Semantic Web

Rules in Semantic Web

Morteza Amini
Outline

- Rules and Their Usage in Web
- RuleML
- SWRL
- Existing Rule Engines
Outline

- **Rules and Their Usage in Web**
  - RuleML
  - SWRL
  - Existing Rule Engines
Introduction

- Rules are being used for many interconnected purposes, capturing regularities in application domains such as the following:
  - **Engineering:** Diagnosis rules
  - **Commerce:** Business rules (including XML versions such as the Business Rules Markup Language (BRML) of IBM's Business Rules for Electronic Commerce project)
  - **Law:** Legal reasoning (Robert Kowalski and Marek Sergot have been formalizing legal rules in an Imperial College group)
  - **Internet:** Access authentication (Tim Berners-Lee proposed registration engines that use authentication rules such as the following: Any person who was some time in the last 2 months an employee of an organization which was some time in the last 2 months a W3C member may register.)
Expert Systems...

- Are in a branch of artificial intelligence.
- Simulate human reasoning in some domain.
- “Reason” by heuristic or approximate methods.
- Explain and justify solutions in user-friendly terms.
Rule-Based Expert Systems

- Originated from AI research in the 70s and 80s.
- Problem data stored as facts.
- “Reason” using IF…THEN…ELSE rules.
- Can “reason” forward-chaining or backward-chaining.
When to Use Rule-Based Systems

- **Problem Domain** = narrow, well-understood domain theory
- **Knowledge Representation** = facts and rules
- **Output** = recommendation
- **Explanation** = rule firing trace
- **Learning Ability** = generally no
Inference Process

1. Rules and facts compared using **pattern matcher**.

2. Matched rules activated into a **conflict set**.

3. Conflict set resolved into agenda (process called **conflict resolution**).

4. Rule engine **fires** on agenda.

5. Engine cycles until **all** rules are satisfied.
Example rules

- The discount for a customer is 5.0 percent if the customer is premium and the product is regular.

- A customer is premium if their spending has been min 5000 euro in the previous year.

- Those who are members of CE can access CE wireless network.
**Rules in SW**

Emerging Standards pioneered in DARPA Agent Markup Language (DAML) program: e.g.
- RuleML
- OWL/DAML+OIL

- Trust
  - Proof
  - Logic framework
  - Rules
  - Signature
  - Encryption

- Ontology
- RDF Schema
- RDF M&S

- XML
- Namespaces
- URI
- Unicode
Outline

- Rules and Their Usage in Web
  - RuleML
  - SWRL
- Existing Rule Engines
RuleML Initiatives

- **Mission:** Enable semantic exchange of rules/facts between most commercially important rule systems.

- Standards specification: 1st version 2001; basic now fairly stable.

- Current Version is 1.0 published in 2012/April/03.

- A number of tools (~12 engines, translators, editors), demo applications.

- Has now a “home” (www.ruleml.org).

- Initial Core: Horn Logic Programs KR …Webized (in markup)… and with expressive extensions.
Logical Basis of RuleML

- The **Datalog** (constructor-function-free) sublanguage of Horn logic is the foundation for the kernel of **RuleML**.

- Datalog is the language in the intersection of **SQL** and **Prolog**.

- In Datalog, we can define
  - **facts** corresponding to explicit rows of relational tables,
  - **rules** corresponding to tables defined implicitly by views.
Datalog

- Datalog is a query and rule language for deductive databases that syntactically is a subset of Prolog.
- Datalog is a function-free Horn clause form logic such that
  - Disallows using functions in arguments of predicates.
  - Imposes certain restrictions on the use of negation and recursion.
  - Only allows range-restricted variables (preserving safety condition), i.e., each variable in the conclusion of a rule must also appear in a clause in the premise of the rule.
- Horn clause: \( u \leftarrow (p \land q \land \ldots \land r) \)
RuleML Top-Level Hierarchy

- **Deliberation** rules for inference and **Reaction** rules for (re)action.

