Computer Languages, Systems & Structures 51 (2018) 102–117

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Domain-specific languages in Prolog for declarative expert knowledge in rules and ontologies

Dietmar Seipel [a,*], Falco Nogatz [a], Salvador Abreu [b]

[a] Department of Computer Science, University of Würzburg, Am Hubland, 97074 Würzburg, Germany
[b] LISP and Department of Computer Science, University of Évora, Rua Romão Ramalho, 59, 7000 Évora, Portugal

ARTICLE INFO

Article history:
Received 27 November 2016
Revised 25 June 2017
Accepted 26 June 2017
Available online 4 July 2017

Keywords:
Domain-specific language
Expert knowledge
Declarative rule
Prolog
Deductive database

ABSTRACT

Declarative *if–then* rules have proven very useful in many applications of expert systems. They can be managed in deductive databases and evaluated using the well-known forward-chaining approach. For domain-experts, however, the syntax of rules becomes complicated quickly, and already many different knowledge representation formalisms exist. Expert knowledge is often acquired in story form using interviews. In this paper, we discuss its representation by defining domain-specific languages (DSLs) for declarative expert rules. They can be embedded in Prolog systems in internal DSLs using term expansion and as external DSLs using definite clause grammars and quasi-quotations – for more sophisticated syntaxes.

Based on the declarative rules and the integration with the Prolog-based deductive database system DDBase, multiple rules acquired in practical case studies can be combined, compared, graphically analysed by domain-experts, and evaluated, resulting in an extensible system for expert knowledge. As a result, the actual modeling DSL becomes executable; the declarative forward-chaining evaluation of deductive databases can be understood by the domain experts. Our DSL for rules can be further improved by integrating ontologies and rule annotations.

1. Introduction

Domain-Specific Languages (DSL) [1] are a common approach to model knowledge in a specific application area. Their aim is to provide an interface that can be easily used by domain experts of the given field, who are often not familiar with a general-purpose programming language (GPL). Due to the close affiliation to their application areas, DSLs are often more appropriately called specifications, definitions, or descriptions [2]. There are many cases where DSLs are just a formal representation of expert knowledge without the ability to be executable at all. Popular examples are the Unified Modeling Language (UML), which is used for the visualisation of a system's architecture, or the extended Backus–Naur form (EBNF), which can be used to express a grammar. DSLs which are not intended to be executed at first, are called *modeling languages*, in contrast to *programming languages* [3]. It is hard to define DSLs that suit both needs, i.e., to be very expressive in the specification of knowledge as well as to be precise on expressing instructions for computation.

* Corresponding author.
E-mail address: dietmar.seipel@uni-wuerzburg.de (D. Seipel).

http://dx.doi.org/10.1016/j.cl.2017.06.006
1477-8424/

But especially expert knowledge is often stated as rules with conditions and consequences. The first is in general a description of an initial, dependent state, which can be expressed very declaratively; the consequences are often defined using formulas or by the addition of new constraints. Using a declarative DSL to model this expert knowledge based on facts and rules enables the connection with deductive databases. This allows to execute and query the already defined expert knowledge and therefore reduce the gap between modeling and programming DSLs. The knowledge, which is in most cases not just about facts, but more about the relationships between the examined entities, could be used to create new insights.

Our approach is to combine an intuitive, declarative modeling DSL with a declarative rule evaluation mechanism given by DATALOG. The combination of both seems very promising, as it results in a reasonable and easily adaptable rule base, while still being consultable and executable. The knowledge could finally be managed in a deductive database system like the PROLOG-based DDBASE [4]. It supports the analysis of the rules, including a graphical visualiser interface. The automated reasoning facilitates the linking of conclusions and helps to detect contradictions. Furthermore, PROLOG suits very well for the definition of new DSLs, providing three mechanisms to extend the programming language: (1) the definition of user-defined operators; (2) a macro-like term and goal expansion; and (3) a powerful yet intuitive way to express grammars using Definite Clause Grammars (DCG) [5]. Together with the quasi-quotations [6] recently introduced in SWI-PROLOG – a concept adopted from languages like Haskell and JavaScript –, external DSLs can be directly used within existing PROLOG programs. Given the flexibility of these mechanisms, PROLOG is very fit for the rapid-prototyping of a newly defined DSL [7].

In our work, we define a DSL whose main part is a set of *if–then* rules which hold the expert knowledge in a declarative, textual format. We discuss the definition of this modeling language as a DSL in PROLOG. In previous work, we had described a rule concept for expert knowledge and some methods for querying and visualising the rules in various application domains including, e.g., in medical diagnosis [8]. In [9], we have adapted this concept to the field of organisational psychology, and we have given a simple definition of the rule language as an internal DSL in PROLOG. In [10], we have given a formal description of the rule language using a context-free grammar with a focus on the integration into the workflow for collecting rules in the field of change management.

The present article is an extended version of our earlier conference paper [10] on *if–then* rules in change management, and it contains some concepts developed in earlier papers of Seipel et al. on medical diagnosis [8,11]. It contributes the following new aspects: we discuss the two approaches of implementing the rule language both as an internal and an external DSL in PROLOG. In order to provide a simple interface with meaningful error messages, we integrate the rule language directly into PROLOG source code using quasi-quotations. We also show how forward-chaining derivations can be visualised using proof trees. Moreover, we have extended the rule language for improved modularity. The expert knowledge can be enriched by ontologies and annotated by provenance and meta-information.

The rest of this paper is organised as follows: Section 2 recalls some basic ideas from declarative programming, domain-specific languages, deductive databases, and ontologies. Section 3 presents a DSL for *if–then* rules, defines its syntax and semantics, and gives some example rules. Then, we present and discuss the implementation as an internal DSL, respectively, external DSL in PROLOG. The analysis of the resulting rule base is investigated in Section 4; ideas for queries and the visual analysis in interactive rule editors are given. Section 5 shows how the knowledge base can be augmented by contextual information given in ontologies and how rules can be annotated with confidence information. The paper is concluded with some final remarks.

2. Declarative expert knowledge in rules and ontologies

In this section, we summarise and highlight some general concepts from declarative programming and expert rules in deductive databases, logic programming, and ontologies, that are useful for the rest of the article.

2.1. Declarative programming

Declarative programming is a programming paradigm that expresses the logic of a computation in an abstract way, without having to describe its control flow. Thus, the *semantics* of a declarative language becomes easier to grasp for domain experts. Declarative programming offers, e.g., the following advantages for data and knowledge engineering: security, safety, and shorter development time, as known from information systems with relational databases. There exists a plethora of results about query optimisation in relational and deductive databases, e.g., [12–16]. For instance, Minker and his students have interesting results in the field of semantic query optimisation [17]: evaluation plans can be derived and cached results can be included.

Languages which claim to be declarative usually attempt to minimise or eliminate side effects by describing *what* the program must accomplish in terms of the problem domain rather than describing *how* to accomplish it as a sequence of programming language instructions – the how is implicitly left to be decided by the implementation of the language. In contrast, imperative (or procedural) languages require algorithms to be implemented in explicit steps. Declarative programming often considers programs as theories of a formal logic, and computations as deductions, or proofs. Declarative programming may greatly simplify writing parallel programs, as it does away with explicit control. Examples of declarative languages include database query languages (e.g., SQL, DATALOG, XQuery), regular expressions, logic and constraint logic programming (e.g., PROLOG, CLP), and functional programming.

2.2. Rules in deductive databases and logic programming

A deductive database (DDB) is a database which may carry out *deductions* based not only on facts but also on rules which are also stored in the database itself [13]. DDBs combine logic programming languages and relational databases, as they share the querying flexibility of the former while retaining the performance and scalability of the latter. DDBs commonly use variants of the logic programming language DATALOG, whose syntax restricts the standard logic programming language PROLOG [18,19], and whose declarative semantics is closer to relational databases and uses forward-chaining and bottom-up evaluation of rules. Typically DDBs will operate on data which are more restrictive than that of PROLOG, yet more general than that which may structurally be accessed with SQL. Deductive database languages have been used in many applications, such as data integration, computer networking, program analysis or security [16,20]. A comprehensive description of the semantics of deductive databases and logic programming is given, e.g., in [16]. For this paper, it is sufficient to have an intuitive understanding of the meaning of *if–then* rules. For the interested reader, we will give a summary of some relevant concepts for the semantics and the evaluation of deductive databases and logic programming in the appendix.

