Knowledge Representation and Reasoning

Rule-Based Systems

Prof. Dr. Hans Jürgen Ohlbach
Contents

• Rule-Based Systems
• Datalog
• OPS 5
• Prolog
• SWRL
**Motivation**

- Description Logics can be used to describe *static* relationships between sets and members, but not *context dependent* relationships between members.

  (e.g. If robot R2D2 *wants* to move from Room X to Room Y, and there is a door between Room X and Room Y, and the door is open, then he *can* move from Room X to Room Y)

- More Flexibility and Expressiveness is obtained by **Rule-Based Systems**

  $$B_1, \ldots, B_n \Rightarrow A$$
Rules: Direction of Application

**backward:**
In order to solve Problem A, one must solve Problem $B_1, \ldots, B_n$

$$A \leftarrow B_1, \ldots, B_n$$

**Ex.:**
$$\text{buy}(X) \leftarrow \text{searchShop}(Y,X), \text{goIntoShop}(Y), \text{get}(X), \text{pay}(X)$$

**forward:**
if the conditions $B_1, \ldots, B_n$ are satisfied, then it holds/you can do C

$$B_1, \ldots, B_n \Rightarrow C$$

**Ex.:**
$$\text{FireInRoom}(F,R), \neg\text{FireBrigadeIn}(R) \Rightarrow \text{ExtinguishYourself}(F)$$

If a forward-rule is applied, its called „**it fires**“
Operational vs. Logical

**Forward**: \( B_1, \ldots, B_n \Rightarrow C \)

**operational**: as soon as conditions \( B_1, \ldots, B_n \) become true, do something in order to make \( C \) true.

**Ex.**:
- FireInRoom(F,R), \( \neg \)FireBrigadeIn(R) \( \Rightarrow \) ExtinguishYourself(F)

**logical**:
- explizit derivation of implicitly true facts

**Ex.**:
- Ancestor(x,y), Ancestor(y,z) \( \Rightarrow \) Ancestor(x,z)

**backwards**: \( A \Leftarrow B_1, \ldots, B_n \)

**operational**: decompose complex actions into their components

**Ex.**:
- buy(X) \( \Leftarrow \) searchShop(Y,X), goIntoShop(Y), get(X), pay(X)

**logical**: like forward inferencing
Fact Base for Closed Systems

Your start with a set of facts describing a particular situation.
The rules operate on these facts

**Operational:**
one can add and remove facts from the fact base:

*Ex.:*
inRoom(Robot,X), goesThroughDoor(Robot,X,Y) ⇒
delete(inRoom(Robot,X)), add(inRoom(Robot,Y))

**Logical:**

- **Monotone:** only derived facts can be added (Ex. Transitivity of Ancestor). The information does not change.
- **Non-monotonic:** one can delete previously derived facts:

*Ex:*
rain
rain ⇒ wet (wet is added)
have_Umbreller
rain, have_Umbreller ⇒ ¬wet (wet must be deleted)
The facts are not available at the beginning, but are submitted from outside the rule systems

**Ex:** Sensor Data

**Rule:** Temperature(CPU) > threshold ⇒ halve(Frequency)

This is the area of *Complex Event Processing (CEP)*

See http://www.complexeevents.com/
Closed World Assumption

Closed World:

**Assumption**: the fact basis is a complete model of the world. That means facts not contained in the fact bases or not derivable from the fact basis are wrong.

**Ex.**: bus timetable

If bus(Tivolistraße,10:00) is not in the timetable then one can conclude \( \neg \text{bus(Tivolistraße,10:00)} \)

The program tests whether bus(Tivolistraße,10:00) can be derived. If not, \( \neg \text{bus(Tivolistraße,10:00)} \) must be true.

(Negation By Failure)
Open World Assumption

**Open World**: the fact basis might be incomplete.

Ex.:

- `hasChild(Boris_Becker, Noah_Gabriel).
- `hasChild(Boris_Becker, Elias_Balthasar).
- `hasChild(Boris_Becker, Anna_Ermakova).
- `hasChild(Boris_Becker, Amadeus).

A query „how many children has Boris Becker?“ can only get an incomplete answer: \((\geq 4)\).

A negated query:

\(\neg (\text{NumberOfChildren}(Boris\_Becker) = 5)\)

cannot be answered at all.

