Declarative Specification and Interpretation of Rule-Based Systems

Dietmar Seipel and Joachim Baumeister
Institute of Computer Science
University of Würzburg, Germany
{seipel|baumeister}@informatik.uni-wuerzburg.de

Abstract

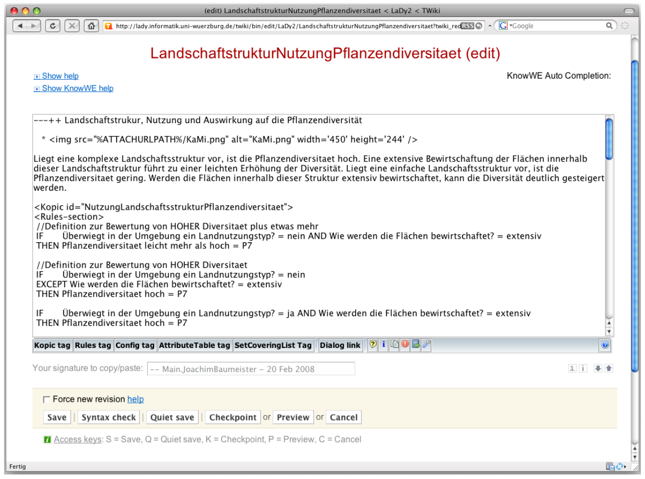
A declarative rule inference engine can be very useful for modern distributed or heterogeneous applications, if it can be configured for multiple semantics and evaluation strategies, and if it allows for arbitrary function calls in rule bodies.

We present a Prolog-based declarative extension of the deductive database language Datalog to more general rule-based applications.

Introduction

Recently, the specification of rule-based knowledge in a general way has gained reasonable attention. Rules are an intuitive knowledge representation for building intelligent systems, and they are used in many applications, for example in the context of business rules, the development of medical knowledge bases, and for the control of technical devices. Additionally, in the context of the semantic web initiative, the definition of a general rule interchange format (RIF) is under discussion. This enables distributed services to understand and to use rule-based knowledge from different sources (W3C Rules Working Group 2006). Similarly, typical use cases are the implementation and distribution of business rules, medical knowledge bases, vocabulary mapping used for data integration, and the extension of general ontologies by expressive rule-based statements.

In open environments like knowledge wikis (Baumeister & Puppe 2008), volunteer users are able to declaratively specify rule-based knowledge through a web-based interface. For example, Figure 1 shows the edit pane of a knowledge wiki, where the user can not only specify the textual knowledge of a specific domain, but is also able to define explicit problem solving knowledge. In the given example, a knowledge wiki for capturing knowledge on landscape diversity is shown (Baumeister, Reutelshoefer, Nadrowski, & Misok 2007). The rule base is formulated in a textual IF/THEN/EXCEPT manner. Conditioned findings are placed in the IF block, and the rule action – here the derivation of some solutions – is described in the THEN block. Exceptions are defined in the EXCEPT block and prevent the rules from firing, if the corresponding conditions evaluate to true.


(edit) LandschaftstrukturNutzungPflanzendiversitaet < LaDy2 < TWiki
http://lady.informatik.uni-wuerzburg.de/twiki/bin/edit/LaDy2/LandschaftstrukturNutzungPflanzendiversitaet?twiki_red
LandschaftstrukturNutzungPflanzendiversitaet (edit)
Show help
Show KnowWE help
KnowWE Auto Completion:

```
---++ Landschaftstrukur, Nutzung und Auswirkung auf die Pflanzendiversität

  * <img src="%ATTACHURLPATH%/KaMi.png" alt="KaMi.png" width='450' height='244' />

Liegt eine komplexe Landschaftsstruktur vor, ist die Pflanzendiversitaet hoch. Eine extensive Bewirtschaftung der Flächen innerhalb
dieser Landschaftstruktur führt zu einer leichten Erhöhung der Diversität. Liegt eine einfache Landschaftsstruktur vor, ist die
Pflanzendiversitaet gering. Werden die Flächen innerhalb dieser Struktur extensiv bewirtschaftet, kann die Diversität deutlich gesteigert
werden.

<Kopic id="NutzungLandschaftsstrukturPflanzendiversitaet">
<Rules-section>
 //Definition zur Bewertung von HOHER Diversitaet plus etwas mehr
 IF      Überwiegt in der Umgebung ein Landnutzungstyp? = nein AND Wie werden die Flächen bewirtschaftet? = extensiv
 THEN Pflanzendiversitaet leicht mehr als hoch = P7

 //Definition zur Bewertung von HOHER Diversitaet
 IF      Überwiegt in der Umgebung ein Landnutzungstyp? = nein
 EXCEPT Wie werden die Flächen bewirtschaftet? = extensiv
 THEN Pflanzendiversitaet hoch = P7

 IF      Überwiegt in der Umgebung ein Landnutzungstyp? = ja AND Wie werden die Flächen bewirtschaftet? = extensiv
 THEN Pflanzendiversitaet hoch = P7
```

