# Declarative Rules for Annotated Expert Knowledge in Change Management

**Dietmar Seipel[1], Rüdiger von der Weth[2], Salvador Abreu[3], Falco Nogatz[4], and Alexander Werner[5]**

1 **Department of Computer Science, University of Würzburg, Würzburg, Germany**
dietmar.seipel@uni-wuerzburg.de
2 **Fac. of Business Admin., Dresden University of Applied Sciences, Dresden, Germany**
weth@htw-dresden.de
3 **LISP and Department of Computer Science, University of Évora, Évora, Portugal**
spa@di.uevora.pt
4 **Department of Computer Science, University of Würzburg, Würzburg, Germany**
falco.nogatz@uni-wuerzburg.de
5 **Fac. of Business Admin., Dresden University of Applied Sciences, Dresden, Germany**
alexander.werner@htw-dresden.de

## Abstract

In this paper, we use declarative and domain–specific languages for representing expert knowledge in the field of change management in organisational psychology. Expert rules obtained in practical case studies are represented as declarative rules in a deductive database. The expert rules are annotated by information describing their provenance and confidence. Additional provenance information for the whole – or parts of the – rule base can be given by ontologies.

Deductive databases allow for declaratively defining the *semantics* of the expert knowledge with rules; the evaluation of the rules can be optimised and the inference mechanisms could be changed, since they are specified in an abstract way. As the logical syntax of rules had been a problem in previous applications of deductive databases, we use specially designed domain–specific languages to make the rule *syntax* easier for non–programmers.

The semantics of the whole knowledge base is declarative. The rules are written declaratively in an extension DATALOG* of the well–known deductive database language DATALOG on the data level, and additional DATALOG* rules can configure the processing of the annotated rules and the ontologies.

**1998 ACM Subject Classification** H.2 Database Management, I.2.1 Applications and Expert Systems, I.2.4 Knowledge Representation Formalisms and Methods

**Keywords and phrases** declarative, DATALOG, PROLOG, domain-specific, change management

**Digital Object Identifier** 10.4230/OASIcs.SLATE.2016.7

## 1 Introduction

There have been many rule–based approaches for knowledge representation, but the declarative approach of *deductive databases* appears very promising. Especially when armed with


domain–specific languages, it becomes much easier to incorporate domain expertise into the development process of an information system.

The knowledge in organisations plays an important role in day–to–day business, even more if routines and organisational structures have to be changed. Knowledge in organisations is represented in many ways. Some rules are documented explicitly in rules but there are also informal procedural rules, which are mostly undocumented. A reason why it is so difficult to compare these implicit and explicit rules is because they exist fragmented in different sources (e.g. individuals, groups) and can not be collected in a standardised manner. An approach to solve this problem is to transfer differently collected information about procedures in a standard format using content analysis, which allows to manage the rules in a deductive database system. The system DDBASE [17] allows to analyse rules during input and to link them in the evaluation process. Both a graphical visualiser interface and automated reasoning facilitate the linking of conclusions and help to detect contradictions. An expandable rule base enables the extension of an overall model and its validation across methodologically different studies. Having a semantically sound and functionally rich declarative rule language is an enabling factor when presented as a domain–specific language (DSL). These results pave the way for the comparison of informal knowledge and official rules as well as documenting and modelling their history accurately.

We have described a rule concept for knowledge in change management and some methods for querying and visualizing the rules from the point of view of organizational psychology in [21]; the syntax of the rules has been modelled using operator precedences yielding an internal DSL in PROLOG. In the present paper, we give a context–free grammar for the rule language, which might be textended later, and we provide a theory about the semantics and evaluation of declarative rule bases following concepts from DATALOG and logic programming in general. Moreover, it is necessary to include provenance information, that can be given by ontologies, and confidence annotations of the rules.

### Organization of the Paper

The rest of this paper is organised as follows: Section 2 recalls some basic ideas from declarative programming, domain–specific languages, and deductive databases. Section 3 is about declarative rule bases; it introduces rules for change management and defines syntax, semantics, and evaluation using deductive databases and logic programming. Section 4 shows how the rule base can be represented using a suitable DSL. The analysis of rule bases is investigated in Section 5; ideas for queries and the visual analysis in interactive rule editors are given. Section 6 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 Background Concepts

In this section, we recall some concepts useful for reading the rest of the article, namely declarative programming, domain–specific languages, deductive databases and logic programming. We will mainly summarize and highlight some of the general statements found in literature.

