Submission to Business Modeling and Integration Domain Taskforce

Production Rule Representation

Ver. 1.01

Submitted by

Fair Isaac Corporation

ILOG SA

March 19, 2007

Content

Deleted:

Deleted:

1 Preface

This document contains a revised joint submission to a Request for Proposal titled *Production Rule Representation* (br/2003-09-03).

1.1 Submitters

This initial submission in response to the Production Rule Representation RFP is made by Fair Isaac Corporation and ILOG SA.

1.2 Contact points

Questions about this submission should be addressed to: James Taylor

1.2.1 Submitters

- James Taylor
  Fair Isaac Corporation
  jamestaylor@fairisaac.com
- Christian de Sainte Marie
  ILOG SA
  csma@ilog.com

1.2.2 Supporters

- Representing the RuleML Initiative:
  Said Tabet of RuleML: stabet@ruleml.org
  Gert Wagner of RuleML: G.Wagner@tu-cottbus.de .
- Representing commercial rule development vendors:
  Silvie Spreeuwenberg of LibRT: silvie@librt.com
  James Taylor of Fair Isaac Corporation: jamestaylor@fairisaac.com
  Christian de Sainte Marie of ILOG SA: csma@ILOG.com
  Jon Pellant of Pega Systems: jon.pellant@pega.com
  David Springgay of IBM: David_Springgay@ca.ibm.com
  Pedram Abrari of Corticon: Pedram@Corticon.com
  Paul Vincent of Tibco pvincent@tibco.com
- Representing associated tool vendors for UML and BPM:
  Jim Frank of IBM: joachim_frank@us.ibm.com
  Mark Linehan of IBM: mlinehan@us.ibm.com
  Jacques Durand of Fujitsu: mmjdurand@us.fujitsu.com
  Sridhar Iyengar of IBM siyengar@us.ibm.com

1.3 Guide to the Submission

The Production Rule Representation (PRR) standard was first proposed as the first technology-based rule-related standards in the OMG Business Rules Working Group, now part of the

Business Modeling and Integration (BMI) domain task force. PRR addresses the requirement for a common production rule representation, as used in rule engines from a variety of vendors today.

Although OMG standards are traditionally associated with "software modeling" tasks, the BMI task force (as well as many vendors represented in OMG) is associated with more "business-oriented" approaches to system automation, such as business rule automation and business process automation. This is fully compliant with the OMG Model Driven Architecture, and production rules provide an alternative, convenient representation for the many business rules that define behavior (ie actions). Many of the vendors involved in this standard all provide their own production rule representations, and these have been used as the basis for this standard.

As noted in the RFP, production rules in this context should not be confused with XMI production rules as defined in XMI 1.1 specification (formal/2000-11-02), production rules as defined in OCL 2.0, or other model or grammar transformation rules specified by the OMG standards such as SVBR.. With respect of production rules, the RFP solicited proposals for the following:

- A MOF2 compliant metamodel with precise dynamic semantics to represent production rules, where "production rules" refers to rules that are executed by an inference engine. This metamodel is intended to support a language that can be used with UML models for explicitly representing production rules as visible, separate and primary model elements in UML models.
- An XMI W3C XML Schema Description (xsd) for production rules, based on the proposed metamodel, in order to support the exchange of production rules between modeling tools and inference engines.
- An example of a syntax that is compliant with the proposed metamodel for expressing production rules in UML models. This syntax will be considered non-normative.

1.4 How to read this document

This submission is organized as follows.

Chapter 2 - Overview of the PRR and its relationships

Chapter 3 - PRR Proposal – definition, semantics and metamodel, scope, UML diagram notation examples and compliance requirements.

Chapter 4 – Comparison to, and interactions with, OMG Standards such as SBVR, BPDM, ODM.

Chapter 5 – References

Appendix A – Complete Metamodel

Appendix B – Glossary – definitions used in this document.

Appendix C – Guidance for Users

Appendix D – Relationship with: W3C Rule Interchange Format

Appendix E – Abstract Syntax Examples

Appendix F - Other Rule Types – Notes on how other rule types would relate to PRR

1.5 Standards Bodies Involved

There are currently a number of standards bodies and other initiatives involved with defining domain-independent production rule representations:

- **OMG** – represented by the Business Modeling Integration group and developers of the RFP to which this proposal responds.
- **W3C** - http://www.w3.org/2005/rules/ - has chartered a working group to define a rule interchange format for rule-driven systems. See also "Relationship with W3C Rule Interchange Format" on page 61.
- **RuleML** – http://www.ruleml.org/ - a family of related rule markup initiatives, with specific focus on W3C and the Semantic Web

The PRR is developed in collaboration with these bodies, with the goal that other standards in this area should be related for maximum standard interoperability and minimal vendor and user cost.

1.6 Commercial availability

The Letter of Intents states companies' intentions regarding commercial availability of this submission.

1.7 Edits since last version

1. IBM changed from submitter to supporter
2. Revised discussion of relationship with W3C RIF
3. Updated example syntax

2 Overview

This chapter provides an overview of the scope and objectives of this specification, and describes the resolution to RFP requirements and compliance points.

2.1 Scope

The PRR fulfils a number of requirements related to business rules, software systems, OMG standards, and other rule standards.

a. It provides a standard production rule representation that is compatible with rule engine vendors' definitions of production rules. It can therefore be used for interchange of business rules amongst rule modeling tools (and other tools that support rule modeling as a function of some other task).

b. It provides a standard production rule representation that is readily mappable to business rules, as defined by business rule management tool vendors.

c. It provides a standard production rule definition that supports and encourages system vendors to support production rule execution.

d. It provides an OMG MDA PIM model with a high probability of support at the PSM level from the contributing rule engine vendors and others, and can be included to add production rule capabilities to other OMG metamodels.

e. It provides examples of how the OMG UML can be used to support production rules in a standardized and useful way.

f. It provides a standard production rule representation that can be used as the basis for other efforts such as the W3C Rule Interchange Format and a production rule engine focused version of RuleML.

2.2 Objectives

The Production Rule Representation is proposed as a new OMG standard in order to:

- improve the modeling of production rules, especially with respect to the UML and MDA;
- allow interoperability across different vendors who define production rules in models.
- Support traceability of production rules to and from other UML and MDA artifacts

The development and adoption of this standard will encourage:

- accelerated adoption of production rule components and/or the business rules approach in everyday software systems;
- improved confidence in the use of production rule execution mechanisms such as rule engines.

2.3 OMG MDA Context

The Model-Driven Architecture (MDA) defines a model-driven approach to software development. An MDA specification consists of a definitive platform-independent base "UML model", plus one or more platform-specific models (PSM) and interface definition sets, each

describing how the base model is implemented on a different “platform”. The MDA also allows for an optional Business Model known as a CIM, or computation-independent model, which can be used as guidance to specify the PIM. It is expected that PRR will both be mappable to and from other OMG metamodels in conformance with the principles of the MDA, and embeddable in new metamodels that require production rule support.

2.3.1 Class of Platform

The target implementation platform for the PRR is the “forward chaining rule engine” or “inference engine”, hereafter described as “production rule engine”. The execution semantics are respectively referred to as "sequential rule processing" and "forward chaining” or “inferencing". The PRR defines a PIM for a “production rule engine class of platform”, that can be subsequently transformed to a vendor-specific model (PSM) executable by a vendor-specific rule engine.

2.3.2 MDA layers

The PRR assumes the following usage of the MDA in business rule driven software systems:

- The Business Model (or CIM): non-ambiguous representation of business policies, procedures, constraints as business rules in natural language and independent of assumptions regarding the platform on which an information system will be delivered.
- PIM: representation of production rules in UML targeted to the production rule engine class-of-platform that is independent of a vendor specific engine.
- PSM: representation of production rules in vendor-proprietary form executable by vendor-specific production rule engine.

The PRR scope includes only the PIM layer of this vision, and is limited to specifying requirements for representing production rules targeted at the forward chaining procedural and inferencing engine class-of-platform.

Production rule engine vendors will be able to provide a mapping from the PRR PIM to the PSM specific to their products, depending on whether procedural or Inferencing rules are specified and whether they support those types. The means to implement the PSM models is provided by such production rule engine products.

The Business Model (CIM) layer – representation of business rules – is addressed by a separate RFP requesting business rule semantics for business users, OMG document br/03-06-03, Business Semantics of Business Rules RFP. This standard is being finalized as the Semantics of Business Vocabularies and Rules (SBVR).

Section 4 below has a discussion of PRR’s relationship with other standards related to the MDA layers.

2.4 UML and Business Rules

Modeling business rules in UML is a broad topic. For example:

- Business rules defined as formal texts for documentation or requirements purposes would be covered by the SBVR standard;
- Business rules such as simple data relationships and constraints have simple counterparts within a UML model. For example, the business rule that “orders must have at least one line item” would typically be represented as a multiplicity constraint on an association. Other rules easily translate into constraint expressions. For example, the rules that “a

birthday must be equal to or earlier than the current date" or that "an account can never have a negative balance" are easily expressed as invariants using OCL (see below).

- Process-oriented business rules define conditional action / behavior / state changes as production rules that are not handled by OCL. Business rules that are expressed by production rules are very common and the representation (modeling) of production rules in UML is not standardized.

The two UML mechanisms for defining constraints and behavior are the Object Constraint Language (OCL) and action semantics (AS). However, neither of these provides an "out-of-the-box" solution for representing production rules. More details on the differences between PRR and these mechanisms is given in section 4.

2.5 Resolution of RFP requirements and requests

2.5.1 Execution Modes

Note that this specification is a change over the RFP, which specified inference rule engines only (not procedural rules as specified in this version of PRR), and both forward and backward chaining rules (where backward chaining rules are not specified in this version of PRR). This change is in order to accommodate industry requirements:

- many vendors that do not use inferencing technologies use instead procedural rules, and PRR accommodates these simple semantics
- few vendors use backward chaining techniques at this time, and in particular this is not used by the vendors involved in PRR.

The authors expect that future extensions of PRR may accommodate backward chaining inference engines.

2.5.2 XMI Schema

Note that an XMI W3C XML Schema Description (xsd) for the production rule metamodel is not provided in this submission. This is deliberate so as to allow the W3C Rule Interchange Format (RIF) to become the standard format for interchange of production rules between modeling tools and inference engines. The RIF standard is still under development (see Appendix D) and includes members of the PRR submission team as well as a number of PRR supporters.

2.5.3 Ruleset Aggregation

Ruleset aggregation is not supported in the submission to avoid complications involving different signatures and execution modes of rulesets.

2.6 Table of RFP Compliance Items

RFP requirement		Compliance
Par. #	**Description**	
6.2	**Required deliverables**	
	A MOF2 compliant metamodel for production rules that are executed by typical inference engines using standard processing algorithms for forward and backward chaining This metamodel is intended to support a language that can be used with UML models for explicitly representing production rules as visible, separate and primary model elements in UML models.	3.4,3.6, Appendix A
	An XMI W3C XML Schema Description (xsd) for the production rule metamodel in order to support interchanging of production rules between modeling tools and inference engines.	See 2.5.2
	An example of a transformation from an instance of the metamodel to a current implementation of a Production Rule language	See Appendix E
6.5	**Definition of Production Rule**	
	Production rules are statements of programming logic of the general form; IF Conditions THEN Actions	3.2 3.4.10
	The Conditions clause may be any (possibly complex) Boolean expression; the Actions clause may be any arbitrary action or collection of actions.	3.2.3, 3.2.4,3.2.5 3.4.11, 3.4.12, 3.4.13
	Production rules are executed by means of an inference engine, specifically their Actions clause is executed only when an inference engine evaluates the Conditions clause and determines that the conditions are true. Different types of production rule inference engines provide different algorithms for determining when a production rule is to be evaluated.	3.2.5
	The inference engine handles the execution, or firing, order the rules based on run-time conditions and its own internal algorithms. The rule modeler does not specify this order.	3.2.5
6.5.1	**Reuse of existing UML metamodels**	
	The proposed production rule metamodel shall use relevant packages of the UML 2.0 Superstructure for all static modeling constructs (see section 6.5.2.1 for the types of static modeling constructs to which production rules need to be able to reference	3.4.5, 3.6.1

RFP requirement		Compliance
Par. #	**Description**	
	It is anticipated that the production rule metamodel will define new behavioral constructs (see section 6.1.3.3.2). The proposed metamodel shall align with the behavioral modeling constructs of the UML 2.0 Superstructure where appropriate. (See section 6.7.1 for discussion of the issue of metamodel alignment.) The production rule metamodel shall extend the UML metamodel as appropriate per this RFP (see sections 6.5.6.4 and 6.6.2)	3.4.1, 3.4.2 3.4.5, 3.6.1
6.5.2.1	**Model element reference** Proposals shall provide a means for production rules to specify access to the types of model elements that can be used in UML Class models including, but not necessarily limited to, instances of classes (typically through quantifier expressions), the values of attributes of classes and instances, the operations/methods of classes and instances.	3.5
6.5.2.2	**Collections** Proposals shall provide a means to specify a collection of instances matching conditions. Once defined, a collection shall be treatable as an object instance. For example, collections may be referenced in production rule conditions as objects and tested for their properties (e.g., sizeOf(*collection*)) and manipulated as sets.	3.2.3
6.5.2.3	**Condition part structure** Proposals shall provide the means to express conditions of production rules with, but not limited to, arithmetic tests ($<$, $\leq$, $=$, $\geq$, $>$, $\neq$), comparison tests ($=$, $\neq$), set membership tests (is member of) and arithmetic expressions (+, -, /, *).	3.6.2
6.5.2.4	**Logical operators** Proposals shall provide the ability to use the AND, OR and NOT logical operators to combine the conditions, and nest logical operations.	3.6.3
6.5.2.5	**Action part structure** Proposals shall provide a means to express arithmetic expressions (+, -, /, *) and execution structures (loops, conditional expressions, nested expressions) in the actions of the production rule	3.6.4, 3.7.4, 3.7.5
6.5.2.6	**Rule scoping** Proposals shall provide an ***optional*** mechanism of scoping a rule to a namespace or a specific namespace element such as a class. A rule's scope provides a ***context*** in which the rule is evaluated. Scoped rules support the use of the notion of ***self***. If scoping production rules to a namespaces other than classes is defined, appropriate semantics must be provided.	3.6.2