- In **Derivation** rules (as a sublanguage of FOL) when certain conditions (premises) are fulfilled, **conclusion** is added or asserted.

- **Facts** are considered as special derivation rules that have an empty (hence, 'true') conjunction of premises.

- **Integrity Constraints** are conclusion-less derivation rules.
  - In some references, they are considered as "denials" or special reaction rules whose only possible kind of action is to signal inconsistency when certain conditions are fulfilled.
Application Direction

- General reaction rules can only be applied in the forward direction in a natural fashion, observing/checking events/conditions and performing an action if and when all events/conditions have been perceived/fulfilled.

- Integrity constraints are usually also forward-oriented, i.e. triggered by updates, mainly for efficiency reasons.

- Derivation rules, on the other hand, can be applied in the forward direction as well as in a backward direction,
  - A backward approach reduces the proof of a goal (conclusion) to proofs of all its subgoals (premises).
  - Since in different situations, different application directions of derivation rules may be optimal (forward, backward, or mixed), RuleML does not prescribe any one of these.

- For facts or 'unit clauses' it makes little sense to talk of an application direction.
RuleML Tutorial

- Peter Miller's spending has been min 5000 euros in the previous year.

<Atom>
  <Rel>spending</Rel>
  <Ind>Peter Miller</Ind>
  <Ind>min 5000 euro</Ind>
  <Ind>previous year</Ind>
</Atom>

- This is a fact.
"spending" is marked up as the relation name (table name) for the fact.

"Peter Miller", "min 5000 euro", and "previous year" are marked up as individual constants that are the three arguments (table columns) of the relation.

The entire relation application constitutes an atomic formula, marked up by `<Atom> ... </Atom>`.

A relation can be n-ary, i.e. have any fixed number, $n = 0, 1, 2, 3, ...$, of arguments.

Null values are specified via empty individuals, `<Ind/>`. 
Representing as Tree

```
  Atom
     /\  
    /  
  /    
spending
     |
     Rel
     /\  
    /  
  /    
Peter Miller
     |
     Ind
     /\  
    /  
  /    
min 5000 euro
     |
     Ind
     /\  
    /  
  /    
previous year
     |
     Ind
```
An Example Rule

- A customer is premium if their spending has been min 5000 euro in the previous year.
  - <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>

- <if> and <then> was <body> and <head> in previous versions, respectively.
In Tree Form

```
# Rule: premium ~ customer → spending ~ customer ∩ min 5000 euro ~ previous year

Atomic

- head: premium ~ customer
- body: spending ~ customer ∩ min 5000 euro ~ previous year

Relationships

- Rel: premium
- Var: customer
- Rel: spending
- Var: customer
- Ind: min 5000 euro
- Ind: previous year
```

*Sharif Univ. of Tech. Rules in Semantic Web - Morteza Amini*
Non-Atomic Constructs

- The discount for a customer buying a product is 7.5 percent if the customer is premium and the product is luxury.

```xml
<Implies>
  <if>
    <And>
      <Atom>
        <Rel>premium</Rel>   <Var>customer</Var>
      </Atom>
      <Atom>
        <Rel>luxury</Rel>    <Var>product</Var>
      </Atom>
    </And>
  </if>
  <then>
    <Atom>
      <Rel>discount</Rel>  <Var>customer</Var>
      <Var>product</Var>   <Ind>7.5 percent</Ind>
    </Atom>
  </then>
</Implies>
```
Non-Atomic Constructs

In some family of RuleML we can use other constructs as follows:

- `<Or>` A disjunctive formula, where `<Or>Atom</Or>` is equivalent to `Atom`.

- `<Exists>` Explicit existential quantifier. It consists of one or more variables `<Var>`, each optionally surrounded by a `<declare>` role, followed by a logical formula.

- `<Forall>` Explicit universal quantifier. It consists of one or more variables `<Var>`, each optionally surrounded by a `<declare>` role, followed by a logical formula.

- `<Naf>` A "by default" negation of a logical atom `<Atom>` (i.e. "weak" negation or negation as failure).