### 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 times, as known from information systems with relational databases. There exists a plethora of results about query optimisation in relational and deductive databases, e.g., [7, 4, 19, 20, 14]. For instance, Minker and his students have interesting results in the field of semantic query optimisation [5]: 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 the 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 programming (e.g. PROLOG), and functional programming.

## 2.2 Domain Specific Languages

A domain–specific language (DSL) is a computer language specifically tailored to a particular application domain, in contrast to a general–purpose language (GPL), which aims for broad applicablility across domains. The *syntax* of a DSL is meant to be intelligible for domain experts. According to Martin Fowler [9], *DSLs are small languages, focused on a particular aspect of a software system*, i.e. they cannot be used to write a whole program, although it is frequent to resort to multiple DSLs in a single system, which is basically written in a GPL.

DSLs can take on two forms: *external* or *internal*. The first form is parsed independently of the host GPL, for instance CSS or regular expressions. Internal DSLs, also known as *embedded* DSLs, are a dialect of a host programming language, and are intrinsically part of the host syntax; they amount to an API in a general–purpose language.

DSLs are appreciated because, for its target domain, a DSL is much easier to wield than either a GPL or a traditional library. The outcome is increased programmer productivity, which is always welcome. Having a DSL also improves communication with the domain experts. In short, a DSL raises the level of abstraction required to program an application, allowing non–computer savvy experts to work more productively.

There are DSLs for numerous areas of application, such as, e.g., expert rules, business rules, configuration rules/constraints, and queries to databases. A systematic mapping study has been given in [12].

## 2.3 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 [4]. 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 [3, 22], and whose declarative bottom–up semantics, given in Section 3, is closer to relational databases.

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 [1, 14].

## 3 Declarative Rule Bases

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. We have built tools for managing and analysing relations, ontologies, and rules. Techniques from deductive databases and logic programming can integrate hybrid knowledge bases with structured knowledge. We will show how domain–specific languages and declarative languages can offer a clear syntax and semantics to modern information systems. 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.

### 3.1 Declarative Rules in Change Management

Currently, the results obtained in psychological studies of organisations are not collected in a uniform data format. The data are kept in proprietary systems, which so far only serve for persistent storage without trying to obtain new insights. Some databases are used, but joining the data records and the underlying research results remained almost impossible for lack of integration. The field of *change management* offered a perfect case study for an integrated rule management [21]. 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. To collect and manage the rules systematically, we have used the Prolog–based deductive database system DDbase [17]. We have investigated the analysis, evaluation, and visualisation of the rule base as well as reasoning techniques. Since we use a deductive database system, we are able to do a *continuous integration* of further rules. Based on the facilities of DDbase, we could perform a comparison of informal and official rules.

We are developing a textual, logic–based rule format, which tries to represent the rules – as far as possible – in a natural language syntax. We have modelled the emotional processes in connection with projects for introducing new software. Currently, we have about 50 rules including the rules shown below. The rules are relevant for companies that are considering to introduce an ERP system. E.g., the second rule states that the acceptance of an ERP system is decreasing if other software exists and the functionality and the acceptance of other Software is increasing.

```
if 'Processes in ERP System' = partly
then 'Processes in other Software' = partly .

if 'Existence of other Software' = yes
and 'Functionality of other Software' = increasing
and 'Acceptance of other Software' = increasing
then 'Acceptance of ERP System' = decreasing .

if 'Use of other Software' = increasing/constant
```

```
then 'Acceptance of ERP System by Users' = decreasing .

if 'Test of ERP System by Users' = yes
then 'Discovery of ERP Function by Users' = yes .
```

We have developed a DSL for intuitively representing the business rules, which maps to PROLOG in simple manner. We have also implemented mechanisms for analysing and visualising the rule base. We use the deductive database system DDBASE, which works with an extension of DATALOG, a logic programming language extending the well–known relational query language SQL.