RFP requirement		Compliance
Par. #	**Description**	
6.5.3	**Precise dynamic semantics** Proposals shall explicitly state the computational semantics of the elements in the proposed metamodel in an appropriate manner to specifying rule behavior with respect to an inferencing algorithm under which the rules are intended to be executed. The description of an inferencing algorithm should be sufficiently general that the prescribed algorithm is applicable across different implementations of the algorithm in specific inference engines. The semantics must be sufficient to describe which behaviors can be guaranteed across inference engines and which ones cannot	3.2.5
6.5.3.1	**Inference modes** The dynamic semantics of production rules must support both forward chaining and backward chaining algorithms. Support for the Rete algorithm is required	Forward Chaining: 3.2.5.1, Backward Chaining Omitted see: 2.5
6.5.4	**XML schema** Submitters shall supply an XMI W3C XML Schema Description (xsd) for the production rules constructed according to the proposed metamodel.	Omitted, see 2.5.2
6.5.5	**Compliance points** Submitters shall specify points at which vendor inference engines can be considered compliant with the production rule metamodel. It is possible that a proposal may permit partial fulfillment of the metamodel to constitute a degree of compliance.	3.8
6.5.6	**RuleSet** Proposals shall provide a means to specify a collection of production rules. Such a collection is typically called a **ruleset**. From a user's perspective, a ruleset typically contains rules that serve a specific functional purpose. From the modeler's perspective, the rules in a ruleset share semantic properties, for example, reasoning mode	3.2.2 3.4.7, 3.4.9
6.5.6.1	**Reasoning mode** Proposals shall provide a means to specify the reasoning mode for a ruleset. The reasoning mode can be either forward chaining or backward chaining. To support backward chaining, proposals shall provide the means to specify the condition (or "goal"), usually an attribute, whose value is to be determined through inferencing over the ruleset. (A goal is optional for forward chaining.)	3.4.9.1
6.5.6.2	**Ruleset aggregation** Proposals shall provide a means to aggregate rulesets in the sense that one ruleset may be included in another (set inclusion).	See 2.5.3

RFP requirement		Compliance
Par. #	**Description**	
6.5.6.3	**Ruleset invocation** Proposals shall provide a means of indicating methods that invoke rulesets as their (principal) behavior and of indicating what rulesets are so invoked by the method. NOTE: No assumption should be made regarding the precise invocation mechanisms, as this is an engine specific (i.e., a PSM and code level consideration). For example, a Java-based inference engine that is compliant with the JSR-94 specification will require different ruleset invocation mechanisms at the code level than a language that provides inferencing as a native feature.	3.4.7
6.5.6.4	**UML activities** Proposals shall provide a means to associate production rules to UML activities in order to model a ruleset as being invoked within an activity. For further discussion, see 6.6.2.	3.4.7
6.5.7	**Naming** Proposals shall provide a means to identify production rules and ruleset using a name. That is, rules and rulesets must be NamedElements	3.4.6, 3.4.7
6.5.8	**Example of a concrete syntax** Submitters shall include, as a non-normative element, at least one example of a concrete syntax for the production rule metamodel that supports the language elements described in section 6.5.2.	Examples in text and Appendix E
6.6	**Optional requirements**	
6.6.1	**Ordering Rules Consideration with Priorities** Programming with production rules occasionally encounters situations in which pre-specifying the order in which the rules are considered by the inference engine may be desirable. Typically, specifying the order in which rules are considered is done through assigning ***priorities*** to production rules. (NOTE: Although rule priorities are a feature of most commercial production rule inference engines, they compromise the declarative status of production rules. Therefore, specification of rule priority at the PIM level is considered to be optional requirement.) The execution semantics of rule priorities should be described as part of the dynamic semantics of production rules (see section 6.5.3)	3.4.10.1

RFP requirement		Compliance
Par. #	**Description**	
6.6.2	**UML Metamodel Extensions: UML Profile** To diagrammatically represent the relationships of rules and rulesets to model elements ***other than those to which the rules explicitly refer*** (see section 6.5.2.1) ***or are scoped*** (see section 6.5.2.6), new UML stereotypes and/or notations may be proposed through a UML Profile. An example of such an extension is the requirement to model the relationship of rulesets to action states in Activity Diagrams in a semantically significant manner (see section 6.5.6.4). Submitters may suggest further mechanisms to relate rules and rulesets to other model elements in other diagrams	Not in this release
6.7	**Issues to be discussed**	
6.7.1	**Alignment with Other Metamodels** When developing the Production Rule metamodel, submitters will need to use metamodel elements that need to align with metamodel concepts of other metamodels, specifically the action semantics and the OCL metamodel. Points of agreement and disagreement as well as points of extension should be identified. This discussion should be conducted from the perspective of ***unifying*** the production rule metamodel with actions semantics. Specifically, the production rule metamodel may be discussed in terms of (1) an extension of the UML Superstructure, or (2) integrated into the UML Superstructure, or (3) a “child” of a parent metamodel from which it is also possible to derive other elements of the UML Superstructure.	See Section 4
6.7.2	**Integration of UML Events and Inferencing** During modeling at the PIM level, it may be necessary for the modeler to refer to an event that occurred in the past within the conditions of a production rule (i.e., prior the current time at which the rule is being evaluated by the inference engine). Submitters are encouraged to present an approach for modeling the relationship of UML events to inferencing and for integrating such UML events into inferencing at the PIM level. NOTE: To pursue this topic it will be necessary to discuss the semantics of such references and to integrate this semantics into the semantics of the inference engine (see section 6.5.3). Alternatively, submitters may make some simplifying assumptions about modeling UML events within an object model, for example as instances of special "event" classes. In this case, there would be nothing special about referring to "events" within rules because the events would appear to the inference engine is standard instances. UML events may also have the role of invoking rulesets as opposed to methods that invoke rulesets (as described in section 6.5.8.2). An integrated view of UML events and inferencing may also be explored in submissions.	This submission assumes UML events are handled within an object model, for example as instances of special "event" classes

RFP requirement		Compliance
Par. #	**Description**	
6.7.3	**Production Rule Transformation from the Business Rules Metamodel** To the extent that is possible, submitters should discuss technical and methodological issues regarding how production rules might be transformed from business rules at the CIM level to production rules at the PIM level. A standard for business rules at the CIM level is currently being formulated through proposals to the Business Semantics for Business Rules RFP (br/2003-06-03). This standard will also include a standard for representing business vocabulary, which would map to classes and attributes. In principle, there should also be a mapping from (a subset of) business rules into (a subset of) production rules expressed in the UML model. Submitters are required to discuss, as far as possible at the time of submission, mapping issues for constructing production rules on the basis of business rules in the CIM.	See 4.5

3 Production Rule Representation

3.1 Introduction to PRR-Core and PRR-OCL

The following MOF2 compliant metamodel defines the PRR. It features:

- A definition of production rules for forward chaining inference and procedural processing.
- A definition for an interchangeable expression language (PRR OCL) for rule condition and action expressions, so they can be replaced by alternative representations for vendor-specific usage or in other standards.
- A definition of rulesets as collections of rules for a particular class of platform (procedural or inference rule engine).

The metamodel is composed of:

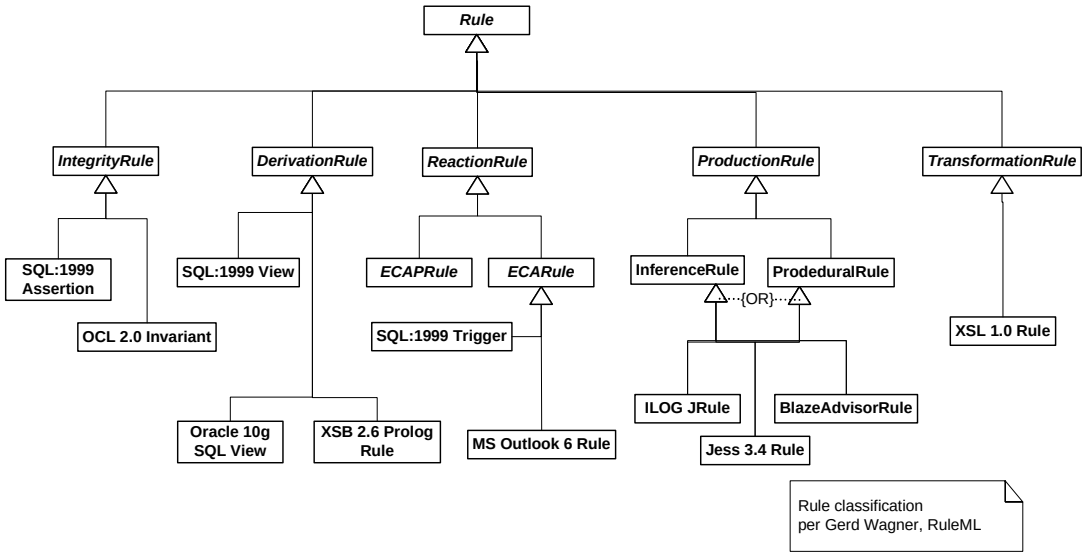
- a core structure referred to as PRR Core
- an abstract OCL-based syntax for PRR expressions, defined as an extended PRR Core metamodel referred to as PRR OCL.

Future extensions of PRR may address:

- rule metamodels for other classes of rules, such as Event-Condition-Action (ECA), backward chaining, and constraints
- rule representations that are specific to graphical notations, such as decision tables and decision trees
- representations of sequences of rulesets within larger decisions
- transformations between PRR and other MDA models such as SBVR.

Other concrete syntaxes may be applied to PRR Core in future. To this end, the PRR is designed to be extensible.

Production Rules fit into the following rule classification scheme (supplied by the RuleML Initiative), although they are a subclass of Computer Executable Rule rather than Rule to avoid confusion with other uses of "Rule" as a metamodel class.



3.2 Production Rules

3.2.1 Production Rule definition

A *production rule*[1] is a **statement of programming logic** that specifies the execution of one or more actions in the case that its conditions are satisfied. Production rules therefore have an operational semantic (formalizing state changes, e.g., on the basis of a state transition system formalism).

The effect of executing production rules may depend on the ordering of the rules, irrespective of whether such ordering is defined by the rule execution mechanism or the ordered representation of the rules.

The production rule is typically[2] represented as:

if [condition] then [action-list]

Some implementations extend this definition to include an "else" construct as follows:

if [condition] then [action-list] else [alternative-action-list]

although this form is not considered for PRR; all rules that contain an "else" statement can be reduced to the first form without an "else", and the semantics for interpreting when "else" actions are executed may be complex in some Inferencing schemes. Note that this implies that a conversion from a PSM to a PIM might be complete but not reversible. Rules with "else" statements in a PSM would result in multiple PIM rules which could not then be translated back into the original rules. The new rules would be functionally equivalent, however.

3.2.2 Production Ruleset definition

The container for production rules is the *production ruleset*. The production ruleset provides

- a means of collecting rules related to some business process or activity as a functional unit,
- a runtime unit of execution in a rule engine together with the interface for rule invocation.

From an architecture and framework perspective, a ruleset is a Behavior in UML terms,

The rules in a ruleset operate on a set of objects, called the "data source" in this document. The objects are provided by the ruleset's:

- parameters
- context at invocation time.

The changed values at the end of execution represent the result or "output" of a ruleset invocation.

3.2.3 Rule Variable definition

The condition and action lists contain expressions (Boolean for condition) that refer to 2 different types of variables (which we term as standard variables and rule variables).

At definition time:

---

[1] From the [RFP].

[2] If.. then.. rules are sometimes represented as when… then… rules by some vendors.

- a *standard variable* has a type and an optional initial expression. In some systems, there may also be a constraint applied to the variable, but the latter is outside the scope of PRR. Standard variables are defined at the ruleset level.

- a *rule variable* has a type and a domain specified optionally by a filter applied to a data source. With no filter, its domain defaults to all objects conforming to its type that are within scope / in the data source. Rule variables may be defined at the rule level, or at the ruleset level; in the latter case the rule variable definitions are available to all rules.

3.2.4 Semantics of Rule Variables

At runtime:

- standard variables are bound to a single value (that could itself be a collection) within their domain. The value may be assigned by an initial expression, or assigned or reassigned in a rule action.

- rule variables are associated with the set of values within their domain specified by their type and filter. Each combination of values associated with each of the rule variables for a given rule is a tuple called a *binding*. It binds each rule variable to a value (object or collection) in the data source. These bindings are execution concepts: they are not modeled explicitly but are the result of referencing rule variables in rule definitions.

This means that a production rule is considered for instantiating against ALL the rule variable values. The use of rule variables means that the definition of a production rule is in fact:

for [rule variables] if [condition] then [action-list]

Note that there is an implied product of rule variables when multiple rule variables are defined e.g.:

for [rule variable 1] for [rule variable 2] if [condition] then [action-list]

3.2.5 Semantics of Production Rules

The operational semantics of production rules in general for forward chaining rules (via a rule engine) are as follows:

i. *Match:* the rules are instantiated based on the definition of the rule conditions and the current state of the data source

ii. *Conflict resolution:* select rule instances to be executed, per strategy

iii. *Act:* change state of data source, by executing the selected rule instances' actions

However, where rule engines are not used and a simpler sequential processing of rules takes place, there is no conflict resolution and a simpler strategy for executing rules.

3.2.5.1 Operational Semantics for Forward-chaining production rules

A forward chaining production ruleset is defined **without** consideration of the explicit ordering of the rules; execution ordering is under the control of the inference engine that maintains a stateful representation of rule bindings.