Your signature to copy/paste: -- Main.JoachimBaumeister - 20 Feb 2008

Force new revision help

Access keys: S = Save, Q = Quiet save, K = Checkpoint, P = Preview, C = Cancel

Figure 1: The edit pane of a knowledge wiki defining rules for the effective use of plant diversity.

Besides the general markup of rules, the intended semantics of the embodied knowledge is of prime importance. Especially in an open environment, the interpretation of the rule base is an important issue and affects the design and the development of the rule base. Even for the same knowledge representation, the interpretation may depend on the particular application project. In our approach, the declarative configuration of the rule semantics is intended to be defined by an administrative knowledge engineer and not by the actual developers of the rule base.

In this paper, rule-based knowledge is represented in Prolog syntax and interpreted by a powerful and configurable rule engine in Prolog, which allows for the declarative configuration of the interpretation process. The presented approach is general in its application and can be adapted to reimplement and extend rule-based systems that were originally using rule engines such as the well-known systems JBossRules or Jess. In the context of this paper, we will demonstrate the approach for the declarative specification of different rule types from the *d3web* diagnostic knowledge system (Puppe 1999).

The rest of this paper is organized as follows: In the next section, we introduce two rule formalisms of diagnostic systems: scoring rules and set covering rules. They cannot be implemented based on the well-known forward iteration of deductive databases. Therefore, we show how forward iteration can be generalized by incorporating backward rules, such that scoring and set covering rules as well as the aggregation of scores can be implemented, and we describe an application scenario taken from the diagnostic knowledge system *d3web*. Finally, we present our implementation of the generalized forward iteration in Prolog.

Variants of Rule Interpretation

Todays diagnostic systems use quite complex inference approaches (Stefik 2004), and there exist many variants of interpreting rule-based knowledge. The interpretation of rules commonly depends on the type of application and the requirements of the domain specialists that are developing the knowledge base.

In the following, we will describe two formalisms for knowledge representation, that have proven to be suitable in a large number of projects in the context of the *d3web* system. *Scoring rules* are a very intuitive and powerful representation often used for developing diagnosis systems, e.g., in the medical domain, and *set covering rules* are very useful for the implementation of knowledge bases in the context of recommender systems.

First we introduce some basic notions that are required for the further definitions, and then we introduce the knowledge representations in more detail.

Basic Notions

In general, a rule-based system uses a rule base to derive new facts from a given fact base or facts provided (interactively) by the user. We partition the facts into two disjoint sets S and F. S is the universal set of all *solutions* of the problem, i.e., facts that are derived by the system and presented to the user as outputs; in this paper, we use the synonym *diagnosis* for solution. F is the universal set of *findings* that are used as input facts in rules: there are input findings $q = v$, where v is the answer to the question q, and intermediate findings $a = v$, where a is an abstraction.

A general rule $r = \alpha \to \beta$ has an antecedent α, which is a formula over conjunctions, disjunctions, and negations of conditions over diagnoses, questions, abstractions, and their corresponding values; the consequent β of the rule is derived if α evaluates to truefor the given set of findings. β commonly depends on the rule representation and will be discussed in the following sections. β can be an intermediate finding $a = v$ or a pair $d = s$, where d is a diagnosis and s is a score, or just a diagnosis d.

Scoring Rules

Scoring rules are used to weight the diagnoses (solutions) $d \in S$ with respect to the given set of facts. A scoring rule is of the form

$$r = \alpha \to (d = s),$$

where α is the condition of the rule and s is a score for d. E.g., $(a_4 = v_1) \vee (\,\mathrm{not}(q_4 = v_1) \wedge (q_5 \geq v_2)\,) \to (d_1 = 5)$.