- `<Neg>` A classical negation of a logical atom `<Atom>` (i.e. classical or "strong" negation).
Outline

- Rules and Their Usage in Web
- RuleML
- **SWRL**
- Existing Rule Engines
Introduction to SWRL

- SWRL is an acronym for Semantic Web Rule Language.
- SWRL is intended to be the rule language of the Semantic Web.
- SWRL includes a high-level abstract syntax for Horn-like rules.
- SWRL is based on OWL-DL: all rules are expressed in terms of OWL concepts (classes, properties, individuals, literals...).
Why Do We Need A Rule Language?

OWL Limitations:
- The OWL reasoning tools are mostly related to classes and classification.
- OWL reasoning is able to compute all the property values that are implied by the property characteristic.

A rule language is needed for several reasons:
- Expressivity can be added to OWL.
- Although expressivity always comes with a price, i.e. Decidability.
SWRL: Combining Ontologies and Rules

- A proposal to combine ontologies and rules:
  - Ontologies: OWL-DL more precisely OWL-Lite
  - Rules: RuleML

- SWRL = OWL-DL + RuleML
  - OWL-DL: variable free
    - corresponding to SHOIN(D)
  - RuleML: variables are used.
SWRL Characteristics

- W3C Submission in 2004: http://www.w3.org/Submission/SWRL
- Has a formal semantics.
- Rules saved as part of ontology.
- Increasing tool support: Bossam, R2ML, Hoolet, Pellet, KAON2, RacerPro, SWRLTab.
- Can work with reasoners.
SWRL rules have the form of an implication between an antecedent (body) and consequent (head).

The intended meaning can be read as: whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold.

Both the antecedent (body) and consequent (head) consist of zero or more atoms.
An empty antecedent is treated as trivially **true** (i.e. satisfied by every interpretation), so the consequent must also be **satisfied** by every interpretation.

An empty consequent is treated as trivially **false** (i.e., not satisfied by any interpretation), so the antecedent must also **not be satisfied** by any interpretation.

Multiple atoms are treated as a conjunction.
SWRL Example

- Has uncle rule:
  - hasParent(?x, ?y) \land hasBrother(?y, ?z) \rightarrow hasUncle(?x, ?z)

- With named individual; has brother rule:
  - Person(Fred) \land hasSibling(Fred, ?s) \land Man(?s)
    \rightarrow hasBrother(Fred, ?s)
SWRL Built-ins

- Core SWRL built-ins defined by: http://www.w3.org/2003/11/swrlb
- Provides commonly needed built-ins, e.g., add, subtract, string manipulation, etc.
- For each operation: performs the operation and returns true.
- Normally aliased as ‘swrlb’.

Example (is adult rule):
- Person(?p) \land hasAge(?p, ?age) \land swrlb:greaterThan(?age, 17) \rightarrow Adult(?p)

- Person(?p) \land bornOnDate(?p, ?date) \land xsd:date(?date) \land swrlb:date(?date, ?year, ?month, ?day, ?timezone) \rightarrow bornInYear(?p, ?year)
SWRL Built-ins

- Built-ins for Comparison
  - swrlb:equal, swrlb:notEqual, swrlb:lessThan, swrlb:lessThanOrEqual, ...

- Math Built-ins
  - swrlb:add, swrlb:subtract, swrlb:multiply, swrlb:divide, swrlb:mod, ...

- Built-ins for Boolean Values
  - swrlb:booleanNot

- Built-ins for String
  - swrlb:stringEqualIgnoreCase, swrlb:stringConcat, swrlb:substring, ...

- Built-ins for Date, Time, and Duration
  - swrlb:yearMonthDuration, swrlb:date, swrlb:subtractDates, ...

- Built-ins for URIs
  - swrlb:resolveURI, swrlb:anyURI

- Built-ins for Lists
  - swrlb:listConcat, swrlb:listIntersection, swrlb:member, swrlb:listSubtraction, ...
<ruleml:imp>
  <ruleml:_rlab ruleml:href="#example1"/>
  <ruleml:_body>
    <swrlx:individualPropertyAtom swrlx:property="hasParent">
      <ruleml:var>x1</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>
Combining OWL and SWRL Reasonings

- OWL has inference capabilities through the OWL terms and characteristics of properties, like inversion, symmetry and transitivity.