## 3.2 Deductive Databases and Logic Programming

A comprehensive description of the syntax and semantics of deductive databases and logic programming is given, e.g., in [14]. A logic program $\mathcal{P}$ is a set of rules, which are range–restricted implications $A \leftarrow \beta$, where $A$ is an atom and $\beta$ can be any formula over atoms built with the junctors $\vee$, $\wedge$, and not (default negation). Some extensions can also handle literals with classical negation ($\neg$), rather than just atoms in the rules. We allow for function symbols and an arbitray use of the junctors in the rules, whereas frequently in deductive databases $\beta = B_1 \wedge \ldots \wedge B_m \wedge \mathsf{not}\ C_1 \wedge \ldots \wedge \mathsf{not}\ C_n$ is just taken as a conjunction of atoms $B_i$ or default negated atoms not $C_i$ without function symbols. $A$ is called the head, and $\beta$ 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 $\phi$ 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. For instance, the well–known transitive closure rules can be expressed as:

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

## Semantics and Evaluation

In logic programming, terms are defined inductively: terms can be variable symbols or constant symbols or of the form $f(t_1,...,t_n)$, where $f$ is a function symbol and $t_1,...,t_n$ are terms themselves. An atom is of the form $p(t_1,...,t_n)$, where $p$ is a predicate symbol and $t_1,...,t_n$ 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 $P$. An Herbrand interpretation $I$ is a subset of $HB_{\mathcal{P}}$.

### Consequences and Evaluation

Assuming the standard definition, we write $I \models \beta$, if $I$ models $\beta$. Here, $I(\mathsf{not}\ \phi) = \neg I(\phi)$ and $I(\phi_1 \odot \phi_2) = I(\phi_1) \odot I(\phi_2)$, for formulas $\phi$, $\phi_1$, $\phi_2$, and junctors $\odot = \vee, \wedge$. 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:

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

For $n \in I\!N_0$, the interpretation $I_n = \mathcal{T}_\mathcal{P}^n$ is obtained by the repeated application of $\mathcal{T}_\mathcal{P}$, starting with $I_0 = \mathcal{T}_\mathcal{P}^0 = \emptyset$, 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 $I_\omega = \mathcal{T}_\mathcal{P}^\omega = \cup_{n=0}^\omega \mathcal{T}_\mathcal{P}^n$. In theory, it can be infinite, if the Herbrand base is infinte 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

In general, the *semantics* of a logic program with default negation is given by its answer sets, cf. [14]. 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 distiguish 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 the negated atom $\neg A$ can be derived.

During the first phase of the case study in change management, we needed only basic semantic concepts, since the main focus had been on the language and the structural analysis and visualisation of the rule base. For the future, it will be important that the semantics of the target language is precisely defined, a characteristic we inherit from logic programming and DATALOG.

## Renaissance of Datalog

For several years, we can observe what is sometimes called a *renaissance* of DATALOG [1]. 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 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

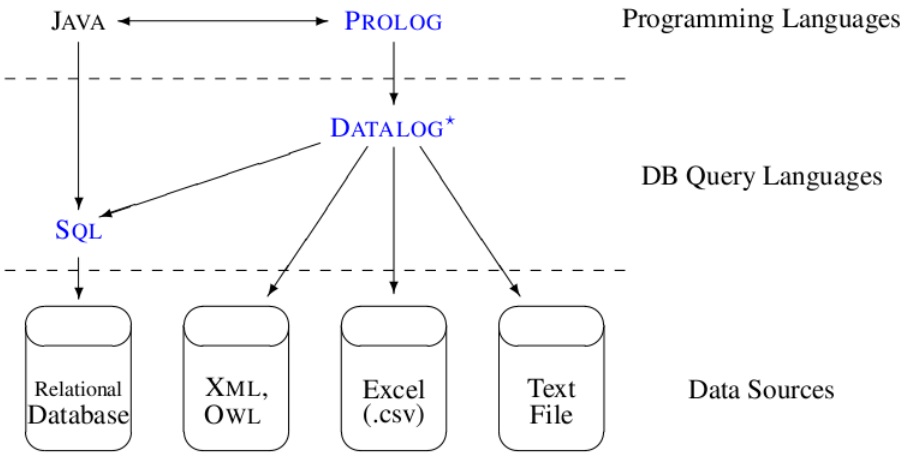


**Figure 1** Architecture of DDBASE.

company Lixto on declarative web data extraction and annotation exists since the beginning of the millennium, and Gottlob's database group has recent publications with Oracle. SAP uses SWI–PROLOG [22] in its Cloud Platform HANA (configuration with source repository git/gerrit), just to name a few.

