systems, cf., e.g., [12, 18]. We have also developed concepts for analyzing rule bases and components for integrated development environments: [14] investigates the analysis and refactoring of logic programs, [2] presents anomaly tests for rules and ontologies in the semantic web rules language SWRL. We would like to transfer these concepts to business rules and extend them to a more general framework based on DROOLS, PROLOG, JAVA, etc. For exchanging data between PROLOG and standard business rule systems, there exist XML–based formats.

Recently, *domain specific languages* (DSL) [9] have proven to be useful for supporting knowledge engineering with business rules. Techniques from logic programming, such as meta–programming, have been used for a long time to support abstraction. Building DSLs with logic programming can be much simpler than the standard approach, where compilers have to be developed. We think that the syntax and the semantics of PROLOG are well–suited for developing an internal DSL. We hope to achieve this goal in an ongoing project (using, e.g., infix notation with specially defined operators), and we want to additionally support this by a graphical user interface.

The rest of this paper is organized as follows: In Section 2, we report some related work on business rules systems and business process modeling. In Section 3, we illustrate our approach using an example from e–commerce. In Section 4, we describe a DATALOG–like, stratified bottom–up evaluation of business rules; we have developed a visualization by proof trees, whose generation can be configured flexibly by the developer. Section 5 summarizes important concepts of knowledge engineering with business rules – some are already included in our approach, others are ideas for future work.

2 Business Process Modelling

Detailed knowledge about the business logic is essential for a company. This may sound obvious, but most IT systems have been developed over a long period of time, and parts of the program that reflect business processes are hidden behind thousands of lines of code.

Refactoring and updating business logic or even understanding how the processes are modeled may be a challenge. This challenge could be solved by business process modeling with JBPM and by business rules systems such as DROOLS.

2.1 JAVA Business Process Modelling (JBPM)

JAVA business process modeling (JBPM) is based on BPMN 2.0 , which is an OMG standard [1] for modeling business processes; it uses the JBoss process definition language JPDL to define process definitions in files. JPDL is a graphic–oriented programming language. The modeling uses nodes, transitions, and actions. Each node has a type defining its runtime behavior. During the flow of a process, the nodes trigger commands that are executed as the nodes are reached. The flow of a process is directed by transitions. Actions perform the logic, when a transition event or a node is reached in a process.

The JPDL process engine runs through a directed process graph. It usually consists of nodes, transitions, one start state, and one end state. The process graph represents the process definitions in a clear manner. Thus, traceability is always given.

In our project we paid special attention to *decision nodes*. When a decision node is reached in a process, the action is that a request – in an XML format – for a decision is sent to a server. The server is implemented in SWI PROLOG, and it uses the HTTP protocol [19]. Thus, we can easily communicate between SWI PROLOG and JBPM and transfer the business logic to PROLOG. In parallel, we also investigate JAVA–based frameworks for evaluating business rules. Moreover, frameworks blending logic programming with JAVA as well as Eclipse Plugins for logic programming might be helpful [12, 18].

2.2 The Business Rules System DROOLS

The JAVA–based expert system DROOLS has been developed by the JBOSS community. Its core module *Expert* provides the rule engine. For efficiently matching rules and facts, an enhanced implementation of the Rete algorithm [8] is used, the object–oriented Rete algorithm. Knowledge representation in DROOLS uses rules and decision tables. The special module *Flow* provides an extensible workflow engine.

However, as a non–standardized solution, DROOLS comes with some flaws and deficiencies [11]. The structure of the knowledge representation is still flat. In Version 5, new nodes and timers have been added, but the problem of a non–existing rule hierarchy persists. Since Version 5.4.0, the conversion of the DROOLS rule language to XML is no longer supported [10]. Furthermore, DROOLS has no tools for fully supporting the *knowledge engineering process;* in particular, there is no automated identification of rule anomalies, such as redundant or contradictory rules. Finally, the strong language and platform dependency makes a broader applicability difficult.