- SWRL has inference capabilities through the SWRL rules.

- In order to avoid the necessity of iteration between OWL inferences and SWRL inferences, it would be good if rule engines could also apply the OWL characteristics.

- This implies that OWL terms and characteristics would be ‘translated’ to a SWRL equivalent.

- In SWRL it is perfectly possible to define rules for symmetry, inversion, or transitivity characteristics.
SWRLTab

- A Protégé-OWL development environment for working with SWRL rules.
- Supports editing and execution of rules.
- Extension mechanisms to work with third-party rule engines.
- Supports querying of ontologies.
### SWRLTab Rule Editor

**Table: SWRL Rules**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Name</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 1</td>
<td>hasSibling(?x1, ?x2) ^ Man(?x2) → hasBrother(?x1, ?x2)</td>
<td></td>
</tr>
<tr>
<td>Rule 10</td>
<td>hasParent(?x1, ?x2) ^ Woman(?x2) → hasMother(?x1, ?x2)</td>
<td></td>
</tr>
<tr>
<td>Rule 11</td>
<td>hasSibling(?x1, ?x2) ^ Woman(?x2) → hasSister(?x1, ?x2)</td>
<td></td>
</tr>
<tr>
<td>Rule 12</td>
<td>hasParent(?x1, ?x2) ^ hasSister(?x2, ?x3) → hasAunt(?x1, ?x3)</td>
<td></td>
</tr>
<tr>
<td>Rule 13</td>
<td>hasParent(?x1, ?x2) ^ Man(?x2) → hasFather(?x1, ?x2)</td>
<td></td>
</tr>
<tr>
<td>Rule 14</td>
<td>hasChild(?x1, ?x2) ^ Man(?x1) → hasSon(?x1, ?x2)</td>
<td></td>
</tr>
<tr>
<td>Rule 15</td>
<td>hasCo consort(?x2, ?x3) → hasParent(?x1, ?x2)</td>
<td></td>
</tr>
<tr>
<td>Rule 16</td>
<td>hasSibling(?x1, ?x2) ^ hasDaughter(?x2, ?x3) → hasNiece(?x1, ?x3)</td>
<td></td>
</tr>
<tr>
<td>Rule 17</td>
<td>hasChild(?x1, ?x2) ^ Woman(?x1) → hasDaughter(?x1, ?x2)</td>
<td></td>
</tr>
<tr>
<td>Rule 18</td>
<td>hasChild(?x1, ?x2) ^ hasChild(?x3, ?x2) → differentFrom(?x1, ?x3) → hasSibling(?x1, ?x3)</td>
<td></td>
</tr>
<tr>
<td>Rule 19</td>
<td>hasSibling(?x1, ?x2) ^ hasSon(?x2, ?x3) → hasNephew(?x1, ?x3)</td>
<td></td>
</tr>
<tr>
<td>Rule 20</td>
<td>hasParent(?x1, ?x2) ^ hasBrother(?x2, ?x3) → hasUncle(?x1, ?x3)</td>
<td></td>
</tr>
</tbody>
</table>
SWRL Rules in KnowledgeBase
Need for Rule Engine

- The execution of SWRL rules requires the availability of a rule engine.

- The most general picture of a rule engine:
  - The rule engine can perform reasoning using a set of rules and a set of facts as input.
  - Any new facts that are inferred are used as input to potentially fire more rules (in forward chaining).

- Rules and facts should be available in a format that is accessible to the rule engine.
Rule Engine Execution

RULES

FACTS

RULE ENGINE

NEW FACTS
Translations that are necessary in the current state-of-the-art to be able to run SWRL rules on a Protégé data set.

(1) The rules have to be translated and introduced in the rule engine.

