SWAT4LS 2014 Tutorial

Introduction to Rule Technologies and Systems

9-11 Dec., Berlin, Germany

Prof. Dr. Adrian Paschke and Ralph Schäfermeier

Arbeitsgruppe Corporate Semantic Web (AG-CSW)
Institut für Informatik, Freie Universität Berlin

paschke@inf.fu-berlin
http://www.inf.fu-berlin/groups/ag-csw/
Agenda

- Theory
  - Difference Rule-based Knowledge Representation and Logic Programming vs. Ontological KR and Description Logics

- Example for Platform-Specific Rule System
  - Beyond Prolog – Prova Reactive Rule Language

- Rule Markup Languages for Platform-Independent Rule Interchange
  - Deliberation and Reaction RuleML
  - W3C SWRL (+Protege SWRL)
  - W3C RIF
Declarative Knowledge Representation

- **Declarative Knowledge Representation:**
  
  *express what is valid, the responsibility to interpret this and to decide on how to do it is delegated to an interpreter / reasoner*

- **Logic** is a discipline concerned with the principles of inference and reasoning
  - Logic-based languages used for declarative knowledge representation
Semantics Technologies for Declarative Knowledge Representation

1. Rules
   - Goal: Derived conclusions and reactions from given knowledge (rule inference / rule chaining)

2. Ontologies
   - Goal: Ontologies described the conceptual knowledge of a domain (concept semantics)
Usage of Rules

1. Rules:
   • **Derivation rules** (deliberative rules): establish / derive new knowledge.
   • **Reaction rules** that establish when certain activities should take place:
     • *Condition-Action rules* (**production rules**)
     • *Event-Condition-Action (ECA) rules* + varians (e.g. ECAP).
     • *Messaging reaction rules* (event message / query reaction rules)

2. **Constraints** on system’s structure, behavior or information:
   • **Structural constraints**.
   • **State constraints**.
   • **Process / flow constraints**.
Logic Languages for Declarative KR

- **Predicate Logic** (First Order Logic)
  - Expressive (variables, functions, quantifier, negation, …)
  - But: undecidable and complex reasoning

=> Subsets of First Order Logic are used for practical knowledge representation

⇒ **Horn Logic** (Logic Programming: Rules)

⇒ **Description Logics** (Ontologies)
Logic Programming

- Logic Programming uses **Rules** for Knowledge Representation

\[ A_1 \land \ldots \land A_n \rightarrow B \]

Antecedent / Body \(\rightarrow\) Consequent / Head

- **Rules**: antecedent \(\rightarrow\) consequent;
  - e.g. \(A \rightarrow B\) or in Prolog notation: \(B :- A\).

- **Facts**: consequent only;
  - e.g. \(-\rightarrow B\) or in Prolog notation \(B\).

- **Goals/Queries**: antecedent only;
  - e.g. \(B?\) or in Prolog \(:- B\) or in Prova \(:-\text{solve}(B)\).
Types of Logic Programs

- **Datalog Program**
  - no functions, restrictions on variables
  - extensions such as Datalog +/- (existential quantification, negation and equality in rule heads).

- **Definite Horn Logic Programs**
  - without negation

- **Disjunctive Horn Logic Programs**
  - with disjunctive connective

- **Normal Horn Logic Program**
  - with negation (as failure)

- **Extended Horn Logic Program**
  - with classical negation, e.g. in the head of a rule
Simplified Example

- Rule: All birds can fly
  
  \[\text{bird}(X) \rightarrow \text{fly}(X).\]  
  (Variables in upper case letters)

- Fact: tweety is a bird
  
  \[\text{bird}(\text{tweety}).\]  
  (Constants in lower case letters)

- Query/Goal: Which animals can fly?
  
  \[\text{fly}(X)?\]

- Result: \(x = \text{tweety}\)
Logic Programming with Negation as Failure

- **Negation** needed in many practical knowledge representation applications

- **Closed World Assumption**
  - “what is not (currently) known to be true in my knowledge base, is assumed as false“

- **Negation-as(-finite)-Failure** (NAF)
  - If we cannot prove the truth, we assume failure
Example: Negation as Failure

- **Rule:** All birds can fly
  
  \[ \text{bird (}X\text{)} \rightarrow \text{fly (}X\text{)} \].
  
  **Rule:** If an animal cannot fly it died (e.g. during the flood)
  
  \[ \sim\text{fly (}X\text{)} \rightarrow \text{died (}X\text{)} \].

- **Facts:**

  \[ \text{bird(tweety)} \].
  
  \[ \text{rabbit(jack)} \].

- **Query:** \[ \text{died(tweety)} \]?
  
  **Result:** \[ \text{no} \]

- **Query:** \[ \text{died(jack)} \]?
  
  **Result:** \[ \text{yes} \]
Logic Programming with Classical Negation

- **Open World Assumption**
  - Opposite of closed world assumption
  - "we can only assume something to be false, if we explicitly can prove the negation"

- **Classical Negation**
  - Something is false, if we explicitly prove it to be false
Example: Classical Negation

- **Rules:**
  
  *bird* (X) -> *fly* (X).
  
  *rabbits* (X) -> ¬ *fly* (X).

  **Rule:** If an animal cannot fly it dies (during the flood)
  
  ¬ *fly* (X) -> *died* (X).

- **Facts:**
  
  *bird* (tweety).
  
  *rabbit* (jack).

- **Query:**  
  
  *died* (tweety)?

  **Result:** 
  
  no

- **Query:**  
  
  *died* (jack)?

  **Result:** 
  
  no (closed world with naf)

  => yes (open world with classical negation)
Classical Negation might lead to logical conflicts

- **Rules:**
  
  \[
  \text{bird}(X) \rightarrow \text{fly}(X).
  \]
  
  \[
  \text{pinguin}(X) \rightarrow \neg \text{fly}(X).
  \]
  
  \[
  \text{pinguin}(X) \rightarrow \text{bird}(X).
  \]

- **Rule:**
  
  \[
  \neg \text{fly}(X) \rightarrow \text{died}(X).
  \]
  
  \[
  \text{fly}(X) \rightarrow \neg \text{died}(X).
  \]

- **Facts:**
  
  \[
  \text{pinguin}(\text{tweety}).
  \]

- **Query:** \( \text{died}(\text{tweety})? \) \( \neg \text{died}(\text{tweety})? \)

- **Result:** yes

\[\Rightarrow \text{logical conflict which needs to be solved (e.g. by defining non-monotonic priorities)}\]
Solution: Solving conflicts with priorities (non-monotonic Defeasible Logic)

- Rules:
  \[(r1) \text{ bird } (X) \rightarrow \text{ fly}(X).\]
  \[(r2) \text{ pinguins } (X) \rightarrow \text{ fly } (X).\]
  \[\text{ pinguin}(X) \rightarrow \text{ bird}(X).\]
  \[\text{ overrides}(r2, r1).\]

- Rule:
  \[\neg \text{ fly } (X) \rightarrow \text{ died}(X).\]
  \[\text{ fly } (X) \rightarrow \neg \text{ died}(X).\]

- Facts:
  \[\text{ pinguin}(\text{twitty}).\]

- Query: \[\text{ died}(\text{twetty})?\] \[\neg \text{ died}(\text{twetty})?\]

- Result: yes no

(but luckily pinguins can swim, so Tweety did not die! HAPPY END:-)
Monotonic vs. Non-Monotonic Logic

- Classical logic (like in DL) is **monotonic** \( \neq \) never gets smaller
  - If something is true, it’s true for all time
    - \( 3 > 2 \) always was and always will be true
- A **non-monotonic logic** is one in which a proposition’s true value can change in time
  - e.g., learning a new fact may cause the number of true propositions to decrease.
- Example: Prolog and many rule systems are non-monotonic, e.g.:
  - **assert and retract** knowledge updates
  - “negation as failure”, cut-fail, …
Unique-Names Assumption

- Logic Programming makes a **unique-names assumption (UNA)**, like in database systems
  - If two instances have a different name or ID this does imply that they are different individuals

In the example above,
- a rule reasoned with UNA **does flag an error**
- an ontology reasoner without UNA instead would **infer that the two resources are equal**

A tutorial is presented by **at most one lecturer**. The tutorial „Rule-based Systems“ is taught by „Adrian Paschke“ **and** „Ralph Schäfermeier“
Further Examples of LP Expressiveness not representable in DL

- DLs with restriction on binary predicates cannot directly support n-ary predicates
  - workaround with e.g. reification

- DL cannot represent “more than one free variable at a time”.

  \[ \text{friendshipBetween}(X,Y) \leftarrow \text{man}(X) \land \text{woman}(Y). \]

- Traditional expressive DLs support transitive role axioms but they cannot derive values of property chains*

  \[ \text{uncleOf} (X,Z) \leftarrow \text{brotherOf}(X,Y) \land \text{parentOf}(Y,Z). \]

  \[ \text{homeWorker}(X) \leftarrow \text{Work}(X, Y) \land \text{live}(X, Z) \land \text{loc}(Y,W) \land \text{loc}(Z,W). \]

* but OWL 2 introduces property chains
Summary Theory: Main Differences between DLs (Ontologies) and LPs (Rules)

**Description Logics**
- Open World Assumption (OWA)
- Generally no Unique Name Assumption (UNA)
- Classical negation (NEG)
- Monotonic

**Logic Programs**
- Closed World Assumption (CWA)
- Unique Name Assumption (UNA)
- Negation As Failure (NAF)
+ many non-monotonic extension and many extra logical built-ins
Ontologies + Rules Integration

- Combine Rules and Ontologies
  - Goal: Extend Expressiveness

- Homogeneous Integration
  - Combination of rules and ontologies (description logics) in one homogenous framework, which uses the same language symbols
  - e.g. DLP, KAON2, SWRL, OWL-RL

- Heterogeneous Integration
  - Hybrid approach (rule engine + DL reasoner)
  - Heterogeneous integration of DL inference techniques and tools in combination with rule languages and rule engines
  - e.g. CARIN, Life, Prova/ContractLog KR, Al-log, DatalogDL, non-monotonic dl-programs, r-hybrid KBs
## Comparison

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DLP</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>Reduction</td>
</tr>
<tr>
<td>SWRL</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>Reduction</td>
</tr>
<tr>
<td>KAON2</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AL-log</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>SLD-resolution</td>
</tr>
<tr>
<td>CARIN</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>Entailment</td>
</tr>
<tr>
<td>dl-programs</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>Fixpoint iteration</td>
</tr>
<tr>
<td>r-hybrid KBs</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>–</td>
</tr>
<tr>
<td>Prova</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>SLD-resolution</td>
</tr>
</tbody>
</table>

**Notes:**

1. **DLP**: Expressivity restrictions
2. **SWRL**: Undecidable
3. **KAON2**: DL-safe rules

**Notes:**

1. **AL-log**: Only concept constraints
2. **CARIN**: Recursive CARIN-ALCNR undecidable
3. **dl-programs**: Nonmonotonic semantics
4. **r-hybrid KBs**: Nonmonotonic semantics
Agenda

- **Theory**: Rule-based Knowledge Representation and Logic Programming
- **Example**: Beyond Prolog – Prova Rule Language
- **Rule Markup Languages**
  - Deliberation and Reaction RuleML
  - W3C SWRL (+Protege SWRL)
  - W3C RIF
Logic Programming beyond Prolog

Prova

Prolog + Java

http://prova.ws
Prova: Declarative Rules for Java

- Combine the benefits of declarative and object-oriented programming (and workflow languages);
- Interpreted scripting syntax combines those of ISO Prolog and Java + Ontologies (as type system);
- Access data sources via wrappers written in Java, query language built-ins (e.g. SQL, SPARQL, RDF Triples, XQuery) or message-driven (Web) service interfaces;
- Make all Java APIs (e.g. EJBs) from available packages directly accessible from rules;
- Be compatible with modern distributed enterprise service and agent-based software architectures
  - Including event messaging and complex event processing
Introduction: Prova Prolog Syntax

- Variables (upper case): X, Y, _, ...
- Constants (lower case): a, b, ab, "String", 1, 2, 3 ...
- Functions (used as complex terms): p(f(x))

- Fact: availability(s1, 99%).
- Rule: qos(S, high):-
  availability(S, 99%).

