Primer

Reaction RuleML 0.2

20 July 2007

Location:

http://ibis.in.tum.de/research/ReactionRuleML/0.2/docs/ReactionRuleML-v0.2-Primer.doc
http://ibis.in.tum.de/research/ReactionRuleML/0.2/docs/ReactionRuleML-v0.2-Primer.pdf
http://ibis.in.tum.de/research/ReactionRuleML/0.2/docs/ReactionRuleML-v0.2-Primer.htm

Editors and Contributors:

Adrian Paschke, RuleML, Inc. <adrian.paschke AT gmx.de>
Alexander Kozlenkov, Betfair, Ltd. <alex.kozlenkov AT betfair.com>
Harold Boley, National Research Council Canada, <harold.boley AT nrc-cnrc.gc.ca>
Said Tabet, Inferware, Corp., <stabet AT comcast.net>
Michael Kifer,
Mike Dean

Abstract

Reaction RuleML is a general, practical, compact and user-friendly XML-serialized rule language and rule interchange format for the family of reaction rules. It incorporates different kinds of production, action, reaction, complex event notification / messaging and KR temporal/event/action logic rules into the native RuleML syntax using a system of step-wise extensions. In particular, the approach covers different kinds of reaction rules from various domains such as global active-database ECA rules and triggers, forward-directed production rules, temporal-KR event/action/process logics, rule-based complex event processing, notification & messaging and active update, transition and transaction logics.

Status

This is the approved version of the Reaction RuleML 0.2 Primer.

Notice

RuleML takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on RuleML's procedures with respect to rights in RuleML specifications can be found at the RuleML website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification, can be obtained from RuleML.

RuleML invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to implement this specification. Please address the information to the RuleML.

Table of Contents

1. Introduction

1.1. Terminology

Reaction RuleML is an acronym for for Reaction Rule Markup and Modelling Language. Reaction RuleML 0.2 is a revision of Reaction RuleML 0.1

1.2. Objective

This document, Reaction RuleML 0.2 primer, is a supplementary document to the Reaction RuleML 0.2 specification [XXX Reaction RuleML 0.2]. The primer provides a brief explanation of all the key features of Reaction RuleML 0.2 with the help of a practical use case and numerous examples. The primer is intended towards e.g. software developers/architects, system integrators, technology consultants who want to know the basics features and syntax of Reaction RuleML. A basic knowledge of declarative rule-based programming, XML, and any programming language is essential for a better understanding of this document. More information on declarative rule-based programming and RuleML can be found in [XXX RuleML Manifesto].

1.3. Non-Normative Status

The primer is a non-normative document and not a definitive specification of Reaction RuleML. The primer contains examples and other information for a better understanding of Reaction RuleML. However, these examples and information would not cover all possible scenarios that are syntactically expressed and covered in Reaction RuleML specifications. For any specific information, one is advised to refer to the Reaction RuleML specification.

1.4. Relationship to the RuleML and Reaction RuleML specification

RuleML 0.91 specification [XXX RuleML 0.91 Reaction RuleML 0.2] provides a complete normative description to the RuleML language family version 0.91. Reaction RuleML 0.2 specification [XXX Reaction RuleML 0.2] provides a complete normative description to the Reaction RuleML language version 0.2 of RuleML. This Primer provides an overview of using and developing with Reaction RuleML. This document should be considered as supplementary and in terms of its scope and completeness, it is not a replacement to the original specifications.

1.5. Outline of this Document

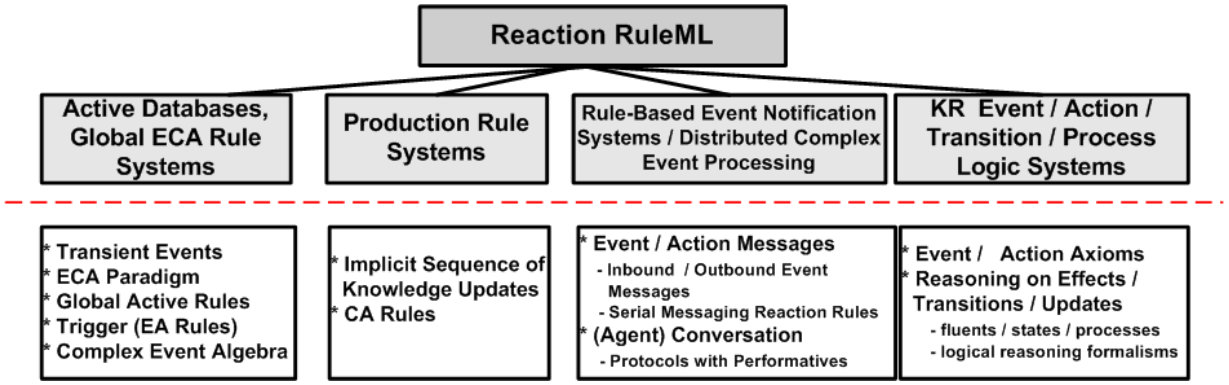
2. Background and History of Reaction RuleML

2.1. Scope

Reaction RuleML is a general, practical, compact and user-friendly XML-serialized language for the family of reaction rules. It is intended for distributed rule-based applications, where reaction rules of the various kinds need to be:

- serialized in a homogeneous combination with other rule types such as derivation rules, normative rules or integrity constraints;
- managed, maintained and interchanged in a common rule markup and interchange language;
- internally layered to capture sublanguages such as production rules, ECA rules, event messaging rules, KR event/action/state processing and reasoning rules;
- translated and executed in different target environments with different operational, execution and declarative semantics;

Reaction RuleML underpins rule-based systems for (pro-)active real-time or just-in-time (re-)actions to events (short-term perspective) but also retrospective and prospective reasoning on events, actions and their effects on changeable knowledge states (long-term perspective akin to state machines, process algebras or transition systems). It incorporates different kinds of production, action, reaction, and KR temporal/event/action logic rules into the native RuleML syntax using a system of step-wise extensions. In particular, the approach covers different kinds of reaction rules from various domains such as active-database ECA rules and triggers, forward-directed production rules, temporal-KR event/action/process logics, complex event notification and messaging and active update, transition and transaction logics.



Reaction RuleML covers constructs for (complex) event, action and state/fluent/transition definitions and specifications of different derivation rule, production rule and reaction rule programs according to the Reaction RuleML Classification of the Event / Action / State Definition and Processing Space" [XXX].

2.2. Design Goals

Reaction RuleML follows the modularization design principle of RuleML and defines new constructs within separated *modules* which are added to the RuleML family as additional *layers*. This layered and uniform design of Reaction RuleML makes it easier to learn the language and to understand the relationship between the different features and it provides certain guidance to vendors who might be interested only in a particular subset of the features and do not need support for the full expressiveness of Reaction RuleML.