The basic idea for the evaluation of scoring rules is very simple. For every possible solution d, a numerical account $acc(d)$ is instantiated at the beginning of each problem solving session as $acc(d) = 0$. During the problem solving session, every scoring rule for d, that is able to fire, simply adds its score s to $acc(d)$ as the rule action. If the aggregated scoring account of a solution d exceeds a given threshold, then d is presented to the user as a possible solution. Like in the classic scheme of the Internist system (Miller, Pople, & Myers 1982), aggregation is done on a logarithmic scale, and we have set the threshold to 7. Every scoring rule only fires once during a problem solving session.

Set Covering Rules

Set covering rules provide an intuitive representation, which is especially well-suited for building recommender systems. A set covering rule is of the form $r = \alpha \to d$; for a solution $d \in S$, the conjunction α contains all findings or negated findings that are expected to be observed if we assume d. E.g., $\alpha = (a_4 = v_1) \wedge not(q_4 = v1) \wedge (q_5 \geq v_2)$. The coverage of d depends on how many elements of α evaluate to true.

For a concrete problem statement, a list of appropriate solutions is derived using a simple hypothesize-and-test method: For every solution we compare the expected findings of the set covering rules with the actually present findings, and we select the best-matching solutions, i.e., the solutions whose expected set of findings overlaps most with the observed findings. This simple knowledge representation can be iteratively extended by additional background knowledge like weights, similarities between finding values and frequencies (Baumeister, Seipel, & Puppe 2003).

Deductive Databases and Logic Programming

Deductive databases and logic programming (Ceri, Gottlob, & Tanca 1990) can be used as a declarative mediator technology between different data sources and rule-based problem solvers. Deductive databases normally use Datalog and the standard inference operator $\mathcal{T}_{\mathcal{P}}$.

But practical rule-based applications often require an extensible set of user-defined built-in predicates, and they need function symbols for handling complex data structures. For some applications the standard form of conjunctive rule bodies is not adequate, since the knowledge representation is more complicated. If non-conjunctive rule bodies are normalized to conjunctive rule bodies, then the $\mathcal{T}_{\mathcal{P}}$-evaluation becomes unnecessarily complex due to redundancy.

Applications such as diagnostic reasoning in the *d3web* system could not be formulated using Prolog backward chaining or Datalog forward chaining alone, since we need recursion on cyclic data, function symbols, negation and disjunction in rule bodies, aggregation, and stratification, and we might also need non-ground facts. This observation also holds in general for other rule formalisms.

But, using *meta interpreters* which we write in Prolog, we can overcome the limitations of both Datalog and Prolog.

Datalog* Thus we have developed a new approach that extends the Datalog paradigm to Datalog* and mixes it with Prolog. This makes deductive databases also able to access and manage a variety of external data sources, such as relational databases, XML databases or XML documents, and Excel data.

We distinguish between Datalog* rules and Prolog rules. Datalog* rules are forward chaining rules (not necessarily range-restricted) that may contain function symbols (in rule heads and bodies) as well as negation, conjunction, disjunction, and Prolog predicates in rule bodies. Datalog* rules are evaluated bottom-up, and all possible conclusions are derived.

The supporting Prolog rules are evaluated top-down, and – for efficiency reasons – only on demand, and they can refer to Datalog* facts. The Prolog rules are necessary for computations on complex terms, and – more importantly – for aggregating the derived Datalog* facts after each iteration.

Declarative Rules in Datalog*

Datalog* allows for general, user-defined built-in predicates, which can be implemented in Prolog, and function symbols. A Datalog* program consists of forward and backward rules in Prolog syntax, all of which can access the Prolog database and are subject to the same backtracking mechanism.

- The forward rules are *interpreted* in a bottom-up iteration. Every elementary step calls all rule bodies in Prolog and adds the derived facts to the Prolog database. Thus, elementary derivations can only refer to already derived facts and not to forward rules, which breaks recursion.
- The backward rules are evaluated top-down in Prolog using standard SLDNF-resolution. This allows for embedding arbitrary Prolog computations in Datalog*. Thus, by defining predicates using backward rules, the user can extend the set of allowed built-in predicates.

Optimizations such as Δ-Iteration (Ceri, Gottlob, & Tanca 1990) can be implemented flexibly based on program transformations; the basic declarative rule interpreter does not have to be changed for this.

Forward Rules

The forward rules use the connectives "`,`" (and), "`;`" (or), and "`not`", where "and" binds stronger than "or".