- Query 1: :-solve (qos(S, high)).
  returns all solutions with all variable bindings for S
- Query 2: :-eval (qos(S, high)).
Prova Syntax – Quick Overview

- Variables (upper case), Constants (lower case)
- Fact: \texttt{availability(s1,99\%)}.
- Rule: \texttt{qos(S,high):- availability(S,99\%).}
- Query 1: \texttt{:-solve (not(qos(S,high)).}
- Query 2: \texttt{:-eval (not(qos(S,high)).}
- Derive: \texttt{, derive([X|Args]), ...}
- Scoping: \texttt{@label("www.prova.ws") p(X), ...}
- Memoization: \texttt{cache(p(X)), ...}
- Lists: \texttt{[Head|Tail] = [Head,Arg2,...,ArgN] = Head(Arg2,...,ArgN)}
- Module Imports:
  \texttt{:- eval(consult("ContractLog/list.prova")).}
  \texttt{:- eval(consult("http://rule.org/list.prova")).}
- Meta data annotation:
  \texttt{@label(r1), @src("www.prova.ws"), @dc_author("AP"))}
  \texttt{qos(S,medium) :- availability(S,98\%).}
Prova Example: General language constructs

Prova extends ISO Prolog syntax:

% Facts
i_am("mediator").
portfolio("balanced",P).
reachable(X,X).
is_a("anticoagulant","molecular_function").

% Clauses (head true if conditions true in the body)
parent(X,Y) :-
    is_a(Y,X).
parent(X,Y) :-
    has_a(X,Y).

% Goals (note there is no head)
:- solve(parent(Parent,"anticoagulant")). % Print solutions
:- eval(parent(Parent,"anticoagulant")). % Just run exhaustive search

Format of the output for the goal solve above:
Parent="molecular_function"
Syntax design features of Prova

- The syntax combines ISO Prolog and Java but the key is simplicity:

  % Prolog
  N2 is N + 1,

  ----------------------------------------

  % Prova
  N2 = N + 1,

  // Java
  List l = new java.util.ArrayList();

  % Prova
  L=java.util.ArrayList(),

- Low-cost creation of distributed integration and computation workflows.
- Prova separates logic, data, and computation.
- Low-cost integration of rule-base scripted agents inside Java and Web applications.
- For more information check the User`s Guide in the prova web page
  - http://prova.ws
Pervasive use of the Java type system

- Java typed and untyped variables;
  - Natural rules for subclasses unification;
- Java variables prefixed by full package
  - prefix with *java.lang* being default;

```prolog
:- solve(member(X,[1,Double.D,"3"])).
:- solve(member(Integer.X,[1,Double.D,"3"])).

% Standard type-less rules for the standard member predicate
member(X,[X|Xs]).  % X is a member of a list if it is the first element
member(X,[_|Xs]) :- % X is a member of a list if it is in the list tail
    member(X,Xs).

-----------------------------
> X=1
> X=java.lang.Double.D
> X=3

> java.lang.Integer.X=1
```
Types in Prova

- **Sorted Logic**
  - Order-Sorted Logic Programs (Typed Rules)

- **Dynamic Type Checking**
  - at runtime

- **Polymorphic Order-Sorted (Subtyping) Unification**
  - *Ad-hoc polymorphism*: Variables might change their types during unification
    - Type-casting in the sense of *coercion* is supported by typed unification
  - **Overloading**: rule/function overloading, i.e. rules with the same head but different type signatures
Example – Typed Function Overloading

\[
\textbf{add}(\text{Integer} \cdot I_1, \text{Integer} \cdot I_2, \text{Result}) : - \\
\quad \text{Result} = I_1 + I_2.
\]

\[
\textbf{add}(\text{String} \cdot I_1, \text{String} \cdot I_2, \text{Result}) : - \\
\quad \textbf{concat}([I_1, I_2], \text{Result}).
\]
Description Logic (DL) type system

- DL-typed and untyped variables;
- Uses **Semantic Web ontologies** as type systems;
- Uses external DL reasoner for dynamic type checking;
- Syntax: `[Variable]^^[nameSpace]:[ClassType]
  [individualConstant]^^[nameSpace]:[ClassType]`

```prolog
% needed
import('ContractLog/owl.prova').

% import external type system (T-Box model) and individuals (A-Box)
import('http://example.org/WineProjectOWL.owl').

% use OWL-DL reasoner; for a list of available predefined reasoners see OWL2PROVA.java
reasoner ('dl').

% typed rule
serve (X^^default:Wine) :- recommended(X^^default:Wine).
% ground fact; defines an instance of class
recommended("Chardonnay"^^default:Wine).
% non ground DL facts are interpreted as queries on external ontology
recommended(X^^default:White_Wine).
recommended(X^^default:Red_Wine).
:- solve(recommended(X^^default:Wine)).
:- solve (recommended(X^^default:White_Wine)).
```
Procedural Attachments - Java Method Calls (runtime reflection)

- Constructors, instance and static methods, and public field access;

- Ability to embed Java calls makes the Prolog-like programming style more suitable for object integration and computation workflows.

```
hello(Name):-
    S = java.lang.StringBuffer("Hello"),
    S.append (Name),
    java.lang.System.out.println (S).
```

```
hello(Name):-
    S1 = java.lang.String("Hello"),
    S2 = S1.concat (Name),
    java.lang.System.out.println (S2).
```
Positional Notation

- "The **discount** for a *customer* buying a *product* is 5 percent if the *customer* is **premium** and the *product* is **regular**."

- Positional notation of predicates terms

\[
\text{discount}(\text{Customer, Product, 5}) :- \\
\text{premium}(\text{Customer}), \\
\text{regular}(\text{Product}).
\]

Application: Standard Logic Programming
Unpositional Notation

- Unpositional Notation = Slotted terms

  Position independent with user-defined key

  (notation in Prova: “key -> value” or “key : value”)

  \[
  \text{discount}({\text{buyer}->\text{Customer}, \text{item}->\text{Product}, \text{rebate}->5}) :\- \\
  \text{discount}({\text{item}->\text{Product}, \text{buyer}->\text{Customer}, \text{rebate}->5}) :\- \\
  \text{discount}({\text{rebate}->5, \text{item}->\text{Product}, \text{buyer}->\text{Customer}}) :\-
  \]

  Application: usefull e.g. for order independent database relations (deductive database) and object-centric relations
Example

- Slotted terms is a highly valuable feature
  - to represent unpositional arguments in relations, e.g. relations from databases

<table>
<thead>
<tr>
<th>Firstname</th>
<th>Semester</th>
<th>ID</th>
<th>Lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>5</td>
<td>123</td>
<td>Do</td>
</tr>
</tbody>
</table>

  \[
  \text{student(id->123, firstname->John, lastname->Do, semester->5)}
  \]

- to represent object-relational knowledge

  \[
  \text{student(id->123, person(firstname->John, lastname->Do), semester->5)}
  \]

- Semantics: see Frame Logic and PSOA Semantics [Boley RuleML 2011]
Prova Built-In Examples

- Simple arithmetic relations (+ - = ...)

```
% Prova
N2 = N + 1,
```

- Negations (not, neg)

```
% Default Negation
register(User) :- not(known(User)), ...
```

- Fact base updates

```
% Update global facts
register(User) :- not(known(User)), assert(known(User)).
```

- Variable mode tests (free, bound, type)

- String manipulation predicates

```
% Concat Strings
concat(["{","In,"}"],Out), % prepend "{" and append "}"
```
External Data and Object Integration

- **File Input / Output**
  
  ..., `fopen(File,Reader), ...`

- **XML (DOM)**
  
  `document(DomTree,DocumentReader) :- XML(DocumenReader),`

- **SQL**
  
  `... sql_select(DB,cla,[pdb_id,"lalx"],[px,Domain]).`

- **RDF**
  
  `... rdf(http://...,"rdfs",Subject,"rdf_type","genel_Gene"),`

- **XQuery**
  
  `... XQuery = ' for $name in StatisticsURL//Author[0]/@name/text() return $name', xquery_select(XQuery,name(ExpertName))`,

- **SPARQL**
  
  `... sparql_select(SparqlQuery,name(Name),class(Class), definition(Def)),`
Examples: File I/O and SPARQL

test_fopen() :-
    fopen(File,Reader),
    % Non-deterministically enumerate lines in the file
    read_enum(Reader,Line),
    println([Line]). % Print one line at a time

exampleSPARQLQuery(URL,Type|X) :-
    QueryString =
        ' PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    SELECT ?contributor ?url ?type
    FROM <http://planetrdf.com/bloggers.rdf>
    WHERE {
        ?contributor foaf:name "Bob DuCharme" .
        ?contributor rdf:type ?type . }
    ,
    sparql_select(QueryString,url(URL),type(Type)|X),
    println([[url,URL],[type,Type]|X],",",").
Distributed Prova Agents (http://prova.ws)

- Java JMS based, open source rule language for reactive agents and event processing
- Leverages declarative ISO Prolog standard extended with (event,message) reaction logic, type systems (Java, Ontologies), query built-ins, dynamic Java integration.
- Combines declarative, imperative (object-oriented) and functional programming styles
- Designed to work in distributed Enterprise Service Bus and OSGi environments
- Supports strong, loose and decoupled interaction
- Compatible with rule interchange standards such as Reaction RuleML

![Diagram of Prova Agent Architecture]

- **Strong Coupling**
  - Direct API implementation
  - Prova

- **Loose Coupling**
  - Message Interchange
  - Rule Service Consumer

- **Decoupled**
  - Event Streams
    - (ordered events)
    - Event Cloud
      - (unordered events)
      - new auto pay
      - account login
      - deposit
      - logout
      - withdrawal
      - new auto pay
      - transfer
      - enquiry
      - incident
      - book
      - request
      - account balance
      - account history

- **Agent / Service Interface**
  - Request
  - Response
  - Prova Agent
  - Prova Agent
  - Prova Agent

- **Computer**
  - Prova
  - Prova Agent
  - Prova Agent

- **Network**
  - Prova Event Processing Network
  - Prova Agent
  - Prova Agent
  - Prova Agent
Prova Messaging

- Send a message
  \[sendMsg(XID,\text{Protocol},\text{Agent},\text{Performative},[\text{Predicate}|\text{Args}]|\text{Context})]\]

- Receive a message
  \[rcvMsg(XID,\text{Protocol},\text{Agent},\text{Performative},[\text{Predicate}|\text{Args}]|\text{Context})]\]

- Receive multiple messages
  \[rcvMult(XID,\text{Protocol},\text{Agent},\text{Performative},[\text{Predicate}|\text{Args}]|\text{Context})]\]

Description:
- \textit{XID} is the conversation identifier
- \textit{Protocol}: protocol e.g. \textit{self}, jms, esb etc.
- \textit{Agent}: denotes the target or sender of the message
- \textit{Performative}: pragmatic context, e.g. FIPA Agent Communication
- \([\text{Predicate}|\text{Args}]\) or \text{Predicate}(\text{Arg}_1,..,\text{Arg}_n): Message payload
rcvMsg(XID, esb, From, query-ref, buy(Product)) :-
  routeTo(Agent, Product), % derive processing agent
  % send order to Agent in new subconversation SID2
sendMsg(SID2, esb, Agent, query-ref, order(From, Product)),
  % receive confirmation from Agent for Product order
rcvMsg(SID2, esb, Agent, inform-ref, order(From, Product)).