The language fulfils typical criteria for good language design such as *minimality*, *symmetry* and *orthogonality*. With minimality we mean that Reaction RuleML provides only a small set of needed language constructs in addition to the existing constructs in RuleML, i.e., the same meaning cannot be expressed by different language constructs. Symmetry is fulfilled in so far as the same language constructs always expresses the same semantics regardless of the context they are used in. Orthogonality permits every meaningful combination of a language constructs to be applicable. Moreover, Reaction RuleML satisfies typical knowledge representation adequacy criteria such as epistemological adequacy w.r.t. the various application domains of reaction/action/production rules and KR event / action logics:

"A representation is called epistemologically adequate for a person or a machine if it can be used practically to express the facts that one actually has about the aspects of the world.„ [MCarthy1969]

2.3. RuleML

The Rule Markup Language (RuleML) [XXX] is a modular, interchangeable rule specification standard to express both forward (bottom-up) and backward (top-down) rules for deduction, reaction, rewriting, and further inferential-transformational tasks. It is defined by the Rule Markup Initiative [XXX], an open network of individuals and groups from both industry and academia that was formed to develop a canonical Web language for rule serialization using XML and for transformation from and to other rule standards/systems. The language family of RuleML covers the entire rule spectrum, from derivation rules to reaction rules including rule-based event processing and messaging (Reaction RuleML), as well as verification and transformation rules.

The building blocks of the derivation rules layer of RuleML (hornlogic layer) are:

- **Predicates** are n-ary relations introduced via an <*Atom*> element in RuleML. The main terms within an atom are variables <*Var*> to be instantiated by values when the rules are applied, individual constants <*Ind*>, data values <*Data*>, and complex expressions <*Expr*>.

- **Derivation Rules** are defined by an <*Implies*> element which consists of a body part (<*body*>) with one or more atomic conditions connected via <*And*> or <*Or*>, possibly negated by <*Neg*> (for classical negation) or <*Naf*> (for negation as failure), and of a conclusion part (<*head*>) that is implied by the body, where rule application can be in a forward or backward manner.
- **Facts** are stated as atoms deemed to be true: <*Atom*>
- **Queries** <*Query*> can be proven backward as top-down goals or forward via bottom-up processing, where several goals may be connected within a query, possibly negated.
- **Rulebases** <Rulebase> are a collection of rules that can be ordered or unordered, without or with duplicates.
- Ordered transactions of **performatives** (<Assert>, <Query>, <Retract>) making an 'implicitly sequential' assumption.

Besides facts, derivation rules, and queries, RuleML defines further rule types such as integrity constraints and transformation rules. A typical derivation rule in RuleML 0.91 might look like this:

```
<Implies>
  <head>
    <Atom>
      <op><Rel>own</Rel></op>
      <Var>Person</Var>
      <Var>Object</Var>
    </Atom>
  </head>
  <body>
    <!-- explicit 'And' -->
    <And>
      <Atom>
        <op><Rel>buy</Rel></op>
        <Var>Person</Var>
        <Var>Merchant</Var>
        <Var>Object</Var>
      </Atom>
      <Atom>
        <op><Rel>keep</Rel></op>
        <Var>Person</Var>
        <Var>Object</Var>
      </Atom>
    </And>
  </body>
</Implies>
```

Note that the <Implies> rule of RuleML 0.91 is generalized to an if-then <Rule> in Reaction RuleML 0.2.

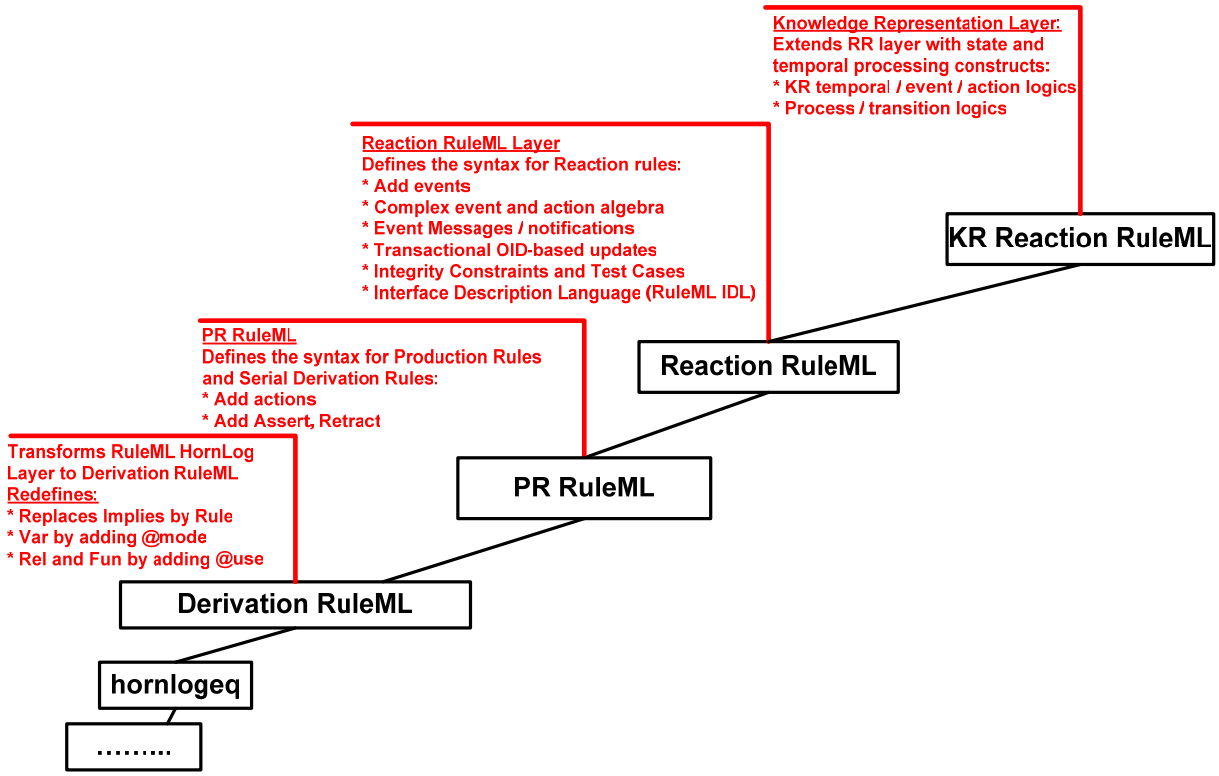
TODO: Explain role tags and type tags + role skipping

2.4. History

TODO

3. Basic Constructs

Reaction RuleML v0.2 generalizes the syntax of RuleML rules to a core rule format which can be specialized in different Reaction RuleML language families to different rule types such as derivation rules, production rules, reaction rules as well as mixed formats such as serial derivation rules which contain locally nested reaction rules and event/action constructs, e.g. update literals.



- **Derivation RuleML**: Defines Syntax and Semantics for Derivation Rules
- **PR RuleML**: Defines Syntax and Semantics for Production Rules
- **Reaction RuleML**: Defines Syntax and Semantics for Reaction Rules
- **KR RuleML**: Defines Syntax and Semantics for KR event/action logic formalisms and state, transition, process logic formalisms

The layers are not organized around complexity, but add different modelling expressiveness to the Reaction RuleML core for the representation of behavioural (re)action logic, decision logic, workflow-like logic, and KR event/action logic.