(2) Afterwards, the ontology and the knowledgebase have to be translated and introduced into the rule engine.

(3) After reasoning,

(4) the results of the reasoning should be translated back into the Protégé format.
Actions for Execution of SWRL Rules

1._classes
2._rules
3. facts
4. new facts

APPLICATION

CLASSES

RULES

RULE ENGINE

NEW FACTS
SWRL and Querying

- SWRL is a rule language, not a query language.
- However, a rule antecedent can be viewed as a pattern matching specification, i.e., a query.
- With built-ins, language compliant query extensions are possible.
SWRLQ: SWRL Query

- Cleaner semantics than SPARQL.
- OWL-based, not RDF-based.
- The SWRL Query Built-in Library is one the SWRLTab built-in libraries.
- It provides SQL-like operations to format knowledge retrieved from an OWL ontology.
- Can work with reasoners.

**Example:** Return all adults in ontology:

Person(?p) ∧ hasAge(?p, ?age) ∧ swrlb:greaterThan(?age, 17) → swrlq:select(?p) ∧ swrlq:orderBy(?age)
Querying: Semantic Issues

- SWRL is based on OWL-DL so assumes open world semantics.

- Querying closes the world, e.g., how many adults in ontology?

- So, it results non-monotonicity!

- Unlike most built-ins, it does not evaluate the arguments and return true if the arguments satisfy some predicate.

- It acts as accumulators and built up data structure outside of an ontology.
SWRL Query

- Basic queries
  - select

- Counting
  - count

- Aggregation
  - avg, max, min, sum, …

- Ordering of Results
  - orderBy, orderByDescending, …

- Eliminating Duplicate Results
  - selectDistinct

- Naming of Result Columns
  - columnNames
SWRL Query Examples (1)

- Return persons under 25 year old with their age.
  \[
  \text{Person}(?p) \land \text{hasAge}(?p, ?a) \land \text{swrlb:lessThan}(?a, 25) \rightarrow \\
  \text{swrlq:select}(?p, ?a)
  \]

- Return car owners with the number of their cars.
  \[
  \text{Person}(?p) \land \text{hasCar}(?p, ?c) \rightarrow \text{swrlq:select}(?p) \land \text{swrlq:count}(?c)
  \]

- Return the oldest people’s age.
  \[
  \text{Person}(?p) \land \text{hasAge}(?p, ?age) \rightarrow \text{swrlq:max}(?age)
  \]
SWRL Query Examples (1)

- Return car owners’ names (in alphabetical order) with the number of their cars.
  \[ \text{Person(?p)} \land \text{hasName(?p, ?name)} \land \text{hasCar(?p, ?c)} \rightarrow \text{swrlq:select(?name)} \land \text{swrlq:count(?c)} \land \text{swrlq:orderBy(?name)} \]

- Return distinct people’s name
  \[ \text{Person(?p)} \land \text{hasName(?p, ?name)} \rightarrow \text{swrlq:selectDistinct(?name)} \]

- Return car owner’s names with the number of their cars where the returned columns named “Name” and “Count” respectively.
  \[ \text{Person(?p)} \land \text{hasName(?p, ?name)} \land \text{hasCar(?p, ?c)} \rightarrow \text{swrlq:select(?name)} \land \text{swrlq:count(?c)} \land \text{swrlq:columnNames("Name", "Count")} \]
SWRL Query Examples (2)

- Return all male persons’ name with prefix “Mr.”.
  \[
  \text{Person(?p) \land Male(?p) \land hasName(?p, ?name) \land}
  \]
  \[
  \text{swrlb:stringConcat(?fullname, "Mr.", ?name) \rightarrow swrlq:select(?fullname)}
  \]

- Return all super-properties of “hasName”.
  \[
  \text{tbox:isSubPropertyOf(?supProperty, hasName) \rightarrow swrlq:select(?subProperty)}
  \]

- Return all sub-classes of “Person”.
  \[
  \text{tbox:isDirectSubClassOf(?subClass, Person) \rightarrow swrlq:select(?subClass)}
  \]
### SWRL Rules