% route to event processing agent 1 if Product is luxury
routeTo(epa1, Product) :- luxury(Product).
% route to epa 2 if Product is regular
routeTo(epa2, Product) :- regular(Product).

% a Product is luxury if the Product has a value over ...
luxury(Product) :- price(Product, Value), Value >= 10000.
% a Product is regular if the Product has a value below ...
regular(Product) :- price(Product, Value), Value < 10000.

**corresponding serialization with Reaction RuleML <Send> and <Receive>**
Message Driven Routing
Prova : Event Routing in Event-Driven Workflows

```
rcvMsg(XID, Process, From, event, ["A"]) :-
    fork_b_c(XID, Process).

fork_b_c(XID, Process) :-
    @group(p1) rcvMsg(XID, Process, From, event, ["B"]),
    execute(Task1), sendMsg(XID, self, 0, event, ["D"]).

fork_b_c(XID, Process) :-
    @group(p1) rcvMsg(XID, Process, From, event, ["C"]),
    execute(Task2), sendMsg(XID, self, 0, event, ["E"]).

fork_b_c(XID, Process) :-
    % OR reaction group "p1" waits for either of the two
    % event message handlers "B" or "C" and terminates the
    % alternative reaction if one arrives
    @or(p1) rcvMsg(XID, Process, From, or, _).
```
Example: RuleResponder e-Scientist

Rule-based Workflow Specification
(Described by Reaction and Derivation Rules)

Event Messaging in Real Time

Enterprise Service Bus

Rule Agent

Rule Agent

Process Engine

Rule Agent

Rule Agent

Human Agent

Exception Handling Agent

Agent-Task-Service Ontology

Client

http://www.csw.inf.fu-berlin.de/sswfs/
% Manager

upload_mobile_code(Remote, File) :
Writer = java.io.StringWriter(), % Opening a file fopen(File, Reader),
copy(Reader, Writer),
Text = Writer.toString(),
SB = StringBuffer(Text),
 sendMsg(XID, esb, Remote, eval, consult(SB)).

% Service (Contractor)

rcvMsg(XID, esb, Sender, eval, [Predicate|Args]) :- derive([Predicate|Args]).
Summary – Prova: beyond Prolog

- Combines declarative programming (rules) with object-oriented programming (Java)
- **Typed Logic Programming**: order-sorted type system
  - supports ad-hoc polymorphism and overloading
- Dynamic Objects construction and **procedural attachments**
  - benefit: object APIs accessible from rules at runtime
- **External data access** via query built-ins and object APIs
Agenda

- **Theory**: Rule-based Knowledge Representation and Logic Programming
- **Example**: Beyond Prolog – Prova Rule Language

**Rule Markup Languages**
- Deliberation and Reaction RuleML
- W3C SWRL (+Protege SWRL)
- W3C RIF
Rule Markup Languages

Platform-Independent Rule Interchange with RuleML Standards

http://ruleml.org
Rule interchange with RuleML (on the platform independent level)
RuleML Organization

- Has an open **non-profit** standardization structure
- Drives the specification of standard **semantic technology & business rules**
- Coordinates rule research & development and holds international meetings
RuleML Interoperation Effort

- XML-based interchange between (sublanguages of) RIF, SWRL, SWSL, SBVR, PRR, DMN, N3, Prolog, Rulelog, CL etc.
RuleML Enables ...

Rule modelling markup translation interchange execution publication archiving in UML RDF XML ASCII..
RuleML Core Technology

- XML schemas
  - In Relax NG
    - Based on MYNG customization approach and Web GUI
  - In XML Schema Definition Language (XSD)

- Presentation & Visualization syntaxes
  - Positional-Slotted Language (POSL) and Prolog (Prova)
  - Grailog (graphical)

- Transformations
  - XSLT normalizer, ANTLR parsers, JAXB unmarshalling of RuleML/XML into Java objects

- Translators (interchange/interoperation tools)
  - Prolog, Jess, N3, ...

- Model-theoretic semantics
  - For (Naf-free, OID/slot-free) FOL, Hornlog, Datalog RuleML: Classical
  - Positional-Slotted Object-Applicative (PSOA) RuleML
RuleML Extended Technology

- Translators (interchange/interoperation tools), e.g.
  - Translators between RuleML and Prolog (Prova), Jess, N3, …
  - RuleML ↔ POSL
  - RuleML → TPTP
  - Attempto Controlled English (ACE) → RuleML
- IDEs (editors, generators)
- Engines (e.g. OO jDREW, Prova, DR-DEVICE)
- APIs (e.g. Rulestore API, API4KB)
- Multi-agent frameworks (e.g. Rule Responder, EMERALD)
- Other tools
  (http://wiki.ruleml.org/index.php/RuleML_Implementations)
RuleML Family of Sublanguages

RuleML

Equality → Naf
Rewriting

Deliberation

Modal

HOL
FOL

Derivation

Fact
Query

Hornlog
Datalog

Reaction

CEP
ECAP
KR

ECA

Trigger (EA)
Production (CA)

Consumption / Selection Policies
Event Algebra
Action Algebra

Optional mix-in of
subClassOf
Syntactic specialization of

Reaction RuleML

http://reaction.ruleml.org
Tutorials about Reaction RuleML at http://www.slideshare.net/swadpasc/
Reaction Rules: Four Sub-branches

- **PR Reaction RuleML**: Production Rules (Condition-Action rules)
- **ECA Reaction RuleML**: Event-Condition-Action (ECA) rules
- **CEP Reaction RuleML**: Rule-based Complex Event Processing (complex event processing reaction rules, (distributed) event messaging reaction rules, query reaction rules, etc.)
- **DR and KR Reaction RuleML**: Knowledge Representation with Temporal/Event/Action/Situation/Transition/Process Logics and Calculi
Quick Overview: Reaction RuleML Dialects

- **Spatio-Temporal Derivation RuleML** (*if-then)*
  - Time, Spatial, Interval

- **KR RuleML** (*if-then or on-if-do)*
  - Situation, Happens (@type), Initiates, Terminates, Holds, fluent

- **Production RuleML** (*if-do)*
  - Assert, Retract, Update, Action

- **ECA RuleML** (*on-if-do)*
  - Event, Action, + (event / action algebra operators)

- **CEP** (arbitrary combination of *on, if, do*)
  - Receive, Send, Message

* + variants and alternatives
Modular SYNtax ConfiGurator (MYNG)

- RuleML schemas customized by MYNG
- MYNG GUI and REST interface, e.g.
  - GUI access to automatically generated monolithic XSD schemas that are compatible with XML tools, e.g. JAXB
  - Myng code display and myng-code URL access to user-configured RuleML sublanguages
RuleML – Constants / Individuals

- Logical constants (individuals) are represented with `<Ind>` tags in RuleML

<table>
<thead>
<tr>
<th>Prova/Prolog</th>
<th>RuleML</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td><code>&lt;Ind&gt;abc&lt;/Ind&gt;</code></td>
</tr>
<tr>
<td>“Adrian Paschke”</td>
<td><code>&lt;Ind&gt;Adrian Paschke&lt;/Ind&gt;</code></td>
</tr>
<tr>
<td>42</td>
<td><code>&lt;Ind&gt;42&lt;/Ind&gt;</code></td>
</tr>
</tbody>
</table>
RuleML - Data

- Data is represented with `<Data>` tags in RuleML.

```
1
<Data xsi:type="xs:integer">1</Data>

datetime(2002,10,10,17,00,00)

<Data xsi:type="xs:dateTime">2002-10-10T17:00:00Z</Data>
```
RuleML - Variables

- Logical variable terms are represented with `<Var>` tags in RuleML

Prova: X

RuleML: `<Var>X</Var>`
RuleML - Functions

- Function expressions / complex terms are represented using the `<Expr>` tag
- The function name is represented by embedded `<Fun>` tag
- Arguments can be `<Ind>`, `<Var>`, `<Data>`, `<Expr>` or `<Plex>`

**Prova**

\[ f(X) \]

**RuleML**

\[
<Expr>
  <Fun>f</Fun>
  <Var>X</Var>
</Expr>
\]
RuleML - Lists

- Lists are represented using the `<Plex>` tag

Prova

[p,1,2]
p(1,2)

RuleML

```xml
<Plex>
  <Rel>p</Rel>
  <Ind>1</Ind>
  <Ind>2</Ind>
</Plex>
```
RuleML - Lists

- The RuleML `<repo>` (rest, positional) tag for the Prolog/Prova “|” operator

Prova
[ Head | Rest ]

RuleML

```xml
<Plex>
  <Var>Head</Var>
  <repo>
    <Var>Rest</Var>
  </repo>
</Plex>
```
RuleML – Atomic Formula / Predicates

- Atomic formulas (predicates) are represented using the <Atom> tag
- Relation name is represented as an embedded <Rel> tag
- Arguments are embedded in <Atom> - these can be <Ind>, <Data>, <Var>, <Expr>, and <Plex>

Prova/Prolog
spending(“Peter Miller”, “min 500 euro”, “previous year”).

RuleML
<Atom>
   <Rel>spending</Rel>
   <Ind>Peter Miller</Ind>
   <Ind>min 500 euro</Ind>
   <Ind>previous year</Ind>
</Atom>
RuleML - Quantification

- Universal quantification is represente by a <Forall> tag, existential by an <Exist> tag, and a generic quantifier by a <Quantifier> tag.

\[
\forall X \exists Y \ldots
\]

FOL

RuleML

<Forall>

<Var>X</Var>

<Exists>

<Var>Y</Var>

</Exists>

...</n

</Expr>
RuleML - Negation

- Negation as Failure is represented by a <Naf> tag, classical/strong negation by a <Neg> tag, and a generic polymorphic negation by a <Negation> tag.