## 3.3 The Deductive Database System DDbase

The deductive database system DDBASE uses the extension DATALOG* with function symbols [16, 17]. Rule bodies can contain embedded PROLOG calls and default negation. In DDBASE, it is possible to have bottom–up and top–down evaluation in one system. DATALOG* can evaluate logic programs with PROLOG syntax (extended DATALOG programs) in a bottom–up style. Thus, the main evaluation method of DATALOG* is bottom–up; but DATALOG* is designed to evaluate embedded PROLOG calls in a top–down manner. In DDBASE, a logic program can be abstracted by a predicate dependency or a rule predicate graph, and a derivation can be visualised by its proof tree, cf. [4].

The architecture of DDBASE is given in Figure 1, which also shows that DDBASE can access hybrid data sources.

DDBASE is part of the DDK, the DISLOG developers' kit, a collection of PROLOG libraries written in SWI–PROLOG [22] including features from data and knowledge engineering, databases (relational, XML, and deductive), ontologies, and non–monotonic reasoning. It can be obtained from `http://www.ddbase.de`.

# 4 A Domain Specific Language for Rules

For expressing rules, we chose to follow the general form:

if *Condition* then *Consequence*.

Notwithstanding the previous schematic statement form, every rule is required to *end with a dot*. The dot allows the language to behave as an embedded DSL for PROLOG. With minor changes, it could be embedded into other host languages such as Python or Javascript.

The DSL has been conveniently defined in Prolog by a collection of suitable operator precedences. After the keywords `if` and `then`, a *Condition* and *Consequence*, respectively, is expected. Both are so–called junctions of findings. If *Condition* is empty, the rule is also called a *fact*. A finding always has the form: `Feature = Value`, where `Feature` and `Value` should generally be included in quotation marks. Only in strings that neither contain spaces nor start with a capital letter, the quotes can be omitted. The values are not limited to `yes` and `no`, for instance other literal descriptions as in the finding `'Acceptance' = increases` or numerical information, which can assume significant practical importance, are possible. Besides equality, additional comparators may be used.

## 4.1 Syntax of Formulas

Several findings in `Condition` and `Consequence` can be linked to formulas by connectives. 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.

Our domain–specific rule language could be described by a context–free grammar, which we could also implement using the following *definite clause grammar* (DCG) in Prolog.

```
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.
```

By further rules, we can define that features and values are certains 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. At the moment, however, we are not using the DCG. Instead, we have embedded the rule DSL internally into Prolog by providing suitable operator defintions for the junctors `if`, `then`, `and`, `or`, `neg`, etc. Thechnically, the rules `if Formula then Conjunction` can be parsed by Prolog into Prolog structures `then(if(Formula), Conjunction)`. From these structures, the rule base of our system can be derived easily and analyzed by our tool. In future work, we might need a more powerful rule language that cannot be an internal DSL in Prolog. In that case, will try to use refinements and extensions of the given DCG formalism to define a suitable external DSL.

W.r.t. the syntax given in Section 3.2, the findings `A=V` are the atoms in the rules; they have the binary predicate symbol "`=`". More general findings `A`$\odot$`V` can use other comparator predicate symbols "$\odot$", which could be, e.g., one of `=,<,>,=<,>=`. Moreover, also other atoms are possible in Datalog$^*$, e.g. for embedded calls with built–in predicates. In Datalog$^*$, the rules are evaluated bottom–up, and the embedded calls are evaluated in a top–down style, as it is common in logic programming approaches, cf. Section 3.3. Within a formula `not` $F$ with default negation "`not`", no other default negation is allowed, but classical negation "$\neg$" is allowed; i.e., $F$ can only be built using the classcical junctors $\wedge$, $\vee$, and $\neg$.

## An Example from Change Management

As a more complex example, consider the following statement:

> *In 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.*

The syntax of the rules must follow the form `if` *Condition* `then` *Consequence*. The statement above is only true in the case of exclusive ors: either the company is small – then no meetings are needed – or it is so large that team meetings are needed to track work processes without an excess of information. If both premises are met (i.e., there would be frequent business meetings in a small business), then the consequence `'information overload' = no` would not be true.