3 Case Study: Taxes in International E–Commerce

Electronic commerce (E–Commerce) has experienced an accelerated growth in recent years. Figure 1 shows the relative number of individuals in the EU–27 ordering goods or services over the internet in the 12 months prior to the survey in 2010. For keeping the accountancy transparent and traceable (24 hours a day, 7 days a week), an automated approach is indispensable. Rule–based systems can support the bookkeeping by automated decision making.

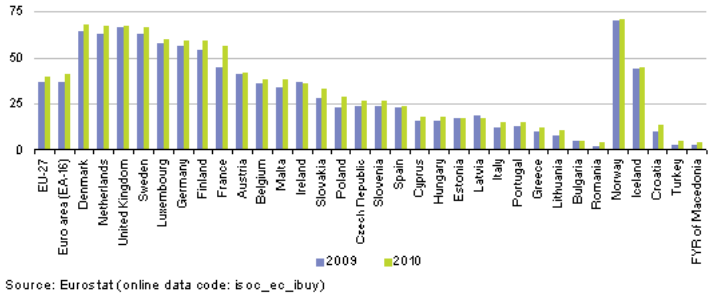

Fig. 1: Growth in E–Commerce

The Delivery Threshold. In the European Union, the sales tax for shipments to foreign countries has to be paid in the country where the shipment starts (home country). However, the sales tax has to be paid to the country of dispatch [4], as soon as the accumulated shipments abroad exceed a certain net merchandise value per year. We call this limit the *delivery threshold*; it is, e.g., 100000 € for shipments from Germany to France.

Assume that we have a credit transfer on our banking account that results from a shipment to a foreign country. The legal text determining the accountancy for paying the sales tax can be simplified and translated into two rules:

1. *When* the delivery threshold has been exceeded last year or previously this year, *then* the sales tax has to be paid in the country of dispatch. For this years calculation, we include a preliminary profit of the currently processed invoice. To determine this preliminary profit, the sales tax of the home country has to be used.
2. *When* the first rule does not fire, *then* the sales tax has to be paid in the home country.

It is a difficult task to arrange rules clearly and precisely. In the underlying law texts, four rules are used with redundant conditions in the rule body. Formulating rules without ambiguity and redundancy is very essential for knowledge representation.

In the following, we will first describe the XML data formats for the input and the output, and then we will illustrate the PROLOG implementation of our computations. Internally, the program consists of basic predicates that reflect the circumstances of the business case, and a business rule part with the program logic. At any time, only a single invoice is processed.

3.1 Input and Output in XML

In our simplified example, only the sales amount, the kind of taxation (e.g., food or non–food) and the country of dispatch are of interest. In reality, for every combination of taxation and country, there exists an account with a unique account number to guarantee consistent traceability.

For an already paid invoice, we have to compute the resulting bookings for a company's internal accountancy. We assume that the invoice is in XML format with various invoice positions that are classified by `line`, tax `type` and `total` amount, e.g.:

```
<invoice country="France" year="2012">
   <position line="1" type="food" total="211.00"/>
   <position line="2" type="nonfood" total="119.60"/>
</invoice>
```

The resulting bookings will be represented in an enriched XML format. Before we describe how the program works, we want to show the resulting bookings in XML. All amounts were rounded to a full cent amount:

```
<booking id="123" country="France" year="2012">
   <position line="1" type="food" total="211.00">
      <profit country="France" amount="200.00"/>
      <taxes  country="France" amount="11.00"/>
   </position>
   <position line="2" type="nonfood" total="119.60">
      <profit country="France" amount="100.00"/>
      <taxes  country="France" amount="19.60"/>
   </position>
</booking>
```

Using the above mentioned tax rules, we split the total amount of an invoice position into two parts: `profit` and `taxes`. Each of them specifies the country and the corresponding amount of money.

3.2 Basic Predicates

There are two groups of basic predicates: the first one describes the currently processed invoice and the current status for last year and this year. Note that every invoice is represented by its own fact base. The second group provides the configuration of the system. We show in our example only the necessary part of the configuration.