<table>
<thead>
<tr>
<th>FOL</th>
<th>RuleML</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg p(X) )</td>
<td>(&lt;\text{Neg}&gt;)</td>
</tr>
<tr>
<td></td>
<td>(&lt;\text{Atom}&gt;&lt;\text{Rel}&gt;p&lt;\text{Rel}&gt;&lt;\text{Var}&gt;X&lt;\text{Var}&gt;&lt;/\text{Atom}&gt;)</td>
</tr>
<tr>
<td></td>
<td>(&lt;\text{Neg}&gt;&lt;/\text{Neg}&gt;)</td>
</tr>
<tr>
<td>( \sim p(X) )</td>
<td>(&lt;\text{Naf}&gt;)</td>
</tr>
<tr>
<td></td>
<td>(&lt;\text{Atom}&gt;&lt;\text{Rel}&gt;p&lt;\text{Rel}&gt;&lt;\text{Var}&gt;X&lt;\text{Var}&gt;&lt;/\text{Atom}&gt;)</td>
</tr>
<tr>
<td></td>
<td>(&lt;\text{Naf}&gt;&lt;/\text{Naf}&gt;)</td>
</tr>
</tbody>
</table>
RuleML - Equality

- An equational formula consisting of two expressions is represented by an `<Equal>` tag.

Prova

\[ X = 1 \]

RuleML

\[
\begin{align*}
\text{<Equal>}
\text{<Var>X</Var>}
\text{<Ind>1</Ind>}
\end{align*}
\]
Rules: Specializable Syntax

<Rule @key @keyref @type @style ...(and further attributes)>

<meta> <!-- descriptive metadata of the rule --> </meta>

<scope> <!-- scope of the rule e.g. a rule module --> </scope>

<guard> <!-- scope of the rule e.g. a rule module --> </guard>

<evaluation> <!-- intended semantics --> </evaluation>

<signature> <!-- rule signature --> </signature>

<qualification> <!-- e.g. qualifying rule metadata, e.g. priorities, validity, strategy --> </qualification>

<quantification> <!-- inner quantifying rule declarations, e.g. variable bindings --> </quantification>

<oid> <!-- object identifier --> </oid>

<on> <!-- event part --> </on>

<if> <!-- condition part --> </if>

<then> <!-- (logical) conclusion part --> </then>

<do> <!-- action part --> </do>

<after> <!-- postcondition part after action, e.g. to check effects of execution --> </after>

<else> <!-- (logical) else conclusion --> </else>

<elsedo> <!-- alternative/else action, e.g. for default handling --> </elsedo>

</Rule>
Example Rule Types

- Derivation Rule:
  
  ```xml
  <Rule style="reasoning">
    <if>...</if>
    <then>...</then>
  </Rule>
  ```

- Production Rule:
  
  ```xml
  <Rule style="active">
    <if>...</if>
    <do>...</do>
  </Rule>
  ```

- Trigger Rule:
  
  ```xml
  <Rule style="active">
    <on>...</on>
    <do>...</do>
  </Rule>
  ```

- ECA Rule:
  
  ```xml
  <Rule style="active">
    <on>...</on>
    <if>...</if>
    <do>...</do>
  </Rule>
  ```
Example Production Rule

```xml
<Rule style="active"> <!-- production rule -->
  <if>
    <And>
      <Atom><Rel>available</Rel><Var>Service</Var></Atom>
      <Atom><Rel>request</Rel><Var>Task</Var></Atom>
    </And>
  </if>
  <do>
    <Assert>
      <Atom>
        <Rel>loaded</Rel>
        <Var>Service</Var><Var>Task</Var>
      </Atom>
    </Assert>
  </do>
</Rule>
```

Further examples on Github:
https://github.com/RuleML/reaction-ruleml/tree/master/exa
RuleML – Implicational Derivation Rules

- Derivation Rules (in Prolog: head :- body)

```xml
<Rule style="reasoning">
  <if>...</if>
  <then>...</then>
</Rule>
```

This can be also represented with the short cut `<Implies>` tag

- First child element is the body of the rule can be either a single `<Atom>` or a conjunction of `<Atom>`s in an `<And>` tag
- Second child element is the head of the rule this must be an atomic formula (Atom)
RuleML – Implies Example

"The **discount** for a **customer** buying a **product** is **5.0 percent** if the **customer** is **premium** and the **product** is **regular**."

[Diagram showing the RuleML representation and mapping to an RDF graph]
Striped Syntax vs. Stripe-skiped Syntax

<Implies>
  <if>
    <Atom>
      <Rel>spending</Rel> <Var>customer</Var>
      <Ind>min 5000 euro</Ind> <Ind>previous year</Ind>
    </Atom>
  </if>
  <then>
    <Atom>
      <Rel>premium</Rel> <Var>customer</Var>
    </Atom>
  </then>
</Implies>

Striped-skiped = without the role /edge tags

<Implies>
  <Atom>
    <Rel>spending</Rel> <Var>customer</Var>
    <Ind>min 5000 euro</Ind> <Ind>previous year</Ind>
  </Atom>
</Implies>

<Implies>
  <Atom>
    <Rel>premium</Rel> <Var>customer</Var>
  </Atom>
</Implies>
Example: Datalog RuleML (without functions)

Ternary Relation discount conditional on unary premium and regular

"The discount for a customer buying a product is 5.0 % if the customer is premium and the product is regular."

```xml
<Implies>
  <then>
    <Atom>
      <Atom>
        <Rel>discount</Rel><Var>cust</Var><Var>prod</Var><Data>5.0 percent</Data>
      </Atom>
    </Atom>
  </then>
  <if>
    <And>
      <Atom><Rel>premium</Rel><Var>cust</Var></Atom>
      <Atom><Rel>regular</Rel><Var>prod</Var></Atom>
    </And>
  </if>
</Implies>
```
Example: Hornlog RuleML with functions

Functional Expressions as terms in Atoms and in other Exprs can be uninterpreted, using attribute per with filler "copy" or interpreted, using it with filler "value"

```xml
<Implies>
  <then>
    <Atom>
      <Rel>discount</Rel>
      <Var>cust</Var>
      <Var>prod</Var>
      <Expr><Fun per="copy">percent</Fun><Data>5.0</Data></Expr>
    </Atom>
  </then>
  <if> . . . </if>
</Implies>
```
Example: RuleML with Equality

Equality formulas act as extension to sublanguages such as Datalog RuleML, Hornlog RuleML, and FOL RuleML. Equal has oriented attribute with value "no" default.

```
<Implies>
  <then>
    <Equal oriented="yes">
      <Expr>
        <Fun per="value">discount</Fun>
        <Var>cust</Var>
        <Var>prod</Var>
      </Expr>
      <Data>5.0 percent</Data>
    </Equal>
  </then>
  <if> . . . </if>
</Implies>
```
Example: First Order Logic (FOL) RuleML

Extension of Hornlog RuleML mainly adding classical negation and (explicit) quantifiers

"A customer receives either a discount of 5.0 percent for buying a product or a bonus of 200.00 dollar if the customer is premium and the product is regular."

\[
\text{Xor}(A,B) \leftrightarrow \text{And}(\text{Or}(A,B),\text{Not}(\text{And}(A,B)))
\]

\[\text{<Implies>}
\text{<then>}
\text{<Xor>}
\text{<Atom>}
\text{<Rel>discount</Rel>}
\text{<Var>cust</Var>}
\text{<Var>prod</Var>}
\text{<Data>5.0 percent</Data>}
\text{</Atom>}
\text{<Atom><Rel>bonus</Rel><Var>cust</Var><Data>200.00 dollar</Data></Atom>}
\text{</Xor>}
\text{</then>}
\text{<if> . . . </if>}
\text{</Implies>}

"The discount for a customer buying a product is 5 percent if the customer is premium and the product is regular."

Positional Notation

discount(Customer, Product, 5) :-
    premium(Customer),
    regular(Product).

Application: Standard Logic Programming
Unpositional KR Notation

- Unpositional Notation = Slotted

Position independent with user-defined key

(notation in Prova: “key -> value” or “key : value”)

discount({buyer->Customer, item->Product, rebate->5}) :-
discount({item->Product, buyer->Customer, rebate->5}) :-
discount({rebate->5, item->Product, buyer->Customer}) :-

Application: usefull e.g. for order independent database relations (deductive database) and object-centric relations
Slotted Un-positional Object Oriented Representation

- Position independent user-defined role -> filler term pairs
- Useful e.g. for order independent database relations (deductive database)

```xml
<Implies>
  <Atom>
    <Rel>spending</Rel>
    <slot><Ind>spender</Ind><Var>customer</Var></slot>
    <slot><Ind>amount</Ind><Ind>min 5000 euro</Ind></slot>
    <slot><Ind>period</Ind><Ind>previous year</Ind></slot>
  </Atom>
  <Atom>
    <Rel>premium</Rel>
    <slot><Ind>client</Ind><Var>customer</Var></slot>
  </Atom>
</Implies>
```
RuleML Types (Sorted RuleML)

- External and internal types (sorts) can be assigned by using the `@type` attribute
- External vocabularies / ontologies define types, e.g.,
  
  ```xml
  <Var type="&vo;DatetimeEvent">E</Var>
  <Ind iri="&vo;e1" type="&vo;DatetimeEvent"/>
  ```

- Semantics defined by **Semantic Profiles**
  - e.g. multi-sorted or order-sorted logic for type interpretation
  - e.g., import semantics (not just union of imported sorts)
  - e.g., mapping semantic with isomorphic structures interpreting composite structures as flat FOL structures, etc.
RuleML as „Webized“ Language Prefix and Vocabulary Mapping

- RuleML has attributes which act as local or external identifiers and references, e.g., type, iri, node, key, keyref, ...
  - including references to the (Reaction) RuleML vocabulary
- There values can be terms, Curies and relative IRIs which need to be mapped to absolute IRIs
  - to enable Web-based imports, interchange and modularization
- @prefix
  - attribute defining a list of prefixes for the mapping of Compact URIs (Curies) to Internationalized Resource Identifiers (IRIs) as a white space separated list of prefix-name IRI pairs
- @vocab
  - an Internationalized Resource Identifier (IRI) that defines the vocabulary to be used when a term is referenced

```
<Var vocab="http://www.w3.org/2001/XMLSchema" type="date"/>
```
Example - Descriptive and Qualifying Metadata

Descriptive Metadata

<Assert key="#module1">
  <meta> <!-- descriptive metadata -->
    <Atom><Rel iri="dc:creator"/> <Ind> Adrian Paschke </Ind> </arg> </Atom>
</meta>
<meta>
  <Time type="&ruleml;TimeInstant"> <Data xsi:type="xs:date"> 2011-01-01 </Data> </Time>
</meta>
<meta>
  <Atom><Rel>src</Rel> <Ind> ./module1.prova </Ind> </Atom>
</meta>
<qualification> <!-- qualifying metadata -->
  <Atom> <!-- the module is valid for one year from 2011 to 2012 -->
    <Rel>valid</Rel>
    <Interval type="&ruleml;TimeInstant">
      <Time type="&ruleml;TimeInstant"> <Data xsi:type="xs:date"> 2011-01-01 </Data> </Time>
      <Time type="&ruleml;TimeInstant"> <Data xsi:type="xs:date"> 2012-01-01 </Data> </Time>
    </Interval>
  </Atom>
</qualification>