3.1. The Structure of a Reaction RuleML Document

The root element of a Reaction RuleML document is the <RuleML> construct from RuleML. It permits (ordered) transactions of KQML-like performatives (<Assert>, <Query> or <Retract>), making an 'implicitly sequential' assumption.

```
<RuleML
    xmlns="http://www.ruleml.org/0.91/xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ruleml.org/0.91/xsd
                        http://ibis.in.tum.de/research/ReactionRuleML/0.2/dr.xsd">

 <Assert>  add a rule base to the knowledge base  </Assert>
 <Query>  query knowledge base    </Query>
<Retract> remove rule base from the knowledge base   </Retract>
 <Assert>  add rule base   </Assert>

</RuleML>
```

The XML schema location defines the selected RuleML or Reaction RuleML language layer, e.g., the Derivation RuleML layer “http://ibis.in.tum.de/research/ReactionRuleML/0.2/dr.xsd” of Reaction RuleML 0.2. The sequential performatives act as wrappers specifying that their contents (optionally surrounded by a <formula> role) are to be added (<Assert>), deleted (<Retract>) or queried (<Query>), making an 'implicit <Rulebase>' assumption. This allows the separation of declarative content from such procedural performatives. A rule base (also known as module) consists of a set of rules (<Rule>) and facts (<Atom>).

```
<RuleML
    xmlns="http://www.ruleml.org/0.91/xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dc="http://purl.org/dc/elements/1.1/""
    xsi:schemaLocation="http://www.ruleml.org/0.91/xsd
                        http://ibis.in.tum.de/research/ReactionRuleML/0.2/dr.xsd">
<label>
      <Plex>
         <Expr><Fun uri="dc:title"><Ind>Reaction RuleML Example1</Ind></Expr>
         <Expr><Fun uri="dc:author"><Ind>Adrian Paschke</Ind></Expr>
         <Expr><Fun uri="dc:date"><Ind>2007-06-10</Ind></Expr>
      </Plex>
</label>
</oid></Ind>example1</Ind></oid>

<Assert>
  <Rulebase>
       <Atom>
             <Rel>keep</Rel>
             <Expr>
                   <Fun>person</Fun>
                   <Ind>Adrian</Ind>
                   <Ind>Paschke</Ind>
             </Expr>
             <Ind>key123</Ind>
       </Atom>
  </Rulebase>
</Assert>
<Query>
      <Atom>
             <Rel>keep</Rel>
             <Var>Person</Var>
             <Ind>key123</Ind>
       </Atom>
</Query>
```

The example asserts a fact about the person “Adrian Paschke” keeping the key “key123” to the KB and then queries the KB for all persons who keep the key “key123”.

The building blocks of a rule in Reaction RuleML 0.2 are:

- One general rule form (`<Rule>`) that can be specialized by to different rule types such as derivation rules, production rules, reaction rules, messaging rules.
- Three execution styles defined by the optional attribute `@style`; default value is "*reasoning*".
    - *active*: 'actively' polls/detects occurred events in global ECA style or changed conditions in production rules style
    - *messaging*: waits for incoming complex event message (inbound) and sends messages (outbound) as actions

  - *reasoning*: Logical reasoning as e.g., in logic programming (derivation rules) and KR formalisms such as event/action/transition logics (as e.g. in Event Calculus, Situation Calculus, TAL formalizations)
- "*weak*" and "*strong*" evaluation/execution semantics defined by the optional attribute `@evaluation` which is used to manage the "justification lifecycle" of local inner nested rules in the derivation/execution process of the outer rules.

Reaction RuleML 0.2 defines an optional meta data label `<label>`, an optional scope label `<scope>`, an optional qualification label `<qualification>`, and an optional object id lable `<oid>` for a RuleML specification `<RuleML>`, rule bases (module) `<Rulebase>` or `<Assert>` (implicit rule base), Queries `<Query>`, rules `<Rule>`, and facts/goals `<Atom>`.

- The *meta data label* can be used to define additional meta data (binary propety-value pairs) about the rule, e.g. the user-defined rule name or Dublin Core annotations such as dc_data or dc_author.
- The *scope* defines a view on the rule base, i.e. it constructs a view using query expressions on an explicitly closed part of the (possible distributed) modularized knowledge base. More specific scopes of queries and goals (e.g. goal literals in the body of a rule) overwrite defined default scopes of rule bases which apply by default to all goals/queries in the respective rule bases. If no scope is specified a query/goal applies on the maximum scope which is the entire knowledge base (note, might be possibly an open distributed KB on the Web).
- The *qualification* defines an optional set of rule qualifications such as a validity value, fuzzy value or a (defeasible) priority value. Again more specific qualifications of e.g. rules overwrite the general qualifcations of rule bases (modules).
- The *object identifier* (oid) denotes the object (identity) of the object. The oid can be e.g. used to link/point to the object within the knowledge representation, e.g. to define priorities between two rules or