The given example illustrates that the formal recording of statements by predicate logic formulas can sharpen the uniqueness of the resulting statements. In the case of the more general $F$ `or` $G$ instead of `either` $F$ `or` $G$, the application of the rule along with other statements might lead to inconsistencies. The detection of such situations is part of the functionality of our tool.

The syntax for the rule storage allows the basic conjunction `and` and disjunction `or`. Moreover, classical negation is supported, which is denoted by `neg` in rules. The exclusive or, $F \oplus G$ (either $F$ or $G$), can be expressed as $(F \wedge \neg G) \vee (\neg F \wedge G)$ using elementary connectives. Thus, the statement sketched in the example above can be modeled using elementary connectives as follows:

```
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.
```

In this case, we do not need any brackets. The precedences 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`.

## Variants of Rules and the DSL

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 text editor. It may be gradually expanded by adding new rules. The end result is an incremental rule storage containing all statements found from research results to be analysed later.

We allow for formulas linking findings by the connectives `and` and `or`. If `Consequences` is a conjunction, then the rule can be normalized to several rules using macro expansion

techniques in PROLOG. More general rules over the junctors `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 that new feature into applications. Especially the handling of confidence values together with disjunctive `Consequences` will be an interesting research field.

## 4.2 The Integrated Development Environment (IDE)

The current tool will allow sloppy input that gets normalized. More powerful rule *editors* are planned for the future, so as to avoid having to directly alter the rule file. These advanced editors will facilitate the insertion of rules, to embody the intention that rules should look like natural language statements. Additionally, the editing tool will try to correct frequently occurring syntactic errors, for example, forgetting the dot at the end of the rule or the wrong notation and use of the junctors; these are potential sources of easy–to–correct errors.

Having a graphical integrated development environment will also facilitate the reuse of previously existing features and values. It is clear that conclusions from the given statements are only possible if the same names are consistently used for the same features.

At present, the tool implements a text–based approach for the handling and a declarative approach for the analysis and the visualisation of the rule base.

# 5 Design Analysis of the Rule Base

The individual records of the rule memory are usually stored in the memory of DDBASE and analysed with our tool. It is possible to read rules as PROLOG source code, to inspect the rules and even directly query them.

## 5.1 Declarative Queries

Using this deductive knowledge base, it is possible to answer the following exemplary questions with our tool:

- Which factors affect the acceptance of the new ERP system?
- Which constellation of findings is necessary to derive another finding?
  Are there findings, which are a particularly common cause of a change?
  Are there any *killer* findings, that block many developments?
- What are the necessary conditions for a finding? Which ones are optional?
- If a different value is assigned to a single feature, how does this affect the overall structure?
- Are there any redundant rules?
  Can some individual rules be expressed by more accurate rules?
- Where do some findings form opposite or even contradictory relationships?

The questions above underline the diversity of queries than can be asked. The current prototype is 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. [1] The predicate `depends_on` is built to work also for pairs of features instead of just findings.

[1] This can be done by entering a semicolon "`;`" after each answer, standard procedure in a PROLOG top–level interpreter.

```
?- F1 = finding:Consequence, F2 = finding:Condition,
   depends_on(F1, F2).

Consequence = ('Emergence of ERP Knowledge in Employees' = yes),
Condition = ('Existence of ERP Knowledge in Employees' = yes) ;
Consequence = ('Emergence of ERP Knowledge in Employees' = yes),
Condition = ('Cooperation/Communication between Employees
   and Employees with ERP Knowledge' = yes) ;
...
```