Open and Closed World Assumption can be mixed.
Stateless Fact-Basis

Every change is done destructively on the fact-basis.

**Ex.**:  
\[
in\text{Room}(\text{Robot},X) , \text{movesThroughDoor}(\text{Robot},X,Y) \implies \text{delete}(\text{inRoom}(\text{Robot},X)) , \text{add}(\text{inRoom}(\text{Robot},Y))
\]

**Advantage:**  
Facts not affected by the rule are still valid.  
**Ex.**: \text{delete}(\text{inRoom}(\text{R2D2},\text{R1})) does not change \text{inRoom}(\text{C-3PO},\text{R1}).

All consequences of the changes must be computed after every rule application.  
**Ex.**: \text{delete}(\text{inRoom}(\text{Robot},X)) changes the visibility graphs for all other robots in the room.
Stateful Fact-Basis

Every rule application yields a copy of the fact-basis such that the old version is still available and the rules can access it.

Ex.: 

\[\text{sentenced-for}(X,Y), \neg\text{previously-convicted}(X) \Rightarrow \text{on-Probation}(X,Y)\]

\[\neg\text{previously-convicted}(X)\]

must check all previous versions of the fact-basis.

Frame Problem: A new version of the fact-basis must be filled with all facts not affected by the rule-base.

Ex.: \(\text{inRoom}(\text{Robot},X), \text{movesThroughDoor}(\text{Robot},X,Y) \Rightarrow \neg\text{inRoom}(\text{Robot},X), \text{inRoom}(\text{Roboter},Y)\)

In the new state not only \(\text{inRoom}(\text{Robot},Y)\) is true, but all the rest of the world which has not changed.
Datalog has been developed for deductive databases.

**Ex.:**
Database:
has_Father(Tom, Klaus).
has_Father(Klaus, Karl).

Datalog: with

has_Ancestor(X,Y) :- has_Father(X,Y).
has_Ancestor(X,Y) :- has_Father(X,Z) & has_Ancestor(Z,Y).

one can deduce,

has_Ancestor(Tom,Karl). (X,Y,Z are variables)
Datalog Clauses

A Datalog clause has the form

\[ P(t_1, \ldots, t_n). \]  

(fact) or

\[ P(t_1, \ldots, t_n) :\neg P_1(s_1, \ldots, s_k) \land \ldots \land P_l(r_1, \ldots, r_m). \]  

(rule)

where:

- the \( P_i \) are predicates (not necessarily in the database)
- The arguments of the predicates are either constant symbols or variable symbols (no function terms!)
- all variables in the „head“ \( P(t_1, \ldots, t_n) \) must also be in the „body“ \( P_1(s_1, \ldots, s_k) \land \ldots \land P_l(r_1, \ldots, r_m) \).

Datalog Queries have the form

\[ ?- P_1(s_1, \ldots, s_k) \land \ldots \land P_l(r_1, \ldots, r_m). \]

with the meaning: give me all bindings for variables such that, \( P_1(s_1, \ldots, s_k) \land \ldots \land P_l(r_1, \ldots, r_m) \) becomes true.
A Datalog Program with forward-evaluation (forward chaining) has always
- a database
- a list of derived facts (always without variables)

A rule

\[ P(t_1, \ldots, t_n) :- P_1(s_1, \ldots, s_k) \land \ldots \land P_l(r_1, \ldots, r_m). \]

is evaluated, by evaluating the \( P_i(\ldots) \) from left to right by:
- either searching the database for instances of \( P_i(\ldots) \), or
- searching the derived facts for instances of \( P_i(\ldots) \).

Variables in the \( P_i(\ldots) \) are bound to corresponding constants. The instances of \( P(t_1, \ldots, t_n) \) are then added to the list of derived facts.
Example for Forward-Evaluation

Database:
  has_Father(Tom,Klaus).
  has_Father(Klaus,Karl).
  has_Mother(Klaus,Maria).

Datalog:
  has_Ancestor(X,Y) :- has_Father(X,Y).
  has_Ancestor(X,Y) :- has_Mother(X,Y).
  has_Ancestor(X,Y) :- has_Father(X,Z) & has_Ancestor(Z,Y).
  has_Ancestor(X,Y) :- has_Mother(X,Z) & has_Ancestor(Z,Y).