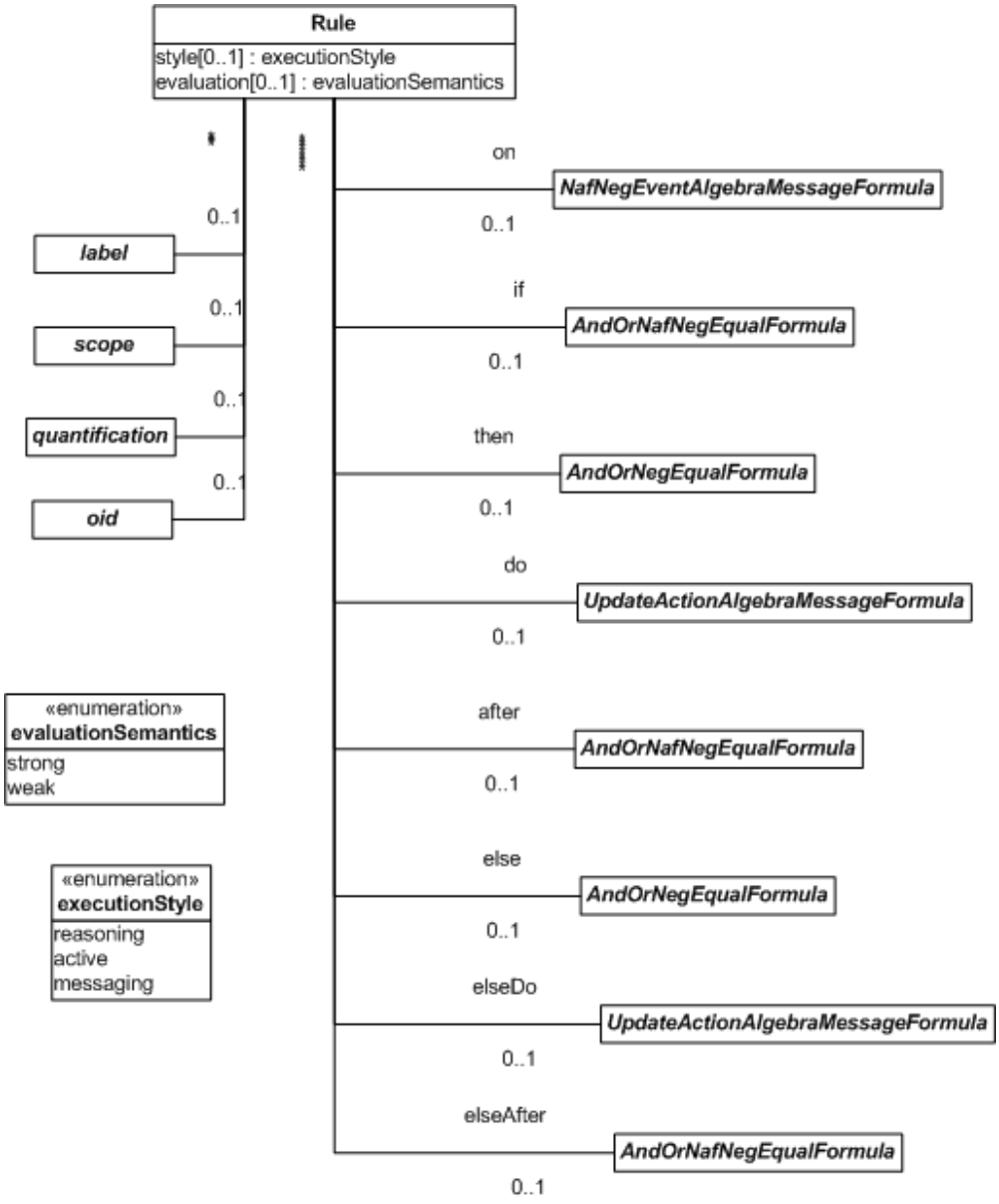
The full syntax of a rule in Reaction RuleML 0.2 is as follows:

```
<Rule style="active" evaluation="strong">
  <label> <!-- optional meta data --> </label>
  <scope> <!-- optional general scope of rule --> </scope>
  <qualification> <!-- optinal qualifications --> </qualification>
  <oid> <!-- optinal object identifier --> </oid>

  <on> <!-- event --> </on>
  <if> <!-- condition --> </if>
  <then> <!-- conclusion --> </then>
  <do> <!-- action --> </do>
  <after> <!-- postcondition --> </after>
  <else> <!-- else conclusion --> </else>
  <elseDo> <!-- else/alternative action --> </elseDo>
  <elseAfter> <!-- else postcondition --> </elseAfter>

</Rule>
```

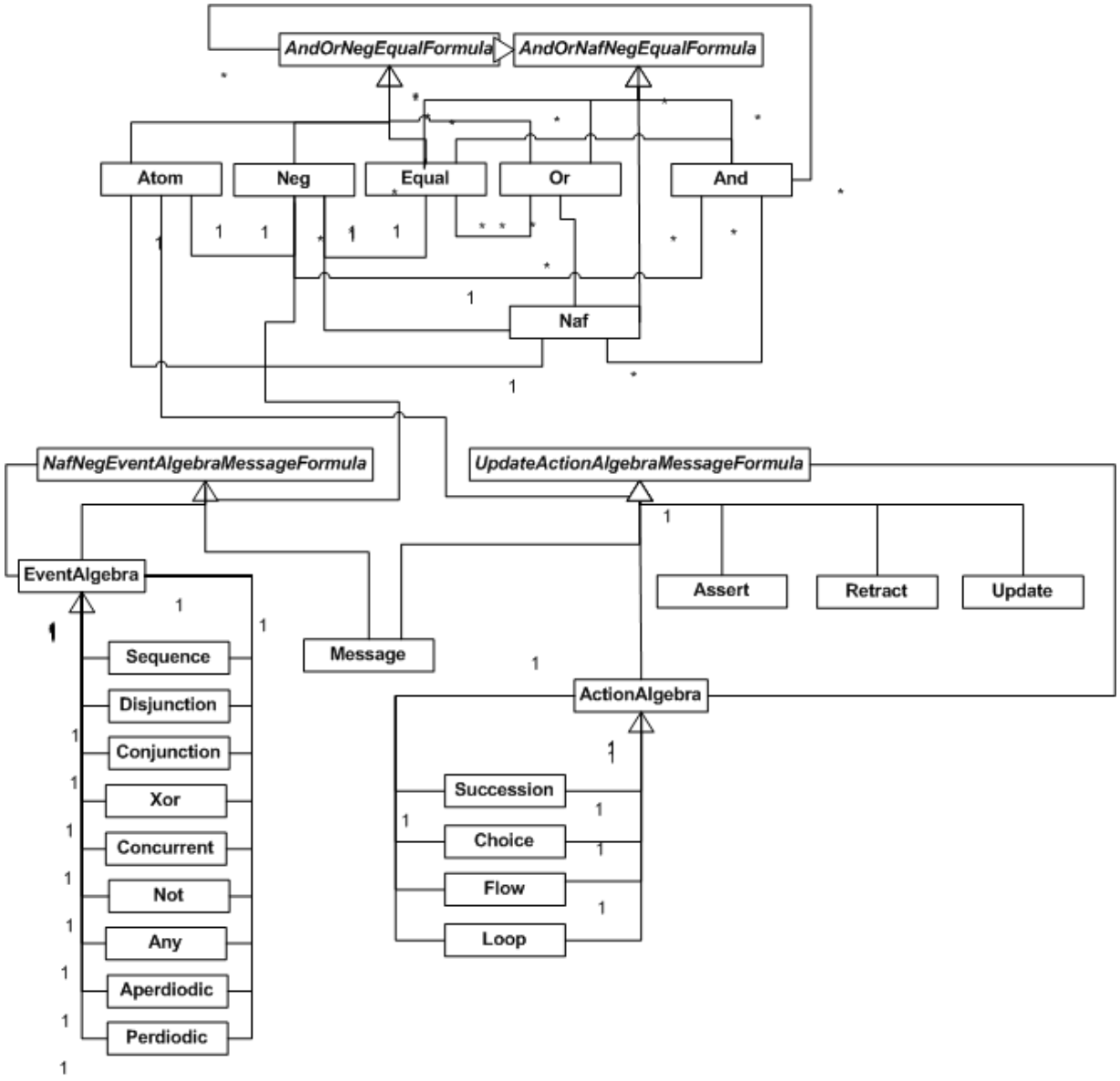
According to the selected and omitted rule parts a rule specializes, e.g. to a derivation rule (if-then or if-then-else; *reasoning* style), a trigger rule (on-do; *active* style), a production rule (if-do; *active* style), an ECA rule (on-if-do; *active* style) and special cases such as ECAP rule with a post condition (on-if-do-after; *active* style) or mixed rule types such as derivation rule with alternative actions (if-then-elseDo; *reasoning* style), e.g. to trigger an update action (add/remove from KB) or send an event message (e.g. to a log system) in case a query on the if-then derivation rule fails. A rule might apply globally as e.g. global ECA rules or locally nested within other rules. With weak evaluation semantics the inference / execution process of the outer rule simply proceeds in case of failure of an inner rule, whereas with strong interpretation the outer rule inference/execution fails possibly leading to backtracking.