<table>
<thead>
<tr>
<th>Name</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule-1</td>
<td>A(?a) ∧ hasIntProperty1(?a, ?i1) ∧ hasIntProperty2(?a, ?i2) ∧ swrlb:add(?i3, ?i1, ?i2) → hasIntPr...</td>
</tr>
<tr>
<td>Rule-2</td>
<td>C(?c) ∧ hasStringProperty1(?c, ?s1) ∧ hasStringProperty2(?c, ?s2) ∧ swrlb:stringConcat(?s3, ?s1...</td>
</tr>
<tr>
<td>Rule-3</td>
<td>C(?c) ∧ hasStringProperty1(?c, ?s1) ∧ hasStringProperty2(?c, ?s2) ∧ swrlb:equal(?s1, ?s2) → has...</td>
</tr>
<tr>
<td>Rule-4</td>
<td>C(c2) ∧ swrlb:stringConcat(?s3, &quot;ABC&quot;, &quot;DEF&quot;) → hasStringProperty3(c2, ?s3)</td>
</tr>
<tr>
<td>Rule-5</td>
<td>C(c4) ∧ hasStringProperty1(c4, ?s1) ∧ hasStringProperty2(c4, ?s2) ∧ swrlb:stringEqualIgnoreCase...</td>
</tr>
<tr>
<td>Rule-6</td>
<td>C(c4) ∧ hasStringProperty1(c4, ?s1) ∧ hasStringProperty2(c4, ?s2) ∧ swrlb:stringLength(?l1, ?s1)...</td>
</tr>
<tr>
<td>Rule-7</td>
<td>C(?ccc) ∧ hasStringProperty1(?ccc, ?d) → query:select(?ccc, ?d)</td>
</tr>
</tbody>
</table>

### SWRLQueryControl

Select a rule with query built-ins from the list above and press the Run button. If the rule generates a result, the result will appear in a new tab.
Query functionality added with built-ins.

Interactive query execution with tabular results display.

Low-level JDBC-like API for use in embedded applications.

Can use any existing rule engine back end.
Outline

- Rules and Their Usage in Web
- RuleML
- SWRL
- **Existing Rule Engines**
The Java Expert System Shell

- Developed at Sandia National Laboratories in late 1990s.
- Created by Dr. Ernest J. Friedman-Hill.
- Inspired by the AI production rule language CLIPS.
- Fully developed Java API for creating rule-based expert systems.
Rule-Based Expert System Architecture

- Rule Base (knowledge base)
- Working Memory (fact base)
- Inference Engine (rule engine)
Inference (Rule) Engines

- **Pattern Matcher** – decides what rules to fire and when.
- **Agenda** – schedules the order in which activated rules will fire.
- **Execution Engine** – responsible for firing rules and executing other code.
How Does Jess Work?

- Jess matches facts in the **fact** base to **rules** in the rule base.

- The **rules** contain function calls that manipulate the fact base and/or other Java code.

- Jess uses the **Rete** (ree-tee) **algorithm** to match patterns.

- **Rete network** = an interconnected collection of nodes = working memory.
Jess Architecture Diagram

WORKING MEMORY

RULE BASE

INFERENC ENGINE

PATTERN MATCHER

AGENDA

EXECUTION ENGINE

Sharif Univ. of Tech.
jDREW

- An easily configured, powerful deductive reasoning engine for clausal first order logic (facts and rules)

- Knowledge-based systems to process the declarative information and rules can use jDREW as an embedded reasoning engine through its various application programmer's interfaces (APIs).

- jDREW can be easily deployed as part of a larger Java system, on a server or, with its small memory footprint, on a client.

- jDREW was designed to be flexible also in its capabilities; It currently provides modules to process rules in Prolog and RuleML format.
References

- RuleML.org
- http://herzberg.ca.sandia.gov/jess/
- http://www.jdrew.org/jDREWebsite/jDREW.html
Any Question...

amin@sharif.edu
Extension of RuleML in three dimensions:

- **User-level roles** provide frame-like slot representations as unordered argument collections.