Forward-Evaluation yields:
  has_Ancestor(Tom,Klaus). has_Ancestor(Klaus,Karl).
  has_Ancestor(Klaus,Maria). has_Ancestor(Tom,Maria).
  has_Ancestor(Tom,Karl).
Advantage:
• very simple evaluation strategy
• no backtracking

Disadvantage:
• not specific for queries.
• therefore there may be too many derived facts

Alternative: magic sets Method.
(simulation of Prolog strategy)
Negation is only allowed in the body!

**Example:**

\[
\text{greenPath}(X,Y) \ :- \ \text{green}(X,Y)
\]
\[
\text{greenPath}(X,Y) \ :- \ \text{greenPath}(X,Z) \ \& \ \text{greenPath}(Z,Y)
\]
\[
\text{save}(X,Y) \ :- \ \text{red}(X,Y) \ \& \ \text{NOT} \ \text{greenPath}(X,Y)
\]

The program must be **stratified**, i.e.

the negated predicate (greenpath) must be completely evaluated before the head predicate (save) can be evaluated.

I.e. the negated predicate must not depend on the head predicate.

Then one can apply **Negation by Failure**.
OPS 5 (Official Production System) forward rule system developed in the 70th (at that time for expert systems).

The most important features are

- simple class definition
- working memory
- Forward rules for modifying the working Memory
- Strategies for conflict resolution if several rule fire at the same time.
- efficient evaluation algorithm for the rules (Rete Algorithm)

See

http://www.math-cs.gordon.edu/courses/cs323/OPS5/ops5.html
Class Declaration in OPS5

Example:

(literalize Student
  Name
  Matrikelnummer
  Semester
  Durchschnittsnote
  Module)

(vector-attribute Module)

Instantiation in the Working Memory

(make Student ^Name |Karl Meyer| ^Matrikelnummer 12345
  ^ Semester 1)
Production Rules

Rule Syntax
(p rulename
   condition --> action)

Example for the condition part
(student ^Matrikelnummer <mn> ^Name <n> ^Semester <s>
   ^Durchschnittsnote {> 4})
• mn, n and s are variables, bound at the corresponding student data.
• Matches all students in the working memory with Durchschnittsnote > 4. The variables are bound at the corresponding attribute values.
Example for the Action Part

- (make Student ...) creates a new instance in the working memory
- (remove 1) removes the instance matching with the first condition part.
- (modify 1 ^Semester (compute s – 1)) the student's semester matching the first condition part is changed (s would be the Semester number)

All elements in the working memory get a timestamp recording the time of the last change.
Evaluation Strategy

For the current state of the Working Memory

- Determine all rules which can fire ("conflict set")
- Select a rule ("selection strategy")
- Execute the action part.

Selection Strategies:
- REFRACTION: no rule instance can fire twice in succession
- RECENCY: Elements in the working memory with newer timestamp are preferred
- SPECIFICITY: more specific rule instances (with more condition parts) are preferred
Aspects

• The Working Memory can become very large.
• Every rule application modifies only a small part of the working memory.
• The number of applicable rules (conflict set) may become very large.
• After every rule application one must update the conflict set although not much may have been changed.

The Rete-Algorithm is an intelligent indexing technique for quickly determining the relevant parts of the conflict set after changing the working memory.
OPS 5 Summary

- OPS 5 was one of the first production rule system.
- It had one of the first class systems.
- There was no Frame Problem, because only the current state is stored. Therefore the rules cannot refer to earlier states.
- Conflict Set und Selection Strategies
- Ancestor of today's rule systems with forward reasoning

**Modern Implementation**: Jess Rule Engine for Java
Prolog (Programming in Logik) is a rule based programming language with backward reasoning, also developed in the 70th.

There was also a kind of Working Memory.
These are the Prolog Facts e.g.

mann(adam).
mann(tobias).
mann(frank).
frau(eva).
frau(daniela).
frau(ulrike).
vater(adam,tobias).
vater(tobias,frank).
mutter(tobias,ulrike).
vater(eva,tobias).
vater(daniela,frank).
mutter(daniela,ulrike).