For several years, we can observe what is sometimes called a *renaissance* of DATALOG [20]. Zaniolo started with LDL at Austin; subsequently, many DDBs have been developed. New DATALOG applications have been developed, e.g., at Berkeley, where Boom and Bloom handle distributed computing, parallelism, and concurrency. New DATALOG and logic programming companies have been created and became successful: the company LogicBlox provides a unified database foundation for the next generation of smart analytical and transactional applications; the company Coherent Knowledge focusses on technology specialising in advanced artificial intelligence (AI) combining declarative, rule-based knowledge representation and reasoning (KRR) tightly with natural language processing (NLP) and complementing machine learning (ML); the company Lixto on declarative web data extraction and annotation exists since the beginning of the millennium, and Gottlob's database group has recent connections and publications with Oracle. SAP uses SWI-PROLOG [19] in its cloud platform HANA for the configuration with the source repository git/gerrit, just to name a few.

2.2.1. Logic programs and DATALOG

Syntactically, a logic program $\mathcal{P}$ is a set of rules, which are range-restricted implications $A \leftarrow \beta$, where A is an atom and β can be any formula over atoms built with the sentential connectives $\vee$, $\wedge$, and not (default negation). In general, the connective not cannot occur within the scope of another connective not. Some extensions can also handle literals with classical negation ($\neg$), rather than just atoms in the rules. Basic DATALOG does not allow for function symbols, and β has to be a conjunction (connective $\wedge$) of atoms. For instance, the well-known transitive closure rules can be expressed in DATALOG:

$$tc(X,Y) \leftarrow arc(X,Y),$$
$$tc(X,Y) \leftarrow arc(X,Z) \wedge tc(Z,Y).$$

Our DATALOG extension allows for function symbols and an arbitrary use of the sentential connectives in the rules, whereas frequently in deductive databases $\beta = B_1 \wedge \cdots \wedge B_m \wedge \text{not } C_1 \wedge \cdots \wedge \text{not } C_n$ is just taken as a conjunction of atoms B_i or default negated atoms not C_i. A is called the head, and β is called the body of the rule. The property *range-restricted* means that every variable symbol in the head must also occur in the body, where variable symbols within default negated formulas not ϕ are not counted. Facts A are rules with an empty body and thus correspond to tuples in a relational database; rules $A \leftarrow \beta$ are implications.

2.2.2. Forward-chaining evaluation

In deductive databases, DATALOG programs can be evaluated using forward-chaining (bottom-up) or backward-chaining (top-down). Usually, bottom-up evaluation is done using an iteration of the hyperresolution consequence operator $\mathcal{T}_\mathcal{P}$, which iteratively applies the rules starting with the facts until now new facts can be inferred. For the DATALOG program $\mathcal{P}$ containing the two transitive closure rules above together with the three facts

$$arc(a,b), \quad arc(b,c), \quad arc(c,d),$$

the *bottom-up evaluation* derives the following monotonically increasing sequence of sets of facts by repeatedly applying the rules to the already derived facts:

$$I_0 = \emptyset,$$
$$I_1 = \{\, arc(a,b),\ arc(b,c),\ arc(c,d) \,\},$$
$$I_2 = I_1 \cup \{\, tc(a,b),\ tc(b,c),\ tc(c,d) \,\},$$
$$I_3 = I_2 \cup \{\, tc(a,c),\ tc(b,d) \,\},$$
$$I_n = I_3 \cup \{\, tc(a,d) \,\}, \text{ for all } n \geq 4.$$

The concept of top-down evaluation is not relevant for our approach and thus not described here. A comprehensive description of the syntax and semantics of deductive databases and logic programming is given, e.g., in [16]. In the appendix, we give a brief summary of the formal semantics. In [11], we have defined refined declarative evaluation methods for DATALOG rules allowing also embedded calls to PROLOG, which, e.g., allows for handling interactive user dialogs during medical diagnosis; furthermore, if diagnoses are inferred with several certainty values, then these values can be aggregated after each iteration of the consequence operator.

2.3. Knowledge acquisition and analysis for if–then rules

Expert knowledge can often be collected in interviews and stated in the form of *if–then* rules, which can be managed in deductive databases. The interview results are typically described with lists and in a story form [21]. Following the steps to gain expert knowledge in a well-structured form as described by Ford and Sterman [22], these interviews are split into three phases: (1) in the *positioning phase* the context and possible input variables are identified and the separate relationships considered; (2) in the *description phase* a single relationship is characterised in a verbal and textual form; (3) and the *discussion phase* compares the different textual representations and identifies inconsistencies. Using a declarative DSL with rules empowers the domain expert to specify knowledge in a textual story form. In connection with a deductive database system, these facts and rules can easily be combined and examined for contradictions, similarities and insufficient descriptions.

Empirical studies often result in certain findings. Currently, there is no standard to collect the results, and it is hard to compare and combine knowledge of different sources. The findings are published in academics, but there is no uniform data format. The same applies for the raw data collected in the studies. Instead, the data are kept in proprietary systems, which so far only serve for persistent storage. Thus, it is impossible to automatically obtain new insights based on the combination of multiple studies. In [8,11] we have used rules for reasoning over findings in medical diagnosis. In [9], we have presented similarly structured rules over findings in change management. This example for *if–then* rules and its analysis will be used later in Section 4.

2.4. Rules in ontologies

Besides relational databases, ontologies have played an important role for building intelligent information systems. Currently, ontology languages like OWL are extended by rule-based elements and links; this extension of ontologies by a rule representation is a very popular research issue. A rule language increases the expressiveness of the underlying knowledge in many ways. Likewise the integration creates new challenges for the design process of such ontologies, but also existing evaluation methodologies have to cope with the extension of ontologies by rules.

The use of ontologies has shown its benefits in many applications of intelligent systems in the last years. Whereas, the implementation of lower parts of the semantic web stack has successfully led to standardisations, the upper parts, especially rules and the logic framework, are still heavily discussed in the research community, e.g., see Horrocks et al. [23]. This insight has led to many proposals for rule languages compatible with the semantic web stack, e.g., the definition of SWRL (semantic web rule language) [24] originating from RULEML [25] and similar approaches [26]. It is well agreed that the combination of ontologies with rule–based knowledge is essential for many interesting semantic web tasks, e.g., the realisation of semantic web agents and services. SWRL allows for the combination of a high-level abstract syntax for Horn-like rules with OWL, and a model theoretic semantics is given for the combination of OWL with SWRL rules. An XML syntax derived from RULEML allows for a syntactical compatibility with OWL. However, with the increased expressiveness of such ontologies new demands for the development and for the maintenance guidelines arise. Thus, conventional approaches for evaluating and maintaining ontologies need to be extended and revised in the light of rules, and new measures need to be defined to cover the implied aspects of rules and their combination with conceptual knowledge in the ontology.

We have built tools for managing and analysing relations, ontologies, and rules [27]. Techniques from deductive databases and logic programming can integrate hybrid knowledge bases with structured knowledge. Nowadays, semantic web technology including linked data (JSON-LD) is also very important. Data and knowledge engineering can clearly benefit from the declarative approach provided by logic programming.

3. Design and implementation of domain-specific languages for declarative expert rules in PROLOG

The logic programming language PROLOG has a long tradition for expert system applications [28,29], and it fits very well for storing, querying and analysing knowledge bases. Recently, SWI-PROLOG has adopted several technologies used in the semantic web [30], which has resulted in the semantic web infrastructure CLIOPATRIA [31] with good support for well-established semantic web technologies like RDF and SPARQL.

However, these languages are hard to read and write for people who acquire the expert knowledge. Even PROLOG's original knowledge representation in the form of Horn clauses is often a great barrier for people not familiar with predicate logic. In contrast, its underlying principle of describing rules in the form of premises and consequences is quite natural and often the preferred form for knowledge collected in interviews [32]. So the usage of a domain-specific language (DSL) based on the statements worded in spoken language seems like a natural consequence. By embedding such a DSL directly into PROLOG, we can still profit from PROLOG's decent deduction and semantic web abilities.

Gupta et al. had presented a first approach for the specification, efficient implementation, and automatic verification of DSLs in (constraint) logic programming [33]. The task of software engineering can be eased by resorting to DSLs, and logic programming – especially PROLOG – can be used to naturally and rapidly obtain an implementation infrastructure for DSLs with backward-chaining.