- **URI grounding** allows for ‘webizing’ using URIs as object identifiers for facts and rules.

- **Order-sortedness** permits typed variables via Web links into taxonomies such as RDF Schema class hierarchies, thus reusing the Semantic Web’s lightweight ontologies.
Object Centering via User-Level Roles

- In KR there has been two unifying data models:
  - **Position-keyed (predicate-centered or pKR):** one predicate or relation symbol is focused, and applied to positionally. In Web it is implemented by languages based on XML (Parent is focused and Childs are accessed with positions)
  - **Role-keyed (object-centered or rKR):** one object identifier is focused, and associated via property roles, unordered, with other objects as arguments. In Web it is implemented by languages based on RDF.

- In RuleML version 0.8 a pKR-rKR-unifying data model that generalizes the data models of both XML and RDF to express clauses (facts and rules) is introduced.

- It is based on differentiating type and role elements in XML, where role tags (distinguished by a leading underscore) accommodate RDF properties.
Example of pKR (1)

\(\text{offer(Ecobile, special, 20000)}:\)

\[
\text{<fact>\n  \text{<_rlab>\text{\textit{pKR fact 1}}\text{</ind>\/_rlab>}}\n  \text{<_head>\n    \text{<atom>\n      \text{<_opr>\text{\textit{offer}}\text{</rel>\/_opr>}}\n      \text{<ind>Ecobile\text{</ind>}}\n      \text{<ind>special\text{</ind>}}\n      \text{<ind>20000\text{</ind>}}\n    \text{</atom>}}\n  \text{</_head>}}\n\text{</fact>}
\]

\(\text{Role elements (tagged by a leading underscore) are differentiated from type elements.}\)
Example of pKR (2)

- fact type has a head role associating it with an atom type. The atom, however, uses a role, _opr, only for its operator association with the rel(ation) type.

- The three arguments of type ind(ividual) are immediate atom children ordered in the spirit of XML and pKR.

- Thus, while the _opr role can be moved from the prefix position to a postfix position without changing its meaning, the ind types are semantically attached to their relative positions.
Example of rKR (1)

offer(name->Ecobile; category->special; price->20000):

<fact>
   <_rlab><ind>rKR fact 1</ind></_rlab>
   <_head>
      <atom>
         <_opr><rel>offer</rel></_opr>
         <_r n="name"><ind>Ecobile</ind></_r>
         <_r n="category"><ind>special</ind></_r>
         <_r n="price"><ind>20000</ind></_r>
      </atom>
   </_head>
</fact>
Now positions of the three arguments is not important.

A processor doesn’t need to process them as an ordered tree.

Having role names (e.g. name, category and price), a processor can process this rule without any concern on positions.
Example of Combining pKR and rKR

\[\text{offer(Ecobile, special, 20000; expiry->2003-12-31; region->North America)}:\]

\[
\text{<fact>}
\]

\[
\text{\_rlab}\text{<ind>prKR fact 1</ind></_rlab>}
\]

\[
\text{\_head>}
\]

\[
\text{\_atom>}
\]

\[
\text{\_opr><rel>offer</rel></_opr>}
\]

\[
\text{<ind>Ecobile</ind>}
\]

\[
\text{<ind>special</ind>}
\]

\[
\text{<ind>20000</ind>}
\]

\[
\text{\_r n="expiry"><ind>2003-12-31</ind></_r>}
\]

\[
\text{\_r n="region"><ind>North America</ind></_r>}
\]

\[
\text{</atom>}
\]

\[
\text{</_head>}
\]

\[
\text{</fact>}
\]
Example of Using Variables

- Discount rule applies for customers of “gold” status and for offers in “special” category:

\[
\text{discount(offer name->}\text{?off}; \text{customer name->}\text{?cust}; \text{awarded amount->10)}
\]

\[
\leftarrow \text{offer(name->}\text{?off}; \text{category->special; price->\_}) \text{ AND}
\]

\[
\text{customer(name->}\text{?cust; status->gold}).
\]
Example: Using Variables (cont.)