3.2. Reaction RuleML 0.2 Meta Model

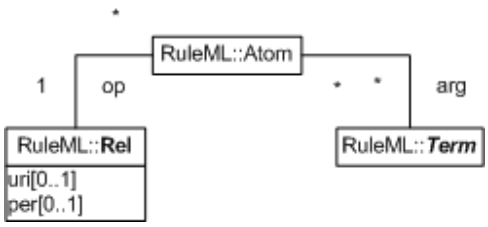
Reaction RuleML 0.2 Rule

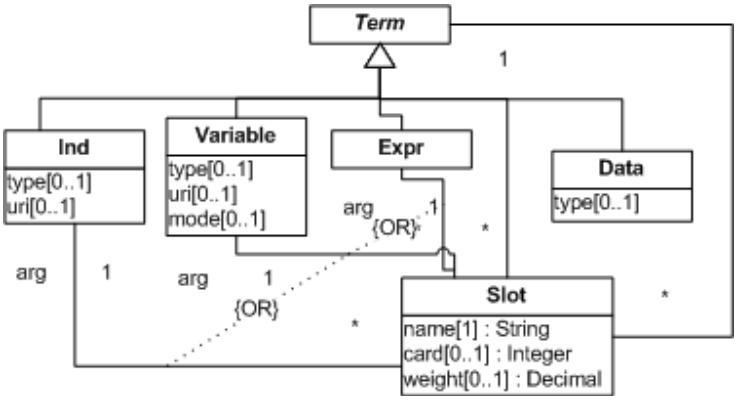


Reaction RuleML 0.2 Formula



Reaction RuleML 0.2 Atom





3.3. Derivation RuleML

The Derivation RuleML (http://ibis.in.tum.de/research/ReactionRule_ML/0.2/dr.xsd) layer of Reaction RuleML 0.2 extends and generalizes the Horn Logic layer with equality of RuleML 0.91 (http://www.ruleml.org/0.91/xsd/hohornlogeq.xsd). It defines syntax and semantics for derivation rules (if-then-(else) rules) which might be processed in forward (bottom-up starting with the facts) or backward-reasoning style (top-down starting with a query). For a discussion of the backward vs. forward reasoning see [XXX]. A derivation rule in Reaction RuleML 0.2 is of execution style "*reasoning*", which is the default value of the *@style* attribute and might be omitted. A derivation rule consists of an if-condition, a then-conclusion, and possibly of an additional else-conclusion (if-then-else rules):

```
<Rule>
   <label> <!-- optinal meta data --> </label>
  <scope> <!-- optinal general scope of rule --> </scope>
  <qualification> <!-- optinal qualifications --> </qualification>
  <oid> <!-- optinal object identifier --> </oid>

  <if> <!-- condition --> </if>
 <then> <!-- conclusion --> </then>
 <else> <!— optional else conclusion --> </else>

</Rule>
```