In the present paper, we investigate *if–then* rules in DATALOG style under forward–chaining, and we provide an implementation infrastructure based on deductive databases. In Section 3.1, we introduce the idea of implementing and using

DSLs in general. The textual, logic-based format of *if–then* rules is presented in Section 3.2. We discuss the implementation techniques for internal and external DSLs in PROLOG in Sections 3.3 and 3.4 and conclude with a comparison of both methods in Section 3.5.

3.1. Domain-specific languages

A domain-specific language is a computer language designed for a particular application domain. It often drives towards either the natural language representation of a given topic or a well-defined formalisation thereof, e.g., a representation of formulas or rules. The DSL is therefore intelligible for domain experts while still being formalised in a programming language.

According to Fowler [34], DSL*s are small languages, focused on a particular aspect of a software system* – in contrast to general-purpose programming languages (GPLs). Although DSLs can generally not be used to write a complete software, they are designed to be very suitable to depict a given domain area and are therefore directly embedded or externally used by GPLs, which results in a good portability and standardisation.

DSLs can take on two forms: *internal* or *external*. In the first form, which is also called *embedded* DSL, the DSL is integrated directly into the host language, i.e., it is an instance or dialect of the host GPL. A popular representative is the JSON file format [35], which is a direct subset of the GPL JavaScript. Although originally designed specifically for data storage and exchange between several JavaScript programs, it is nowadays a well-established DSL for data exchange supported by every major programming language, similarly to XML.

For all programming languages except JavaScript, the JSON file format is not a direct subset of the host GPL and therefore classified as an external DSL. Its format is independent of the concrete syntax of the GPL and has to be supported by using appropriate parsers and APIs. Another popular example for an external DSL are regular expressions, which have a well–defined application area and are integrated in most GPLs, although with slightly varying syntax.

These two examples – JSON and regular expressions – underline the importance of DSLs as a way to ensure portability and to increase programmer productivity across various GPLs. The transition from an internal DSL to an external DSL is often smoothly, as seen with JSON and regular expressions, because with a gaining acceptance across domain experts and programming languages, the syntax gets standardised and widely adopted.

Having a DSL also improves communication with the domain experts. By raising the level of abstraction, non-computer experienced experts can work more productively, and the barrier to model expert knowledge gets lower. Today, there are DSLs for numerous areas of application, such as, e.g., expert rules, business rules, configuration rules/constraints. A systematic mapping study has been given in [36].

3.2. Representation of if–then rules

We have developed a textual, logic-based format for *if–then* rules, which tries to represent the rules – as far as possible – in a natural language syntax. For expressing the rules, we have chosen to follow the general form:

if *Condition* then *Consequence*.

After the keywords `if` and `then`, a *Condition* and a *Consequence*, respectively, is expected. Both are so-called junctions of findings. If *Condition* is empty, then the rule is also called a *fact*. A finding always has the form `Feature = Value`, although besides equality, additional comparators may be used. The simplest values are the constants `yes` and `no`, specifying the existence or absence of a given feature. Besides this, literal descriptions as in the finding `'Some Value' = increases` or numerical information, which can assume significant practical importance, are possible.

Several findings in `Condition` and `Consequence` can be linked by connectives to form formulas. For this, the keywords `and`, `or`, and `neg` are available. If *F* and *G* be formulas, then the following are also allowed formulas: `neg` *F*, *F* `and` *G*, *F* `or` *G*. Note that conjunction binds stronger than disjunction, and classical negation `neg` binds the most strongly. An extension would be to allow for default negation (`not`) to occur in `Condition`, but not in `Consequence`. For representing arbitrary formulas, subformulæ can be included in brackets. Currently, this language does not support more complex operators like the exclusive *either F or G*, formally $F \oplus G$. However, it can be expressed as $(F \wedge \neg G) \vee (\neg F \wedge G)$ using elementary connectives. Although it might be easily possible to define a particular *either–or* operator, we consider the rule using only elementary connectives for the purposes of this introduction. The DSL implementation techniques presented in the following sections can be easily adopted for additional operators.

As mentioned in Section 1, PROLOG suits very well to be used with DSLs and provides several major techniques to implement this rule language. In the following Sections 3.3 and 3.4 we will discuss the implementation as an internal and as an external DSL in PROLOG.

3.3. An internal DSL using PROLOG operators

Simple languages such as the rule representation in the form of *if–then* can be modelled in PROLOG directly by using its built-in language features. It is possible to define terms and declare appropriate operators, so that the given rules can be embedded directly, resulting in an internal DSL.

The DSL has been conveniently defined in PROLOG by a collection of suitable operators and their precedences:

```
:- op(1100, yfx, then).
:- op(1000, fx, if).
:- op(900, yfx, or).
:- op(800, yfx, and).
:- op(700, fx, neg).
```

To support the *if–then* syntax, we define an unary prefix–operator `if` as well as the binary operator `then`. Using these operators, it is valid to state `if a then b` in PROLOG, as it gets parsed as the nested term `then(if(a),b)`. Both terms are equal and valid PROLOG terms after the definition of the operators as presented before.

To be valid PROLOG, features and values beginning with a capital letter or containing spaces have to be enclosed by single quotation marks. Additionally, every rule has to end with a dot.

As a simple example, we want to express that clothes get wet if (1) the weather is rainy and one has no umbrella, or (2) there is a thunderstorm. This statement can be modeled as an instance of this internal PROLOG DSL as follows:

```
if 'Weather' = rainy and 'Umbrella' = no
   or 'Weather' = 'Thunderstorm'
then 'Clothes' = wet.
```

Since the constants `rainy`, `no` and `wet` begin with a lowercase letter and do not contain a space, they do not need to be enclosed in quotation marks. Furthermore, we do not need any brackets. The precedences of the operators, which are given as numerical values between 700 and 1100, ensure that `and` binds before `or` and `if` and `then`, that `or` binds before `if` and `then`, and finally that `if` binds before `then`. In PROLOG, we can declare this more compactly by assigning increasing precedences to the operators in the sequence =, `and`, `or`, `if`, `then`. In general, using brackets we can express a formula where `or` should bind before `and`. As mentioned before, we could make the language more powerful by adding a binary operator `xor` for the exclusive or.

The proposed notation for rules complies with the syntax of PROLOG, which facilitates its usage as an embedded DSL. With the definition of `if`, `then`, `neg`, `and` and `or` as operators, the established rules become valid PROLOG structures. Thus, it is possible to create an externally-backed rule base file including all known statements. As it conforms to user-readable syntax, the rule base may even be updated with a normal text editor. It may be gradually expanded by adding new rules. The end result is an incremental rule storage containing all statements representing the expert knowledge.

This definition of the rule language as an internal DSL in PROLOG is easily extensible and allows for rapid prototyping. Nevertheless, using an internal DSL to represent rules in a form which is based on the natural language representation has some trade-offs in PROLOG: Since of the requirements of the syntax of the host language, strings must not contain spaces and must not start with a capital letter, as they are otherwise recognised as variables. It is also necessary to use the dot as the rule ending, which might not be feasible if findings should contain dots, for example to represent hierarchies using the notation `a.b`.

In addition, the specification of the rules might be hard for users – the experts of the application domain –, which are not very experienced with PROLOG. Since we use only PROLOG operators for the internal DSL definition, the error handling is based only on a rule's syntax. Therefore, potential errors like starting a value with a capital letter will not yield any warning, since it is recognised as a variable which is syntactically allowed in these positions by the PROLOG parser.

3.4. An external DSL using quasi-quotations

Because of the disadvantages of using a PROLOG dialect and to lower the barrier to entry for users not familiar with PROLOG, we also discuss the implementation of the *if–then* rules as an external DSL in PROLOG. The resulting format is only loosely coupled with PROLOG and might therefore be adapted in other applications and programming languages, too, similar to existing, well-known DSLs like EBNF and regular expressions.

Quasi-Quotations in PROLOG. Originally, PROLOG has poor support for long text fragments. There are several mechanisms to specify strings that spawn about multiple lines, but all require the user to adjust the original multi-line string.[1] To support multi-line strings natively and the easy embedding of external DSLs without the need of manually adding escape symbols, multiple mechanisms have been discussed in [37] for extending the PROLOG syntax. Inspired by constructs in other programming languages, for instance Haskell, quasi-quotations have been added to SWI-PROLOG in 2013 [6]. Since then, they have been used to embed several well-known external languages like HTML and SQL. In [38], we have discussed the implementation of GraphQL as a DSL in PROLOG using quasi-quotations to specify queries in deductive databases.