The operational semantics of forward-chaining production rules extend the general semantics as follows:

1. *Match:* bind the rule variables based on the state of the data source, and then instantiate rules using the resulting bindings and the rule conditions. A rule instance consists of a binding and the rule whose condition it satisfies. All rule instances are considered for further processing

2. *Conflict resolution:* the rule instance for execution is selected by some means such as a rule priority, if one has been specified

3. *Act:* the action list for the selected rule instance is executed in some order

This sequence is repeated for each rule instance until no further rules can be matched, or an explicit end state is reached through an action.

It is important to note that:

- In the case where more than one binding satisfies the condition, there is one separate rule instance per binding.

- An action may modify the data source, which can affect current as well as subsequent bindings and condition matches. For example, an existing rule instance may be removed because the match is no longer valid or an additional rule instance may be added due to a newly valid match.

One popular algorithm for implementing such a forward chaining production rules is the Rete algorithm [RETE][1].

3.2.5.2 Operational Semantics for Sequential production rules

A sequential production rule is a production rule defined **without** re-evaluation of rule ordering during execution.

The operational semantics of sequential production rules extends the general semantics by separating the match into bind and evaluate steps, where the bind step is once-only step, as follows:

1. *Bind:* bind the rule variables based on the state of the data source at invocation time, and instantiate rules using the bindings.

2. Evaluate: evaluate the rule conditions based on the current state of the data source. Each instance is treated as a separate rule. If the condition evaluates to false then the rule instance is not considered.

3. Act: execute the action list of the current rule instance

This sequence 2-3 is repeated for one rule instance at a time until all the rules are processed, or an explicit end state is reached through an action.

It is important to note that:

[1] Charles Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Artificial Intelligence, 19, pp 17-37, 1982

- The processing order is defined per rule, not per rule instance. It is specific to the engine what is the ordering of the rule instances.

- The instances to be executed are defined on the initial state of the data source. Side effects from the execution of one instance will not affect the eligibility of other instances for execution. Side effects **may** affect the whether specific conditions of those rules are satisfied.

- Rule execution order is determined by the specified sequence of the rules in the ruleset.

3.3 Overview of PRR-Core

PRR Core is a set of classes that allow for production rules and rulesets to be defined in a purely platform independent way without having to specify OCL to represent conditions and actions. As such all conditions and actions are “opaque” and simply strings. While this limits the ability to transform rules from one production environment (PSM) to another, it would allow for sharing of rules between all tools that understand the basic structure of production rules.

3.4 PRR-Core Metamodels

This section specifies the PRR-Core Metamodel.

3.4.1 Overview of PRR-Core concept classes

The following is a partial model showing the concepts of general rules and rulesets for future extension to other rule types.

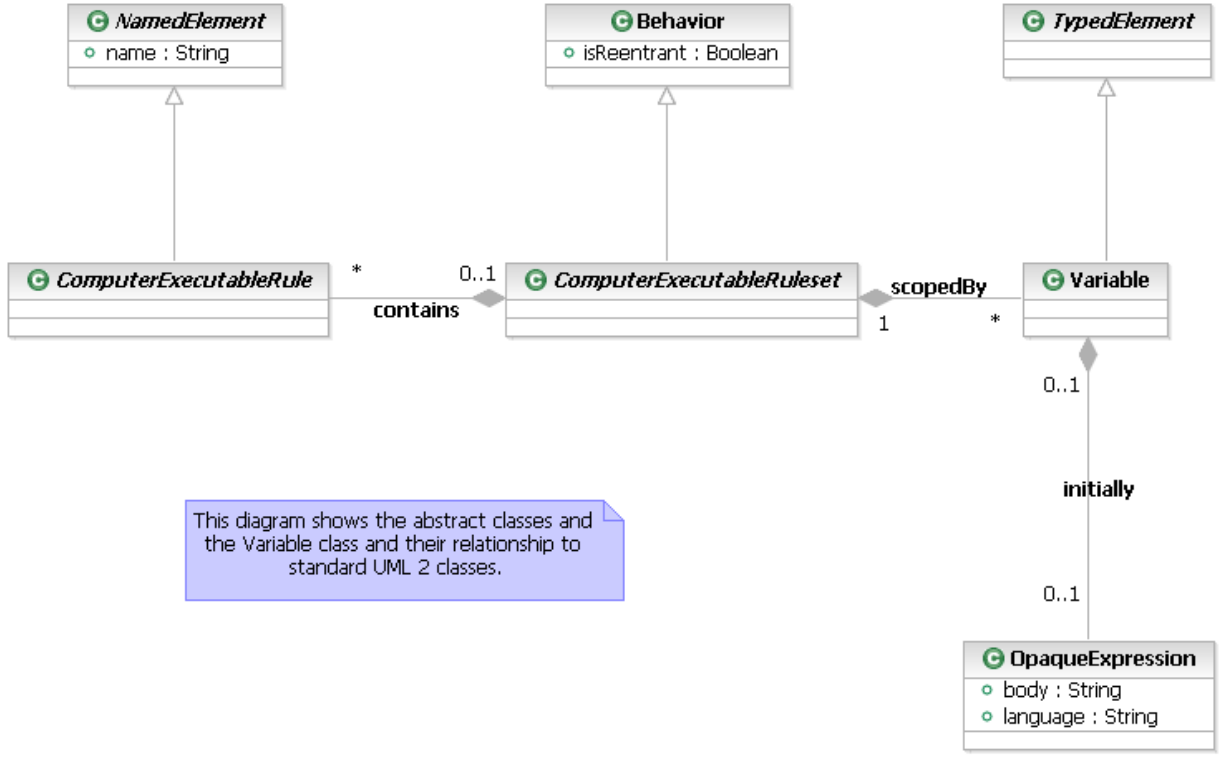


Figure 1: PRR Concept Classes

3.4.2 Overview of PRR-Core Production Ruleset

The following is a partial model showing the ProductionRuleset class and its relationship to other model elements.

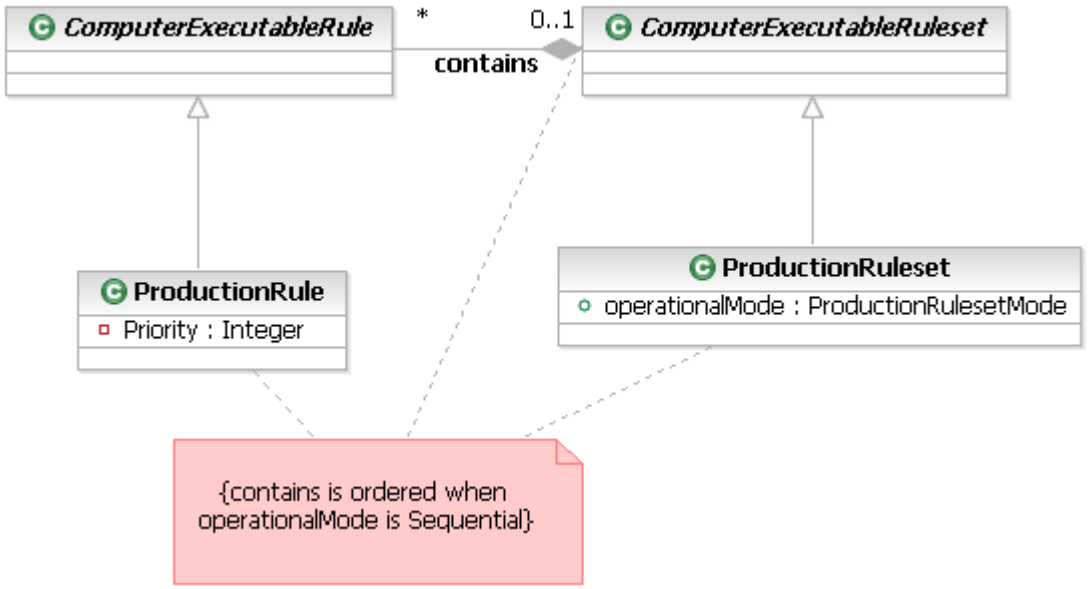


Figure 2: PRR ProductionRuleset Classes

3.4.3 Overview of PRR-Core Production Rule

The following is a partial model showing the ProductionRule class and its relationship to other model elements.

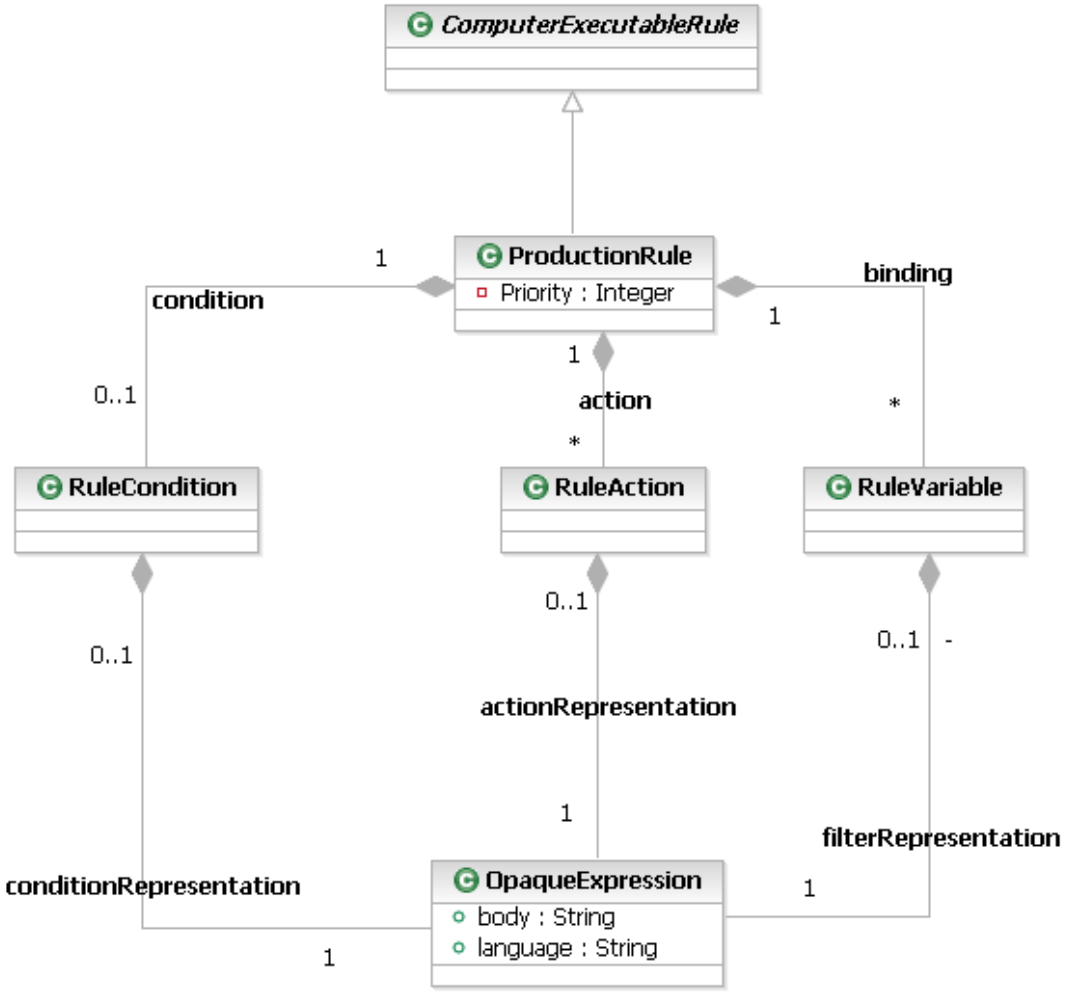


Figure 3: PRR ProductionRule Classes

3.4.4 Overview of PRR-Core RuleVariable

The following is a partial model showing the RuleVariable class and its relationship to other model elements.

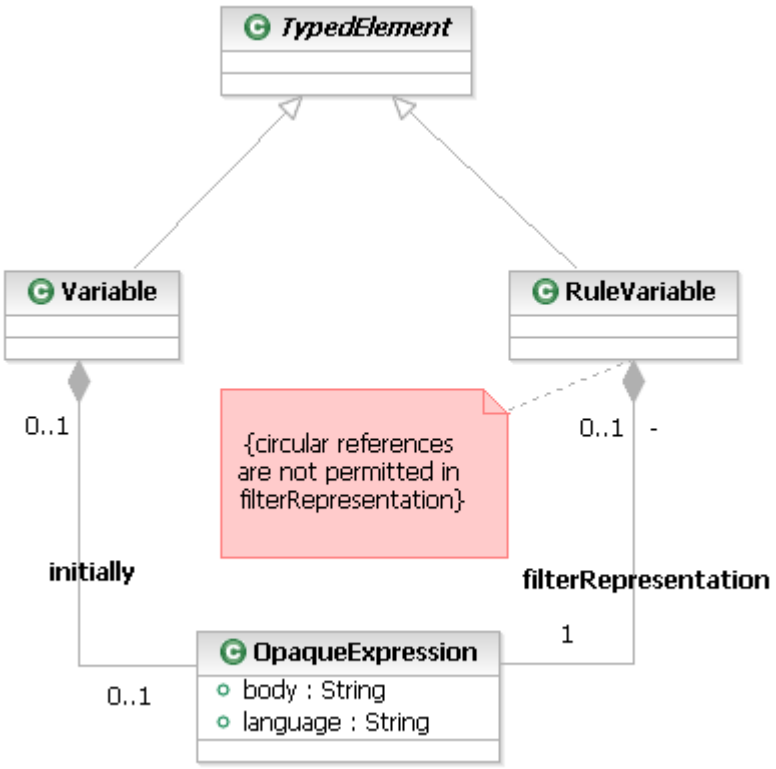


Figure 4: PRR RuleVariable Classes

3.4.5 Classes used in PRR Core

The following classes are used in these models. Each is defined below the table

Name	Description
RuleCore::ComputerExecutableRule	The computer executable rule represents a conditional piece of programmatic logic, including production rules. Future OMG standards may address other computer executable rule types such as event-condition-action rules, which would be derived from this class.
RuleCore::ComputerExecutableRuleset	The computer executable ruleset is a container for computer executable rules, and provides an execution context. In addition, a computer executable ruleset defines the interface for rule invocation, and the unit of execution in a rule engine; it is a Behavior in UML terms (or a service implementation in SOA terminology).
ProductionRule::RuleVariable	The variable represents a programming construct to hold values for use in executing a rule. The values must conform to the variable's type.