Here, not only the contents of individual rules is returned, but also derived knowledge gets computed. If the consequence `'Emergence of Knowledge about ERP System in Employees' = yes` is a prerequisite for a further consequence, then the existence of an employee with knowledge about the ERP System is output as being a prerequisite. Since we use the system DDBASE, is it 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 of ERP System at the Beginning' = partly) ;
Precondition =
   ('Feedback' = no).
```

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

## 5.2 Visualisation of Dependency Graphs

We have already indicated the advantages of an interactive rule editor. Besides facilitating the entry of rules and the dynamic formulation of queries, the rule editor should be used for visualising the statements stored in the rule base.

In a similar form, this had been implemented with the tool VISUR, cf. [18]. The tool visualises a given rule base and thus allows for a graphical interpretation of the rule base. 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 been rendered visually. VISUR has, among other applications, been used for the visualisation of medical diagnoses, whose rules assign symptoms to a diagnosis. We are extending the tool for visualising change management rules from studies in organisational psychology. 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 Figure 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.

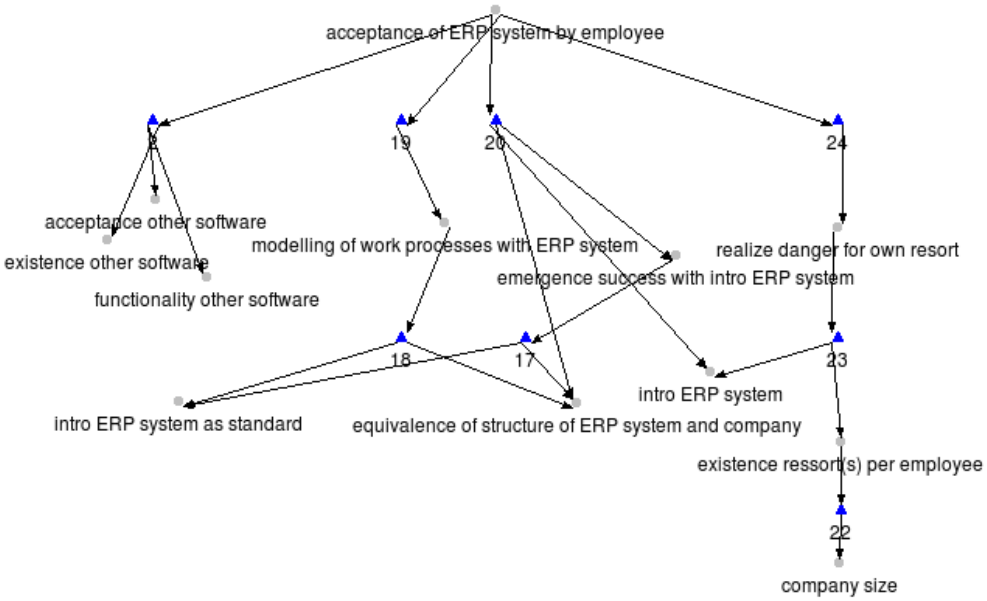


**Figure 2** A dependency graph for the schematic representation of the transitive preconditions and the relevant rules (shown by the blue triangles labelled by `2, 17-20, 22-24`) for deriving the feature `'acceptance of ERP system by employee'`.

## 6 Extensions of the Knowledge Base

As already suggested, besides the pure features, the rules should be annotated by contextual information, such as their source, the method of achieving the rules, the time period of the investigation, and the confidence. The annotations can be used to deduce further constraints and implications from the rule base; the resulting statements about findings can be annotated with confidence values. For reasoning about the queries, provenance information is very important and can influence the usage of the interactive rule editor; for instance, unexpected interaction with other parts of the knowledge base can resort to provenance information to influence whether and how we accept or reject the new knowledge. The extensions can be expressed in DATALOG*, and thus DDBASE can integrate them with the evaluation in a consistent reasoning system.

### 6.1 Provenance Information in Ontologies

*Provenance* is information about entities, activities, and people involved in producing a piece of data, which can be used to assess its quality, reliability or trustworthiness. For collaborations across disciplines, hybrid information systems using data and techniques from many different sources with no preexisting agreement about the semantics of the processes or data, it is important to be able to express provenance. The infrastructure must provide general purpose mechanisms for annotating (i.e., making assertions about), discovering, and reasoning about processes and data.

## The Open Provenance Model (OPM)

Some of the *inferences* require additional reasoning beyond that supported by OWL and SWRL. We assume that the reader is familiar with the basic concepts from ontologies; we do not get formal in this section; a general description of the semantic web rule language SWRL can, e.g., be found in [15, 2]. Also, structures such as the *provenance graph* are very useful for representing causal relationships. The *Open Provenance Model (OPM)* defines logical constraints on the provenance graph [15]. Some constraints cannot be expressed in OWL, but can be expressed based on SWRL rules. The Open Provenance Model provides a way to use semantic web technology and rules to implement semantic metadata. [15] discusses a binding of the OPM written in OWL with rules written in SWRL. This allows for the development of hybrid systems that use OWL, SWRL, and other semantic web software, interoperating through a shared space of RDF triples. PROV is a specification that provides a vocabulary to interchange provenance information. It defines a core data model for the interchange of *provenance* on the web; it allows for building representations of the entities, people and processes involved in producing a piece of data in the domain. The provenance of digital objects represents their origins; the records of a PROV specification can describe the entities and activities involved in producing and delivering or otherwise influencing a given object. Provenance can be used for many purposes, such as understanding how data was collected so it can be used meaningfully, determining ownership and rights over an object, making judgements about information to determine whether to trust it, verifying that the process and steps used to obtain a result complies with some given requirements, and reproducing how something was generated.

## Example in Turtle Syntax

For example, the provenance of a conference paper could be described as follows: The paper was written by author `abc`. The final version of the paper is based on an earlier draft. Some professors made comments on the draft. The author cites prior work from a book. The paper includes a table that was generated by a program. This may be expressed in the so–called turtle syntax, which is a special language that could also be considered as a DSL. Note that the property "`a`" means "*is a*".