Qualifying Metadata

<Rulebase key="#innermodule1.1">
  <meta> ... </meta> <qualification> ... </qualification>
</Rulebase>

<Rule key="#rule1">
  <meta> ... </meta> <qualification> ... </qualification>
</Rule>
<Rule key="#rule2">
  <meta> ... </meta> <qualification> ... </qualification>
</Rule>
</Rulebase>
</Assert>
Sorted RuleML - RuleML with Types

- **Types** can be assigned using the @type attribute

```xml
<Var type="car:Vehicle">Car</Var>
<Ind type="car:Sedan">2000 Toyota Corolla</Ind>
<Data xsi:type="xs:integer">12</Data>
```
Example: Typed Variant

E.g. Variables with attribute `type`, whose values are IRIs pointing to local or externally define sorts / types, e.g., ontological class definitions on the Web specified in RDFS and OWL

```xml
<Implies>
  <then>
    <Atom>
      <Rel>discount</Rel>
      <Var type="http://xmlns.com/foaf/spec/#term_Person">cust</Var>
      <Var type="http://daml.org/.../ProfileHierarchy.owl#Product">prod</Var>
      <Data>5.0 percent</Data>
    </Atom>
  </then>
  <if> . . . </if>
</Implies>
```
Signatures – Defining Sorts / Types

- Local types/sorts info can be locally specified using statements like this (called **signatures**):

  - **Predicate/function signature**: 
    
    $oid = \{\text{signature pattern(s)}\}$
    
    `person = \{ person(name\?).\}`
    
    `spouse = \{ spouse(X\?, Y\?).\}`
    
    `children = \{ child(X\?, Y\?).\}`

  - **Frame signature** 
    
    $oid = \{\text{frame_signature pattern(s)}\}$
    
    `Person = \{ person[name *=> string, spouse *=> person, children *=>>> person]. \}`

- The notion of **well-typedness** relates signatures to data
Signature Pattern for Sort Definition

- A sort $T$ is declared by a signature pattern $T = \{ T_1, \ldots, T_n \}$ in the sort's signature definition.

Example

\[
\begin{align*}
\text{Event} &= \langle \text{Term} = \{\text{Constant} \} \rangle \\
\text{Time} &= \langle \text{Term} = \{\text{Constant} \} \rangle \\
\text{PositionalBinaryPredicate} &= \langle \text{PositionalPredicate} = \{\text{Term, Term} \}, \text{arity} = 2 \rangle \\
\text{Happens} &= \langle \text{PositionalBinaryPredicate} = \{\text{Event, Time} \} \rangle
\end{align*}
\]

- Non-Polymorphic Classical Dialects
  - Unique signature name per sort definition

- Dialects with Polymorphism
  - multiple (to countable infinite) different signatures with the same signature name
Example – Signatures in (Reaction) RuleML

- Predicate Signature Example:

```xml
<signature>
  <Atom arity="2" scope="local">
    <!-- arity of 2 and local scope -->
    <oid><Ind>likes</Ind></oid> <!—predicate sort/type oid -->
    <Rel>likes</Rel>
    <arg><Var mode="+"/></arg> <!— mode=+ i.e. input argument -->
    <arg><Var mode="?"/></arg> <!— input or output argument -->
  </Atom>
</signature>
```

- Frame Signature Example:

```xml
<signature>
  <Atom scope="global">
    <oid><Ind>person</Ind></oid> <!— frame type oid -->
    <slot><Ind>likes</Ind><Var type="person"/></slot>
  </Atom>
</signature>
```
Example User-defined Signature Definition

```
<signature>
  <Time>
    <oid><Ind>Datetime</Ind></oid>
    <slot><Ind>date</Ind><Var type="xs:date"/></slot>
    <slot><Ind>time</Ind><Var type="xs:time"/></slot>
  </Time>
</signature>
```

```
<signature>
  <Event>
    <oid><Ind>TimeEvent</Ind></oid>
    <slot><Ind>event</Ind><Var type="xs:string"></Var></slot>
    <slot><Ind>time</Ind><Var type="Datet ime"></Var></slot>
  </Event>
</signature>
```
Example - Typed Complex Event **Pattern** Definition

```xml
<Event key="#ce2" type="&ruleml;ComplexEvent">
  <signature> <!-- pattern signature definition -->
    <Sequence>
      <signature>
        <Event type="&ruleml;SimpleEvent">
          <signature>
            <Event>...event_A...</Event>
          </signature>
        </Event>
      </signature>
      <signature>
        <Event type="&ruleml;ComplexEvent" keyref="#ce1"/>
      </signature>
    </Sequence>
  </signature>
</Event>

<Event key="#ce1">
  <signature> <!-- event pattern signature -->
    <Concurrent>
      <Event><meta><Time>...t3</Time></meta><signature>...event_B</signature></Event>
      <Event><meta><Time>...t3</Time></meta><signature>...event_C</signature></Event>
    </Concurrent>
  </signature>
</Event>

<Event key="#e1" keyref="#ce2"><content>...</content></Event>
```
Selected Reaction RuleML Algebra Operators

- **Action Algebra**

- **Event Algebra**
  - *Sequence* (Ordered), *Disjunction* (Or), *Xor* (Mutual Exclusive), *Conjunction* (And), *Concurrent*, *Not*, *Any*, *Aperiodic*, *Periodic*, *AtLeast*, *ATMost*, **Operator** (generic Operator)

- **Interval Algebra (Time/Spatio/Event/Action/… Intervals)**
  - *During*, *Overlaps*, *Starts*, *Precedes*, *Meets*, *Equals*, *Finishes*, **Operator** (generic Operator)

- **Counting Algebra**
  - *Counter*, *AtLeast*, *AtMost*, *Nth*, **Operator** (generic Operator)

- **Temporal operators**
  - *Timer*, *Every*, *After*, *Any*, **Operator** (generic Operator)

- **Negation operators**
Example – Modes and Scopes

<signature>
  <Atom type="ruleml:Happens" arity="2" mode="+" scope="global">
    <Rel>planned</Rel>
    <Var type="ruleml:Event" mode="+"></Var>
    <Var type="ruleml:Time" mode="+"></Var>
  </Atom>
</signature>

global scope

<signature>
  <Event key="#ce1" scope="private">
    <Concurrent>…</Concurrent>
  </Event>
</signature>

private scope

<scope>
  <Atom><oid><Ind>m1</Ind><oid><Rel>source</Rel><Ind>module1</Ind></Atom>
</scope>

user-defined scope

<signature>
  <Atom type="ruleml:Happens" arity="2" mode="+" scope="m1">
    …
  </Atom>
</signature>
F-Logic

- Frame Logic - Object-oriented Frame-Based Logic

- An object-oriented first-order logic
- Extends predicate logic with **Frames**
  - Class hierarchies and inheritance
  - Objects with complex internal structure
  - Typing
  - Encapsulation
- A basis for object-oriented logic programming and knowledge representation

<table>
<thead>
<tr>
<th>O-O programming</th>
<th>Relational programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-logic</td>
<td>Predicate calculus</td>
</tr>
</tbody>
</table>

**Background:**

- Basic theory: [Kifer & Lausen SIGMOD-89], [Kifer, Lausen, Wu JACM-95]
- Path expression syntax: [Frohn, Lausen, Uphoff VLDB-84]
- Semantics for non-monotonic inheritance: [Yang & Kifer, ODBASE 2002]
- Meta-programming + other extensions: [Yang & Kifer, Journal on Data Semantics 2003]
- Object-Oriented RuleML semantics: PSOA Semantics [Boley RuleML 2011]
Example – Frames in (Reaction) RuleML

<Rulebase>

<signature>
  <Atom>
    <oid><Ind>person</Ind></oid>
    <slot><Ind>likes</Ind><Var type="person"/></slot>
  </Atom>
</signature>

<Atom>
  <oid><Ind type="person">John</Ind></oid>
  <slot><Ind>likes</Ind><Ind type="person">Mary</Ind></slot>
</Atom>

</Rulebase>
A RuleML knowledge base / rule base is a conjunction of clauses that are asserted (&lt;Assert&gt;) to be true

&lt;Assert&gt;
&lt;Rulebase&gt;
&lt;!-- Clauses (Facts and Rules) here --&gt;
&lt;/Rulebase&gt;
&lt;/Assert&gt;
RuleML – Queries

- Queries are represented with `<Query>` tag
- Contains one child to represent the body of the query - this is an `<Atom>` or a conjunction of `<Atom>`s in an `<And>`
- Example: `premium(Who)?`

```xml
<Query>
  <Atom>
    <Rel>premium</Rel>
    <Var>who</Var>
  </Atom>
</Query>
```
RuleML – Performativs / Actions

- Performatives are knowledge base actions:
  
  `<Assert>, <Retract>, <Update>, <Query>, <Answer>, <Test>, <Send>, <Receive>, <Consult>, and <Action>`

- Imported knowledge ( `<Consult>` action) and knowledge actions ( `<Assert>, <Retract>, <Update>, <Action>`) are added as modules

- Examples:

  `<Retract scope="module1"/>`  Retract module with scope “module1”
  `<Query scope="module1">…`  Query module with scope "module1"
  `<on><Event scope="module1">…`  Detect event within scope
Knowledge Modularization

- (Local) knowledge distribution
  - *key-keyref* attributes
- Knowledge imports
  - *Xinclude* (syntactic) and *Consult* (semantic) import action
- Knowledge modules
  - *Modes* partition the universe into subsets with input (+), output (-), open mode (?)
  - *Scopes* define subsets of the universe as modules with scoped interpretation: *global*, *local*, *private*, dynamic scopes
Example - Imports

1. (Locally) Include external knowledge base

```xml
<xi:include href="../../kb1.rrml"/>
...
```

2. Consult (import action) importing external knowledge base

```xml
<do>
  <Consult iri="../../kb1.rrml"/>
</do>
```

3. Consult enclosed knowledge base

```xml
<Consult>
  <payload>
    <RuleML>
      ...
    </RuleML>
  </payload>
  <enclosed>
    <Message>
      ...
    </Message>
  </enclosed>
</Consult>
```
Example – Truth Degree Valuation in Answers

<Answer>
  <degree><Data>1</Data></degree>
  ...
</Answer>

(Truth degree 1 = true)

<Answer>
  <degree><Data>0</Data></degree>
  ...
</Answer>

(Truth degree 0 = false)