In the following we show some diagnostic forward rules for deriving facts for the unary predicates `finding` and `diagnosis`. After each step of the bottom-up iteration, all scores for the same diagnosis are *aggregated* before the next iteration starts.

Intermediate Findings There can be rules for deriving intermediate findings from input findings and intermediate findings, which have previously been derived during an interactive examination dialog. E.g., the following rule derives $a_4 = v_1$, if the intermediate finding $a_1 = v_7$ and (at least) one of the input findings $q_2 = v_1$ or $q_3 = v_4$ are known:

```
finding(a4 = v1) :-
   finding(a1 = v7),
   ( condition(q2 = v1)
   ; condition(q3 = v4) ).
```

The Prolog predicate `condition/1`, is implemented using backward rules, which we will see later. If an input finding is not yet known, then an interview is started to ask the corresponding question to the user.

Scoring Rules The following rules assign scores to the diagnoses based on the intermediate diagnoses and the answers to the questions; the complex rule structure of the first rule is shown in Figure 2:

```
diagnosis(d1 = 5) :-
   ( finding(a4 = v1)
   ; not(condition(q4 = v1)),
     condition(q5 >= v2) ).
diagnosis(d1 = -3) :-
   not( condition(q4 = v2)
      ; finding(a2 = v3) ).
```

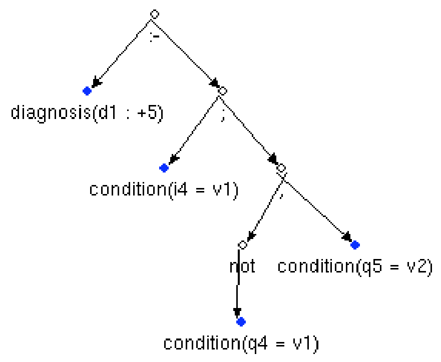


Figure 2: Rule structure of a scoring rule

Set Covering Rules The following set covering rule derives a solution from a conjunctive condition. The Prolog predicate `coverage/2` computes the value `S` indicating how well the condition matches the current fact base; it can be implemented by the user.

```
diagnosis(d = S) :-
   coverage( ( finding(a4 = v1),
      not(condition(q4 = v1)),
      condition(q5 = 2) ), S ).
```

The best-matching solution is determined later by aggregation. More complicated set covering rules can have weights and similarities.

Interview Strategy Question indication rules are used for deriving the information that a question should be asked to the user at the current point of the problem solving process:

```
finding(q1 = indicated) :-
   ( finding(a4 = v1)
   ; not(condition(q4 = v1)) ).
```

Optimization by Blocking Rules We can block a rule as soon as some diagnosis is established – i.e. the score is equal/greater than 7 – by adding a literal `not(diagnosis(_ = 7))` at the beginning of the rule body:

```
finding(a5 = v1) :-
   not(diagnosis(_ = 7)),
   condition(q1 = v7).
```

We can block individual rules or all rules. In the latter case, the derivation process stops.

Usually, a rule without variables should fire at most once. In Datalog*, we provide a second type of Δ-rules which is identified by another rule arrow `<-`.

```
finding(a5 = v1) <-
   condition(q1 = v1).
```

Such a rule is expanded – when the rule base is consulted – to the following regular rule, where `r17` is a unique rule identifier, which is created by the system:

```
finding(a5 = v1) :-
   not(rule_was_fired(r17)),
   condition(q1 = v7),
   assert(rule_was_fired(r17)).
```

In *d3web*, Δ-rules have to be used for scoring rules, since a scoring rule can only fire once (further firings would repeatedly increase the score). In general, Δ-rules can also be used for speeding up the diagnostic process.

Backward Rules

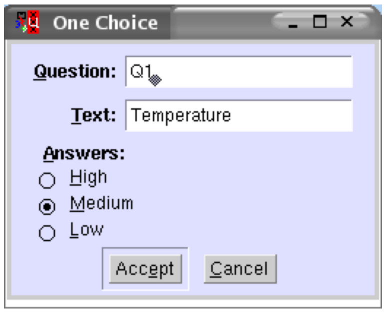
Backward rules are used for the configuration of the declarative inference engine. They are stored together with the derived facts in the Prolog database. There can be rules for evaluating the conditions of forward rules, for initiating interactive interviews during a diagnostic process, and for aggregating the scores of diagnoses.