Queries for the Working Memory:
?- mann(tobias).  Answer: yes
?- mann(X).       Answer: X = adam
                   X = tobias
                   X = frank
                   no.
Prolog Rules (Clauses)

Beispiele:

grandfather(X,Y) :- father(X,Z), father(Z,Y).

X,Y are variables (starting with capital letters).
Meaning: In order to find an X, with a grandfather Y, look for a father Z of X, and for Z a father Y.

Queries:

?- grandfather(adam,frank).    answer: yes.
?- grandfather(X,frank).        answer: X=adam
?- grandfather(adam,X).         answer: X=frank
?- grandfather(X,Y).            answer: X=frank Y = adam
Recursion in Prolog

Example:
\[
\text{ancestor}(X, Z) \quad : \quad \text{parent}(X, Z).
\]
\[
\text{ancestor}(X, Z) \quad : \quad \text{parent}(X, Y), \text{ancestor}(Y, Z).
\]
\[
\text{parent}(X, Y) \quad : \quad \text{mother}(X, Y); \text{father}(X, Y).
\]

Anfrage:
?- ancestor(adam, X).

Anwort:
\[
X = \text{tobias}
\]
\[
X = \text{ulrike}
\]
\[
X = \text{frank}
\]
In order to evaluate a query
?- p(t1,...,tn)
• the system searches the facts and rule heads top down (the sequence is important)
• tries to unify the query with the fact or head (i.e. find variables which make both terms equal).
• if the unification succeeds,
  - establish a „choicepoint“ for later backtracking and
  - and evaluate recursively the body literals left to right
• if the evaluation of the body literals fails, backtract to the last choice point and continue with the next alternative.
Example: ABB - CD = EED
- - *
FD + EF = CE
= = =
EGD * FH = ???

Search bindings 0,1,2,3,4,5,6,7,8,9 for the letters A,...,H, which solve the equation system.

Prolog Solution:

```
gen(A,B,C,D,E,F,G,H) :- permutation([A,B,C,D,E,F,G,H,_,_], [0,1,2,3,4,5,6,7,8,9]).
```

binds all permutationen of 0,...,9 at the variables.
Beispiel: ABB - CD = EED

- - *

FD + EF = CE

= = =

EGD * FH = ???

Prolog Program

gl1(A,B,C,D,E) :- ((A*100 +B*10+B) - (C*10+D)) =:= (E*100+E*10+D).

gl2(C,D,E,F) :- ((F*10+D) + (E*10+F)) =:= (C*10+E).

gl3(A,B,D,E,F,G) :- ((A*100+B*10+B) - (F*10+D)) =:= (E*100+G*10+D).

gl4(C,D,E,F,H) :- ((C*10+D) - (E*10+F)) =:= (F*10+H).

gl5(C,D,E,F,G,H,X) :-

    ((E*100+E*10+D) * (C*10+E)) =:= ((E*100+G*10+D) * (F*10+H)),

    X is ((E*100+G*10+D) * (F*10+H)).

solution :- gen(A,B,C,D,E,F,G,H), gl1(A,B,C,D,E), gl2(C,D,E,F),

          gl3(A,B,D,E,F,G), gl4(C,D,E,F,H), gl5(C,D,E,F,G,H,X), write(X).

tries all permutations of the numbers and tests the equations.
Important Aspects of Prolog

- Functions and arbitrary nested terms
- Unification of the terms (Finds the bindings for the Variables which make the terms equal)
- Backward reasoning with backtracking
- The sequence of the clauses and literals is important
- Built-in datatypes (terms, arithmetics, Lists, ...)
- Negation by Failure
- Many built-in predicates
- Modification of the Working Memory by `assert` and `retract`
- Programmable stop points for the backtracking by `!` (cut-Literal).
- Extensions of Prolog by Constraint Reasoning Systems.
Die Semantic Web Rule Language

Combines Datalog and OWL

It is a *logical* language for forward rules, which may refer to OWL classes.

See http://www.w3.org/Submission/SWRL/
The Semantic Web Stack
SWRL Rules

\[
\text{Man}(?m) \rightarrow \text{Person}(?m)
\]

´Man´ is a subclass of ´Person´

Syntactic Sugar for OWL.