<Answer>
  <degree><Data>0.5</Data></degree>
  ...
</Answer>

(Truth degree 0.5 = undefined)
- cid is the conversation identifier (enabling also subconversations)
- protocol: protocol definition (high-level protocols and transport prot.)
- agent (send, receiver): denotes the target or sender agent of the message
- directive: pragmatic context directive, e.g. FIPA ACL primitives
- payload: Message payload (<content> for arbitrary XML content)
Messaging Reaction Rules

```xml
<Rule>
  ...
  <do><Send><Message> …query1 </Message></Send></do>
  <do><Send><Message> …query2 </Message></Send></do>
  <on><Receive><Message> …response2 </Message></Receive></on>
  <if> prove some conditions, e.g. make decisions on the received answers </if>
    <on><Receive><Message> …response1 </Message></Receive></on>
  ....
</Rule>
```

Note: The „on“, „do“, „if“ parts can be in arbitrary combinations, e.g. to allow for a flexible workflow-style logic with subconversations and parallel branching logic.
Example: Loosley-Coupled Communication via Messages to Agent Interface

- Event Message is local to the **conversation scope** (cid) and **pragmatic context** (directive)
(Reaction) RuleML - Semantic Profiles

- Profiles are represented with a `<Profile>` tag.
  - They define the *intended semantics* for knowledge interpretation, reasoning, execution, …

- A semantic profile (SP) can define, e.g.
  - profile (sub-)signature $S_{SP}$
  - profile language $\Sigma_{SP}$
  - intended interpretation semantics $I_{SP}$
  - axioms $\Phi_{SP}$
  - a semantics preserving translation function into Reaction RuleML $\tau(.)$
Integration of Semantic Profiles

1. Include/Import external Semantic Profile

```xml
<xi:include href="../../profiles/SituationCalculusProfile.rrml" xpointer="xpointer(/RuleML/*)="/>
<evaluation>
  <Profile keyref="&ruleml;ReifiedClassicalSituationCalculus" />
</evaluation>
```

2. Reference pre-defined Semantic Profile as profile type

```xml
<evaluation>
  <Profile type="&rif;RDFS" iri="http://www.w3.org/ns/entailment/RDFS"/>
</evaluation>
```

3. Locally defined Semantic Profile

```xml
<Assert>
  <evaluation>
    <Profile key="&ruleml;ReifiedClassicalSituationCalculus" >
      <formula><Rulebase> ... RuleML definition... </Rulebase></formula>
      <content> ... xs:any XML content, e.g. RIF, Common Logic XML... </content>
    </Profile>
  </evaluation>
  <Rulebase>
    <Rule> ... </Rule>
    <Rule> ... </Rule>
  </Rulebase>
</Assert>
```

Note: also other non RuleML content models are supported.
Example: Use of Semantic LP Profiles for Interpretation

`<-- rule interface with two alternative interpretation semantics and a signature. The interface references the implementation identified by the corresponding key -->`

```
<Rule key="#r1">
  <evaluation index="1">
    <!-- WFS semantic profile -->
    <Profile type="&ruleml;Well-Founded-Semantics" />
  </evaluation>
  <evaluation index="2">
    <!-- alternative ASS semantic profile define in the metamodel -->
    <Profile type="&ruleml;Answer-Set-Semantics" />
  </evaluation>
  <!-- the signature defines the queryable head of the backward-reasoning rule -->
  <signature>
    <Atom><Rel>likes</Rel><Var mode="+"/><Var mode="-"/></Atom>
  </signature>
</Rule>
```

`<-- implementation of rule 1 which is interpreted either by WFS or by ASS semantics and only allows queries according to it’s signature definition. -->`

```
<Rule keyref="#r1" style="reasoning">
  <if>...</if>
  <then>
    <Atom><Rel>likes</Rel><Var>X</Var><Var>Y</Var></Atom>
  </then>
</Rule>
```
Example - Reaction Rules with Semantic Profiles

```xml
<Rule style="active">
  <evaluation>
    <Profile>e.g. selection and consumptions policies</Profile>
  </evaluation>
  <signature>
    <Event key="#ce1">
      … event pattern definition (for event detection)
    </Event>
  </signature>
  <on>
    <Event keyref="#ce1"/>
  </on>
  <if>
    ...
  </if>
  <do>
    <Assert safety="transactional">
      <!-- transactional update -->
      <formula>
        <Atom>
          …
        </Atom>
      </formula>
    </Assert>
  </do>
</Rule>
```

**Interface**
(semantic profile + event pattern signature for event processing / detection)

**Implementation**
(reaction rule triggered by event detection)
Summary RuleML

- RuleML is a **unifying family of rule languages** across all industrially relevant Web rules
  - It supports **untyped and typed logic** terms which might be “webized” or not (in contrast to RIF which is only webized)
    - RuleML uses a general **type attribute** for typed logic terms
  - It supports a general approach towards predefined and user-defined semantics
- It addresses **all rule types** and provides **translators** between sublanguages of RuleML, Reaction RuleML, RIF, PRR, SBVR, Jess, SWRL, Prova (ISO Prolog),…
- It **contributes to relevant standards** such as W3C RIF, OMG PRR, W3C SWRL, OASIS LegalRuleML
RuleML Basis for other Languages - Examples

- Semantic Web Rule Language (SWRL)
  - Uses RuleML Version 0.89
- Semantic Web Services Language (SWSL)
  - Uses RuleML Version 0.89
- W3C Rule Interchange Format
  - Uses RuleML Version 0.91 with frames and slots
- OASIS LegalRuleML
  - Uses RuleML Version 1.0
- OMG PRR and OMG SBVR
  - Input from RuleML
- OMG API4KB
  - Input from Reaction RuleML 1.0
- RuleML RIF-CASPD, RIF-CLPWD, RIF-Rulelog, …
SWRL - A Semantic Web Rule Language Combining OWL and RuleML

- W3C RuleML SWRL Member Submission
- Homogenous combination of RuleML (Datalog Layer) + OWL DL
  - Extends RuleML with ontology axioms:
    
    \[
    C(x), \ P(x,y), \ \text{sameAs}(x,y), \ \text{differentFrom}(x,y), \ \text{builtIn}(r,x,...)
    \]

    \[C = \text{OWL description or data range,} \]
    \[P = \text{OWL property} \]
    \[r = \text{built-in relation,} \]
    \[x \ \text{und} \ y = \text{variable or OWL individuals or OWL data values} \]

  + Homogenous Language
  + Integrated into Ontology tools, e.g. Protégé SWRL Plug-in
  - Undecidable
  - Restricted expressiveness
SWRL

- A Semantic Web Rule Language Combining OWL and RuleML
- SWRL is **undecidable**
- SWRL with the restriction of **DL Safe rules** is decidable
  - Variables in DL Safe rules bind only to explicitly named individuals in the ontology.
SWRL Idea

- Datalog rules which refer to an OWL ontology

\[
\text{mother(?mom1, Ralph)} \land \\
\text{mother(?mom2, Ralph)} \\
\rightarrow \text{owl:sameAs(?mom1, ?mom2)}
\]
SWRL Properties

- Unary / Binary Datalog sublanguages of RuleML
- No disjunction
- No negation
SWRL
Abstract Syntax

rule ::= 'Implies(' [ URIreference ] { annotation } antecedent consequent ')'\nantecedent ::= 'Antecedent(' { atom } ')'\nconsequent ::= 'Consequent(' { atom } ')'
SWRL
Abstract Syntax

atom ::= description '('< i-object '>')'  
      | dataRange '('< d-object '>')'  
      | individualvaluedPropertyID '('< i-object i-object '>')'  
      | datavaluedPropertyID '('< i-object d-object '>')'  
      | sameAs '('< i-object i-object '>')'  
      | differentFrom '('< i-object i-object '>')'  
      | builtin '('< builtinID { d-object } '>')'  

builtinID ::= URIreference
SWRL
Abstract Syntax

An atom can be
- C(x)
- P(x,y)
- sameAs(x,y)
- differentFrom(x,y)
- builtin(r,x,...)
SWRL
Direct Model-Theoretic Semantics

• straightforward extension of the semantics for OWL

<table>
<thead>
<tr>
<th>Atom</th>
<th>Condition on Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C(x)</td>
<td>S(x) ∈ EC(C)</td>
</tr>
<tr>
<td>D(z)</td>
<td>S(z) ∈ EC(D)</td>
</tr>
<tr>
<td>P(x,y)</td>
<td>&lt;S(x),S(y)&gt; ∈ ER(P)</td>
</tr>
<tr>
<td>Q(x,z)</td>
<td>&lt;S(x),L(z)&gt; ∈ ER(Q)</td>
</tr>
<tr>
<td>sameAs(x,y)</td>
<td>S(x) = S(y)</td>
</tr>
<tr>
<td>differentFrom(x,y)</td>
<td>S(x) ≠ S(y)</td>
</tr>
<tr>
<td>builtin(r,z1,...,zn)</td>
<td>&lt;S(z1),...,S(zn)&gt; ∈ D(f)</td>
</tr>
</tbody>
</table>
SWRL:
Several Concrete Syntaxes

- XML concrete syntax (RuleML)

```xml
<ruleml:imp>
  <ruleml:_rlab ruleml:href="#example1"/>
  <ruleml:_body>
    <swrlx:individualPropertyAtom swrlx:property="hasParent">
      <ruleml:var>x1</ruleml:var>
      <ruleml:var>x2</ruleml:var>
    </swrlx:individualPropertyAtom>
    <swrlx:individualPropertyAtom swrlx:property="hasBrother">
      <ruleml:var>x2</ruleml:var>
      <ruleml:var>x3</ruleml:var>
    </swrlx:individualPropertyAtom>
  </ruleml:_body>
  <ruleml:_head>
    <swrlx:individualPropertyAtom swrlx:property="hasUncle">
      <ruleml:var>x1</ruleml:var>
      <ruleml:var>x3</ruleml:var>
    </swrlx:individualPropertyAtom>
  </ruleml:_head>
</ruleml:imp>
```
SWRL: Several Concrete Syntaxes

- Prolog-like syntax

\[
\text{hasParent}(\text{x}_1, \text{x}_2) \land \text{hasBrother}(\text{x}_2, \text{x}_3) \implies \text{hasUncle}(\text{x}_1, \text{x}_3)
\]
SWRL: Several Concrete Syntaxes

- Prolog-like syntax

\[
\text{hasParent}(\text{?x1, ?x2}) \land \text{hasBrother}(\text{?x2, ?x3}) \rightarrow \text{hasUncle}(\text{?x1, ?x3})
\]
Protégé SWRL Tab Plugin

- Protégé 3
- OWL1
- Jess Rule Engine
Protégé SWRL Tab Plugin

- Protégé 4
- OWL 2
- Rudimentary Rule Editor
- No Rule Engine
Protégé SWRL Tab Plugin

- Protégé 5
- OWL 2
- The SWRL Tab is back!
- Drools Rule Engine (JBoss)
Protégé SWRL Tab Plugin: Control View

OWL axioms successfully transferred to rule engine.
Number of SWRL rules exported to rule engine: 4
Number of OWL class declarations exported to rule engine: 4
Number of OWL individual declarations exported to rule engine: 14
Number of OWL object property declarations exported to rule engine: 3
Number of OWL data property declarations exported to rule engine: 6
Total number of OWL axioms exported to rule engine: 139
The transfer took 148 milliseconds.
Press the 'Run Drools' button to run the rule engine.