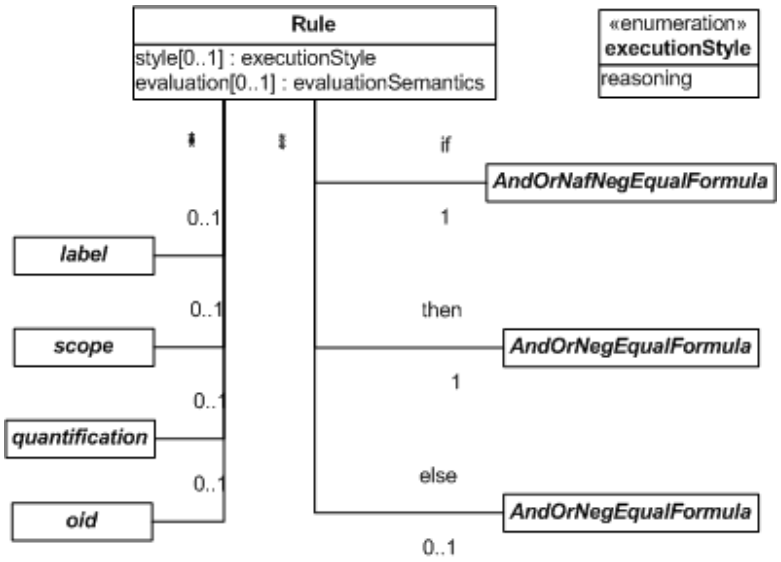


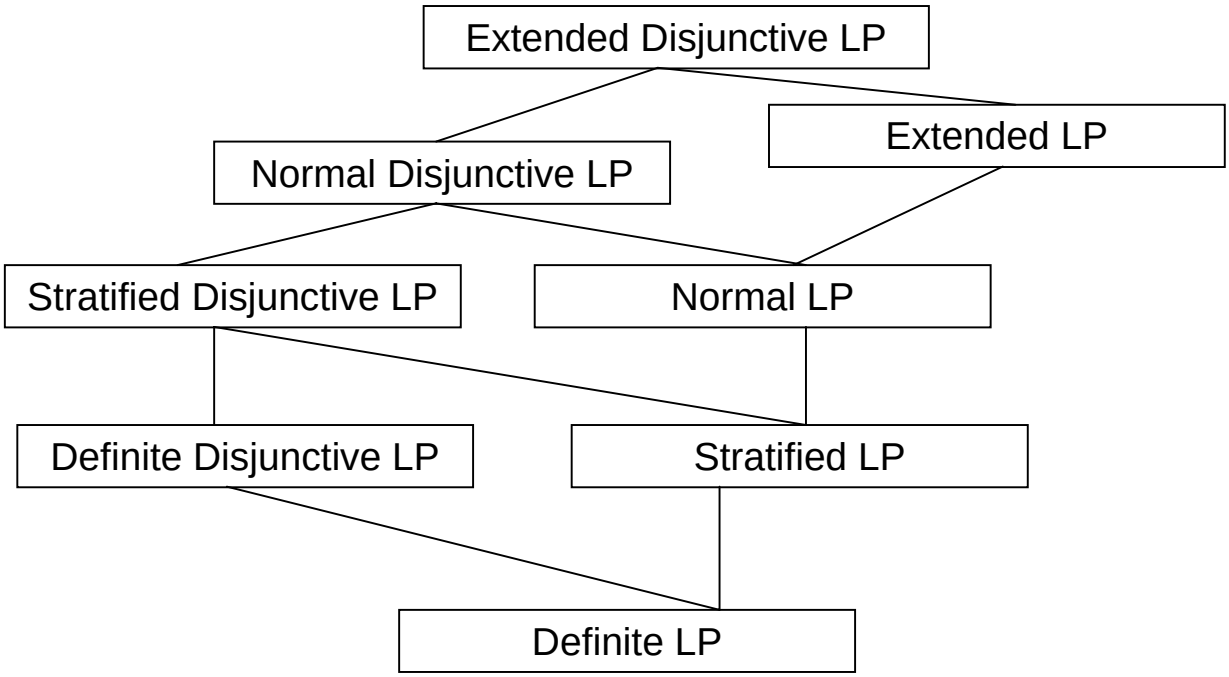
Figure 1: Derivation RuleML Meta Model

For instance, a derivation rule "*if a Customer is a premium customer then the Customer receives a discount of 10 percent*" can be formalized as follows.

```
<Rule>
   <if>
      <Atom>
            <Rel>premium</Rel>
            <Var>Customer</Var>
      </Atom>
   </if>
   <then>
      <Atom>
            <Rel>discount</Rel>
            <Var>Customer</Var>
            <Ind type="math:Percentage">10</Ind>
       </Atom>
  </then>
</Rule>
```

The example also shows the use of external type systems to define an explicit type for a term; here the individual constant "*10*" is of type "*math:Percentage*", where "*math*" is the name space prefix defined in the preamble of the RuleML document and "*Percentage*" is the type which is a class/concept defined in the external type system. These external type systems might be many-sorted, order-sorted, and polymorphic. Reaction RuleML supports typical type systems as in object-oriented languages such as Java or C++ (class hierarchies) with support for coercion (dynamic down-casting) and ad-hoc polymorphism or Semantic Web type systems such as RDFS taxonomies and OWL ontologies.

Derivation RuleML support different logic programming expressiveness classes, e.g., without variables (propsitional), without expression functions (datalog), with default negation-(as finite)-failure (normal), with explicit negation (extended), with or-disjunctions (disjunctive), with equality.



3.3.1. Basic Constructs

Rule bases (a.k.a. modules, clause sets) are represented by the <Rulebase> tag, queries by the <Query> tag, rules by <Rule>, and facts/goals by <Atom>. Conjunctions are represented by <And> and disjunctions by <Or> expressions. Logical constant terms are represented with <Ind> tag, variables with <Var> tag.

```
<Ind>Peter</Ind>                    "Peter"
<Ind>John Doe</Ind>                 "John Doe"
<Ind>42</Ind>                       42
<Var>X</Var>                        X
<Var/>                              _
```

3.3.2. Complex Terms/Functions and Lists

Complex terms (structures) describing logical functions are represented using the <Expr> tag. The function name (<Fun>) of an expression is followed by a sequence of zero or more arguments (<arg>). Optional user-defined slots (<slot>) are also permitted before and/or after the arguments, just like an atom (<Atom>). Rest variables (<repo> and <resl>) are also permitted.

```
<Expr per="plain">
      <Fun>book</Fun>
      <Var>Title</Var>
      <Var>Author</Var>
      <Var>Content</Var>
      <Var>Chapters</Var>
</Expr>
```

A <Fun> with "*plain*" use is left uninterpreted (is default value and might be omitted); with "*value*" use is interpreted purely for its value and used for interpreted functions with return value(s).

Lists are collections of (ordered) arguments (optionally enclosed by <arg>) and/or (unordered) user-defined slots (<slot>).

```
<Plex>
      <Ind>book</Ind>
      <Var>Title</Var>
      <Var>Author</Var>
      <Var>Content</Var>
      <Var>Chapters</Var>
</Plex>
```

Note that by convention the first argument of a list might be interpreted as function relation and hence the list *[book,Title,Author,Content,Chapters]* might be equivalently written as *book(Title,Author,Content,Chapter)*, i.e. as <Expr>. Rest variables (<repo> and <resl>) are also permitted, e.g. *[Head|Tail]* is serialized as:

```
<Plex>
      <Var>Head</Var>
      <repo><Var>Tail</Var></repo>
</Plex>
```

3.3.3. Equality

Equality, denote by <Equal>, in Derivation RuleML is used in a declarative way for equality comparisons and assignment activities to free variables.

```
<Assert>
  <Rule>
      <if>
            <Equal>
                  <Var>Value</Var>
                  <Data xsi:type="xs:int">1</Data>
            </Equal>
      </if>
      <then>
            <Atom>
                  <Rel>on</Rel>
                  <Var>Value</Var>
            </Atom>
       </then>
  </Rule>
</Assert>
<Query>
      <Atom>
            <Rel>on</Rel>
            <Var>Value</Var>
      </Atom>
</Query>
<Query>
      <Atom>
            <Rel>on</Rel>
            <Ind>1</Ind>
      </Atom>
</Query>
```

In the first query the variable "*Value*" is free (unbound) and hence is bound via equality unification to the value *1* of type *xs:int* in the condition part (if) of the rule by the equality formula. The query returns *1* as result of the variable. That is, here the equality function applies as assignment operator. In the second query the argument is a constant individual *1*, which is bound to the variable "Value" in the conclusion part of the rule and compared with the second argument in the equality function "1=1" of the condition, which means the query succeeds. That is, the equality function applies as comparison operator respectively object equality operator. Note, that there is an implicit assumption that numeric individuals such as 1,2,3 are of type (sort) Integer which unify by typed unification (which is implemented in the rule execution environment / inference engine) with equal type sorts from different type systems, e.g. java.lang.Integer=xs:int.
With the equal formula it is possible to assign arbitrary complex objects such as complex terms, lists, external objects, or constructive views created from external data sources or by scopes on the knowledge base, to variables and make copies of previously bound variables or parts of variables. Access to external data sources and constructive views will be described later on in this document.

```
<Equal>
      <Var>List</Var>
      <Plex>
            <Ind>discount</Ind>
            <Plex>
                  <Ind>customer</Ind>
                  <Ind>Jon Doe</Ind>
            </Plex>
            <Ind>10</Ind>
      </Plex>
</Equal>
```

The example assign the list *[discount,[customer,"Jon Doe"],10]* to the variable *List*. Note that by interpreting the first argument of a list as predicate/function name this list is equivalent to the complex term *discount(customer("Jon Doe"),10)* which is serialized as follows:

```
<Equal>
      <Var>List</Var>
      <Expr>
            <Fun per="value">discount</Fun>
            <Expr>
                  <Fun>customer</Fun>
                  <Ind>Jon Doe</Ind>
            </Expr>
            <Ind>10</Ind>
      </Plex>
</Equal>
```

3.3.4. Default Negation and Explicit Negation

Reaction RuleML distinguishes two forms of negation, namely default negation (weak negation) in terms of negation as failure (<Naf>) as in normal logic programs and explicit negation (strong negation) (<Neg>) as in extended logic programs or first order theories (classical negation).

```
<Rule>
	<if>
		<Atom>
			<Rel>reputation</Rel>
			<Var>Client</Var>
			<Ind>bad</Ind>
		</Atom>
	</if>
	<then>
		<Neg>
			<Atom>
				<Rel>trust</Rel>
				<Var>Client</Var>
			</Atom>
		</Neg>
	</then>
</Rule>
```

The rule states that there is no trust to a client if the reputation of the client is bad.