Predicates for Representing the Invoices. The facts for `annual_sales_so_far/4` provide the business volume for a country in a given year. Only these facts change after the processing of an invoice, since the business volume for the current year must be updated. The facts for `invoice/2` and `position/3` are extracted from the XML document for the invoice. In our example, Germany is the home country.

```
% annual_sales_so_far(Destination, Year, Total)
annual_sales_so_far('France', 2011, 92300).
annual_sales_so_far('France', 2012, 99800).
```

```
% invoice(Destination, Year)
invoice('France', 2012).

% position(Id, Type, Total).
position(1, food,    211.00).
position(2, nonfood, 119.60).

% home_country(Country)
home_country('Germany').
```

Predicates for the Configuration. The facts for `tax/3` provide the information about the taxes for a given country, i.e., the tax rates for the different tax types. The facts for `delivery_threshold/2` specify the delivery threshold for the different countries; for our current invoice, only France is of interest.

```
% tax(Country, Type, Tax_Rate)
tax('France',     food, 0.055).
tax('France',  nonfood, 0.196).
tax('Germany',    food, 0.070).
tax('Germany', nonfood, 0.190).

% delivery_threshold(Destination, Threshold)
delivery_threshold('France', 100000).
```

3.3 Business Rules

The rule for `annual_sales/3` aggregates the business volume for the current year including the recent invoice using the predicate `ddbase_aggregate/3` from the DISLOG Developers' Kit (DDK) [17]. We have to add the business volume given by `annual_sales_so_far/3` to the net value of the current invoice, which sums up the net values of its positions. For obtaining these net values, the tax rate of the home country is used, which is given by `tax/3`.

```
annual_sales(Destination, Year, Total) :-
   ddbase_aggregate( [Destination, Year, sum(Value)],
      ( annual_sales_so_far(Destination, Year, Value)
      ; invoice_position(Destination, Year, Value) ),
      Tuples ),
   member([Destination, Year, Total], Tuples).

invoice_position(Destination, Year, Net_Value) :-
   invoice(Destination, Year),
   position(_, Type, Total),
   home_country(Home_Country),
   tax(Home_Country, Type, T),
   Net_Value is Total/(1+T).
```

The predicate `tax_country/1` implements the two business rules for the delivery threshold: it determines the country where the taxes have to be paid.

```
tax_country(Tax_Country) :-
   invoice(Destination, Year),
   delivery_threshold(Destination, Threshold),
   ( ( Y is Year - 1 ; Y is Year ),
     annual_sales(Destination, Y, Total),
     Total > Threshold ->
     Tax_Country = Destination
   ; home_country(Home_Country),
     Tax_Country = Home_Country ).
```

Knowing the tax country, the rule for `booking_position/5` calculates the correct amount for every position and the modes `profit` and `taxes`. The derived head atoms can easily be transformed to an XML document of the form shown above.

```
booking_position(Line, Type, Mode, Country, Amount) :-
   invoice(Destination, _),
   position(Line, Type, Value),
   tax_country(Tax_Country),
   tax(Tax_Country, Type, T),
   ( Mode = profit, Country = Destination,
     Amount is Value/(1+T)
   ; Mode = taxes, Country = Tax_Country,
     Amount is Value*T/(1+T) ).
```

The business rules are evaluated bottom–up in a stratified manner [5]. The predicate `invoice_position/3` has to be evaluated before `annual_sales/3`; this precedency is due to the meta–predicate `ddbase_aggregate/3` for aggregation. Another well–known meta–predicate that requires stratification in deductive databases is default negation `not/1`. In principle, the other rules could be evaluated in a simultaneous bottom–up iteration – in our system, however, they are evaluated subsequently.

Some of the used predicates are deterministic for a given input. The all–results–inference–capability is especially essential for the two non–deterministic predicates `invoice_position/3` and `booking_position/5`. In JAVA–based systems such as DROOLS, it is much more complicated to perform such computations – this turned out when we compared the PROLOG implementation of the business rules with an alternative implementation in DROOLS.