The basic form of a quasi-quotation in PROLOG is of the following, where `Tag` is an atom and `Content` is a string:

`{|Tag||Content|}`, e.g., `{|html(doc)||<p>Hello, Name!</p>|}`

Quasi-quotations within a PROLOG program are translated using term expansion. Given a quasi-quotation with the tag `tag(SyntaxArgs)`, the compiler will call the corresponding user–defined predicate as `tag(+Content,`

[1] In particular, a trailing \ (single backslash) and \c (backslash and c) are common, which differ only in the whitespace handling.

+SyntaxArgs, +Vars, -Result) to calculate the Prolog term the quasi-quotation gets replaced by. In this way, the term {|html(doc)|| Hello, Name! |} has the tag html(doc) and invokes the following call to the predicate html/4: html(' Hello, Name! ', [doc], ['Name'='Alice'], Result).

In order to prevent a name clash with existing Prolog predicates, we propose to use a compound tag for quasi-quotations starting with a prefix for the used module, for example to use {|a_rule||...|} for the quasi-quotation to embed rules of the module called a. In future work, this tag prefixing could be done automatically by extending Prolog's term expansion for quasi-quotations. For better readability in this paper, we will omit the module prefixes and refer to quasi-quotations simply by their canonical name, for instance rule.

By using the given argument Vars of the called predicate tag/4, it is possible to refer to Prolog variables of the current goal. In this way, it is possible to check for possible variable names within the given string Content and bind them to the appropriate variables. A more detailed introduction to quasi-quotations can be found in [6], a more detailed example for the implementation of a Dsl in Prolog in [38].

3.4.1. Parsing with definite clause grammars

The definition of how to parse the given content is declared using a context-free grammar, which can be implemented using definite clause grammars (DCGs) in Prolog. This technique offers a huge freedom in defining the Dsl, because with DCGs it is possible to process any string input. Nevertheless, the required source code, especially the definition of the grammar, becomes large even for our simple Dsl for the specification of *if–then* rules. In the following we present an extract of the grammar to parse the Dsl specified in Section 3.2.

```
rule --> "if ", formula, " then ", conjunction.
formula --> conjunction | disjunction | classical_negation.
conjunction --> literal | literal, " and ", formula.
disjunction --> literal | literal, " or ", formula.
classical_negation --> "-", formula.
literal --> finding | "not(", finding, ")".
finding --> feature, "=", value.
```

For the sake of simplicity, we omit the generation of the internal representation the quasi-quotation gets replaced by, and also the handling of non-mandatory whitespaces and the specification of minor DCG rules.

By further rules, we can define features and values as certain strings without the character "=". This DCG can be used for verifying that a rule is in the language. The grammar formalism can help to clarify the syntax for people who are not experts in logic programming or Prolog by returning meaningful error messages. In addition, the domain-specific language can be defined to be more relaxed: rules do not necessarily have to end with a dot, and could be separated by newlines; strings starting with an uppercase letter or containing strings do not have to be encapsulated by quotes. In addition, rules are not required to end with a dot.

Using this grammar, the expert rules can be directly embedded into any Prolog program using the quasi-quotation {|rule|| ... |}:

```
{|rule||
   if Weather = rainy and Umbrella = no
   or Weather = Thunderstorm
   then Clothes = wet. |}
```

Since DCGs are a well-established part of the Prolog standard, this grammar can also be used to parse rules which are specified in a separate text file, without the need of using quasi-quotations.

3.4.2. Dsl portability by the example of JavaScript

Using a separate text file to store the rules it is possible to use and manipulate the rule base with different tools and programming languages. As mentioned before, the approach of extending the Prolog syntax by quasi-quotations has been adopted by other languages. Similar to Prolog, JavaScript had poor support for multi-line strings. The JavaScript equivalent to Prolog's and Haskell's quasi-quotations is called *tagged template strings*. To compare our approach of defining the Dsl in Prolog, we want to present a similar implementation in JavaScript.

Tagged template strings have been introduced to JavaScript in the standard ECMAScript 2015 (formerly ECMAScript 6) [39]. They are of the following basic form:

tag`Content`, e.g., html`<p>Hello, ${name}!</p>`

Similarly to the Prolog approach, a tagged template string of the previous form invokes a function call of the user-defined function tag, i.e., in this example html(content). Within the string content it is possible to embed expressions by using the syntax ${expression}, which is generally used to integrate variables of the same scope, as with name in the example presented before. To process the given string content, it is split by the embedded expressions:

```
let name = 'Alice'
function tag(strings, ...values) {
   return strings[0] + values[0] + strings[1]
}
tag`Hello, ${name}!`   // returns 'Hello, Alice!'
```

Unlike PROLOG, JavaScript has no built-in methods to modify the program parser using a grammar. Instead, third-party libraries have to be used to parse the given content strings and generate the corresponding replacement. Similarly to other languages providing language features like quasi-quotations and tagged template strings, there is no standard like the DCGs known from PROLOG. This makes PROLOG a very good fit to implement DSLs, especially in a rapid prototyping development approach. In the future it might be desirable to support DCGs in other programming languages as well – since it is also just a domain-specific language for specifying how to parse a given text –, so that our language implementation can be used in other environments, too.

3.5. Comparison of the approaches

In the previous Sections 3.3 and 3.4, we have introduced two methods to implement a DSL for *if–then* rules in PROLOG. Both have their advantages: The definition as an internal DSL is only a lightweight extension of the GPL PROLOG and the usage of self-defined operators seems very natural for PROLOG users. In addition, the user can benefit from existing tools to develop PROLOG programs, e.g., an integrated development environment (IDE) or existing workflows for continuous integration (CI), since its DSL extension is in fact consultable and executable PROLOG source code. On the other hand, these advantages apply only for very experienced PROLOG users. For a target audience, which is unlikely to be familiar with PROLOG, the implementation using a more gracious grammar seems to be appropriate.

4. Analysis and evaluation of declarative expert rules

Expert knowledge acquired in empirical studies is a good use-case for the application of deductive databases with *if–then* rules. E.g., in the domain of *change management*, there are many influencing factors in organisations which have been examined in various psychological studies. *If–then* rules over findings have also occurred in the domain of *medical diagnosis* [8].

Both applications have provided perfect case studies for data integration of rules obtained by studies. In [9] we have presented the psychological background from change management, where the emotional processes are modelled for projects introducing new software, and a simple format to represent the obtained rules. The definition of the DSL has been formalised in [10]. The following types of rules have been considered in organisations: explicit, official business processes, and informal rules. Often, the sources of the rules are fragmented, distributed, and hybrid.

As an example, consider the following statement resulting from an expert interview: *In a small business, work processes are comprehensible without frequent team meetings, and no abundance of information arises.* The same applies to large companies with frequent meetings. In natural language, this may be formulated as follows:

> *If either the size of the company is small or the meetings are frequent, then the transparency of the work processes increases and there is no information overload.*

Since we are currently using only the basic operators *and, or*, and *neg*, the statement sketched in the example above can be more formally modeled as follows, resolving the implicit exclusive disjunction:

```
if neg Company Size = small and Meeting = often
   or Company Size = small and neg Meeting = often
then Traceability of Work Processes = rises
    and Information Overload = no
```

Currently, we are integrating about 50 rules obtained by studies in change management. In medical diagnosis, we have been dealing with rule bases of thousands of rules obtained over a longer time from medical doctors.

Embedding this kind of rule as a DSL is a very natural thing to do using the PROLOG user-facing syntax devices, such as the operator precedence and associativity declarations. Conceptually, a DSL may benefit from hooks into the underlying language, which would stand as simple domain-specific constructs. A consequence of having the abstract syntax tree so directly accessible, in the form of PROLOG terms, is that it may easily be stored, investigated, interpreted, combined or rewritten.