```
ex:draft
   a prov:Entity ;
   a abc:Manuscript ;
   dcterms:title "Latest results" .
ex:article a prov:Entity ;
   a abc:ConferencePaper ;
   dcterms:title "Results from case study" .
ex:dataset a prov:Entity ;
   a abc:Dataset .
ex:book
   a prov:Entity ;
   a abc:Thesis .
ex:result a prov:Entity ;
   a abc:Table .
ex:comment a prov:Entity ;
   a abc:Review .
```

### Evaluation in Datalog*

Several of the key constraints and inferences of the OPM cannot be expressed in Owl and Swrl, due to fundamental limitations of the semantics of these languages. E.g., it is not possible to modify the value of an asserted property, or to write a rule to determine the number of times an artifact is used, or to detect a cycle in the provenance graph. Storing the OPM records in triples makes it possible to use other reasoning engines or languages such as Prolog or Datalog to implement queries or inferences. Owl and Swrl's Rdf representations provide a simple and well–understood means of exchanging provenance information with other tools, such as Rdf databases or *declarative* programming languages.

The hybrid system DDbase shows that semantic web technologies are not only useful for provenance information but also provide a base level of interoperability that can enable loosely–coupled tools with varying levels of capability and expressiveness. We do not need specialized reasoners for different knowledge bases. Instead, the rules are encoded in Datalog* and the provenance information is given in ontologies. Further Datalog* rules can encode the profile of the ontology. E.g., [10] study the controlled query evaluation for Datalog and Owl 2 profile ontologies. Then, we obtain a Datalog* knowledge base that can be evaluated in DDbase. Observe, that standard Datalog rules would not be sufficient here, since we need function symbols and embedded calls to Prolog.

## 6.2 Annotation of the Rule Base

The infrastructure must provide universal mechanisms for the annotation of rules for argumenting about processes and data. Similarly, the treatment of confidence values can be achieved. Frequently, collected values can be ambiguous. In the example above, our 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.

### Annotated Findings

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 domain–specifc language 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
and accumulate(conjunction_independence, [A,B,C], D)
then D:'Acceptance of ERP System' = decreasing .
```

The symbols A, B, C, and D in the rule represent logical variables, which always begin with a capital letter – which is common in the logic programming language Prolog. They are distinguishable from normal character strings, since they are included in quotation marks as 'emergence conflicts'. The variables in the rule body are universally quantified; i.e., the statement 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 domain–specific language 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 as follows:

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

## Annotated Datalog Rules

In general, the handling of annotated DATALOG rules has also been investigated by Lakshmanan, Subrahmanian, Kifer, et. al. [13, 11]. We have implemented the rules of Subrahmanian in DDBASE. Observe, that the implementation of `accumulate` above works for arbitrary lists `Xs` of values to obtain the product. In DDBASE, `multiply` 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*.

Our approach is specified in terms of declarative rules that are given in domain–specifc languages. Rather than changing the inference engine in various forms, we keep the inference engine of DDBASE and change the declarative knowledge base of rules. We are thinking of adding a mode for specifying how to interpret variables as matching or evaluate. By analysing the individual rules, their dependencies, and the number of occurrences of individual features or findings, it is possible to determine the approximate impact of a single features, findings, or rules.