3.4 Calculation of Profits and Taxes – An Example

In the following, we want to process the given invoice with the program bookings step by step. First, the business volume in `annual_sales/3` is determined. We summarize the derived facts below. As one can see, the delivery threshold for 2011 was not exceeded, but for 2012 it was. Apparently, the company's profit has increased from 2011 to 2012.

```
annual_sales('France', 2011,  92300).
annual_sales('France', 2012, 100098).
```

Since the annual sales to the destination country France – including the currently processed invoice – exceed the delivery threshold in 2012, the tax has to be paid in France:

```
tax_country('France').
```

Finally, `booking_position/5` calculates the resulting booking positions for the modes `profit` and `tax`:

```
booking_position(1, food,    profit, 'France', 200.00).
booking_position(1, food,     taxes, 'France',  11.00).
booking_position(2, nonfood, profit, 'France', 100.00).
booking_position(2, nonfood,  taxes, 'France',  19.60).
```

The derivation of the first fact for `booking_position/5` is visualized in Figure 2. For a better overview, we use abbreviated predicate names in the proof trees.

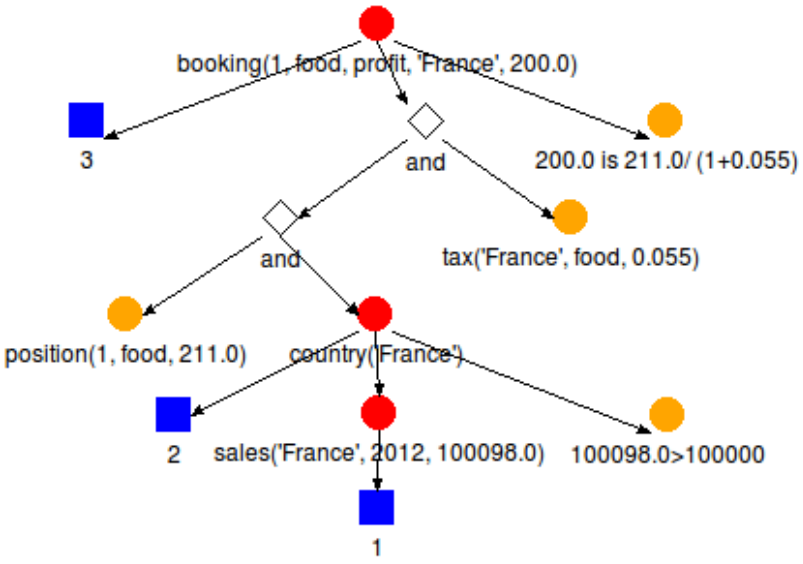

Fig. 2: Visualization of Proof Trees

If we assume a different value for `annual_sales_so_far/3` in 2012, then the annual sales to the destination country France – including the currently processed invoice – do no longer exceed the delivery threshold in 2012:

```
% annual_sales_so_far(Destination, Year, Total)
annual_sales_so_far('France', 2011, 92300).
annual_sales_so_far('France', 2012, 80000).
```

Thus, the tax has to be paid in the home country Germany:

```
tax_country('Germany').
```

We derive the following facts for `booking_position/5`:

```
booking_position(1, food,    profit,  'France', 197.20).
booking_position(1, food,     taxes, 'Germany',  13.80).
booking_position(2, nonfood, profit,  'France', 100.50).
booking_position(2, nonfood,  taxes, 'Germany',  19.10).
```

The derivation of the first fact for `booking_position/5` is visualized in Figure 3. The `(If -> Then; Else)` statement has been replaced by pure PROLOG with conjunction and disjunction.