ProductionRule::ProductionRule	A ProductionRule is a statement of programming logic that specifies the execution of one or more actions in the case that its conditions are satisfied. See section below.
ProductionRule::ProductionRuleset	The ProductionRuleset represents a ruleset for production rules
ProductionRule::RuleCondition	The condition represents a Boolean expression that is matched against available data to determine whether a ProductionRule can be instantiated
ProductionRule::RuleAction	The action association defines an ordered list of actions.
UML2::Kernel::NamedElement	See UML2
UML2::Kernel::TypedElement	See UML2
UML2::BasicBehaviors::Behavior	See UML2

3.4.6 Computer Executable Rule

The rule represents a conditional piece of programmatic logic, including production rules. Future OMG standards may address other rule types such as event-condition-action rules, which would be derived from this class. A Computer Executable Rule is a Named Element.

3.4.6.1 Attributes

None

3.4.6.2 Associations

partOf:ComputerExecutableRuleset[0..1]
A rule may be part of a ruleset

3.4.6.3 Constraints

None

3.4.6.4 Semantics

The semantics for **computer executable rules** are determined by their subtypes such as **production rules**.

3.4.7 Computer Executable Ruleset

The ruleset is a container for rules, and provides an execution context for rule execution. In addition, a ruleset defines the interface for rule invocation, and the unit of execution in a rule engine; it is a Behavior in UML terms (or a service implementation in SOA terminology) and so a Named Element.

3.4.7.1 Attributes

None

3.4.7.2 Associations

contains:ComputerExecutableRule[*]
The rules contained in the ruleset.

scopedBy:Variable[*]
The variables defined in the ruleset

3.4.7.3 Constraints

None

3.4.7.4 Semantics

The semantics for computer executable rulesets are determined by their subtypes such as production rulesets.

3.4.8 Variable

The variable represents a programming construct to hold values for use in executing a rule. The values must conform to the variable's type.

3.4.8.1 Attributes

None

3.4.8.2 Associations

initially:OpaqueExpression[0..1]
An optional expression specifying an initialization on the variable.

scopes:ComputerExecutableRuleset[0..1]
The variable may be part of the scope of a ruleset

3.4.8.3 Constraints

None

3.4.8.4 Semantics

The variable represents a typed element that is used in rule expressions as a substitute for an explicit object reference.

3.4.9 ProductionRuleset

The ProductionRuleset represents a ruleset for production rules.

3.4.9.1 Attributes

operationalMode:enumeration{ProductionRulesetMode}
The operational semantics of the ruleset are described in its operationalMode attribute. The domain is open, but each model consumer (rule engine) will only understand a limited set of operational modes: this specification of PRR defines the semantics of rulesets with operation modes "Sequential" or "Forward Chaining".

3.4.9.2 Associations

None

3.4.9.3 Constraints

A ProductionRuleset may only contain ProductionRules

3.4.9.4 Semantics

The ProductionRuleset defines the operational semantics of the production rules it contains via the operationalMode attribute. Generally rule execution cycle is defined in 3 stages, and is repeated until some state is met:

1. Match: identify eligible rules

2. Conflict resolution: rule selection per strategy

3. Act: change state per rule definition

The eligible rules are identified during the match step by binding their rule variables and checking their conditions' OpaqueExpression against specified data. All the instances of eligible rules, obtained by substituting the rule variables with the values within their domain, are considered for further processing. See section 3.2.5 "Semantics of Production Rules" on page 19 for further detail.

3.4.10 ProductionRule

A ProductionRule is a statement of programming logic that specifies the execution of one or more actions in the case that its conditions are satisfied.

The execution of a production rule will depend on the type of rule engine and the other rules in the ruleset in which it is contained.

The production rule is represented as:

for [rule variables] if [condition] then [action-list]

3.4.10.1 Attributes

priority:integer
An optional attribute specifying the priority of a rule for use in determining the sequence of execution of Production Rules in Production Rulesets. Rules with higher priority values have higher priority than those with lower priority values.

3.4.10.2 Associations

condition:RuleCondition[0..1]
The rule condition that is required to be satisfied for the rule to be triggered.

action:RuleAction[*]
The ordered list of actions that are executed when the rule is fired.

binding:RuleVariable[*]
The list of RuleVariables that define the bindings in rule instantiation.

3.4.10.3 Constraints

There must be at least one RuleVariable or one RuleCondition specified.

3.4.10.4 Semantics

The operational semantics of production rules is defined in relation to the execution of the containing ruleset:

1. Given a set of objects assigned to its RuleVariables, the condition specifies whether the rule is eligible for execution / can be instantiated.

2. An instantiated rule can be chosen for execution (criteria being conflict resolution, strategy for execution sequencing, etc.), and if so, its actions are executed in order.

3.4.11 RuleCondition

The condition[1] represents a Boolean expression that is matched against available data to determine whether a ProductionRule can be instantiated. A tuple of RuleVariable values, known as a binding, defines a ProductionRule instance provided that with the binding the rule condition is satisfied. ProductionRule instances may be executed, subject to the operational mode of the containing ruleset. The condition filters the bindings that satisfy its expression, and then these values are used in the rule actions.

3.4.11.1 Attributes

None

3.4.11.2 Associations

conditionRepresentation:OpaqueExpression[1]
The expression specifying the rule condition.

3.4.11.3 Constraints

The OpaqueExpression evaluates to a Boolean result.

3.4.11.4 Semantics

The condition is used in the match step in the ProductionRuleset semantics, and gates the instantiation of the rules and the execution of the actions.

3.4.12 RuleAction

The action association defines an ordered list of actions. These actions may affect objects within the domain of a ruleset invocation (data source) or some external invocation.

3.4.12.1 Attributes

None

3.4.12.2 Associations

actionRepresentation:OpaqueExpression[*]
The expression used to specify an action.

3.4.12.3 Constraints

The actions form an ordered list.

3.4.12.4 Semantics

When a rule is executed, the list of actions is executed in sequential order.

---

[1] Note that production rules are popularly defined in terms of multiple conditions (eg a set of Boolean expressions that include ANDs and ORs to create a single logical expression). For the purposes of PRR, we define that a condition in a ProductionRule is a single Boolean expression.

3.4.13 RuleVariable

The RuleVariable defines a domain to be used in rule execution. If nothing else is specified, its domain is the contents of the data source conforming to this type. Oftentimes, however, it is necessary to further restrict the domain of a rule variable (for example, if the data source contains different sets of objects with the same type, such as applicant: Person [*], landlord: Person [*], tenant: Person [*], a rule variable with type Person would likely be restricted to one of these sets). The range of values that a rule variable can take may be further constrained by a filter expression..

3.4.13.1 Attributes

None

3.4.13.2 Associations

FilterExpression: OpaqueExpression [*] The expression used to specify a collection and/or filter for the domain represented by the RuleVariable

3.4.13.3 Constraints

The filter expression for a Rule Variable must not create circular references through references to other Rule Variables.

3.4.13.4 Semantics

At runtime, RuleVariables are used to specify the bindings that define applicable rule instances for specified values from the data source.

3.5 Overview of PRR OCL

PRR OCL makes use of the OCL metamodel to represent the expressions attached to the RuleVariable, Condition and Action parts of the production rules.

The version of the OCL specification that has been used in this document is OCL 2.0 ptc/2005-06-06 (issues for OCL 2.0 can be found here http://www.omg.org/issues/ocl2-ftf.open.html). The subset of OCL metaclasses that is used in PRR comes exclusively from BasicOCL. Metaclasses coming from complete OCL are not used.

PRR OCL is composed of:

- a selection of classes from the BasicOCL package (and consequently EssentialOCL) and a set of specific constraints that define the use of OCL classes in the context of PRR OCL
- a PRRActionOCL package that extends the BasicOCL package and provide the classes to represent the action part of the production rules.
- a PRR OCL Standard Library based on the OCL Standard Library that gives the predefined types and operations that any implementation of PRR OCL must support.

3.6 PRR OCL Metamodel

The following is a model of the classes involved in PRR OCL

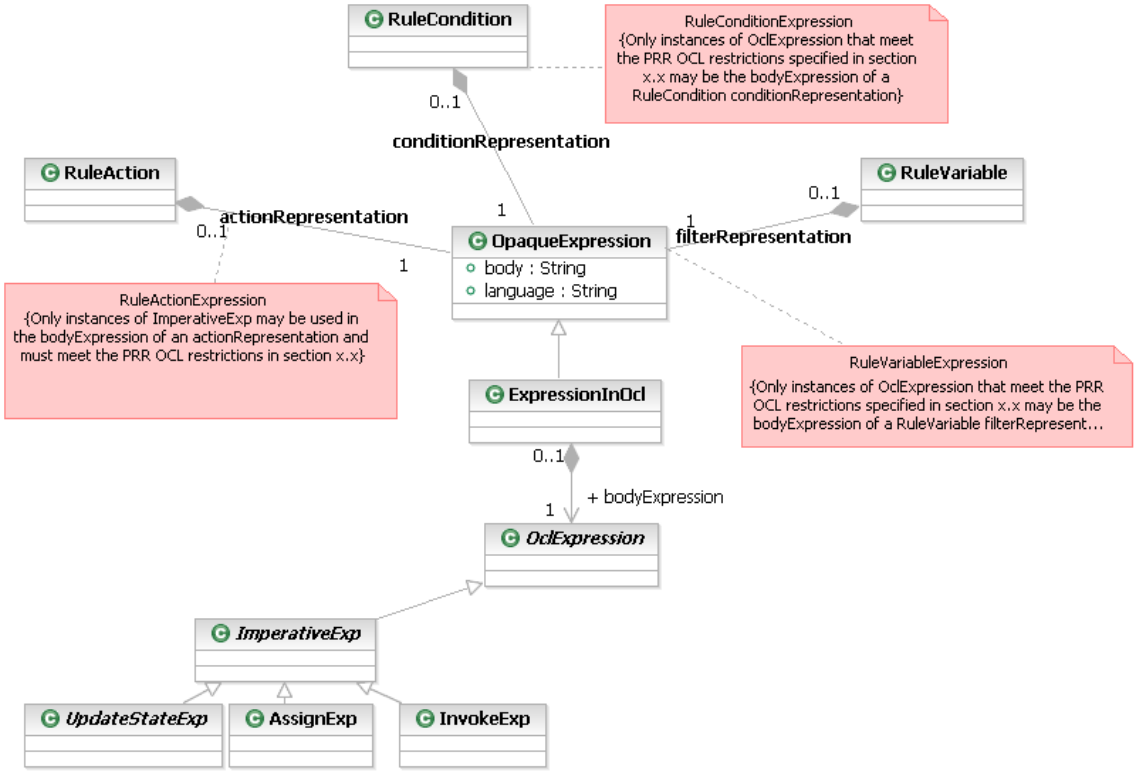


Figure 5 –Metamodel for PRR OCL

Figure 6, Figure 7, and Figure 8 show the subset of BasicOCL package that is used by PRR OCL. The classes that are not part of OCL are shown with a transparent fill color.

Deleted:

Deleted:

Deleted:

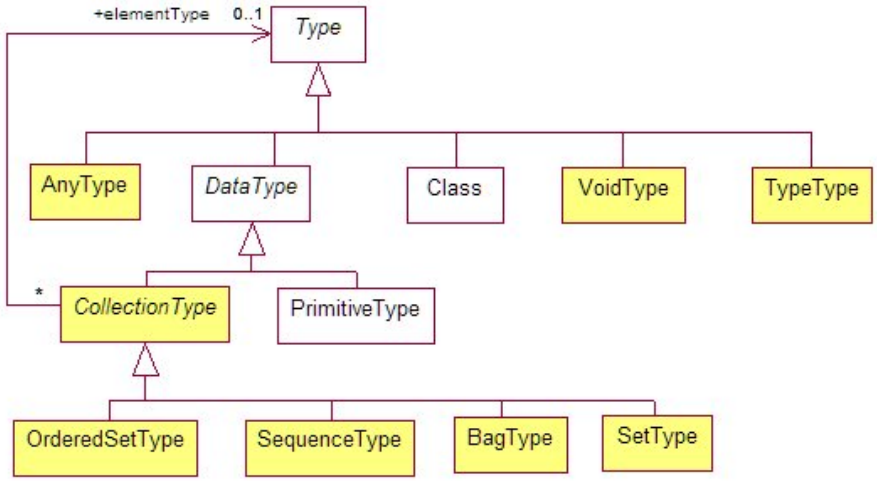


Figure 6: Types

The following types are not used:

- TupleType: TupleType (informaly known as record type or struct) combines different types into a single aggregate type
- InvalidType: In OCL, the only instance of InvalidType is Invalid, which is further defined in the OCL standard library. Furthermore Invalid has exactly one runtime instance called OclInvalid. In OCL, the invalid value is returned when invalid expressions are evaluated, such as division of zero for instance. In PRR OCL, the result of the evaluation of an invalid expression is not specified and is specific of the implementation.