In short, having a good match with the underlying general purpose language while retaining a convenient user–friendly syntax and clear semantics is useful for an application–oriented DSL. This is clearly the case with DDBASE and PROLOG.

## 7 Final Remarks

Deductive databases allow for declaratively defining the *semantics* of the expert knowledge with rules. For representing the *syntax* of the rules, we use concepts from domain–specific languages – trying to remain usable within an adequate host language. In a case study for change management in organisational psychology, we have demonstrated the usefulness of the proposed approach in a practical situation. The analysis and visualisation of rules is also used successfully by AI people for medical diagnosis.

In the future, we are planning to apply knowledege engineering techniques, such as refactoring approaches [8], to the deductive rule bases. 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 [13, 11] and probabilistic–enabled logic programming languages, such as ProbLog [6], 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., [14]. We expect that, especially, the combined handling of confidence values and disjunctive rules will be an interesting research field.

## References

1 Serge Abiteboul. DATALOG: La renaissance. `http://www.college-de-france.fr/site/serge-abiteboul/course-2012-05-09-10h00.htm`, 2012.

2 Joachim Baumeister and Dietmar Seipel. Anomalies in ontologies with rules. *Journal of Web Semantics, Science, Services and Agents on the World Wide Web*, 8(1):55–68, 2010.

3 Ivan Bratko. PROLOG *Programming for Artificial Intelligence*. Addison–Wesley Longman, 4th edition, 2011.

4 Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Springer, Berlin, 1990.

5 Upen S. Chakravarthy, Dan H. Fishman, and Jack Minker. Semantic query optimization in expert systems and database systems. In *Proceedings of the 1st International Workshop on Expert Database Systems*, pages 659–674, 1986.

6 Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, pages 2468–2473, 2007.

7 Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Benjamin Cummings, 7th edition, 2015.

8 Martin Fowler. *Refactoring – Improving the Design of Existing Code*. Addison–Wesley, 1999.

9 Martin Fowler. *Domain–Specific Languages*. Addison–Wesley, 2011.

10 Bernardo Cuenca Grau, Evgeny Kharlamov, Egor V. Kostylev, and Dmitriy Zheleznyakov. Controlled query evaluation for DATALOG and OWL 2 profile ontologies. `arXiv:1504.06529`, 2015.

11 Michel Kifer and V. S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, 12(4):335–368, 1992.

12 Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain–specific languages: A systematic mapping study. *Information and Software Technology*, 71:77–91, 2016.

13 Laks V. Lakshmanan and Fereidoon Sadri. On a theory of probabilistic deductive databases. *Theory and Practice of Logic Programming*, 1:5–42, 2001.

14 Jack Minker, Dietmar Seipel, and Carlo Zaniolo. Logic and databases: History of deductive databases. In *Handbook of the History of Logic*, volume 9, Computational Logic. North Holland, 2014.

15 Robert E. McGrath and Joeg Futrelle. Reasoning About Provenance with OWL and SWRL Rules. In *AAAI Spring Symposium*, 2008.

16 Dietmar Seipel. Practical applications of extended deductive databases in DATALOG*. In *Proceedings of the 23rd Workshop on Logic Programming (WLP 2009)*, September 2009.

17 Dietmar Seipel. Knowledge engineering for hybrid deductive databases. In *Proceedings of the 29th Workshop on Logic Programming (WLP 2015)*, September 2015.

18 Dietmar Seipel, Joachim Baumeister, and Marbod Hopfner. Declaratively querying and visualizing knowledge bases in XML. In *Proceedings of the 15th International Conference on Applications of Declarative Programming and Knowledge Management*, LNAI 3392, pages 16–31. Springer, 2005.

19 Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1988.

20 Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, 1989.

21 Rüdiger von der Weth, Dietmar Seipel, Falco Nogatz, Katrin Schubach, Alexander Werner, and Franz Wortha. Modellierung von handlungswissen aus fragmentiertem und heterogenem rohdatenmaterial durch inkrementelle verfeinerung in einem regelbanksystem. *Journal Psychologie des Alltagshandelns*, 2016.

22 Jan Wielemaker. An overview of the SWI–Prolog programming environment. In *Proceedings of the 13th International Workshop on Logic Programming Environments (WLPE)*, pages 1–16, 2003.