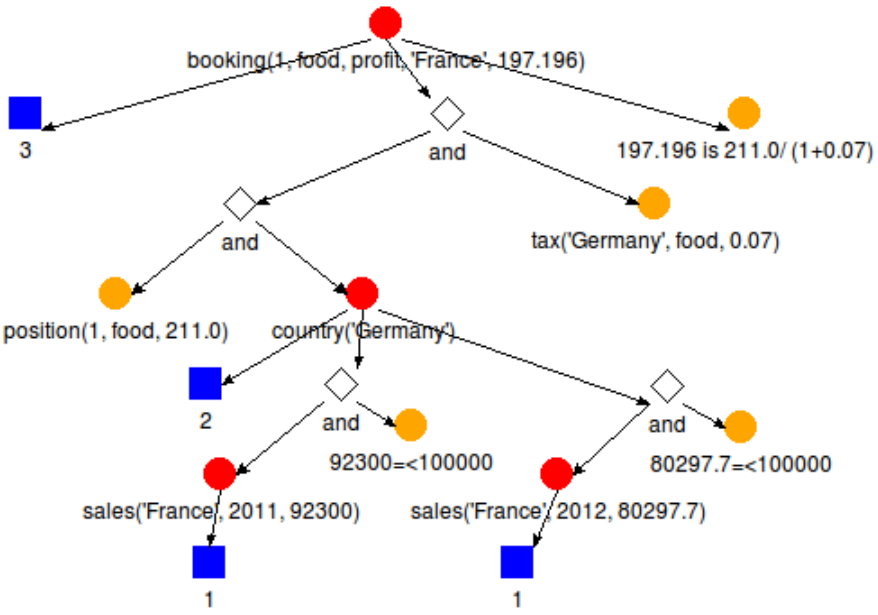

Fig. 3: Visualization of Proof Trees

4 Bottom–Up Evaluation with Proof Trees

We have extended the DDK package DATALOG* [16] such that it can generate proof trees [5]. The proof trees are obtained based on an (automatic) program transformation. Compared to standard DATALOG, the extension 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. DATALOG* programs are evaluated bottom–up – just like standard DATALOG programs.

The rule nodes in the proof trees – see Figures 2 and 3 – are labeled by numbers, and they are shown as blue boxes. The nodes for the derived atoms are shown as red circles, that are labeled with the atoms. Basic predicates from the configuration or calls to PROLOG for built–in predicates (such as `is/2` and `>/2`) are given by nodes in the form of an orange circle.

The visualization of the proof trees can be configured by the developer. Firstly, it is possible to group body atoms in the proof trees for better readability; these atoms are then joined by an `and` node, that is depicted as a white rhombus. Secondly, trivial body atoms can be excluded to reduce the complexity of the visualization: e.g., the atoms for `invoice/2` and `delivery_threshold/2` are excluded, since they are part of the input, and they are not modified during the computation. Thirdly, the included atoms can be simplified suitably: e.g., the predicate symbol `annual_sales/4` is abbreviated to `sales/4`, `tax_country/1` is abbreviated to `country/1`, and `booking_position/5` is abbreviated to `booking/5`.

A proof tree is encoded in a term structure that resembles XML. We process this XML using the query, transformation and update language FNQUERY of the DDK. In principle, however, the proof trees can be analyzed and further processed using standard XML tools as well. The term structures do not lead to any termination problems even for recursive rule sets, since – during the bottom–up evaluation – we do not allow that an atom is used in its own derivation. This can be tested by investigating the proof tree. Obviously, the construction of the proof trees costs some extra runtime. But for realistic examples of business rules, this is by far no problem. The main issue is to assist the business analyst during the knowledge engineering phase by a suitable, not too verbose visualization of the derivations.

5 Knowledge Engineering for Business Rules

In practice, it is a long, iterative process to model the application domain in cooperation with the customer (domain expert), and to develop the business rules. There are many communication problems and misunderstandings. The example, that we have shown above, is simplified; we have shown only a small portion of the business rules, and we have omitted details, such as the account numbers which depend on the different countries, types (food or nonfood), and modes (profit or tax). However, even for the small example, it took a few sessions with the customer to completely understand and suitably model the problem.

It is crucial that the business rules are formal and to some extent understandable for the customer as well – who usually has very little experience with PROLOG. Thus, we

need an integrated development environment, that supports – among others – domain specific languages, visualization, and anomaly tests.