4.1. Querying knowledge stored in if–then rules

Both implementation mechanisms presented in Sections 3.3 and 3.4 have in common that the given expert knowledge base is directly executable and that it is possible to run queries on the data. This is trivial for an implementation using PROLOG operators, but it can also be easily seen for the implementation using quasi-quotations, since they get replaced by

nested Prolog terms (as defined in the grammar) at compile time during term expansion. As introduced in Section 3.3, this nested term could, e.g., be retrieved in the interactive Prolog top-level[2] by calling the following query:

```
?- then(if(Condition),Consequence).
Condition = neg 'Company Size' = small and ...,
Consequence = 'Traceability of Work Processes' = rises and ...
```

We have developed the Dsl for intuitively representing the *if–then* rules, which map to Prolog in a simple manner. We use the deductive database system DDbase [4], which works with an extension of Datalog. DDbase is part of the system Declare – formerly called DisLog Developers' Kit (Ddk) – a large collection of Prolog libraries written in Swi-Prolog [19] including features from data and knowledge engineering, databases (relational, Xml, and deductive), ontologies, and non–monotonic reasoning. It can be obtained from `www.ddbase.de`.

4.2. Querying the structure of a declarative rule base

The individual records of the expert knowledge specified as *if–then* rules are usually loaded in the memory of DDbase and analysed with our tool. As mentioned before, it is possible to read the rules as Prolog source code, no matter which of the two presented mechanisms has been used, so they can be inspected and even directly queried from within DDbase.

Using this deductive knowledge base, it is for instance possible to answer the following questions about the given declarative expert knowledge:

- Insights about findings. Which constellation of findings is necessary to derive another finding; are there findings, which are a particularly common cause of a consequence; are there any *killer* findings, that block the application of many rules; what are the necessary conditions for a finding; which ones are optional; where do some findings form opposite or even contradictory relationships?
- Insights about features. If a different value is assigned to a single feature, how does this affect the overall structure?
- Insights about rules. Are there any redundant rules; can some individual rules be expressed by more accurate rules; is it possible to combine multiple rules into a more general form without changing the overall statement?

These questions underline the diversity of queries that can be asked once the expert knowledge has been captured in the story form using our Dsl. We are currently already supporting queries for conditions and consequences of individual findings. For example, by means of the predicate `depends_on`, the following query can be formulated in DDbase (we do not show the encoding here); we can iterate through all answers.[3] The predicate `depends_on` is built to work also for pairs of features instead of just findings. By entering "';" after each answer of the form `Condition = ..., Consequence = ...`, all answers can be listed subsequently:

```
?- F1 = finding:Consequence, F2 = finding:Condition,
   depends_on(F1, F2).
Condition = ('Existence of ERP Knowledge ...' = yes),
Consequence = ('Emergence of ERP Knowledge ...' = yes) ;
Condition = ('Cooperation/Communication ...' = yes),
Consequence = ('Emergence of ERP Knowledge ...' = yes) ;
...
```

Here, not only the contents of individual rules is returned, but also derived knowledge can be computed. As an example, the deductive database can infer that the existence of knowledge about enterprise resource planning (ERP) is a prerequisite for its propagation among employees.

If there is a rule with a condition `A` and the consequence `B` and `B` is a prerequisite for a further consequence `C`, then `A` can be inferred as being a prerequisite for `C`, too. In the system DDbase, it is also possible to immediately determine all causes of an individual finding; for doing this, the consequence can be an argument in the following predicate, as this happens to determine the causes of a conflict:

```
?- F1 = finding:'Emergence of Conflicts' = yes,
   F2 = finding:Precondition,
   depends_on(F1, F2).
Precondition = ('Acceptance ... at Beginning' = partly) ;
Precondition = ('Feedback' = no).
```

For simple values, the tool can also handle classical negations, i.e., above the two findings `neg 'Feedback' = no` and `'Feedback' = yes` are equivalent.

[2] The top-level is the read-eval-print loop (REPL) of Swi-Prolog that takes user input and evaluates them. It is often used to interactively retrieve data that is stored in the program. The prefix `?-` in our code examples denotes the usage of the top-level.

[3] This can be done by entering a semicolon ";" after each answer, standard procedure in a Prolog top-level interpreter.

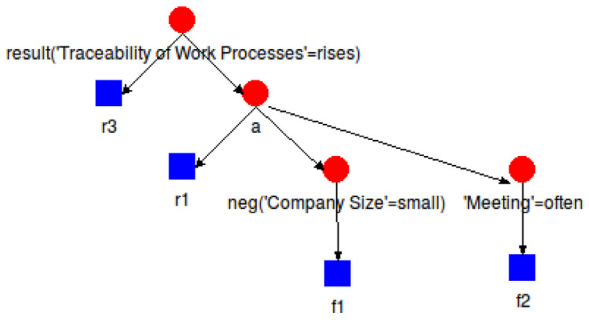

Fig. 1. A derivation tree for expert rules in change management.

Defining similar deduction rules on the findings and values in the rule base, e.g., by specifying synonyms of the variables and input values gained in the positioning phase, this approach strongly supports the discussion phase during the acquisition of the expert knowledge, as introduced in Section 2.3.

4.3. Analysis and visualisation of forward-chaining derivations

The basic DSL of the system DDBASE allows atomic rule heads and conjunctive rule bodies. In DDBASE, it is possible to explain forward–chaining derivations using so-called derivation trees. For a used rule $r = A \leftarrow B_1 \wedge \cdots \wedge B_n$, the derivation tree contains a node A with $n+1$ children representing the rule name r and the body atoms B_1, ..., B_n. For rules with more complicated bodies or heads, it is also possible to define derivation trees.

In the following analysis and visualisation, we quote the features as in an internal DSL, we write the negation of a finding as `neg(A=V)`, and we assume two facts `f1` and `f2`. The rules `r1` and `r2` derive an auxiliary atom `a`, since we cannot use conjunctive heads, from which `result(Finding)` can be derived, which contains a finding.

```
?- Program = [
     f1 = [neg('Company Size' = small)],
     f2 = ['Meeting' = often],
     r1 = [a]-[neg('Company Size' = small), 'Meeting' = often],
     r2 = [a]-['Company Size' = small, neg('Meeting' = often)],
     r3 = [result('Traceability of Work Processes' = rises)]-[a],
     r4 = [result('Information Overload' = no)]-[a] ],
   Query = result(X),
   tp_iteration_dislog_with_proof_trees(Program, Query).
```

For the general query `result(X)`, it is possible to construct two derivations trees. The first one for `result('Traceability of Work Processes' = rises)` is shown in Fig. 1 below, the second one for `result('Information Overload' = no)` would look similar and is thus not shown.

The internal DSL for rules is sufficient here, since the rules can be generated from an external, more human-readable DSL representation.

4.4. Analysis and visualisation of dependency graphs

We have already indicated the advantages of a representation of declarative expert knowledge with *if–then* rules in PROLOG with a focus on the query mechanisms provided by DDBASE. Besides facilitating the dynamic formulation of queries, rules can be used to visualise the expert knowledge and the encoded dependencies.

With the tool VISUR, cf. [8], the given rule base can be visualised, which allows for a graphical interpretation. It had been developed for and used by AI people for the analysis and visualisation of rules in medical diagnosis. Thus findings, which are a prerequisite for a variety of consequences, can also be rendered visually. VISUR has, among other applications, been used for the visualisation of medical diagnoses, where rules assign symptoms to a diagnosis. We have extended the tool for the *if-then* rules specified with our DSL. This provides a schematic representation of the findings: from the features (grey circles), consequences can be visualised depending on the values (which are not shown here). An example application is given in Fig. 2, which illustrates the features and the relevant rules on which the feature `'acceptance of ERP system by employee'` depends transitively. The other nodes (shown by grey circles) are features, which can themselves be influenced by further features.

Interesting applications of dependency graphs are to look for graph-theoretic properties, including, e.g., cycles and connected components in the dependency graphs.