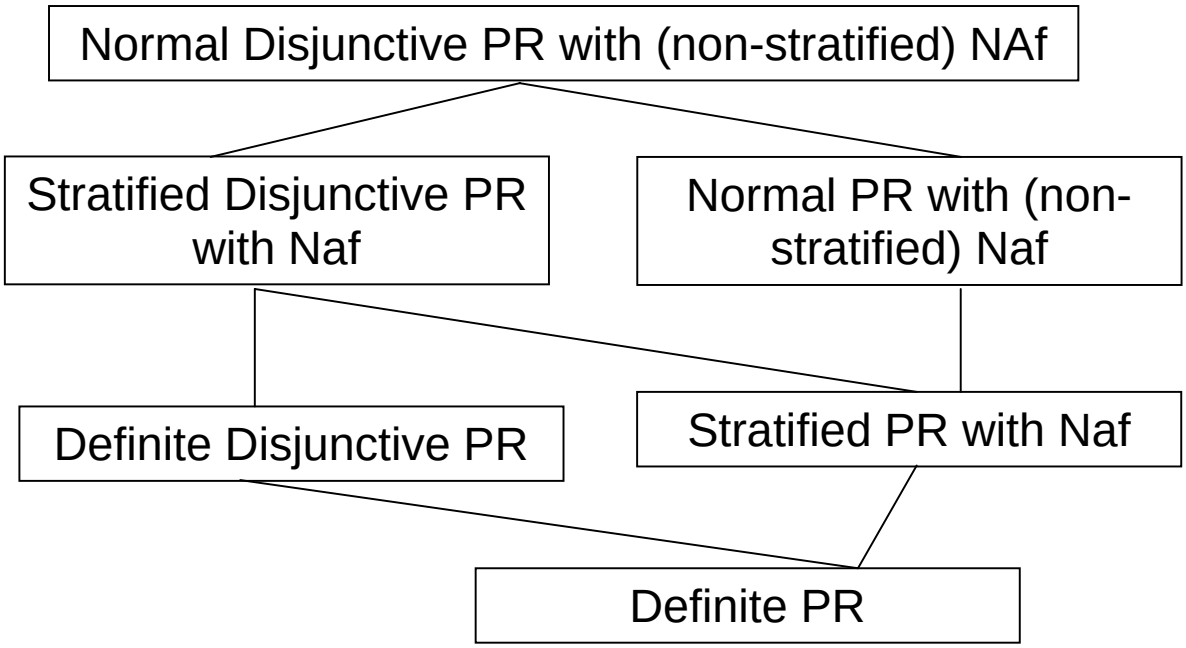
```
<Rule>
	<if>
	    <Naf>
		<Atom>
			<Rel>forbid</Rel>
			<Var>Person</Var>
			<Var>Action</Var>
		</Atom>
	    </Naf>
	</if>
	<then>
		<Atom>
			<Rel>permit</Rel>
			<Var>Person</Var>
			<Var>Action</Var>
		</Atom>
	</then>
</Rule>
```

This rule defines that a person is permitted to perform an action if it can not be proven that the person is forbidden to perform the action. That is, if there is no information that the person is prohibited the person has permission by default.

3.4. PR RuleML

The PR RuleML layer (http://ibis.in.tum.de/research/ReactionRule_ML/0.2/pr.xsd) extends the Derivation RuleML (http://ibis.in.tum.de/research/ReactionRuleML/0.2/dr.xsd) layer of Reaction RuleML 0.2. It defines syntax and semantics for

production rules (if-do rules; style=active) and serial horn rules (if-then rules with serial actions; style=reasoning) as well as mixed forms between derivation and production rules (e.g. if-then-elseDo rules; style=active). Different expressiveness classes of production rule programs are supported:



3.4.1. Production Rules and Stratified Production Rules with Negation

A production rule specifies the execution logic of one or more actions in the case that its conditions are satisfied. The forward-chaining production rules system paradigm typically defines a forward-directed operational semantics where the changes in the conditions due to update actions such as "*assert*" or "*retract*" on the internal working memory are considered as implicit events leading to further update actions and to a sequence of "firing" production rules, i.e.: "*if Condition do Action*".
Production rules have become very popular as a widely used technique to implement large expert systems in the 1980s for diverse domains such as troubleshooting in telecommunication networks or computer configuration systems. There are many sequential forward-chaining procedural implementations in the area of deductive databases and many well-known forward-reasoning engines for production rules such as ILOG's commercial jRules system, Fair Isaac/Blaze Advisor, CA Aion, Haley, ESI Logist or popular open source solutions such as OPS5, CLIPS or Jess which are based on the RETE algorithm. In a nutshell, this algorithm keeps the derivation structure in memory and propagates changes in the fact and rule base. This algorithm can be very effective, e.g. if you just want to find out what new facts are true or when you have a small set of

initial facts and when there tend to be lots of different rules which allow you to draw the same conclusion.

A production rule in Reaction RuleML 0.2 consists of a condition part (if), an action part (do), an optional post-condition part (after) and an alternative action part (elseDo) with an optional alternative post-condition part (elseAfter). The execution style `@style` for production rules is *style="active"*.