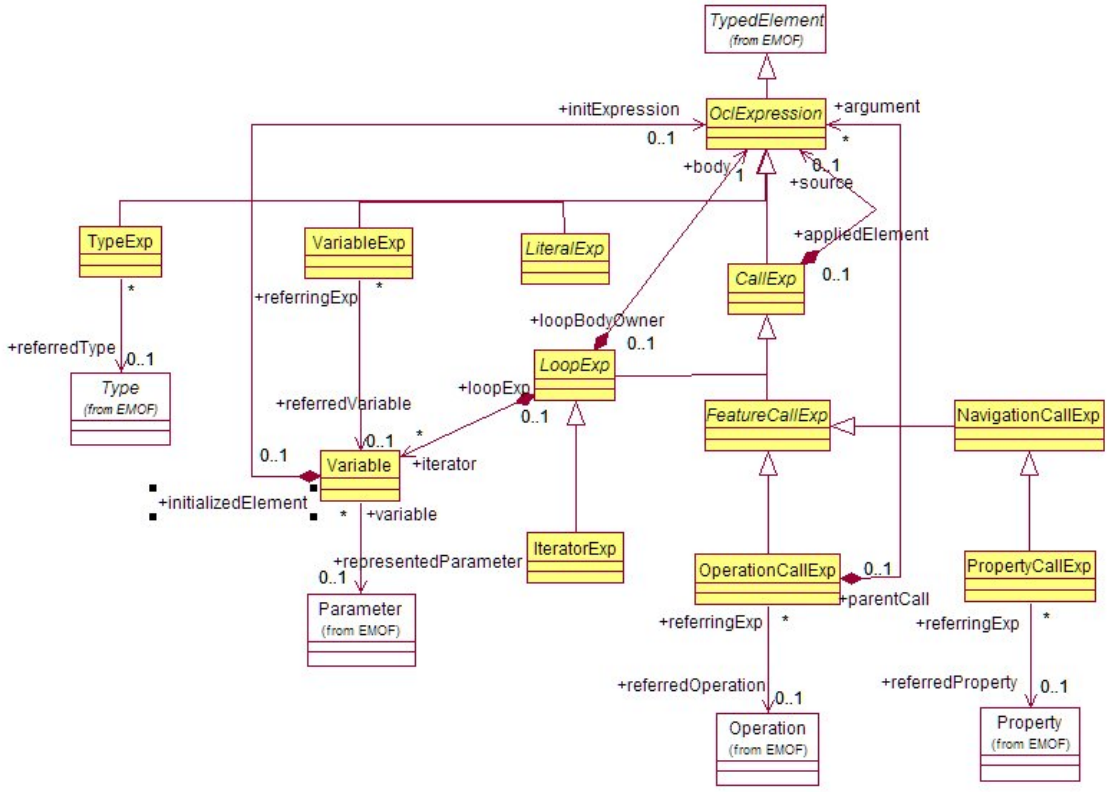


Figure 7: OCL Expressions

The following OCL expressions are not used:

- IfExp: the semantic of if-then-else expression is redefined by the rule structure itself.
- IterateExp: IteratorExp is sufficient for the PRR OCL use
- LetExp: RuleVariable must be used to define variable
- TupleLiteralExp: the tuple type is not used
- InvalidLiteralExp. The invalid type is not used
- UnlimitedNaturalExp: this expression is used to encode the upper value of a multiplicity specification. It is not used in the production rule expression.
- CollectionLiteralExp: the PRR OCL does not authorize defining new collection.

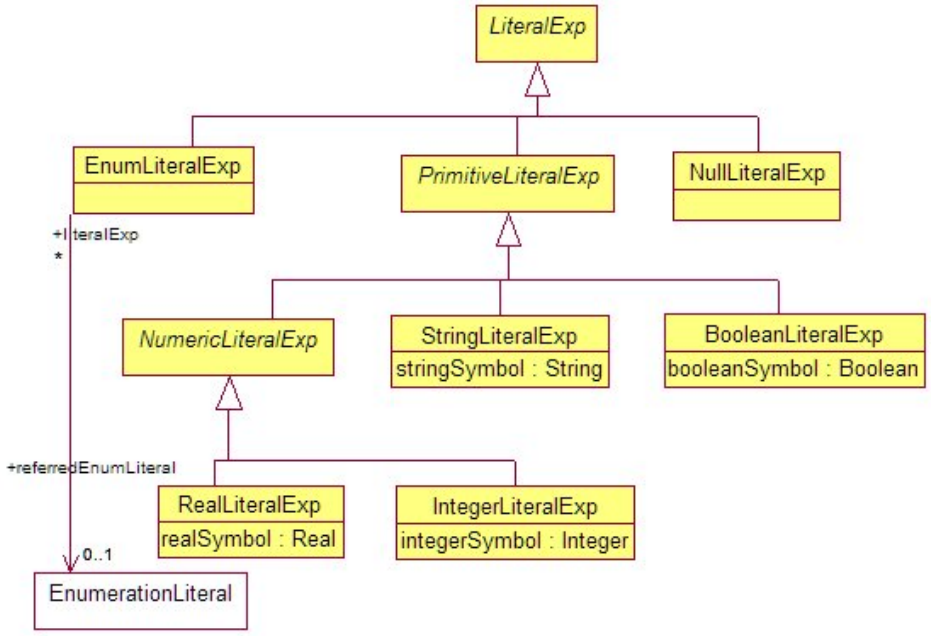


Figure 8: Literals

3.6.1 Classes used in PRR OCL

Name	Description
ProductionRule::RuleVariable	The variable represents a programming construct to hold values for use in executing a rule. The values must conform to the variable's type.
ProductionRule::RuleCondition	The condition represents a Boolean expression that is matched against available data to determine whether a ProductionRule can be instantiated
ProductionRule::RuleAction	The action association defines an ordered list of actions.
ProductionRuleOCL::ImperativeExp	A rule action expression, abstract class.
ProductionRuleOCL::AssignExp	A subclass of ImperativeExp that assigns a value to an expression
ProductionRuleOCL::InvokeExp	A subclass of ImperativeExp that invokes an operation and passes values in as parameters
ProductionRuleOCL::UpdateStateExp	An abstract subclass of ImperativeExp that updates the state of the rules engine only
ProductionRuleOCL::AssertExp	A subclass of UpdateStateExp that adds an object to the execution context of the engine
ProductionRuleOCL::RetractExp	A subclass of UpdateStateExp that removes an object from the execution context of the engine

Name	Description
ProductionRuleOCL::UpdateExp	A subclass of UpdateStateExp that changes an object in the execution context of the engine
UML2:: BasicBehaviors::Operation	See UML2
UML2:: OCL::OclExpression	See UML2

3.6.2 RuleVariable

A *RuleVariable* is associated to a *FilterExpression* used to specify a collection and/or filter for the domain represented by the RuleVariable. This section describes how PRR OCL can be used to define the FilterExpression.

The general structure of the FilterExpression, written in an OCL like syntax, is:

***dataSource* → *operator* (*iterator* | *body*)**

The components are:

- dataSource: the source of data on which the filter must be applied.
- operator: there are two possible values
  - `any`: return one element of the dataSource for which body is true. At runtime, the rule variable will be bound to all the possible elements. The type of the return value must be compatible with the type of the rule variable.
  - `select`: return the subset of the dataSource for which body is true. The return value is a Set.
- iterator: the iterator variable. This variable is bound to every element value of the *source* collection while evaluating the *body* expression
- body: a boolean expression

The following example defines, in an OCL like syntax, a ruleset with an input parameter and a rule with an *`item`* rule variable, no condition and a simple action that print out the name of the type of the filtered items.

```
ruleset ruleset1(in scart : ShoppingCart) :

rule r1
ruleVariable :
      item : Item = scart.items->any(e: Item | e.type=ItemType.CD);
action:
    Util.println(item.name);
```

Although this looks like valid OCL syntax, it would not be executable as such. The term “any” here has a different execution over what would be expected in pure OCL. In PRR OCL this means “each in turn”.

At runtime, the *`item`* rule variable will be associated with each Item found on the ShoppingCart that match the test. If the items associated to the shopping cart instances given as input to the

ruleset are for instance: *cd1 [CD], book1 [Book], cd2[CD]* then the result of the execution will be:

cd1

cd2

The following example defines a ruleset with an input parameter and a rule with an *items* rule variable that is bound to the collection of shopping cart items that match the given test and with an action that prints out the size of the collection.

```
ruleset ruleset2(in scart : ShoppingCart) :

rule r1
ruleVariable :
      items : Set = scart.items->select(e: Item | e.type=ItemType.CD);
action:
     Util.println(items.size());
```

At runtime, the *items* rule variable will be associated to the set of items found on the ShoppingCart and that match the given test. If the items associated to the shopping cart instances given as input to the ruleset are for instance: *cd1 [CD], book1 [Book], cd2[CD]* then the result of the execution will be:

2

In the PRR OCL metamodel, a FilterExpression maps to an *IteratorExp* instance.

The following restrictions apply:

- the IteratorExp must have at most one iterator variable.
    - The type of the iterator variable must be the same at the type of the rule variable when the “any” operator is used.
    - When the “select” operator is used, it is assumed that the type of the elements of the collection is the same as, or included in, the type of the rule variable.
- No CallExp can be applied on IteratorExp in the RuleVariable part
    - the IteratorExp is exclusively used to represent binding. The RuleVariable definition needs to be simple to allow the rule engine, at runtime, to update its state when the instances of the collection are modified.
    - Operation on collection are therefore not authorized on rule variable. It if for instance not possible to write *shoppingCarts->collect(items)* or its implicit form *shoppingCarts.items*.
- No CallExp can use an IteratorExp

Figure 9 shows the abstract syntax of the first example above (the rule action part is not detailed).

Deleted:

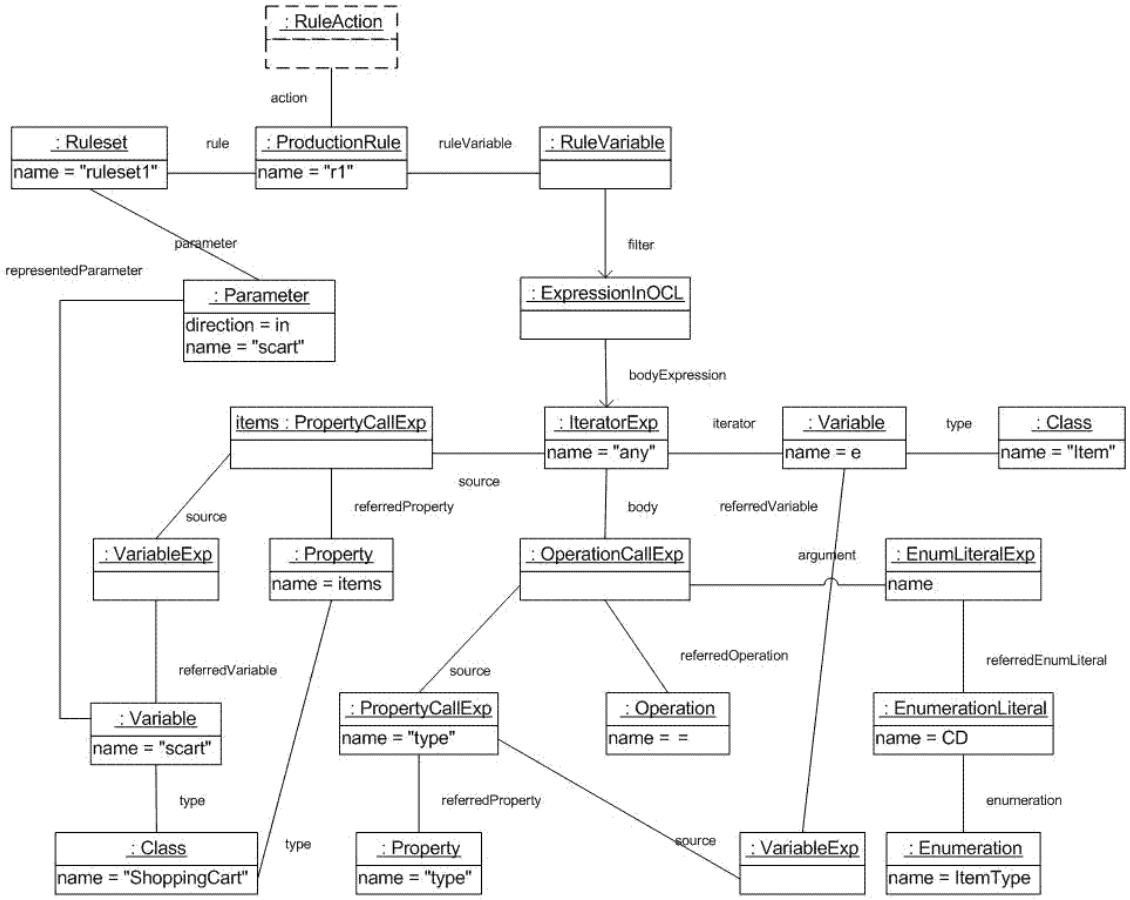


Figure 9 An example of Rule Variable abstract syntax

In OCL, the operators '+', '-', '*'. '/', '<', '>', '<>' '<=' '>=' are used as infix operators. It means, for instance, that the expression *a < b* is conceptually equivalent to the expression *a.<(b)*.

This explains why the "e.type" expression is used as source in the Figure 9.

Deleted:

3.6.3 RuleCondition

The Rule condition represents a Boolean expression that is matched against available data to determine whether a production rule's actions can be executed.

In PRR OCL, rule conditions are defined using a BooleanLiteralExp.

The following restrictions apply:

- IteratorExp cannot be used in RuleCondition: IteratorExp are only used to represent rule variables.

PRR OCL does not provide special operations on collections. Collections are treated as instances like any other objects.