Evaluation of Conditions For evaluating a condition in the body of a forward rule, the following predicate searches for findings in the fact base. If there is no matching finding, then an *interview dialog* is initiated, cf. Figure 3.

A call of the form `condition(C)`, where $C = Q \otimes V$, determines the answer W to the question Q and compares it with V using the operator $\otimes$. If a finding for Q is known, then W can be taken from there. Otherwise, if the question Q is indicated, then a suitable Prolog dialog is initiated, and the answer W is the asserted in a finding.

```
condition(C) :-
   C =.. [Op, Q, V],
   ( finding(Q = W)
   ; finding(Q = indicated),
     d3_dialog(Q = W),
     assert(finding(Q = W)) ),
   !,
   apply(Op, [W, V]).
```

In the rule, `Op` denotes the operator $\otimes$. If there exists no finding for Q and Q is not indicated, or if the comparison $W \otimes V$ yields false, then the call `condition(C)` fails.



Question: Q1
Text: Temperature
Answers:
○ High
[● Medium]
○ Low

Figure 3: Dialog for the question `Q1`

Further Connectives The call `m_to_n(M-N, Cs)` in a forward rule requires that between `M` and `N` of the conditions in the list `Cs` are true.

```
finding(a6 = v2) :-
   m_to_n( 1-2, [ finding(a4 = v1),
      not(condition(q4 = v1)),
      condition(q5 = 2) ] ).
```

The predicate `m_to_n` can be implemented using a backward rule:

```
m_to_n(M-N, Cs) :-
   findall( C,
      ( member(C, Cs), call(C) ),
      Cs_2 )
   length(Cs_2, K), M =< K, K =< N.
```

Obviously, the evaluation of an `m_to_n`-condition has to be delayed until all findings in `Cs` have been evaluated; this is acchieved by a stratified evaluation of Datalog*.

Simple set covering rules could be implemented using the predicate `m_to_n/2`.

Aggregation A method for aggregating the set `Fs1` of derived facts to a new set `Fs2` can be provided by the user. For *d3web*, we determine the set `Ds` of all diagnoses for which scores have been derived. For all diagnoses `D` in `Ds` we aggregate the scores `S` from the derived facts `diagnosis(D = S)` into a single value `Score`:

```
aggregate_facts(Fs1, Fs2) :-
   findall( diagnosis(D = Score),
      ( setof( S,
           member(diagnosis(D = S), Fs1),
           Scores ),
         aggregate_scores(Scores, Score),
      Fs2 ).

aggregate_scores(Scores, Score) :-
```

```
   maplist( power(2),
      Scores, Xs ),
   add(Xs, X), log(2, X, Score).
```

Since scores are on a logarithmic scale in *d3web*, the aggregation function is $S_1 \oplus S_2 = \log_2(2^{S_1} + 2^{S_2})$.

The combination of probabilities in probabilistic reasoning can also be implemented in Datalog*.

Evaluation of Datalog*

Rules with embedded procedural computations and complex data structures cannot be evaluated in Prolog or Datalog alone for the following reasons. Current Datalog engines cannot handle function symbols and non-ground facts, and they do not allow for the embedded computations.

Standard Prolog systems cannot easily handle recursion with cycles, because of non-termination, and are inefficient, because of subqueries that are posed and answered multiply. Thus, they have to be extended by Datalog features (our approach) or by memoing/tabling features (the approach of the Prolog extension XSB).

We have implemented an inference engine for stratified Datalog* as a meta-interpreter using the common Prolog system SWI (Wielemaker 2007); the user dialogs have been built with its publicly available graphical API (Wielemaker, Anjewierden 2007).

The Generalized Consequence Operator $\mathcal{T}_{\mathcal{D},\mathcal{P}}$

The operator $\mathcal{T}_{\mathcal{D},\mathcal{P}}$ operates on a forward program $\mathcal{D}$ and an auxilliary program $\mathcal{P}$. In the implementation below $\mathcal{D}$ is given as a list `D` of rules, whereas $\mathcal{P}$ is stored in the modules `M` and `user` (the standard module) of the Prolog clause database.

The following predicate calls all rule bodies of `D` in the Prolog module `M`. The set `Fs` of derived facts will be stored in `M` only afterwards.

```
tp_operator(D, M, Fs) :-
   findall( A,
      ( member(A :- B, D),
        M:call(B) ),
      Fs ).
```

For all body predicates of $\mathcal{D}$ there have to be either rules (or facts) in $\mathcal{D} \cup \mathcal{P}$ or `dynamic` declarations in $\mathcal{P}$; otherwise, a call to such a predicate would raise an exception.