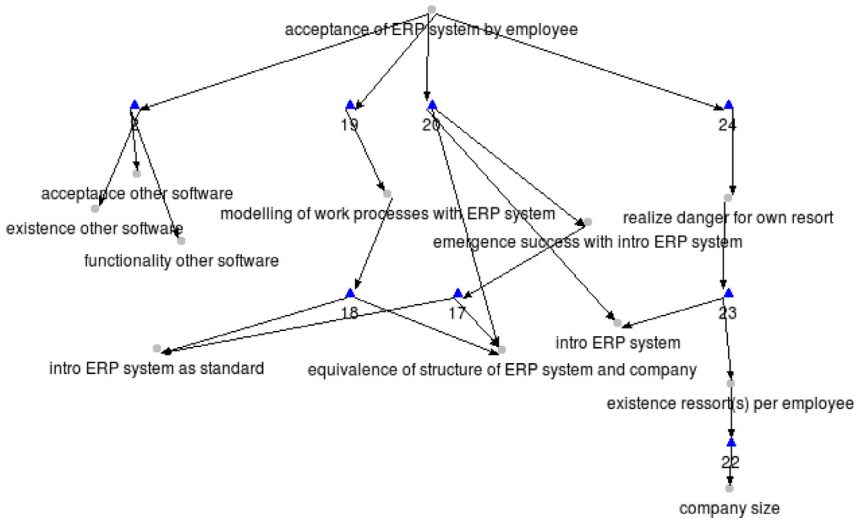

Fig. 2. A dependency graph for the schematic representation of the transitive preconditions and the relevant rules (shown by the blue triangles labelled by `2, 17-20, 22-24`) for deriving the feature `'acceptance of ERP system by employee'`. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article).

Type of Knowledge	Type of Embedding
Annotated Rules	Internal with Grammar
Annotated Findings	Internal with Grammar
Ontologies	External with Quasi Quotations
Disjunctive Rules	Internal with Grammar

Fig. 3. Types of embedding.

4.5. The DSL in the context of other host languages

Even though it is natural to express these rules as a PROLOG dialect, specifying it as a – largely stand-alone – DSL allows for its embedding into other languages and the possible reimplementation using different technologies, in the future. The point is to make the DSL directly usable by domain experts, who are not PROLOG knowledgeable. Its implementation has operational semantics based on the DATALOG engine of DDBASE, relying on a forward-chaining model.

It should be noted that visualisation may not match the front end: intermediately generated atoms are not part of the external DSL; take for instance the auxiliary literal `[a]` in the previous example, which was introduced to model a conjunctive head. One way to handle this sort of situation is to have the visualiser colour these atoms differently, to reflect the fact that they do not have a user-facing meaning.

5. Design of domain-specific languages for more general expert knowledge

One strength of our proposed approach arises from tightly coupled findings. The more the features given in the declarative expert rules are connected, the more the rule base can profit from the design analysis using inference rules and the visualisation of the dependency graphs. Therefore, it might be desirable to improve the rule base in three ways: (1) to extend the rules by meta information, e.g., about its provenance, (2) to annotate the findings with similar information, and (3) to integrate environment knowledge about the features using ontologies. Fig. 3 shows the used types of embeddings.

5.1. Annotated rules

Besides its very good integration into PROLOG, the DSL can be easily extended due to the declarative style of DCGs. As an improvement of the DSL presented in Section 2.3 and in [10], we present a syntax for annotating the rule bases with their provenance and other information. Our aim is to generate a flexible rule base which can take into account additional information provided by probabilities, and the confidence in the rule findings, their authors, and meta-information, e.g., the size of the test group for the psychological studies in change management. This language extension is also the basis for the modularisation of the rule base, as it will simplify the work with a growing number of findings.

We propose the extension of the DCG by several elements to parse rule annotations like in the following example:

```
Author1 says {
   if Meeting = often then Information Overload = no
   if Information Overload = yes
      then Willingness in Meetings = decreasing
 } with Confidence = 0.8
```

This data enhancement by meta-information like the author's name `Author1` and the proposition's confidence will help to not only detect but to resolve contradictory rules. It is possible to trust statements by several authors more than others. Besides this, the extended representation provides a natural way to split large rule bases into logical, self-contained units.

There are various entities which could be taken into account while analysing the given rule base. Besides the pure findings, the rules should be annotated by their contextual information, for example assumptions that have been made for an empirical study. Additionally, provenance information like the data source, the method of achieving the rules, the time period of the investigation, and the confidence should be stored. For an easier integration of the rules, relations of features should be taken into account. In this way, e.g., rules which are verified for a special application are could be generalised to rules about findings in general and vice-versa.

This extension of the given rules can influence the usage of the analysis tools, since additional connections between features will be added. Based on the provenance information, inconsistencies in the rule base can be solved more easily. With the help of rule annotations it is possible to identify highly dependent features that are very unlikely to happen. Especially in a workflow of continuous integration – when new rules are added incrementally – this information can be used to accept or reject the new expert rules. The extensions can be expressed in the extension Datalog* provided by DDbase, which thus can integrate their evaluation in a consistent reasoning system.

5.2. Annotated findings

Similarly, the treatment of confidence values for findings can be achieved. Frequently, collected values can be ambiguous. In our running example, the rule base contains the value `partly` in addition to `yes` and `no`. A more precise value in the form of *relative frequencies* could derive a more accurate form of knowledge.

The following simple example is a general annotated rule, where findings are annotated by values in the form `X:A=V`; the higher precedence of "=" in our Dsl binds the finding `A=V` before it is annotated with the confidence value `X` by ":":

```
if A: Existence of other Software = yes
and B: Functionality of other Software = increasing
and C: Acceptance of other Software = increasing
then D: Acceptance of ERP System = decreasing
with accumulate(conjunction_independence, [A,B,C], D)
```

The symbols `A`, `B`, `C`, and `D` in the rule represent logical variables, which always begin with a capital letter – which is common Prolog. The variables in the rule body are universally quantified; i.e., the statement it is assumed to hold for all suitable findings. The variables are in this case attached to the actual values, so that the unconditional probability can be calculated. Thus, our Dsl makes use of logical variables, and follows the syntax and semantics of predicate logic and its refinements in answer set programming.

In the case of stochastic independence, the predicate `accumulate` can be implemented in Prolog as follows:

```
accumulate(conjunction_independence, Xs, X) :-
   multiply(Xs, X).
```

Observe, that the implementation of `accumulate/3` above works for arbitrary lists `Xs` of values to obtain the product. In DDbase, `multiply/2` is implemented using Prolog meta-predicates. We have also implemented other forms of accumulating lists `Xs` of values, such as, e.g., positive and negative correlation. They can be used within the same knowledge base in Datalog*.

5.3. Integration of ontologies

In addition to the proposed rule base annotations, it might be desirable to integrate external knowledge bases and ontologies, e.g., further meta-information about empirical studies and their publication as conference papers. This can include information about the authors, the paper's citations and reception. As suggested before, ontologies might also help to resolve inconsistencies and to deduct further rules using generalisation.

There are markup languages to acquire rules in a well-defined syntax, e.g., using Owl and RuleML, cf. Section 2.4. Because we want to embrace knowledge experts we chose to implement a simpler syntax to write ontologies as a Dsl. Using Prolog's quasi-quotation syntax, they can easily be integrated into the rule bases.

A Dsl can be based on the Terse Rdf Triple Language *Turtle* [40], which expresses triples in a syntax similar to Sparql. Each triple consists of a subject, a predicate and an object. The Turtle format is suitable for manually editing the triple information. For example, the provenance of an empirical study could be described as follows. The paper was written by

author Author1, whose email address is author1@example.org and who has a reputation of 12.5. The final version of the paper is based on an earlier draft.

```
@prefix pub: <http://example.org/pub#>
<pub:author1>
   pub:name "Author1"
   pub:email "author1@example.org"
   pub:reputation "12.5"
<pub:article>
   pub:title "Results from Case Study"
   pub:author <pub:author1>
   pub:draft <pub:draft>
<pub:draft>
   pub:title "Latest Results"
   pub:author <pub:author1>
```

Each @prefix handles a specific domain. The previous example can be read as follows: *The entity ex:author1 has a name of Author1*. The same syntax can be used to specify relations between features used in the rule base, like in the following example:

```
@prefix ent: <http://example.org/ex#>
<ent:erp-system>
   ent:label "ERP System"
   ent:is <ent:business-software>
<ent:business-software>
   ent:label "Business Software"
   ent:is <ent:software>
<ent:software>
   ent:label "Software"
```

Based on this ontology, we can use the additional information that an ERP system is a business software – which is some software. This can result in additional findings and rules for the more general business software.