Collections in production rules are handled in a different way than in OCL. For instance, the test to check that the city of at least one address of one of the customers of a company is "Paris" could be written like this in OCL, assuming a forward-chaining implementation:

```
context Company
inv : self.customers.addresses->exists(p : Address | p.city = 'Paris'
)
```

In PRR OCL this could be modelled as follows:

```
ruleset ruleset3(in company : Company) :

variable:
    parisCust : List = Util.createList();

rule r1
ruleVariable :
  customer : Customer = company.customers->any();
  addresses : Set = customer.addresses->select(p : Address | p.city =
    'Paris');
condition :
  addresses.size() > 0 and not parisCust.contains(customer);
action:
  parisCust.add(customer);

rule r2
ruleVariable :
  customer : Customer = company.customers->any();
  addresses : Set = customer.addresses->select(p : Address | p.city =
    'Paris');
condition :
  addresses.size() = 0 and parisCust.contains(customer);
action:
  parisCust.remove(customer);

rule r3
condition :
  parisCust.size() = 0;
action:
  Util.sendMessage("There is no customer of company with an address in
  Paris");
```

`r1` and `r2` maintain the list of customers that have at least one address in Paris. `r3` sends a message when there is no customer that has an address in Paris. With this design the check is performed when the number of customer change, the number of address change or an address is modified.

3.6.4 RuleAction

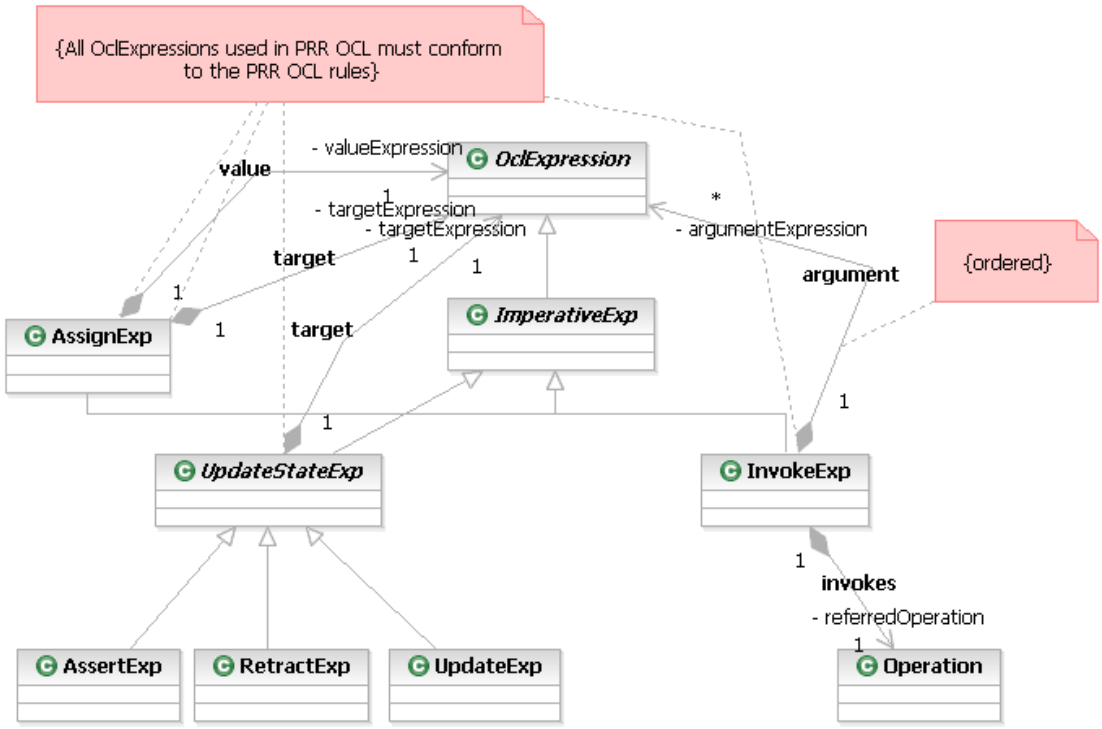


Figure 10: PRR OCL Actions

The rule action part defines an ordered list of actions. These actions may update objects within the domain of a ruleset invocation (data source) or make some external invocation.

The metamodel needed to represent actions must be simple. Three different actions have been selected:

- Update State Expression
  An abstract class of actions that impact the scope of the engine.
  - Assert: Add an object to the scope of the engine
    The only behavior that can we can be sure of and so the only semantic we can describe is that an object is added to the engine. This object maybe newly created or already existing into the system but this is not in the scope of the rule engine.
  - Retract: remove an object from the scope of the engine
    Again the only semantic that we can describe and that is meaningful to the engine is that an object is or is not in the scope of the engine.
  - Update: notification of an object change
    Some operations modify the state of objects and others do not. If the modified objects are in the scope of the engine, the engine must be notified that the objects state has been modified to be able to compute the list of eligible rules. It is not possible from the operation call to determine automatically what objects will be modified so it may

be necessary in the rule to explicitly notify the engine.
We can assume that the notification is done by the application but in that case:
- It is intrusive on the application: the method definition must integrate notification code
- The definition of the rule is not complete: the semantic and so the execution effect depends on code that exists outside of the rule.

- Invoke: operation call – may require associated add, remove, update actions.
- Assign: assign a value to a variable or a property value – includes any relevant update action. The assign operation handle both single valued and multi valued properties.

BasicOCL is extended to provide new types of expression for PRR OCL. This extension is consistent with the way conditions are defined and is similar to the solution that has been chosen in QVT Specification (ptc/05-11-01). Later we can extend the action part by supporting other operations as required.

Note: Some BasicOCL extensions used below are similar to the extensions of the same name specified in the OMG MOF QVT specification.

3.6.5 Class ImperativeExp

3.6.5.1 Description

The *imperative expression* is an abstract concept serving as the base for the definition of all side-effect oriented expressions defined in this specification. Its superclass is OCLExpression.

3.6.5.2 Attributes

No additional attributes defined.

3.6.5.3 Associations

None.

3.6.5.4 Constraints

No additional constraints defined.

3.6.5.5 Semantics

None.

3.6.6 Class AssignExp

3.6.6.1 Description

An *assignment expression* represents the assignment of a variable or the assignment of a Property.

3.6.6.2 Attributes

None.

3.6.6.3 Associations

- `value : OclExpression [1]`
  The expression to be evaluated in order to assign the variable or the property.

- `target : OclExpression [1]`
  The “left hand side” expression of the assignment. Should reference a variable or a property that can be updated.

3.6.6.4 Constraints

The target expression must be a either a VariableExp or a PropertyCallExpr.

The target expression must NOT be a RuleVariable.

The value type must conform to the type of the target.

The value expression must be a PRR OCLExpression.

3.6.6.5 Semantics

In this description we refer to "target field" the referred variable or property.

If the variable or the property is monovalued, the effect is to reset the target field with the new value. If it is multivalued, the effect is to reset the field with the value of the new collection.

An assignment expression returns the assigned value.

3.6.7 Class InvokeExp

3.6.7.1 Description

An InvokeExp refers to an operation defined in a Classifier. The expression may contain an ordered list of argument expressions if the operation is defined to have parameters. In this case, the number and types of the arguments must match the parameters.

3.6.7.2 Attributes

None

3.6.7.3 Associations

- `argument : OclExpression [*]`
  The arguments denote the arguments to the invoke expression. This is only useful when the invoked operation is related to an Operation that takes parameters.
- `referredOperation : Operation [1]`
  The Operation to which this InvokeExp is a reference. This is an Operation of a Classifier that is defined in the UML model.

3.6.7.4 Constraints

No additional constraints defined.

3.6.7.5 Semantics

In this description we refer to "target field" the referred variable or property.

If the variable or the property is monovalued, the effect is to reset the target field with the new value. If it is multivalued, the effect is to reset the field with the value of the new collection.

An assignment expression returns the assigned value.

3.6.8 Class UpdateStateExp

3.6.8.1 Description

UpdateStateExp is an abstract concept serving as the base for the update state expressions: assert, retract and update.

3.6.8.2 Attributes

None.

3.6.8.3 Associations

- `target : OclExpression [1]`
  The expression that return the target object to assert, extract or update.

3.6.8.4 Constraints

None

3.6.8.5 Semantics

3.6.9 Class AssertExp

3.6.9.1 Description

AssertExp represents the addition of an object to the scope of the engine.

3.6.9.2 Attributes

None

3.6.9.3 Associations

None

3.6.9.4 Constraints

None

3.6.9.5 Semantics

If the object returned by the OCL Expression exists and is known to the rule engine then AssertExp does nothing

If the object returned by the OCL Expression does not exists then it is created and added to the scope of the rule engine

If the object returned by the OCL Expression exists but is not known to the rule engine then AssertExp adds the object to the scope of the rule engine.

3.6.10 Class RetractExp

3.6.10.1 Description

RetractExp represent the removal of an object to the scope of the engine.

3.6.10.2 Attributes

None.

3.6.10.3 Associations

None

3.6.10.4 Constraints

None

3.6.10.5 Semantics

If the object returned by the OCL Expression exists and is known to the rule engine then RetractExp removes the object from the scope of the rules engine

If the object returned by the OCL Expression does not exist or is not known to the rule engine then RetractExp does nothing.

3.6.11 Class UpdateExp

3.6.11.1 Description

AssertExp represent the modification of an object that is managed by the engine.

3.6.11.2 Attributes

None.

3.6.11.3 Associations

None

3.6.11.4 Constraints

None

3.6.11.5 Semantics

If the object returned by the OCL Expression exists and is known to the rule engine then UpdateExp informs the rule engine that there is a new value.

If the object returned by the OCL Expression does not exist or is not known to the rule engine then UpdateExp does nothing.

3.7 PRR OCL: Standard Library

This section defines a library of predefined types and operations. Any implementation of PRR OCL must support these types and operations.

3.7.1 The OclAny, OclVoid types

The type OclVoid is a type that conforms to all other types. It has one single instance called null which corresponds with the UML NullLiteral value specification. Any property request on a null object is invalid.

All types in the UML model and the primitive types in the PRR OCL standard library comply with the type OclAny. Conceptually, OclAny behaves as a supertype for all the types except for the pre-defined collection types. Practically OclAny is used to define operations that are useful for every type of PRR OCL instance.

OclAny

= (object2 : OclAny) : Boolean

True if self is the same object as object2. Infix operator.

post: result = (self = object2)

<> (object2 : OclAny) : Boolean

True if self is a different object from object2. Infix operator.

post: result = not (self = object2)

oclAsType(typespec : OclType) : T

Evaluates to self, where self is of the type identified by typespec.

post: (result = self) and result.oclIsTypeOf(typeName)

oclIsTypeOf(typespec : OclType) : Boolean

Evaluates to true if the self is of the type identified by typespec. .

allInstances() : Set(T)

Returns all instances of self that have been added to the rule engine. Type T is equal to self..

pre: self.isKindOf(Classifier) -- self must be a Classifier

oclIsKindOf(typespec : OclType) : Boolean

Evaluates to true if the self conforms to the type identified by typespec.

3.7.2 OclType

The metaclass TypeType is used to represent the type accepted by the oclIsTypeOf and oclAsType operations. The TypeType has a unique instance named 'OclType'.

OclType

= (object : OclType) : Boolean

True if self is the same object as object.

<> (object : OclType) : Boolean

True if self is a different object from object.

post: result = not (self = object)

3.7.3 Primitive Types

The primitive types defined in the OCL standard library are Integer, Real, String and Boolean. They are all instance of the metaclass Primitive from the UML core package.

3.7.4 Real

Note that Integer is a subclass of Real, so for each parameter of type Real, you can use an integer as the actual parameter.

+ (r : Real) : Real

The value of the addition of self and r.

- (r : Real) : Real

The value of the subtraction of r from self.

*** (r : Real) : Real**

The value of the multiplication of self and r.

- : Real

The negative value of self.

/ (r : Real) : Real

The value of self divided by r. Evaluates to OclInvalid if r is equal to zero.

< (r : Real) : Boolean

True if self is less than r.

> (r : Real) : Boolean

True if self is greater than r.

post: result = not (self <= r)

<= (r : Real) : Boolean

True if self is less than or equal to r.

post: result = ((self = r) or (self < r))

>= (r : Real) : Boolean

True if self is greater than or equal to r.

post: result = ((self = r) or (self > r))

abs, floor, round, max, min are not required. They can be provided by the application.

3.7.5 Integer

- : Integer

The negative value of self.

+ (i : Integer) : Integer

The value of the addition of self and i.

- (i : Integer) : Integer

The value of the subtraction of i from self.

*** (i : Integer) : Integer**

The value of the multiplication of self and i.

/ (i : Integer) : Real

The value of self divided by i.Evaluates to OclInvalid if r is equal to zero

3.7.6 String

size() : Integer

The number of characters in self.

concat(s : String) : String

The concatenation of self and s.

post: result.size() = self.size() + string.size()

post: result.substring(1, self.size()) = self

post: result.substring(self.size() + 1, result.size()) = s

substring(lower : Integer, upper : Integer) : String

The sub-string of self starting at character number lower, up to and including character number upper. Character numbers run from 1 to self.size().

pre: 1 <= lower

pre: lower <= upper

pre: upper <= self.size()

toInteger() : Integer

Converts self to an Integer value.

toReal() : Real

Converts self to a Real value.

3.7.7 Boolean

or (b : Boolean) : Boolean

True if either self or b is true.

and (b : Boolean) : Boolean

True if both b1 and b are true.

not : Boolean

True if self is false.

post: if self then result = false else result = true endif

xor(Boolean) and implies(Boolean) are not required. The application can provide them if needed.