Since the evaluated forward program $\mathcal{D}$ is not part of the Prolog database, the forward rules do not call each other recursively within a single $\mathcal{T}_{\mathcal{D},\mathcal{P}}$-operator, and they cannot be called from backward rules. The recursion is only reflected in the bottom-up iteration of the operator.

For all body predicates of $\mathcal{D}$ there can be rules in both $\mathcal{D}$ and $\mathcal{P}$. A body predicate, for which there are only rules in $\mathcal{P}$ which do not refer to predicates from $\mathcal{D}$, could be considered as a built-in predicate of $\mathcal{D}$. The rules of $\mathcal{P}$ are evaluated top-down using Prolog's SLDNF-resolution.

For diagnostic reasoning with *d3web*, the rules were evaluated by such a generalized operator with the built-in predicates `condition/1`, `coverage/2`, `m_to_n/2`, etc.

Managing Facts in a Module

We use two elementary predicates for asserting/retracting a given list `Fs` of facts in/from a module `M` of the Prolog database:

```
assert_facts(M, Fs) :-
   forall( member(F, Fs), assert(M:F) ).
retract_facts(M, Fs) :-
   forall( member(F, Fs), retract(M:F) ).
```

For adding a list `Fs` of facts to `M`, we need to know the list `I` of facts that are already stored in `M`. First, these facts are retracted from `M`, then `I` and `Fs` are aggregated using a user-defined plugin predicate (if the is no such predicate, then no aggregation is done), and finally the result is asserted in `M`.

```
aggregate_facts(M, I, Fs, J) :-
   retract_facts(M, I),
   ord_union(I, Fs, K),
   ( aggregate_facts(K, J) ; J = K ),
   assert_facts(M, J).
```

`aggregate_facts/2` is a plugin predicate that can be specified using backward rules in the application.

The Iteration of $\mathcal{T}_{\mathcal{D},\mathcal{P}}$

For a given set `D` of forward rules and a set `I` of facts, a new set `K` is derived in module `M`:

```
tp_iteration(D, M, I, K) :-
   tp_operator(D, M, Fs),
   ( tp_terminates(I, Fs) -> K = I
   ; aggregate_facts(M, I, Fs, J),
     tp_iteration(D, M, J, K) ).
tp_terminates(I, Fs) :-
   not(member(A, Fs), not(member(A, I))).
```

The derived facts in `tp_iteration/4` are stored in the module `M` and kept on the argument level (`I`, `J`, `K`). The iteration terminates, if all derived facts have already been known.

The derived facts in the module `M` are necessary for `tp_operator/3`. Since they are mixed with the facts and the rules of the auxilliary Prolog program $\mathcal{P}$ in `M`, they cannot be extracted from `M` at the end of the iteration; therefore, they also have to be kept on the argument level.

The initial call starts `tp_iteration/4` with an empty set of facts:

```
tp_iteration(D, M, K) :-
   tp_iteration(D, M, [], K).
```

Stratification

We have implemented a Prolog library for handling stratification in Datalog*. For example, sophisticated, heuristic methods of program analysis, which can also be extended by the user, have to be used to determine embedded calls in Prolog meta-predicates, cf. (Seipel, Hopfner, & Heumesser 2003).

For diagnostic reasoning with *d3web*, meta-predicates such as `not`, `m_to_n`, `maplist`, `checklist`, `findall`, `setof`, and `bagof` have to be handled.

Modification of the Fact Base in Bottom-Up Rules

It is possible to update the Prolog database using the predicates `assert` and `retract` in forward rules. The following transitive closure rules make the derived facts available to further derivations before the iteration is completed.

```
tc(X, Y) :-
   arc(X, Y),
   assert(tc(X, Y)).
tc(X, Y) :-
   arc(X, Z), tc(Z, Y),
   assert(tc(X, Y)).
```

Consequently, a large part of the transitive closure is computed during a single iteration of the $\mathcal{T}_{\mathcal{D},\mathcal{P}}$-operator. For avoiding problems, a more controlled use of update predicates for the Prolog database could be required.

In our Datalog* implementation of *d3web* reasoning `assert` is used only after interactive dialogs. This does not effect the derivation process so drastically; the computation simply behaves as if all findings had been known before the computation.