Business rules – in connection with domain specific languages (DSL) – can bridge the gap between the customer and the developer, since – normally – the customer cannot understand the – often procedural – code written by the developer. A business analyst helps to communicate based on a declarative specification with business rules and DSLs.

5.1 Knowledge Engineering in PROLOG

In a domain specific language, the developers can implement the business rules, and the customer can still – largely – understand this formalization. The formalization should be executable – like in declarative programming. Using PROLOG's infix operators, it should be possible to iteratively develop a domain specific language. Using PROLOG's parsing techniques (DCGs), this language can be improved even further. At any time of the development phase, the developer and the customer can discuss about the DSL specification. In future work, we will investigate the use of DSLs for declarative knowledge engineering.

Usually, business rules have to be evaluated in a bottom–up, forward chaining manner. During the development phase, a *visualization* of the program execution is essential. Using DATALOG*, we can already evaluate the PROLOG specification bottom–up, and during this evaluation we can generate *proof trees*.

We could also investigate the declarative specification by looking for *design anomalies*. In our example, we could manually simplify an over–complicated set of four initial rules for determining the tax country to a single and much more readable rule. In the future, we would like to – partially – automatize such simplifications. We might also detect certain *inconsistencies* or *redundancies* in the set of business rules.

The standard tools for business rules, such as DROOLS, do not support the development phase well. However, it is conceivable, that we develop the business rule specification in PROLOG and later compile it to DROOLS. Up to Version 5.3, there exists an XML format for DROOLS rules, that could be used for the data exchange; the DSL feature of DROOLS itself could also be used. Moreover, technologies of compiling PROLOG code to JAVA, that are known in the PROLOG community, might be used.

5.2 Knowledge Engineering in DROOLS

In PROLOG, the list of all solutions to a given query can be simply computed using the meta–predicate `findall/3`. In DROOLS, such a computation is only possible using a combination of JAVA and a query language like SQL (e.g., embedded SQL programming). This might be very complicated and unnecessarily blow up the implementation in DROOLS.

Although JAVA and DROOLS are connected tightly, there are some flaws. Moreover, DROOLS does not even work together properly with all JAVA concepts. For example, when we need to use a class in the condition block of a business rule, then that class must be instantiated in the DROOLS session, even if the class has just static attributes and static methods, such as, e.g., a constant class. Only when we use such a class in an action block, then we need not instantiate it in the DROOLS session.

When we looked at rules from DROOLS, we noticed that the rule format is not well–suited for analyses. For our investigations, we transformed the original XML into an enriched XML using Extensible Stylesheet Language Transformations (XSLT). Our aim was not only to investigate rules in DROOLS this way, but to export business rules from PROLOG to DROOLS. However, since Version 5.4.0, the conversion from DROOLS to XML is no longer supported, which complicates our undertaking.

5.3 Domain Specific Languages

Domain specific languages (DSL) [9] have recently become popular in knowledge engineering for business rules. Unlike general–purpose programming languages (such as JAVA and C), a DSL is usually a subset of a programming language or a specification language for a special problem domain; we call this an internal DSL. External DSLs are completely new languages with a separate syntax and semantics (such as, e.g., the query language SQL for relational databases).

DSLs are increasingly used in business process modeling, since the standard approaches, that are often based on JAVA, are too difficult to understand and cannot be modified or extended flexibly enough. The idea is to assist business analysts already during the customer contact with developing a formal specification of the business rules based on a DSL. Subsequently, this initial formal specification can be corrected and refined by the developers. Ideally, it should already be executable, but it could also be reimplemented later in a standard programming language. The abstract, declarative DSL representation can, however, be understood and validated by the business analyst at any time during the development phase.

Unfortunately, developing suitable DSLs turned out to be difficult in practical projects, and it brings a non–negligible additional effort at the beginning of the software project. Frequently, projects fail, since the DSL was not developed carefully enough.