3.7.8 Collection-Related Types

3.7.8.1 Collection

size() : Integer

The number of elements in the collection self.

post: result = self->iterate(elem; acc : Integer = 0 | acc + 1)

includes(object : T) : Boolean

True if object is an element of self, false otherwise.

post: result = (self->count(object) > 0)

includesAll(c2 : Collection(T)) : Boolean

Does self contain all the elements of c2 ?

post: result = c2->forAll(elem | self->includes(elem))

isEmpty() : Boolean

Is self the empty collection?

post: result = (self->size() = 0)

excludes(object : T) : Boolean

True if object is not an element of self, false otherwise.

post: result = (self->count(object) = 0)

excludesAll(c2 : Collection(T)) : Boolean

Does self contain none of the elements of c2 ?

post: result = c2->forAll(elem | self->excludes(elem))

3.7.8.2 Set

union(s : Set(T)) : Set(T)

The union of self and s.

post: result->forAll(elem | self->includes(elem) or s->includes(elem))

post: self ->forAll(elem | result->includes(elem))

post: s ->forAll(elem | result->includes(elem))

union(bag : Bag(T)) : Bag(T)

The union of self and bag.

post: result->forAll(elem | result->count(elem) = self->count(elem) + bag->count(elem))

post: self->forAll(elem | result->includes(elem))

post: bag ->forAll(elem | result->includes(elem))

= (s : Set(T)) : Boolean

Evaluates to true if self and s contain the same elements.

post: result = (self->forAll(elem | s->includes(elem)) and

s->forAll(elem | self->includes(elem)))

3.7.8.3 OrderedSet

append (object: T) : OrderedSet(T)

The set of elements, consisting of all elements of self, followed by object.

post: result->size() = self->size() + 1

post: result->at(result->size()) = object

post: Sequence{1..self->size() }->forAll(index : Integer |

result->at(index) = self ->at(index))

prepend(object : T) : OrderedSet(T)

The sequence consisting of object, followed by all elements in self.

post: result->size = self->size() + 1

post: result->at(1) = object

post: Sequence{1..self->size()}->forAll(index : Integer |

self->at(index) = result->at(index + 1))

insertAt(index : Integer, object : T) : OrderedSet(T)

The set consisting of self with object inserted at position index.

post: result->size = self->size() + 1

post: result->at(index) = object

post: Sequence{1..(index - 1)}->forAll(i : Integer |

self->at(i) = result->at(i))

post: Sequence{(index + 1)..self->size()}->forAll(i : Integer |

self->at(i) = result->at(i + 1))

subOrderedSet(lower : Integer, upper : Integer) : OrderedSet(T)

The sub-set of self starting at number lower, up to and including element number upper.

pre : 1 <= lower

pre : lower <= upper

pre : upper <= self->size()

post: result->size() = upper -lower + 1

post: Sequence{lower..upper}->forAll(index |

result->at(index - lower + 1) =

self->at(index))

at(i : Integer) : T

The i-th element of self.

pre : i >= 1 and i <= self->size()

indexOf(obj : T) : Integer

The index of object obj in the sequence.

pre : self->includes(obj)

post : self->at(i) = obj

3.7.8.4 Bag

= (bag : Bag(T)) : Boolean

True if self and bag contain the same elements, the same number of times.

post: result = (self->forAll(elem | self->count(elem) = bag->count(elem)) and

bag->forAll(elem | bag->count(elem) = self->count(elem)))

union(bag : Bag(T)) : Bag(T)

The union of self and bag.

post: result->forAll(elem | result->count(elem) = self->count(elem) + bag->count(elem))

post: self ->forAll(elem | result->count(elem) = self->count(elem) + bag->count(elem))

post: bag ->forAll(elem | result->count(elem) = self->count(elem) + bag->count(elem))

union(set : Set(T)) : Bag(T)

The union of self and set.

post: result->forAll(elem | result->count(elem) = self->count(elem) + set->count(elem))

post: self ->forAll(elem | result->count(elem) = self->count(elem) + set->count(elem))

post: set ->forAll(elem | result->count(elem) = self->count(elem) + set->count(elem))

3.7.8.5 Sequence

= (s : Sequence(T)) : Boolean

True if self contains the same elements as s in the same order.

post: result = Sequence{1..self->size()}->forAll(index : Integer | self->at(index) = s->at(index)) and self->size() = s->size()

union (s : Sequence(T)) : Sequence(T)

The sequence consisting of all elements in self, followed by all elements in s.
post: result->size() = self->size() + s->size()

post: Sequence{1..self->size()}->forAll(index : Integer |
self->at(index) = result->at(index))

post: Sequence{1..s->size()}->forAll(index : Integer |
s->at(index) = result->at(index + self->size())))

append (object: T) : Sequence(T)

The sequence of elements, consisting of all elements of self, followed by object.

post: result->size() = self->size() + 1
post: result->at(result->size()) = object
post: Sequence{1..self->size() }->forAll(index : Integer |
result->at(index) = self ->at(index))

prepend(object : T) : Sequence(T)

The sequence consisting of object, followed by all elements in self.

post: result->size = self->size() + 1
post: result->at(1) = object
post: Sequence{1..self->size()}->forAll(index : Integer |
self->at(index) = result->at(index + 1))

insertAt(index : Integer, object : T) : Sequence(T)

The sequence consisting of self with object inserted at position index.

post: result->size = self->size() + 1
post: result->at(index) = object
post: Sequence{1..(index - 1)}->forAll(i : Integer |
self->at(i) = result->at(i))
post: Sequence{(index + 1)..self->size()}->forAll(i : Integer |
self->at(i) = result->at(i + 1))

at(i : Integer) : T

The i-th element of sequence.

pre : i >= 1 and i <= self->size()

indexOf(obj : T) : Integer

The index of object obj in the sequence.

pre : self->includes(obj)
post : self->at(i) = obj

3.8 Conformance

The Production Rule Representation contains both a PRR Core subset and a PRR OCL set of classes.

The following compliance points are distinguished for both parts

- Syntax compliance
  The tool can read and write PRR definitions in accordance with the grammar, including validating its type conformance and well-formedness rules.
- Execution compliance
  The tool executes PRR definitions (potentially translated to an internal representation) in accordance with their semantics

The following table shows the possible compliance points. Each tool is expected to fill in this table to show which compliance points are supported.

Table Overview of Compliance points

	PRR Core	PRR OCL
Syntax		
Evaluation		

Note: No XMI conformance conditions are specified as the intention is to use W3C RIF for interchange.

4 Comparison with other OMG Standards

4.1 UML

4.1.1 UML Activities

UML Activities can coordinate the execution of Behaviors and, as Production Rulesets are implementations of Behaviors, Activities can thus coordinate the execution of Production Rulesets. A future version of PRR may well specialize Activities to manage “Decisions” made up of multiple, coordinated rulesets. Many commercial rule engine products use a “ruleflow” construct for this that have clear similarities with Activities.

4.1.2 UML Events

Because Production Ruleset are specializations of Behavior they can be invoked by Event in the same way as other subclasses of Behavior. Similarly, because RuleAction supports Operation invocation, they can cause instances of Event to be created by invoking a suitable Operation.

4.2 Alignment with MDA - Model Driven Architecture

The Production Rule Representation represents a Platform-Independent Model (PIM) for the representation of production rules in UML. It is targeted to the production rule engine class-of-platform that is in wide use around the world and is independent of a vendor specific engine. The PRR is further limited to specifying requirements for representing production rules targeted at the two most popular forms of rules engine – forward-chaining / inferencing and procedural engine class-of-platforms. These two types cover all ranges of solutions from complex decision-making to supporting Business Process Management.

Production rule engine vendors will be able to provide a mapping from the PRR PIM to the PSM specific to their products, depending on whether procedural or Inferencing rules are specified and whether they support those types.

4.3 Alignment with OCL - Object Constraint Language

OCL provides a very rich expression language that specifies query operations on a model. OCL however is side-effect free, and therefore does not provide support for the *direct* method invocation of methods that change the state of the system, as required by the actions of a production rule. The critical concept is that of “direct method invocation.” OCL 2.0 does permit reference to operations that change the state of the system in a constraint expression, but the semantics of such a reference is that the operation *will have been* invoked when the truth of the constraint is tested. This semantics, which is permitted only in postconditions, does not satisfy the requirements of the action clause of production rules, which cannot be used as postconditions of operations.

OCL is not used as a syntax for business rule management vendors.

However, re-using the syntax of OCL and redefining the semantics for postconditions allows a derivative of OCL to be used to represent the expressions used in production rules.

4.4 Alignment with Action Semantics

The need to represent behaviors with side effects, such as method invocations in action clauses of production rules, gives rise to the possibility of modeling production rules using action semantics. Indeed, action semantics ready supports statements of the form "If condition, then action". However, there are several points at which the semantics of production rules mismatch action semantics.

- Execution semantics. Action semantics allows two modes for the execution of action statements: parallel execution and sequential execution based on explicitly modeled control flows or data flows between action statements. Inference rules lack explicit modeling of sequence. Indeed, the point of modeling a problem, or decision, space with inference rules is to avoid the need to specify the sequence of rule execution beyond the semantics of the rule statements themselves. The inference engine can be viewed as handling their actual sequencing based on run-time conditions. Note that inferencing behavior defines rule execution order in a data driven, a priori fashion.
- Multiple quantified expressions. Action semantics provides for expressions that yield a set of instances of a classifier (e.g., ReadExtentAction). However, action semantics does not support the use of multiple quantifiers within the same expression; that is, it does not support expressions that yield sets of tuples. For example, within action semantics one cannot easily or clearly write a statement of the following form, which mimics a common production rule structure in the action language TALL:

```
foreach instance a of Applicant and foreach instance r of Residence
    [a.unassigned and r.available and suitableFor(a, r) {
            assignTo(a, r);
        }
    ]
```

Operating with sets of tuples is essential for handling pattern-matching inference rules, which are fundamental to such inferencing algorithms as the Rete algorithm.

4.5 Semantics of Business Vocabulary and Business Rules (SBVR)

The SBVR specification essentially defines two metamodels in the form of "vocabularies": the SBVR "Vocabulary for Describing Business Vocabularies" [henceforth called *business vocabulary metamodel*], and the SBVR "Vocabulary for Describing Business Rules" [henceforth called *business rules metamodel*], which builds on the *Vocabulary for Describing Business Vocabularies*.

A business vocabulary is defined to contain "all the specialized terms and definitions of concepts that a given organization or community uses in their talking and writing in the course of doing business". A business rule is defined as "a rule that is under business jurisdiction", which means that "the business can enact, revise, and discontinue business rules as it sees fit".

The SBVR business vocabulary metamodel is rather large with more than one hundred concept definitions. The SBVR business rule metamodel, containing 33 concept definitions, is more handy but still sizeable.

SBVR being an OMG CIM standard and PRR being an OMG PIM standard, it is natural to expect that there will be guidelines how to derive a PRR PIM from an SBVR CIM. They should include guidelines how to derive a UML design class model (sometimes also called 'Business Object Model') from an SBVR business vocabulary.

The SBVR submission contains a discussion about how to represent a business vocabulary visually in the form of a UML class diagram. The method considers fewer than 20 concepts from the more than 100 concepts of the SBVR business vocabulary metamodel. It is not discussed if such a radical reduction in expressivity creates any problems or not. Also, the resulting class diagram corresponds rather to a UML domain model (or CIM), and not to a design model.(or PIM) because it

- does not contain data types for attributes
- does not include multiplicity elements (which would have to be derived from corresponding "structural business rules")
- does not follow standard naming conventions for design models (e.g. using names starting with upper case for classes and names starting with lower case for properties and associations)
- may contain powertypes (which are typically not used in PIM-level models)

The SBVR submission does not say much about how to derive PIM-level rule expressions from SBVR business rule statements. In fact, it is unclear if any of the conceptual distinctions of the SBVR business rule metamodel can be preserved in a PRR rule model.

One option to consider in any possible RFP for mapping SBVR to PRR is to use OCL invariants and derive expressions as an intermediate representation from which a PRR rule model may be derived.

Despite the difficulties inherent in transforming SBVR to PRR there is clear value in providing traceability between the two standards. Such traceability would allow impact analysis ("which PRR Rulesets are impacted if this SBVR rule is changed") and reduce costs of ongoing maintenance. Such traceability is being actively discussed.

4.6 Business Process Definition Metamodel (BPDM)

BPDM involves developing Activity Models to represent Behavior. Similarly, PRR involves developing rule models to represent Behavior. The integration of these two complimentary approaches can be achieved through standard Behavior modeling.

4.7 Ontology Definition Metamodel (ODM)

Ontologies are used to define Class models that are then used by PRR. As such, ODM represents a possible preparatory process in the production of rules in PRR.

4.8 Enterprise Distributed Object Computing (EDOC) and Enterprise Collaboration Architecture (ECA)

The use of production rules to represent business decision logic associated with UML class diagrams represents the next stage in the evolution of software engineering best practices as previously defined by EDOC and ECA.

5 References

5.1 References Specific to the PRR

[PRR RFP] Production Rule Representation Request For Proposal, *http://www.omg.org/cgi-bin/doc?br/03-09-03.pdf*

[RETE] Discussion of Rete algorithm, Forgy, C. L., "Rete: A Fast Algorithm for the Many Pattern/Many Objects Pattern Match Problem", Artificial Intelligence, 19(1982), pp. 17-37.

5.2 General References

The following documents may be useful to readers of this proposal.

"MDA: The Architecture of Choice for a Changing World", http://www.omg.org/mda.