No Reasoner set. Select a reasoner from the Reasoner menu  

Show Inferences
Protégé SWRL Tab Plugin: Rule Editor

Rule

Square Roots

Comment

Determines if a number a is square root of a number b and sets the squareRootOf property accordingly.

Status

Expecting variable or OWL individual name

\[
\text{NaturalNumber}(a) \land \text{NaturalNumber}(b) \land \text{hasValue}(a, x) \land \text{hasValue}(b, y) \land \text{swrlb:multiply}(y, x, x) \rightarrow \text{squareRootOf}(a),
\]
Protégé SWRL Tab Plugin: Loaded Rules View
Protégé SWRL Tab Plugin: Asserted Axioms View
Protégé SWRL Tab Plugin:
Asserted Axioms View
SWRL Built-Ins

- Allow for extensions to the SWRL language.
- Provide arithmetics beyond DL/Prolog (e.g. String arithmetics).
- Still no clearly defined formal semantics.
- Can only appear in the body of a rule.
SWRL Built-Ins: Comparison

- `swrlb:equal`, `swrlb:notEqual`
- `swrlb:lessThan`, `swrlb:lessThanOrEqual`, ...

\[
\text{Person}(?p1) \land \text{Person}(?p2) \land \\
\text{uniqueID}(?p1, ?id1) \land \\
\text{uniqueID}(?p2, ?id2) \land \\
\text{swrlb:equal}(?id1, ?id2) \\
\rightarrow \text{owl:sameAs}(?p1, ?p2)
\]
SWRL Built-Ins: Comparison

- swrlb:equal, swrlb:notEqual
- **swrlb:lessThan**, swrlb:lessThanOrEqual, ...

```
Person(?p1) ∧ Person(?p2) ∧
birthdate(?p1, ?bd1) ∧
birthdate(?p2, ?bd2) ∧
swrlb:lessThan(?bd1, ?bd2)
→ younger(?p1, ?p2)
```
SWRL Built-Ins: Comparison

- `swrlb:equal`, `swrlb:notEqual`
- `swrlb:lessThan`, `swrlb:lessThanOrEqual`, ...

- Works with
  - Numericals
  - Strings
  - Dates and Times
  - Durations
SWRL Built-Ins: Mathematics

- `swrlb:add`, `swrlb:subtract`
- `swrlb:multiply`, `swrlb:divide`, `swrlb:integerDivide`, ...

```
Number(?a) ∧ value(?a, ?i) ∧
Number(?b) ∧ value(?b, ?k) ∧
swrlb:add(?k, ?i, ?i)
→ doubleOf(?b, ?a)
```
SWRL Built-Ins: Mathematics

- `swrlb:add`, `swrlb:subtract`
- `swrlb:multiply`, `swrlb:divide`, `swrlb:integerDivide`, ...

Item(?i) \(\land\) net_price(?i, ?np) \(\land\) 
\[
\text{swrlb:multiply}(?gp, ?np, 1.19)
\]
\(\rightarrow\) gross_price(?i, ?gp)
SWRL Built-Ins: Mathematics

- `swrlb:mod`, `swrlb:pow`
- `swrlb:unaryPlus`, `swrlb:unaryMinus`, ...

\[
\text{Person}(?p) \land \text{credit}(?p, ?c) \land \text{swrlwb:unaryMinus}(?d, ?c) \\
\rightarrow \text{debt}(?p, d)
\]
SWRL Built-Ins: Mathematics

- `swrlb:abs`, `swrlb:ceiling`
- `swrlb:floor`, `swrlb:round`, `swrlb:roundHalfToEven`
- `swrlb:sin`, `swrlb:cos`, `swrlb:tan`

Price(?price) ∧ amount(?price, ?a) ∧
\[ \text{swrlb:abs}(\text{?euros, ?a}) \land \text{swrlb:subtract}(\text{?cents, ?a, ?euros}) \land \text{amount_euros}(\text{?price, ?euros}) \]
→ amount_cents(?price, ?cents)
SWRL Built-Ins: Booleans

- `swrlb:booleanNot`

```sparql
swrlb:booleanNot(?x, ?x) → Frozen(Hell)
```
SWRL Built-Ins: Strings

- `swrlb:stringEqualIgnoreCase`, `swrlb:stringConcat`
- `swrlb:stringLength`, ...

Person(?p) ∧ first_name(?p, ?fn) ∧ last_name(?p, ?ln) ∧ `swrlb:stringConcat(?n, ?fn, " ", ?ln)` → full_name(?p, ?n)
SWRL Built-Ins: Strings

- `swrlb:stringEqualIgnoreCase`, `swrlb:stringConcat`
- `swrlb:stringLength`, ...

```
Person(?p) ∧ says(?p, ?x) ∧
  swrlb:stringLength(?l, ?x) ∧
  swrlb:greaterThan(?l, 5,000)
→ Verbose (?p)
```
SWRL Built-Ins: Strings

- swrlb:normalizeSpace
- swrlb:upperCase, swrlb:lowerCase
- swrlb:translate
- swrlb:contains, swrlb:containsIgnoreCase
- swrlb:startsWith, swrlb:endsWith
- swrlb:substringBefore, swrlb:substringAfter
- swrlb:matches, swrlb:replace
- swrlb:tokenize
SWRL Built-Ins: Date/Time Arithmetics

- `swrlb:yearMonthDuration`, `swrlb:dayTimeDuration`
- `swrlb:dateTime`, `swrlb:date`, `swrlb:time`

\[\begin{align*}
\text{hasAppointment}(?x, \ ?date) \\
\land \ swrlb:date(?date, \ ?year, \ ?month, \ ?day, \ ?timezone) \to \\
\text{hasAppointmentOnDay}(?x, \ ?day)
\end{align*}\]
SWRL Built-Ins: Date/Time Arithmetics

- `swrlb:addYearMonthDurations`, `swrlb:subtractYearMonthDurations`, `swrlb:multiplyYearMonthDuration`, `swrlb:divideYearMonthDuration`
- `swrlb:addDayTimeDurations`, `swrlb:subtractDayTimeDurations`, `swrlb:multiplyDayTimeDuration`, `swrlb:divideDayTimeDuration`
SWRL Built-Ins: Date/Time Arithmetics

- `swrlb:subtractDates`, `swrlb:subtractTimes`
- `swrlb:addYearMonthDurationToDateTime`,
  `swrlb:addDayTimeDurationToDateTime`,
  `swrlb:subtractYearMonthDurationFromDateTime`,
- `swrlb:subtractDayTimeDurationFromDateTime`
- `swrlb:addYearMonthDurationToDate`,
  `swrlb:addDayTimeDurationToDate`,
- `swrlb:subtractYearMonthDurationFromDate`,
  `swrlb:subtractDayTimeDurationFromDate`,
- `swrlb:addDayTimeDurationToTime`, ...
SWRL Built-Ins: URIs

- `swrlb:resolveURI`, `swrlb:anyURI`

```
→ hasURI(SWAT4LS, ?uri)
```
SWRL Built-Ins: Lists

- `swrlb:listConcat`
- `swrlb:listIntersection`, `swrlb:listSubtraction`
- `swrlb:member`
- `swrlb:length`
- `swrlb:first`, `swrlb:rest`
- `swrlb:sublist`
- `swrlb:empty`
SWRL Built-Ins: Lists

- `swrlb:equal`, `swrlb:notEqual`
- `swrlb:lessThan`, `swrlb:lessThanOrEqual`, ...

\[
\text{Person}(?p1) \land \text{Person}(?p2) \land \\
\text{uniqueID}(?p1, ?id1) \land \\
\text{uniqueID}(?p2, ?id2) \land \\
\text{swrlb:equals}(?id1, ?id2) \\
\rightarrow \text{owl:sameAs}(?p1, ?p2)
\]
Further Reading

- http://www.w3.org/Submission/SWRL/
- https://github.com/protegeproject/swrlapi-drools-engine/wiki/SWRLDroolsTab
RuleML Basis for other Languages - Examples

- Semantic Web Rule Language (SWRL)
  - Uses RuleML Version 0.89
- Semantic Web Services Language (SWSL)
  - Uses RuleML Version 0.89
- W3C Rule Interchange Format
  - Uses RuleML Version 0.91 with frames and slots
- OASIS LegalRuleML
  - Uses RuleML Version 1.0
- OMG PRR
  - Input from RuleML
- OMG API4KB
  - Input from Reaction RuleML 1.0
- RIF-CASPD, RIF-CLPWD, RIF-Rulelog, …
The current W3C Semantic Web Architecture - Rules

W3C Semantic Web Stack since 2007
W3C RIF Dialects

[http://www.w3.org/TR/rif-ucr/]
RIF-FLD (Framework of Logic Dialects)
- framework of mechanisms for specifying the syntax and semantics of logic-based RIF dialects through a number of generic concepts
RIF-PRD (Production Rules Dialect)
- Dialect for Production Rules
RIF-BLD (Basic Logic Dialect)
RIF-Core (shared subset between BLD and PRD)
RIF-SWC (RDF and OWL Compatibility)
- Interoperability between RIF and Semantic Web Data- and Ontology Languages (RDF, RDFS, OWL).
RIF-DTB (Data Types and Builtins)
- document describes RIF data types and built-in functions and predicates
RIF-UCR (Use Cases and Requirements)
RIF-Test (Test Cases)
RIF OWL 2 RL RIF XML Data Integration
W3C RIF Basic Logic Dialect

- **Definite Horn rules** with equality + a number of syntactic extensions
  - Slots, frames, internationalized resource identifiers (IRIs) as identifiers for concepts, XML Schema data types.

- **Syntaxes**
  - **Normative XML syntax** (closely related to striped RuleML XML syntax)
  - Presentation syntax + abridged presentation syntax
RIF a “Webized” Rule Language

- Symbols used to identify constants, variables, functions, predicates

- "literal"^^<symspace-identifier>
  - Datatype constant: “Adrian”^^<xsd:string>
  - Web entity: “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”^^<rif:iri>
  - Local constant: “PersonX”^^<rif:local>
### RIF BLD XML Syntax Examples

<table>
<thead>
<tr>
<th>RIF Condition</th>
<th>XML serialization close to RuleML syntax</th>
</tr>
</thead>
</table>
  <formula><Atom><op><Const type="&rif;iri">&cpt;purchase</Const></op>  
   <args ordered="yes"><Var>Buyer</Var> <Var>Seller</Var>  
    <Expr><op><Const type="&rif;iri">&cpt;book</Const></op>  
     <args ordered="yes">  
      <Var>Author</Var>  
      <Const type="&rif;iri">&bks;LeRif</Const></Expr>  
   </Expr>  
  </args>  
  <Expr><op><Const type="&rif;iri">&curr;USD</Const></op>  
   <ARGS ordered="yes">49</ARGS></Expr>  
</Atom></formula>  
</Exists></formula>  
<formula><Equal>  
  <SIDE><Var>Seller</Var></SIDE>  
  <SIDE><Var>Author</Var></SIDE>  
</Equal></formula>` |
W3C RIF Production Rules Dialect

- Production Rules (Condition-Action)
  - Based on RIF condition language (from RIF Core)
  - Actions: Retract, Assert, Modify, Execute
  - Negation (inflationary negation)

- Syntax
  - **Normative XML syntax**
  - Non-normative EBNF-based presentation syntax
    - for presentation purposes
<Implies>
  <if> FORMULA </if>
  <then rif:ordered="yes">
    <Do>
      ACTION*
    </Do>
  </then>
</Implies>
RIF Semantic Web Compatibility

- RIF-SWC specifies the interoperation between RIF and the data and ontology languages RDF, RDFS, and OWL.