We think that logic programming technology – especially using PROLOG – could be beneficial here by simplifying the development of a DSL. In some sense, DATALOG* can also be considered as an internal DSL that is embedded in PROLOG. In fact, many DATALOG* programs can even equivalently be evaluated using PROLOG's top–down evaluation.

6 Conclusions

In this paper, we have presented a PROLOG–based approach for the bottom–up evaluation of business rules, that supports the visualization of the derivation processes using suitable, customized proof trees.

Knowledge engineering with declarative concepts turned out to be very beneficial for business rules.

Integrated development environments for PROLOG generally provide helpful tools for debugging and tracing. We are planning to additionally use techniques for anomaly analysis, which we have developed before.

In the future, we will extend our knowledge engineering approach by developing suitable domain specific languages based on PROLOG technology.

References

1. T. Allweyer. *BPMN 2.0 – Einführung in den Standard für die Geschäftsprozessmodellierung.* BoD, 2010.
2. J. Baumeister, D. Seipel: *Anomalies in Ontologies with Rules.* Journal of Web Semantics: Science, Services and Agents on the World Wide Web, Volume 8 (1), pp. 55–68, 2010.
3. P. Browne. *JBoss Drools Business Rules.* Packt Publishing, 2009.
4. Bundeszentrale für Steuern. *Merkblatt Umsatzsteuer im In– und Ausland.* `http://www.bzst.de/DE/Steuern_International/USt_im_In_und_Ausland/Merkblatt_USt_Inland_Ausland.html`
5. *S. Ceri, G. Gottlob, L. Tanca:* Logic Programing and Databases, Springer, 1990.
6. A. Charfi, M. Mezini. *Hybrid Web Service Composition: Business Processes Meet Business Rules.* Proc. 2nd International Conference on Service–Oriented Computing, ICSOC 2004, pp. 30–38. ACM, 2004.
7. T. Davenport and J. Short. *Information Technology and Business Process Redesign.* Sloan Management Review, 1990.
8. C. Forgy. *Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.* Artificial Intelligence, Volume 19 (1), pp. 17–37, 1982.
9. M. Fowler. *Domain–Specific Languages.* Addison–Wesley (Pearson Education), 2011.
10. JBOSS Community. DROOLS *User Guide.* 2012, `http://docs.jboss.org/drools/release/5.4.0.Final/drools-expert-docs/html/ch02.html`
11. K. Kaczor, G. Nalepa, L. Lysik, K. Kluza. *Visual Design of Drools Rule Bases Using the XTT2 Method.* Studies in Computational Intelligence, Semantic Methods for Knowledge Management and Communication, Springer, 2011.
12. O. Febraro, G. Grasso, N. Leone, K. Reale, F. Ricca. DATALOG *Development Tools.* Proc. 2nd International Workshop on DATALOG in Academia and Industry (DATALOG 2.0), Springer, LNCS 7494, pp. 81–85.
13. M. Salatino. *jBPM Developer Guide.* 2009, Packt Publishing.
14. D. Seipel, M. Hopfner, B.Heumesser: *Analyzing and Visualizing* PROLOG *Programs Based on* XML *Representations.* Proc. International Workshop on Logic Programming Environments (WLPE), 2003.
15. D. Seipel, J. Baumeister, M. Hopfner: *Declaratively Querying and Visualizing Knowledge Bases in* XML. Proc. 15th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2004), Springer LNAI 3392, pp. 16–31, 2005.
16. D. Seipel. *Practical Applications of Extended Deductive Databases in* DATALOG*. Proc. Workshop on Logic Programming (WLP), 2009.
17. D. Seipel. *The* DISLOG *Developers' Kit (*DDK*).* `http://www1.informatik.uni-wuerzburg.de/database/DisLog/`
18. J. Wielemaker. SWI PROLOG *Reference Manual, Version 5.0.* `http://www.swi-prolog.org/`
19. J. Wielemaker, M. Hildebrand, J. van Ossenbruggen: *Using* PROLOG *as the Fundament for Applications on the Semantic Web*, Proc. of the ICLP Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services (ALPSWS), 2007.