Conclusions

In this paper, we have introduced a novel approach for the declarative configuration of rule engines. With the described method the corresponding rule inference strategy can be defined ad-hoc by experts for every single knowledge base. The approach has been used for an implementation of diagnostic reasoning, and for the detection of anomalies in SWRL ontologies (Baumeister, Kleemann, Seipel 2007).

In the context of open knowledge engineering environments like knowledge wikis, such a configuration approach should be very suitable, where distributed volunteers enter different types of problem solving knowledge, and where the interpretation logic of the provided knowledge heavily depends on the application project. With the presented approach an administrative knowledge engineer can easily configure the intended semantics of the applied rule formalism.

The approach mainly builds on the new language Datalog* which allows for the definition and the use of forward rules and backward rules. Using Datalog* it seems to be possible to implement an extended form of the magic sets transformation method (Ceri, Gottlob, & Tanca 1990) – which also wants to add top-down features to bottom-up processing – for rules with non-conjunctive rule bodies in a very simple way.

References

Abiteboul, S.; Bunemann, P.; Suciu, D. Data on the Web – From Relations to Semi-Structured Data and XML. Morgan Kaufmann, 2000.

Baumeister, J.; and Puppe, F. 2008. Web-based Knowledge Engineering with Knowledge Wikis. In *Proceedings of AAAI Spring Symposium on "Symbiotic Relationships between Semantic Web and Knowledge Engineering"*, Standford University, USA, AAAI Press.

Baumeister, J.; Seipel, D.; and Puppe, F. 2003. Incremental Development of Diagnostic Set Covering Models with Therapy Effects. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 11 (Suppl.), pp. 25–50.

Baumeister, J.; Reutelshoefer, J.; Nadrowski, K.; Misok, A. Using Knowledge Wikis to Support Scientific Communities In *Proc. 1st Workshop on Scientific Communities of Practice (SCOOP)*, Bremen, Germany, 2007.

Baumeister, J.; Kleemann, T., & Seipel, D. Towards the Verification of Ontologies with Rules, In *Proc. 20th International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, 2007, AAAI Press, pp. 524–529.

Berners-Lee, T.; Hendler, J.; Lassila, O. The Semantic Web, In *Scientific American*, May 2001.

Ceri, S.; Gottlob, G.; Tanca, L. Logic Programing and Databases, Springer, 1990.

Kifer, M.; Lausen, G. F-Logic: A Higher-Order Language for Reasoning about Objects, In *Proc. ACM SIGMOD Conference on Management of Data (SIGMOD)*, 1989.

Miller, R.; Pople, H; Myers, J. INTERNIST-1, an Experimental Computer-Based Diagnostic Consultant for General Internal Medicine, New England Journal of Medicine, 1982, Vol. 307, pp. 468–476.

Puppe, F. Systematic Introduction to Expert Systems, Springer, 1993.

Puppe, F. Knowledge Reuse among Diagnostic Problem Solving Methods in the Shell-Kit D3, International Journal of Human-Computer Studies 49, 627-649, 1998

Puppe F.; et al. The *d3web* system. `http://d3web.sourceforge.net`

Seipel, D. DisLog – A Disjunctive Deductive Database Prototype, In *Proc. 12th Workshop on Logic Programing (WLP)*, 1997. `http://www1.informatik.uni-wuerzburg.de/databases/DisLog`

Seipel, D. Processing XML Documents in Prolog, In *Proc. 17th Workshop on Logic Programing (WLP)*, 2002.

Seipel, D.; Hopfner, M.; Heumesser, B. Analyzing and Visualizing Prolog Programs based on XML Representations. In *Proc. International Workshop on Logic Programing Environments (WLPE)*, 2003.

Seipel, D.; Baumeister, J.; Hopfner, M. Declarative Querying and Visualizing Knowledge Bases in XML, In *Proc. 15th International Conference on Applications of Declarative Programing and Knowledge Management (INAP)*, 2004, pp. 140–151.

Stefik, M. Introduction to Knowledge Systems, Morgan Kaufmann, 2004.

Wielemaker, J. SWI-Prolog 5.0 Reference Manual, `http://www.swi-prolog.org/`

Wielemaker, J.; Anjewierden, A. Programming in XPCE/Prolog, `http://www.swi-prolog.org/`

W3C Rules Working Group. 2006. RIF Use Cases and Requirements: http://www.w3.org/TR/rif-ucr.