SWI-PROLOG [19], as one of the most popular PROLOG implementations, already includes PROLOG predicates to process RDF in this Turtle syntax [30]. Using its built-in predicate[4] rdf_read_turtle/3, it is easily possible to define a quasi-quotation with the tag turtle, to embed the Turtle syntax directly into existing PROLOG source code. Then, the RDF triples can be queried using the rdf/3 predicate. When one or more arguments are unbound in the query, all matching solutions are retrieved using PROLOG's backtracking technique. In this way, joins can be stated naturally by calling the rdf/3 predicates with common variables. In the following example we define a PROLOG predicate trusted_article/2 for retrieving all articles with at least one author with a reputation of at least 10:

```
trusted_article(Article, Reputation) :-
   rdf(Article, pub:author, Author),
   rdf(Author, pub:reputation, Reputation),
   Reputation > 10.
```

5.4. Rules with disjunctive consequences

So far, we allow for rule heads with formulas linking findings by the connective and for conjunction; if Consequences is a conjunction, then the rule can be normalised to several rules using macro expansion techniques in PROLOG. More general rules over the sentential connectives and and or can be transformed to several rules with disjunctive Consequences. So far, the domain experts have not used disjunctive Consequences in applications; at the moment, they are not accustomed to use disjunctions in rule heads. In the future, we will try to introduce such a new feature into applications, following the theoretical work of [16]. Especially the handling of confidence values together with disjunctive Consequences will be an interesting research field.

6. Conclusion

Domain-specific languages are often distinguished in modeling and programming languages. Expert knowledge acquired in interviews is a classical example of knowledge which is – if stored in an unstructured format – not intended to be

[4] Note that RDF and PROLOG use the same terminology *predicate*. In PROLOG, predicates of the form name/arity hold the program's information, e.g., member(1,[1,2]).

executed. In our work we have defined a DSL which resembles the textual format of *if–then* rules often stated as interview answers. We have presented the syntax and semantics of this language and introduced two implementations: as an internal DSL using self-defined operators in PROLOG, and as an external DSL using term expansion and quasi-quotations.

Because of its integration in the PROLOG-based deductive database system DDBASE, the expert knowledge in the form of declarative rules becomes executable and can be enhanced with techniques from deductive databases. In case studies in the domains of medical diagnosis [27] and change management in organisational psychology [9], we have demonstrated the practical usefulness of the proposed approach; the analysis and visualisation of rules is used successfully by AI people for medical diagnosis.

In the future, we are planning to apply knowledge engineering techniques, such as refactoring approaches [41], to the deductive rule bases. This will support the discussion phase when collecting expert knowledge and integrating multiple data sources.

We will also incorporate further aspects of hybrid information sources and *contextual annotations* by, e.g., uncertainty and provenance information. Regarding the latter, it could be useful to model confidence and uncertainty with concepts from annotated logic programming [42,43] and probabilistic-enabled logic programming languages, such as ProbLog [44], and to analyse and support the knowledge engineering and reasoning process for hybrid knowledge bases including these concepts.

Other extensions might deal with uncertain knowledge in the form of *disjunction* in the rule heads (conclusions), as described in, e.g., [16]. We expect that, especially, the combined handling of confidence values and disjunctive rules will be an interesting research field.

The expressiveness of the DSL is critical to its success, and thus we shall be looking into integrating useful extensions which relate to logic programming, such as *temporal knowledge*. A notable example is Temporal Constrained Objects (TCOB) [45], which provides a way in which application-domain entities may be viewed as time-varying and have their values at different times integrated in a single sentence. Another possible approach is proposed as Temporal Contextual Logic Programming (TCOP) [46], which integrates notions of change in a logic programming framework, thereby providing groundwork constructs useful in a change management DSL.

Appendix A. Evaluation of declarative expert rules

A comprehensive description of the syntax and semantics of deductive databases and logic programming is given, e.g., in [16]. Often, deductive databases consist of DATALOG programs. For instance, the well-known transitive closure rules can be expressed as a DATALOG program, a logic program without default negation:

$$tc(X,Y) \leftarrow arc(X,Y),$$
$$tc(X,Y) \leftarrow arc(X,Z) \wedge tc(Z,Y).$$

In the following, we give a brief summary some concepts about Herbrand interpretations that are necessary for explaining the semantics of logic programs, and then we sketch the semantics of logic programs with or without default negation.

Herbrand interpretations. In logic programming, terms are defined inductively: terms can be variable symbols or constant symbols or of the form $f(t_1,\ldots,t_n)$, where f is a function symbol and $t_1,\ldots,t_n$ are terms themselves. An atom is of the form $p(t_1,\ldots,t_n)$, where p is a predicate symbol and $t_1,\ldots,t_n$ are terms. A ground atom is an atom without variable symbols. E.g., *arc*(*a, b*) is a ground atom with the predicate symbol *arc*, where the ground terms $t_1 = a$ and $t_2 = b$ are constants (strings starting with a lower case character), and the atom *arc*(*X, Y*) contains the variable symbols X and Y (strings starting with an upper case character). The Herbrand base $HB_{\mathcal{P}}$ is the set of all ground atoms over the logic program $\mathcal{P}$, i.e., their predicate, function, constant and variable symbols must occur in $\mathcal{P}$. An Herbrand interpretation I is a subset of $HB_{\mathcal{P}}$.

Semantics without default negation. The semantics of logic programs without default negation can be defined equivalently in three ways: using a logical model theory, a proof theory (consequence operator), and a fixpoint theory.

Assuming the standard definition, we write $I \models \beta$, if I models β. Here, $I(\text{not } \phi) = \neg I(\phi)$ and $I(\phi_1 \odot \phi_2) = I(\phi_1) \odot I(\phi_2)$, for formulas ϕ, ϕ_1, ϕ_2, and connectives $\odot = \vee, \wedge$. A ground rule $A \leftarrow \beta \in gnd\,(\mathcal{P})$ is obtained by substituting all variable symbols of a rule by ground terms. The immediate consequence operator $\mathcal{T}_{\mathcal{P}}$ derives all ground atoms A, such that there exists a ground rule in $gnd\,(\mathcal{P})$, where I models its body:

$$\mathcal{T}_{\mathcal{P}}(I) = \{\ A \in HB_{\mathcal{P}} \mid A \leftarrow \beta \in gnd\,(\mathcal{P}),\ I \models \beta\ \}.$$

Since the rules are range-restricted, $\mathcal{T}_{\mathcal{P}}(I)$ will be finite, if I is finite. E.g., for the transitive closure rules together with the facts *arc*(*a, b*), *arc*(*b, c*), *arc*(*c, d*), the *bottom-up evaluation* derives the following monotonically increasing sequence of interpretations by repeatedly applying the rules to the already derived facts:

$$I_0 = \emptyset,$$
$$I_1 = \{\ arc(a,b),\ arc(b,c),\ arc(c,d)\ \},$$
$$I_2 = I_1 \cup \{\ tc(a,b),\ tc(b,c),\ tc(c,d)\ \},$$

$$I_3 = I_2 \cup \{\, tc(a,c),\ tc(b,d) \,\},$$
$$I_n = I_3 \cup \{\, tc(a,d) \,\},\ \text{for all } n \geq 4.$$

For $n \in IN_0$, the interpretation $I_n = \mathcal{T}_{\mathcal{P}}^n$ is obtained by the repeated application of $\mathcal{T}_{\mathcal{P}}$, starting with $\mathcal{T}_{\mathcal{P}}^0 = \emptyset$, i.e. $\mathcal{T}_{\mathcal{P}}^{n+1} = \mathcal{T}_{\mathcal{P}}(\mathcal{T}_{\mathcal{P}}^n)$. The least fixpoint of the consequence operator – here I_4 – is also the unique minimal model of the logic program. Observe, that the least fixpoint is $\mathcal{T}_{\mathcal{P}}^{\omega} = \cup_{n=0}^{\omega} \mathcal{T}_{\mathcal{P}}^n$. In theory, it can be infinite, if the Herbrand base is infinite since $\mathcal{P}$ contains function symbols. In practice, the rules have to ensure that the iteration terminates after finitely many steps with a finite fixpoint. The consequence operator and its iteration provide one proof-theoretic (operational) semantics of a logic program without default negation, i.e., an evaluation method.

Semantics with default negation. In general, the semantics of a logic program with default negation is given by its answer sets, cf. [16]. For our purposes, however, it is sufficient to consider logic programs with a limited use of default negation, so-called stratified programs, where there is no recursion through default negation.

The evaluation of stratified programs can be based on logic programs without default negation at all. These programs – the transitive closure program above is an example – can be evaluated bottom-up using hyperresolution in an efficient bottom-up style. Then, *declarativity* is given by the fact that without default negation, three semantics coincide: model, proof, and fixpoint theory.