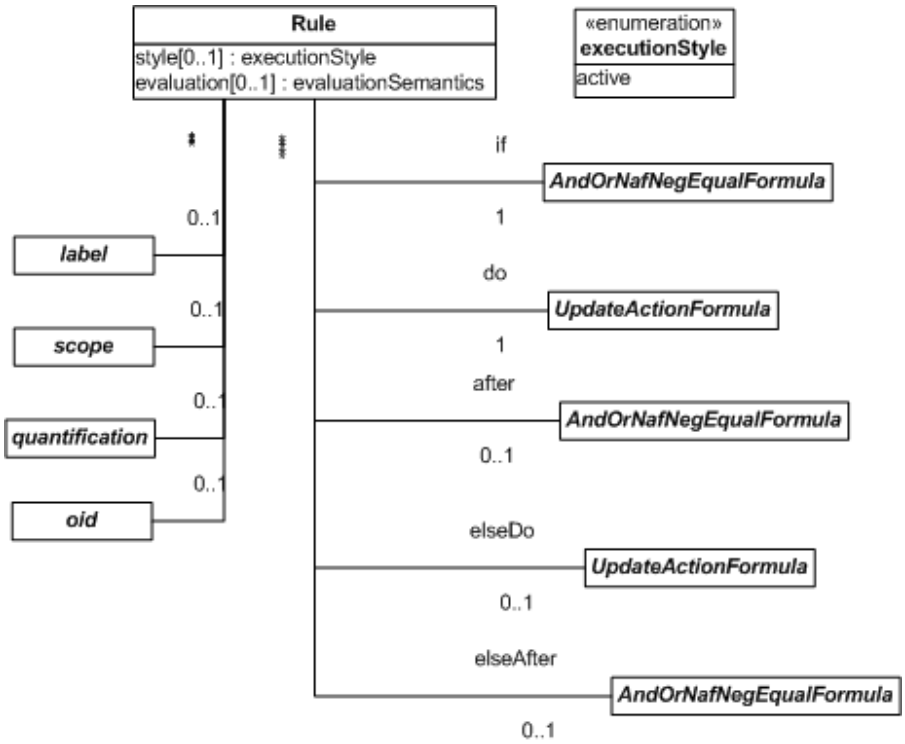


Figure 2: Meta Model for Production Rules

The full syntax for production rules (with conjunctions, disjunctions and negations) is:

```
<Rule style="active">
   <label> <!-- optinal meta data --> </label>
  <scope> <!-- optinal general scope of rule --> </scope>
  <qualification> <!-- optinal qualifications --> </qualification>
  <oid> <!-- optinal object identifier --> </oid>

  <if> <!-- conditions --> </if>
 <do> <!-- actions --> </do>
 <after> <!-- post-conditions --> </after>
 <elseDo> <!— alternative actions --> </else>
 <elseAfter> <!-- post-conditions --> </elseAfter>

</Rule>
```

The most common form of actions in production rules in Reaction RuleML 0.2 are update actions denoted by

3.4.2. Serial Horn Rules with (transactional update) actions

Closely related to production rules are logical update languages such as dynamic logic programs, transaction logic programs and in particular serial Horn rule programs, where the serial Horn rule body is a sequential execution of actions in combination with standard LP goal literals. These serial Horn rules can be processed top-down or bottom up. Other update languages define update rules with update actions in the head of a rule relating directly to production rules.

3.5. Reaction RuleML

3.5.1. Event-Condition-Action Rules

3.5.2. Complex Event Algebra

3.5.3. Complex Action Algebra

3.5.4. Messaging Reaction Rules and Complex Event Processing

Messaging

- promotes loose coupling between components;
- fits into content distribution, distributed workflow (e.g., loan procurement), chain of command, top-down information integration, XA-transactions scenarios;
- matches popular enterprise architectures like JMS, conversational web services (BPEL);
- compatible with the next-generation Enterprise Service Bus architectures, such as Microsoft Indigo, IBM MQ/Websphere, BEA ESB, open source Mule (mule.codehaus.org);
- can be ultimately (trivially) load balanced since nodes can consume messages as soon as they are ready;
- compatible with agent protocols and human-like conversations;
- can be based on speech act theories to categorise messages and reactions;
- compatible with process algebra based distributed software architectures including collaborating Petri nets, pi-calculus, CFSM;

- a message corresponds to a specific information item that is not part of the local knowledge—one can define epistemic knowledge learning rules for messages like ‘inform’ that specify how the information may be assimilated based on sender’s status, internal state, ordering of previous messages, timing, etc.
- compatible with OCL 2.0 action clause (signals, operations).

3.5.5. Reactive Workflows

3.6. KR RuleML

4. Advanced Constructs

4.1. Modules and Distributed Rule Bases

4.2. Constructive Views and Scopes

4.3. Language Extensibility

4.3.1. Constructs of other Namespaces and external Type Systems

4.3.2. Expression Languages and Query Languages

4.3.3. Procedural Attachments

4.4. Reaction RuleML Interface Description Language

5. Use Case

6. Reaction RuleML Tools

6.1. Validator Services

6.2. Translator Services

6.3. Reaction RuleML Editor and Rule Base Manager

6.4. Reaction RuleML Rule Interchange Middleware and Inference Service Broker

7. Changes in Reaction RuleML 0.2

As a result of the Reaction RuleML Technical Committee's issues process, the original Reaction RuleML 0.1 specification has received several updates. Changes in the 0.2 release relative to the previous version 0.1 are detailed below:

Core Rule Syntax

- Reaction RuleML 0.2 replaces <Implies> from RuleML and generalizes the syntax of different rule families by introducing one general <Rule> construct for all types of Rules.
- Reaction RuleML 0.2 adds an optional meta data label <label>, a scope <scope>, and a rule qualification <qualification> to a <Rule>.
- Reaction RuleML 0.2 explicitly denotes the usage (@per="plain|value|effect|modal") of functions and relations.

Complex Event and Action Algebra

- Reaction RuleML 0.2 redefines the complex event and action algebra and adds further operators for complex event and action definitions:

  Action Algebra: Succession (Ordered Succession of Actions), Choice (Non-Determenistic Choice), Flow (Parallel Flow), Loop (Loops)

  Event Algebra: Sequence (Ordered), Disjunction (Or) , Xor (Mutal Exclusive), Conjunction (And), Concurrent , Not, Any, Aperiodic, Periodic

Complex Event Messaging and Rule Interchange

- Reaction RuleML 0.2 defines Complex Event Messages <Message> also on the top-level of RuleML for message interchange
- Reaction RuleML 0.2 defines the RuleML Interface Description Language (RuleML IDL) for describing the signatures of public rule functions together with their mode and type declarations. These interface descriptions are important in open distributed rule-based environments.

External Data Access and Procedural Attachments

- Reaction RuleML 0.2 generalizes the support for Boolean-valued and object-valued procedural attachments
- Reaction RuleML 0.2 supports adding XML elements and/or attributes of non-RuleML namespaces to RuleML terms by using them as types.
- Reaction RuleML 0.2 supports XPointer and XPath expressions as markup and query language to point into and select data (resources and resource sets) from external XML data sources and create constructive views over resource sets
- Constructed views can be bound to variables and accessed using the compact '$' notation

8. Benefits of Reaction RuleML

Reaction RuleML provides a serialization support for systems that automate rule based reactive logic and processes. Full Reaction RuleML does not typically need to be executed directly, but its various sublanguages can be transformed into target execution languages of underlying rule-based systems, e.g., a business rule management system (BRMS), a production rule expert system, an active database system or another kind of event-driven architecture (EDA). Compared to traditional event-driven systems, this approach has the following major advantages:

- rules are externalized and easily shared among multiple applications (avoiding vendor lock-in) ;
- encourages reuse and shortens development time;
- changes can be made faster and with less risk;
- lowers cost incurred in the modification of business and reaction logic;

Reaction rules constitute the next step in the application of information system (IS) technology aimed at automating reactions to events occurring in e.g. open service-oriented Web applications. Automated services that have business and reaction logic embedded inside often take substantial time to change, and such changes can be prone to errors. In a world where the life cycle of business models and applications/services is steadily shortening, it has become increasingly critical to be able to adapt to changes in external environments promptly. These needs are addressed by Reaction RuleML. Reaction RuleML improves the agility of event driven architectures and the

manageability of business processes and behavioural reaction logic e.g. for complex event processing, since rules become more accessible.

9. Appendices

A. References