RDF triples
- `ex:john ex:brotherOf ex:jack`
- `ex:jack ex:parentOf ex:mary`

RIF Rule

- **Importing RDF Graphs in RIF**
- `Import(<IRI> <profile>)`
**Shared RIF / OWL 2 RL**

- based on Description Logic Programs fragment [DLP] and pD*
- is a syntactic profile of OWL 2 DL and RIF (Second Edition).
- allows for scalable reasoning using rule-based technologies.
- trades the full expressivity of the language for efficiency

http://www.w3.org/2007/OWL/wiki/Profiles#OWL_2_RL
**OWL 2 RL**

• achieved by restricting the use of OWL 2 constructs to certain syntactic positions.

• Table 1. Syntactic Restriction on Class Expressions in SubClassOf Axioms

<table>
<thead>
<tr>
<th>Subclass Expressions</th>
<th>Superclass Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>a class</td>
<td>a class</td>
</tr>
<tr>
<td>a nominal class (OneOf)</td>
<td>intersection of classes (ObjectIntersectionOf)</td>
</tr>
<tr>
<td>intersection of class expressions (ObjectIntersectionOf)</td>
<td>universal quantification to a class expressions (ObjectAllValuesFrom)</td>
</tr>
<tr>
<td>union of class expressions (ObjectUnionOf)</td>
<td>at-most 1 cardinality restrictions (ObjectMaxCardinality 1)</td>
</tr>
<tr>
<td>existential quantification to a class expressions (ObjectSomeValuesFrom)</td>
<td>existential quantification to an individual (ObjectHasValue)</td>
</tr>
<tr>
<td>existential quantification to an individual (ObjectHasValue)</td>
<td></td>
</tr>
</tbody>
</table>
Comparison RIF/RuleML/Prova Prolog (1)

<table>
<thead>
<tr>
<th>RIF</th>
<th>RuleML</th>
<th>Prova</th>
</tr>
</thead>
<tbody>
<tr>
<td>`&lt;Const type=&quot;&amp;xs;string&quot;&gt;</td>
<td>`&lt;Data xsi:type=&quot;xs:string&quot;&gt;</td>
<td>&quot;ABC&quot;</td>
</tr>
<tr>
<td>ABC</td>
<td>ABC</td>
<td></td>
</tr>
<tr>
<td><code>&lt;/Const&gt;</code></td>
<td><code>&lt;/Data&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Var&gt;x&lt;/Var&gt;</code></td>
<td><code>&lt;Var&gt;x&lt;/Var&gt;</code></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Expr&gt;</code></td>
<td><code>&lt;Expr&gt;</code></td>
<td>func:f(X)</td>
</tr>
<tr>
<td>`&lt;op&gt;&lt;Const type=&quot;&amp;rif;iri&quot;&gt;</td>
<td>`&lt;Fun iri=&quot;func:f&quot;, per=&quot;value&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&amp;func;f</td>
<td><code>&lt;/Var&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;/Const&gt;&lt;/op&gt;</code></td>
<td><code>&lt;/Var&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;args ordered=&quot;yes&quot;&gt;</code></td>
<td><code>&lt;Expr&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Var&gt;X&lt;/Var&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;/args&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;/Expr&gt;</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Comparison RIF/RuleML/Prova Prolog (2)

<table>
<thead>
<tr>
<th>RIF</th>
<th>RuleML</th>
<th>Prova</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;Atom&gt;</code>&lt;br/&gt; <code>&lt;op&gt;</code>&lt;br/&gt; <code>&lt;Const type=&quot;&amp;rif;iri&quot;&gt;</code>&lt;br/&gt; &amp;cpt;discount&lt;br/&gt; <code>&lt;/Const&gt;</code>&lt;br/&gt; <code>&lt;/op&gt;</code>&lt;br/&gt; <code>&lt;args ordered=&quot;yes&quot;&gt;</code>&lt;br/&gt; <code>&lt;Var&gt;</code>cust<code>&lt;/Var&gt;</code>&lt;br/&gt; <code>&lt;Var&gt;</code>prod<code>&lt;/Var&gt;</code>&lt;br/&gt; <code>&lt;Var&gt;</code>val<code>&lt;/Var&gt;</code>&lt;br/&gt; <code>&lt;/args&gt;</code>&lt;br/&gt; <code>&lt;/Atom&gt;</code></td>
<td><code>&lt;Atom&gt;</code>&lt;br/&gt; <code>&lt;Rel iri=&quot;cpt:discount&quot;/&gt;</code>&lt;br/&gt; <code>&lt;Var&gt;</code>cust<code>&lt;/Var&gt;</code>&lt;br/&gt; <code>&lt;Var&gt;</code>prod<code>&lt;/Var&gt;</code>&lt;br/&gt; <code>&lt;Var&gt;</code>val<code>&lt;/Var&gt;</code>&lt;br/&gt; <code>&lt;/Atom&gt;</code></td>
<td><code>cpt:discount(Cust,Prod,Val)</code></td>
</tr>
<tr>
<td><code>&lt;Equal&gt;</code>&lt;br/&gt; <code>&lt;left&gt;</code>&lt;br/&gt; <code>&lt;Var&gt;</code>X<code>&lt;/Var&gt;</code>&lt;br/&gt; <code>&lt;right&gt;</code>&lt;br/&gt; <code>&lt;Var&gt;</code>Y<code>&lt;/Var&gt;</code>&lt;br/&gt; <code>&lt;/Equal&gt;</code></td>
<td><code>&lt;Equal oriented=&quot;yes&quot;&gt;</code>&lt;br/&gt; <code>&lt;Var&gt;</code>X<code>&lt;/Var&gt;</code>&lt;br/&gt; <code>&lt;Var&gt;</code>Y<code>&lt;/Var&gt;</code>&lt;br/&gt; <code>&lt;/Equal&gt;</code></td>
<td><code>X=Y</code></td>
</tr>
<tr>
<td>RIF</td>
<td>RuleML</td>
<td>Prova</td>
</tr>
<tr>
<td>--------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td><code>&lt;Member&gt;</code></td>
<td></td>
<td>ppl:Adrian^nppl:Person</td>
</tr>
<tr>
<td><code>&lt;instance&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Const type=&quot;&amp;rif;iri&quot;&gt;&amp;ppl;Adrian&lt;/Const&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;class&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Const type=&quot;&amp;rif;iri&quot;&gt;&amp;ppl;Person&lt;/Const&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Member&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Atom&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;op&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Const type=&quot;&amp;rif;iri&quot;&gt;&amp;ex:gold&lt;/Const&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;slot&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Name&gt;customer&lt;/Name&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Var&gt;Customer&lt;/Var&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;slot&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Atom&gt;</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Comparison RIF/RuleML/Prova Prolog (3)
Unifying Logic

• Not standardized in W3C Semantic Web Stack yet
  • **Which semantics?** (e.g., Description Logics, F-Logic, Horn Logic, CL ...)
  • **Which assumptions?** (e.g., Closed World, Open World)
  • How to address non-monotonicity, extra logical features, meta reasoning, ...

W3C Semantic Web Stack since 2007
Summary – Intro to Rule Technologies

- Rule-based Knowledge Representation
  - Goal: derive conclusions and reactions
- Rule types and languages
  - deliberation and derivation rules, reaction rules, transformation rules, constraints
- Platform-specific rule languages
  - Example: Prova: Prolog + Java
- Platform-independent Rule Markup Languages
  - Example: RuleML, W3C SWRL, W3C RIF, …
Thank you ...

Questions?
Further Reading – Rules and Logic Programming, Prova


- Prova Rule Engine [http://www.prova.ws/](http://www.prova.ws/)

- **Prova 3 Semantic Web Branch**
  - Prova 3 version with Semantic Web support on GitHub ([https://github.com/prova/prova/tree/prova3-sw](https://github.com/prova/prova/tree/prova3-sw))


- Prova CEP examples: [http://www.slideshare.net/isvana/epts-debs2012-event-processing-reference-architecture-design-patterns-v204b](http://www.slideshare.net/isvana/epts-debs2012-event-processing-reference-architecture-design-patterns-v204b)
RuleML Online Community

- RuleML MediaWiki (http://wiki.ruleml.org)
- RuleML Blog (http://blog.ruleml.org)
- Mailing lists (http://ruleml.org/mailman/listinfo)
- Technical Groups
  (http://wiki.ruleml.org/index.php/Organizational_Structure#Technical_Groups)
  - Uncertainty Reasoning
  - Defeasible Logic
  - Reaction Rules
  - Multi-Agent Systems
  - Modal Logic …

- RuleML sources are hosted on Github (https://github.com/RuleML)
- W3C RIF http://www.w3.org/standards/techs/rif#w3c_all
- W3C SWRL http://www.w3.org/Submission/SWRL/
Further Reading – RuleML and Reaction RuleML

- Adrian Paschke: Reaction RuleML 1.0 for Rules, Events and Actions in Semantic Complex Event Processing, Proceedings of the 8th International Web Rule Symposium (RuleML 2014), Springer LNCS, Prague, Czech Republic, August, 18-20, 2014
  http://dx.doi.org/10.1007/978-3-642-16289-3_15
- Adrian Paschke, Harold Boley, Zhili Zhao, Kia Teymourian and Tara Athan: Reaction RuleML 1.0: Standardized Semantic Reaction Rules, 6th International Conference on Rules (RuleML 2012), Montpellier, France, August 27-31, 2012
  http://link.springer.com/chapter/10.1007%2F978-3-642-32689-9_9
  http://www.slideshare.net/swadpasc/reaction-ruleml-ruleml2012paschketutorial
  http://www.igi-global.com/book/handbook-research-emerging-rule-based/465
  http://www.igi-global.com/book/handbook-research-emerging-rule-based/465
- Adrian Paschke and Harold Boley: Rule Responder: Rule-Based Agents for the Semantic-Pragmatic Web, in Special Issue on Intelligent Distributed Computing in International Journal on Artificial Intelligence Tools (IJAIT), Vol. 20,6, 2011
Further Reading – Surveys and Tutorials


- Jon Riecke, Opher Etzion, François Bry, Michael Eckert, Adrian Paschke, Event Processing Languages, Tutorial at 3rd ACM International Conference on Distributed Event-Based Systems. July 6-9, 2009 - Nashville, TN

  [http://www.igi-global.com/chapter/rule-markup-languages-semantic-web/35852](http://www.igi-global.com/chapter/rule-markup-languages-semantic-web/35852)