In general, the answer set semantics of logic programs with unlimited default negation is defined by a fixpoint theory. This can also be extended to handle literals with classical negation ($\neg$), rather than just atoms. In non-monotonic reasoning with answer sets, we distinguish between true literals in an answer set and literals derived by the inference process. Intuitively, a default negated literal $\mathsf{not}\, A$ is considered true in an answer set, if the atom A cannot be derived, whereas a classically negated literal $\neg A$ is considered true, if $\neg A$ can be derived.

References

[1] Kosar T, Oliveira N, Mernik M, Pereira MJV, Črepinšek M, da Cruz D, et al. Comparing general-purpose and domain-specific languages: an empirical study. Comput Sci Inf Syst 2010;7(2):247–264.
[2] Mernik M, Heering J, Sloane AM. When and how to develop domain-specific languages. ACM Comput Surv 2005;37(4):316–44.
[3] Sun Y, Demirezen Z, Mernik M, Gray J, Bryant B. Is my DSL a modeling or programming language? In: Proceedings of 2nd International Workshop on Domain-Specific Program Development (DSPD). Tennessee: Nashville; 2008. p. 4.
[4] Seipel D. Knowledge engineering for hybrid deductive databases. In: Proceedings of the 29th workshop on logic programming (WLP 2015); 2015.
[5] Pereira FC, Warren DH. Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. Artif Intell 1980;13(3):231–78.
[6] Wielemaker J, Hendricks M. Why it's nice to be quoted: quasiquoting for prolog. In: Proceedings of the 23rd workshop on logic-based methods in programming environments (WLPE 2013); 2013.
[7] Kosar T, Mernik M. Embedded domain-specific languages in prolog. Acta Electrotech Inf 2006;6(1):3.
[8] Seipel D, Baumeister J, Hopfner M. Declaratively querying and visualizing knowledge bases in XML. In: Proceedings of the 15th international conference on applications of declarative programming and knowledge management (INAP 2004). Springer; 2005. p. 16–31. LNAI 3392
[9] von der Weth R, Seipel D, Schubach K, Nogatz F, Werner A. Modellierung von Handlungswissen aus fragmentiertem und heterogenem Rohdatenmaterial durch inkrementelle Verfeinerung in einem Regelbanksystem. Journal Psychologie des Alltagshandelns 2016;9(2):33–48.
[10] Seipel D, von der Weth R, Abreu S, Nogatz F, Werner A. Declarative rules for annotated expert knowledge in change management. In: Proceedings of the 5th symposium on languages, applications and technologies (SLATE), 51; 2016.
[11] Seipel D, Baumeister J. Declarative specification and interpretation of rule-based systems. In: Proceedings of the 21st international Florida artificial intelligence research society conference (FLAIRS 2008). AAAI Press; 2008. p. 359–64.
[12] Elmasri R, Navathe SB. Fundamentals of database systems. 7th ed. Benjamin Cummings; 2015.
[13] Ceri S, Gottlob G, Tanca L. Logic programming and databases. Berlin: Springer; 1990.
[14] Ullman J. Principles of database and knowledge-base systems, volume I. Computer Science Press; 1988.
[15] Ullman J. Principles of database and knowledge-base systems, volume II. Computer Science Press; 1989.
[16] Minker J, Seipel D, Zaniolo C. Logic and databases: history of deductive databases. Handbook of the History of Logic, 9. NorthHolland: Computational Logic; 2014.
[17] Chakravarthy US, Fishman DH, Minker J. Semantic query optimization in expert systems and database systems. In: Proceedings of the 1st international workshop on expert database systems; 1986. p. 659–74.
[18] Bratko I. Prolog programming for artificial intelligence. 4th ed. Addison-Wesley Longman; 2011.
[19] Wielemaker J. An overview of the SWI-Prolog programming environment. In: Proceedings of the 13th international workshop on logic programming environments (WLPE); 2003. p. 1–16.
[20] Abiteboul S.. DATALOG: La renaissance. http://www.college-de-france.fr/site/serge-abiteboul/course-2012-05-09-10h00.htm; 2012.
[21] Wright G, Ayton P. Eliciting and modelling expert knowledge. Decis Support Syst 1987;3(1):13–26.
[22] Ford DN, Sterman JD. Expert knowledge elicitation to improve formal and mental models. Syst Dyn Rev 1998;14(4):309–40.
[23] Horrocks I, Patel-Schneider PF, Bechhofer S, Tsarkov D. OWL Rules: a proposal and prototype implementation. J Web Semant 2005;3(1):23–40.
[24] I. Horrocks P.F. Patel-Schneider B. Harold T. Said G. Benjamin D. Mike et al. SWRL: a semantic web rule language combining OWL and RuleML. W3C Member submission2004; 21:79.
[25] Boley H, Tabet S, Wagner G. Design rationale of RuleML: a markup language for semantic web rules. In: Proceedings of the first international conference on semantic web working. CEUR-WS. org; 2001. p. 381–401.
[26] Wagner G, Giurca A, Lukichev S. A usable interchange format for rich syntax rules integrating OCL, RuleML and SWRL. In: Proceedings of the workshop reasoning on the web; 2006.
[27] Baumeister J, Seipel D. Anomalies in ontologies with rules. J. Web Semant Sci Serv Agents World Wide Web 2010;8(1):55–68.
[28] Clark KL, McCabe FG. Prolog: a language for implementing expert systems. Imperial College of Science and Technology. Department of Computing; 1980.
[29] Parsaye K. Database management, knowledge base management, and expert system development in prolog. In: Databases for business and office applications, database week. ACM; 1983. p. 159–78.
[30] Wielemaker J, Hildebrand M, van Ossenbruggen J. Using prolog as the fundament for applications on the semantic web. In: Proceedings of the 2nd international workshop on applications of logic programming in the semantic web and semantic web services (ALPSWS2007); 2007. p. 84–98.

[31] Wielemaker J, Beek W, Hildebrand M, van Ossenbruggen J. ClioPatria: a SWI-prolog infrastructure for the semantic web. Semant Web 2016;7(5):529–41.
[32] Cooke NM, McDonald JE. A formal methodology for acquiring and representing expert knowledge. Proc IEEE 1986;74(10):1422–30.
[33] Gupta G, Pontelli E. Specification, implementation, and verification, of domain specific languages: a logic programming-based approach. In: Computational logic: logic programming and beyond. Springer; 2002. p. 211–39. LNCS 2407
[34] Fowler M. Domain-specific languages. Addison–Wesley; 2011.
[35] Crockford D.. The application/json media type for javascript object notation (JSON). Internet RFC 4627, July 2006.
[36] Kosar T, Bohra S, Mernik M. Domain-specific languages: a systematic mapping study. Inf Softw Technol 2016;71:77–91.
[37] Wielemaker J, Angelopoulos N. Syntactic integration of external languages in Prolog. In: Proceedings of the ICLP workshop on logic-based methods in programming environments (WLPE12); 2012. p. 40–50.
[38] Nogatz F, Seipel D. Implementing GraphQL as a query language for deductive databases in swi-prolog using dcgs, quasi quotations, and dicts. In: Proceedings of the 30th workshop on logic programming (WLP 2016); 2016.
[39] W-B, Allen. ECMAScript 2015 language specification. 2015.
[40] D. Beckett, T. Berners–Lee, E. Prudhommeaux. Turtle-terse RDF triple language. W3C Team Submission2008; 14:7.
[41] Fowler M. Refactoring – improving the design of existing code. Addison–Wesley; 1999.
[42] Lakshmanan LV, Sadri F. On a theory of probabilistic deductive databases. Theory Pract Logic Program 2001;1:5–42.
[43] Kifer M, Subrahmanian VS. Theory of generalized annotated logic programming and its applications. J Logic Program 1992;12(4):335–68.
[44] De Raedt L, Kimmig A, Toivonen H. ProbLog: a probabilistic prolog and its application in link discovery. In: Proceedings of the 20th international joint conference on artifical intelligence (IJCAI); 2007. p. 2468–73.
[45] Kannimoola J, Jayaraman B, Achuthan K. Temporal constrained objects: application and implementation. Comput Lang Syst Struct 2017;49:82–100.
[46] Nogueira V, Abreu S. Temporal contextual logic programming. Electron Notes Theor Comput Sci 2007;177:219–33.