<imp>
  <rK_r><ind>rKR rule 1</ind></rK_r>
  <head>
    <atom>
      <opr><rel>discount</rel></opr>
      <r n="offer name">off</r>
      <r n="customer name">cust</r>
      <r n="awarded amount">10</r>
    </atom>
  </head>
  <body>
    <and>
      <atom>
        <opr><rel>offer</rel></opr>
        <r n="name">off</r>
        <r n="category">special</r>
        <r n="price"></r>
      </atom>
    </and>
  </body>
</imp>
URI Grounding

- Describing resources as their URIs is possible through using a \textit{wid} (web id) attribute within the \textit{ind} type.

- this is complemented by a \textit{widref} (web id reference) attribute within the \textit{ind} type

- \textit{wid} and \textit{widref} are dual like XML’s \textit{id} and \textit{idref} and RDF’s \textit{about} and \textit{resource}. 
Example of URI Grounding

```xml
<ruleml:rulebase
    xmlns:ruleml="http://www.ruleml.org/dtd/0.83/ruleml-ooodatalog.dtd"
    xmlns:s="http://offercore.org/offerproperties#"
    xmlns:t="http://productcore.org/productproperties#">
    <fact>
      <_rlab><ind wid="http://catalist.ca/37">grKR fact 1</ind></_rlab>
      <_head>
        <atom>
          <_opr><rel>offer</rel></_opr>
          <_r n="s:name"><ind widref="http://ecobile.com">Ecobile</ind></_r>
          <_r n="s:category"><ind>special</ind></_r>
          <_r n="s:price"><ind>20000</ind></_r>
        </atom>
      </_head>
    </fact>
    <fact>
      <_head>
        <atom>
          <_opr><rel>product</rel></_opr>
          <_r n="t:name"><ind>Ecobile SX</ind></_r>
          <_r n="t:fuel"><ind href="http://naturalgas.org">gas</ind></_r>
          <_r n="t:horsepower"><ind>90</ind></_r>
          <_r n="t:displacement"><ind>1550</ind></_r>
        </atom>
      </_head>
    </fact>
  </ruleml:rulebase>
```
Order-sorted KR (sKR) that is based on a special treatment of sort predicates and sorted (typed) individuals, variables, etc. in clauses.

With sort restrictions directly attached to variables (hence usable during unification), proofs can be kept at a more abstract level, thus reducing the search space.

An independently defined sort hierarchy, e.g., in RDFS (using subClassOf) or OWL, can be employed as the taxonomy that constitutes the partial order of the resulting order-sorted logic.
Example of Term Typing (1)

```xml
<ruleml:rulebase
  xmlns:ruleml="http://www.ruleml.org/dtd/0.83/ruleml-oodatalog.dtd"
  xmlns:t="http://distribcore.org/distribclasses#"
  xmlns:u="http://customercore.org/custclasses#">
  <imp>
    <_rlab><ind>rsKR rule 1</ind></_rlab>
    <_head>
      <atom>
        <_opr><rel>discount</rel></_opr>
        <_r n="offer name"><var type="t:Offer">off</var></_r>
        <_r n="customer name"><var type="u:Customer">cust</var></_r>
        <_r n="awarded amount"><ind>10</ind></_r>
      </atom>
    </_head>
  </imp>
</ruleml:rulebase>
```
Example of Term Typing (2)

<._body>
  <and>
    <atom>
      <._opr><rel>offer</rel></._opr>
      <._r n="name"><var type="t:Offer">off</var></._r>
      <._r n="category"><ind>special</ind></._r>
      <._r n="price"><var/></._r>
    </atom>
    <atom>
      <._opr><rel>customer</rel></._opr>
      <._r n="name"><var type="u:Customer">cust</var></._r>
      <._r n="status"><ind>gold</ind></._r>
    </atom>
  </and>
</._body>
</imp>
</ruleml:rulebase>