Variables are indicated by ?
Property Value Assignment

\[ \text{Person(?p)} \land \text{hasSibling(?p,?s)} \land \text{Man(?s)} \rightarrow \text{hasBrother(?p,?s)} \]

\[ \text{hasParent(?x, ?y)} \land \text{hasBrother(?y, ?z)} \rightarrow \text{hasUncle(?x, ?z)} \]

Is not possible in OWL.
SWRL Rules with Nominals and Built-ins

Person(Fred) ^ hasSibling(Fred, ?s) ^ Man(?s)  
→ hasBrother(Fred, ?s)

not possible in OWL.

SWRL with Built-ins

Person(\(?p\)) ^ hasAge(\(?p,?age\)) ^ swrlb:greaterThan(?age,17)  → Adult(?p)

Person(\(?p\)) ^ hasNumber(\(?p, ?number\)) ^ swrlb:startsWith(?number, "+"")  
→ hasInternationalNumber(?p, true)
Person(?p) \land \text{hasSalaryInPounds}(?p, ?pounds) \land \\
\text{swrlb:multiply}(?dollars, ?pounds, 2.0) \\
\rightarrow \text{hasSalaryInDollars}(?p, ?dollars) \\
(?dollars = ?pounds \times 2.0)
Built-in Libraries

• **Temporale built-ins:**
  – temporal:before("1999-11-01T10:00",
    "2000-02-01T11:12:12.000")
  – temporal:duration(2, "1999-11-01",
    "2001-02-01", temporal:Years)

• **TBox built-ins:**
  – tbox:isDatatypeProperty(?p)
  – tbox:isDirectSubPropertyOf(?sp, ?p)

• **Mathematical built-ins:**
  – swrlm:eval(?circumference, "2 * pi * r", ?r)
\[(\text{hasChild } \geq 1) (\?x) \rightarrow \text{Parent}(\?x)\]

**Meaning:**

The set of objects where the hasChild relation has at least one object is a subset of Parent.

The child may be unknown.

(open world assumption)
OWL class expressions are allowed at all positions where one-place predicates are allowed.

\[
\text{Parent}(\?x) \rightarrow (\text{hasChild} \geq 1)(\?x)
\]

every OWL class expression is allowed
Publication(?p) ^ hasAuthor(?p, ?y) ^ hasAuthor(?p, ?z) ^ differentFrom(?y, ?z) → cooperatedWith(?y, ?z)

Like OWL, SWRL has no unique name assumption

Different individuals must be declared different
There is owl:allDifferent
SWRL Semantics

- Is a direct extension of the OWL set semantics
- Therefore no negation in the rules!

Ex.:
Person(?p) ^ not hasCar(?p, ?c) → CarlessPerson(?p)

A negation at this position is not compatible with the open world assumption.

(?p could have a car, but we don't know)
SWRL ist monoton

Person(?p) ^ hasAge(?p,?age) ^
swrlb:add(?newage, ?age,1)
→ hasAge(?p, ?newage)

No new assignment of attributes!
No counting

Publication(?p) ^ hasAuthor(?p,?a) ^
<hasAuthor has only one entry>
→ SingleAuthorPublication(?p)

Due to the open world assumption the real number of authors is unknown.

Publication(?p) ^ (hasAuthor = 1)(?p)
→ SingleAuthorPublication(?p)

is possible because the publication must contain the information that there is only one author.
Tools

• Protégé for editing OWL and SWRL
• Pellet Rule engine for OWL and SWRL
• Jena framework for Java

The consistency problem for SWRL, however, is no longer decidable.

Therefore every rule engine must be incomplete.
SQWRL: Query Language for SWRL

Semantic Query-Enhanced Web Rule Language (squirrel)

**Ex.**:

Person(?p) ^ hasAge(?p,?age) ^ swrlb:greaterThan(?age,17)
→ sqwrl:select(?p, ?age) ^ sqwrl:orderBy(?age)

Person(?p) ^ hasCar(?p,?car)
→ sqwrl:select(?p) ^ sqwrl:count(?car)
Summary

- SWRL extends OWL by a logic based rule language.
- Logical correlations can be defined like in Datalog, but no actions like in OPS 5.
- Consistency is undecidable.
- Nevertheless operational useful by avoiding non-terminating rules.
- SQWRL as query language.
- There are now efficient rule engines.