The OMG Hitchhiker's Guide, Version 6.1, *http://www.omg.org/cgi-bin/doc?omg/2002-03-03*

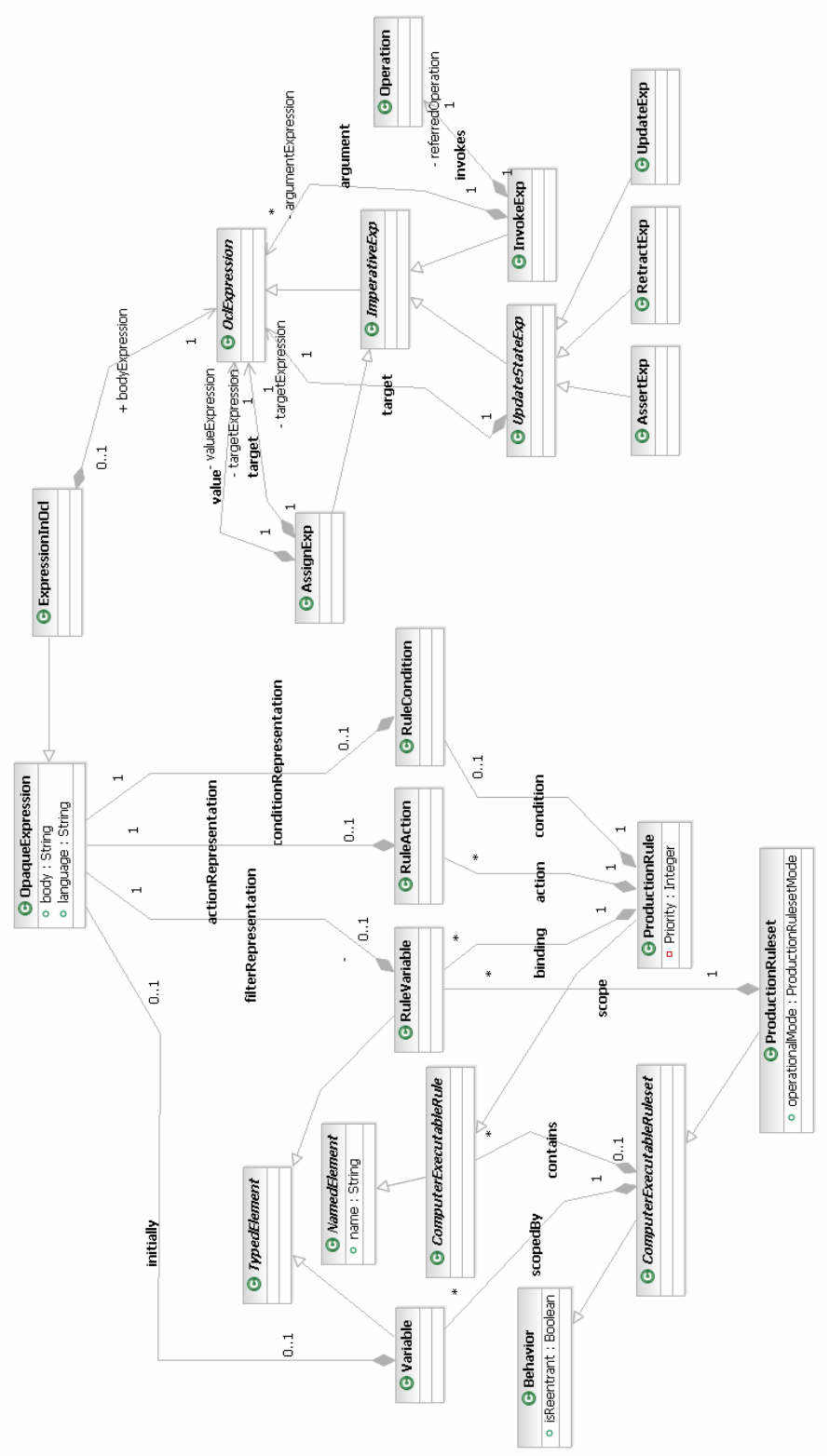
Meta Object Facility Specification, *http://www.omg.org/technology/documents/formal/mof.htm*

RuleML Draft Metamodels (unpublished)

Unified Modeling Language Specification, *http://www.omg.org/technology/documents/formal/uml.htm*

XML Metadata Interchange Specification, *http://www.omg.org/technology/documents/formal/xmi.htm*

A Complete Metamodel



B Glossary

Backward chaining – A recursive algorithm for executing production rules. Also known as ***goal-driven reasoning***, backward chaining seeks to establish a value of an attribute (or "goal") by ascertaining the truth of the conditions of production rules whose action assigns a value to the attribute. Unknown attributes in those conditions are considered subgoals and are similarly pursued.

Business rule – According to the GUIDE definition, "A Business Rule is a statement that defines or constrains some aspect of the business" [GUI] The traditional taxonomy of business rules classifies business rules into (business) terms, facts, and rules. Rules may be further classified as constraints, derivations (e.g., inference and computation rules), and triggers. (An industry-accepted standard classification of rules is not available at the present time.)

Forward chaining – A class of algorithms for executing production rules. Also, known as ***data-driven reasoning***, forward chaining executes production rules by testing whether their condition is true. Simple forward chaining is used to assign attribute values based on other attribute values. More complex forward chaining algorithms support first-order predicate calculus, i.e., quantification over instances of classes, and are executed by means of the Rete algorithm.

Inference engine – Software that provides an algorithm or set of algorithms, such as backward and/or forward chaining, for executing production rules.

Production rule – A production rule is an independent statement of programming logic of the form IF Condition, THEN Action that is executable by an inference engine.

Rete algorithm – Meaning 'net', the Rete algorithm creates a network that computes the path (relationships) between the conditions in all the rules. The Rete algorithm is intended to improve the speed of forward-chaining rule systems by limiting the effort required to recompute the rules available for firing after a rule is fired.

Rule engine – As a general category, rule engine refers to any software that executes rules. In this sense, inference engines are a type of rule engine.

C Guidance for Users

This section describes the expectations of the authors in terms of usage of the Production Rule Representation. This represents “guidelines” only, and is not a normative part of the PRR specification.

1. The PRR metamodel is targeted at UML and business rule modeling vendors, to incorporate production rules in models to support the separation of business logic from business objects. Example use cases for the use of a PRR model are:
    a. <User> specifies a <use case> with business rules defined separately. The following approaches for rules may be used:
        i. Define rules in an informal language as “lists of rules”, annotating the use case.
        ii. Define rules in a formal language mechanism such as OMG SBVR, without any computation context.
        iii. Define rules in a production rule format, with natural language conditions, using PRR Core. Although the conditions and actions will need to be translated to a rule language, the basic structure will be PRR compliant and ease transformations in the development phase.
        iv. Define rules in a production rule format with an existing class model, using PRR OCL and a supporting tool that creates the OCL expressions automatically for the user. In this case, the production rules will be very close to their executable form, subject to the transformations required from use case to design to implementation.
    b. <User> annotates a <class diagram> with required behavior in the form of PRR rules.
        i. At the CIM level, define rules in a formal language mechanism such as OMG SBVR. This will require the mapping of the class model to the appropriate MDA-CIM level constructs for reference in the formal language statements, although tools may provide this automatically. After rule modeling, the appropriate transformations to different types of rules (as well as other behaviors) may be carried out for PIM-level modeling.
        ii. Define rules in a production rule format, with vendor-specific conditions and actions, using PRR Core extended with a vendor condition and action language. Normally the vendor-specific condition and action language will be specified as a “high level language”. However, this use is at an MDA PSM level due to the use of a platform-specific rule language.
        iii. Define rules in a production rule format using PRR OCL and a supporting tool that creates the OCL expressions automatically for the user. In this case, the production rules will be suitable for transformation to a number of different engines in a true MDA PIM format.
2. Other tool types that may choose to implement PRR for MDA compatibility and vendor flexibility at deployment. Such tools may choose a more execution-oriented approach (i.e. OMG PIM layer) rather than the CIM level provided by SBVR. These are:
    a. Enterprise Architecture and Business Modeling tools: these often allow the definition of UML class models and are aimed at business modelers who need to specify behavior.
    b. Business Process Modeling and Management tools: these often define process entities and activity-based behavior that can often be better represented as or augmented by discrete production rules.
    c. Business rule specification tools that, for example, develop SBVR rulesets. Although such tools may do MDA transformations direct to procedural and Object Oriented code, they could also benefit from the intermediate step of PRR-based declarative production rule transformations.
    d. Business Rule Management Systems: these implement vendor specific MDA-like transformations between business language specifications to production rules, and are almost by definition likely to be PRR Core compatible. Although PRR OCL may appear a backward step for such BRMS users, it is likely to be useful for tool interchange until the advent of other technologies for rule interchange (eg W3C RIF for PR).

3. OMG UML developers that are conversant with OCL may also edit and define PRR OCL rules directly in their UML tool of choice.

D Relationship with W3C Rule Interchange Format

In November 2005, the World Wide Web consortium (W3C) chartered the Rule Interchange Format (RIF) working group to specify a format for rules that can be used across diverse systems. This format (defined as a language) will function as an interlingua into which both established and new rule languages can be mapped, allowing rules written for one application to be published, shared, and re-used in other applications and other rule engines.

Because of the great variety in rule languages and rule engine technologies used in academia and emerging technologies, this common format will take the form of a Core language to be used with a set of standard and non-standard extensions. The RIF working group is chartered to first establish the extensible core and possibly a set of initial extensions, and then in a second phase to begin specifying additional extensions based on user requirements, including more common commercial rule systems. These extensions need not, and are unlikely to, all be combinable into a single "unified rule language".

The primary normative syntax of the language must be an XML syntax. Users are expected to work with tools or rule languages that are transformed to and from this format.

In the current first phase, the essential task of the Working Group is to construct an extensible format for rules keeping in mind the various features, requirements and usage scenarios for rule languages (as described, e.g., in the working group's "Use Cases and Requirements" document[1]), to be sure the right kind of extensibility is in place. In order to permit meaningful rule interchange as early as possible, the (Phase 1) core language's rule semantics will be essentially Horn Logic, a well-studied sublanguage of First-Order Logic, which is the basis of Logic Programming, which is especially common amongst semantic web researchers.

The working group plans, at least, to define dialects that extend the core language for production rule interchange, on the one hand; and for interchanging rules in more expressive logic programming languages (e.g. with negation explicitly defined in the condition part).

The Phase 1 of the W3C RIF working group ends in November 2007. At this time, the working group expects the RIF Core language specification to be a W3C Candidate Recommendation, and an initial draft of the production rule dialect to have been produced. The working group is then expected to be re-chartered for a Phase 2 focusing on a limited number of dialects extending RIF Core.

There is an overlap in scope between W3C RIF and PRR, and they share the goal of rule interoperability. There will be a useful division of labor as OMG focuses on the standard metamodel definition and modeling of production rules (and possibly other rule types), while W3C RIF group focuses on a Rule Interchange Format suitable for the real-time "Web" and users of "Web technologies" such as XML. The W3C working group appointed a liaison to work with PRR Core metamodel to maximize the value of these standards efforts in both groups. The liaison effort is effective because of considerable overlap in membership of the PRR and RIF

[1] http://www.w3.org/TR/2006/WD-rif-ucr-20060710/

groups. In addition, the RIF working group is encouraged (by charter[1]) to produce a document showing how these standards work together.

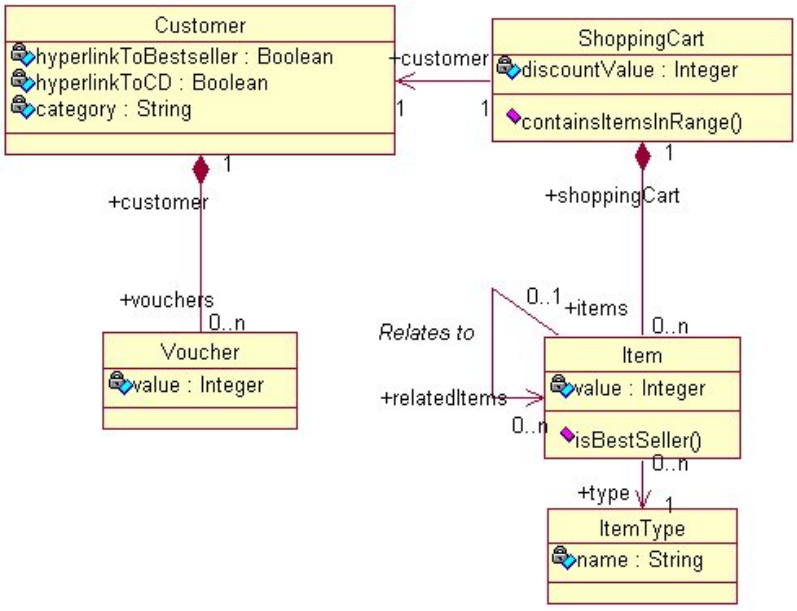
[1] http://www.w3.org/2005/rules/wg/charter#omg-prr

E Abstract Syntax Examples

The following describes an example mapping between a commercial rule engine syntax and PRR OCL.

E.1 Class Diagram

This section presents the UML class diagrams used to model the application rules.



E.2 Production rules with OCL translations

The rule in each of the following examples is first presented in its natural English form, then in the proprietary production rule language of one of the submitters, and, finally, in PRR OCL.

Example 1: Discount rule

English text:

If the shopping cart contains between 2 and 4 items and either the purchase value is greater than $100 and the customer category is gold or the purchase value is greater than $200 and the customer category is Silver then apply a 15% discount on the shopping cart value.

Proprietary rule language:

```
rule discount {
when
{
    ?customer: Customer();
    ?shoppingCart: ShoppingCart(customer == ?customer);
    evaluate((?shoppingCart.containsItemsInRange(2, 4)) &&
             (((((?shoppingCart.getValue() > 100d) &&
                    (?customer.category equals "Gold")) ||
                ((?shoppingCart.getValue() > 200d) &&
                    (?customer.category equals "Silver"))))));
}
then
{
    modify ?shoppingCart
    {
     shoppingCart.discountValue
                = shoppingCart.discountValue + 15f);
   }
}
}
```

PRR OCL:

Rule *discount*

ruleVariable:

```
    ?customer: Customer = Customer->any()
    ?shoppingCart: ShoppingCart =
          ShoppingCart->any(c: customer | c=?customer)
```

Condition:

```
    (?shoppingCart.containsItemsInRange(2, 4)
    and
    (((?shoppingCart.items->collect(i:Item|i.value))->sum()>100
        and
        ?customer.category == “Gold”)
     or
     ((?shoppingCart.items->collect(value))->sum() > 200
        and
        ?customer.category == “Silver”)))
```

Action:

```
    shoppingCart.discountValue = shoppingCart.discountValue+15f
```

Example 2: noCDItem rule

English text:

If there is no CD item in the customer shopping cart then add a hyperlink to the CD page in the customer web page.

Proprietary rule language:

```
rule noCDItem {
when
{
  ?customer1: Customer();
  ?shoppingCart1: ShoppingCart(customer == ?customer1);
  not Item(type == ItemType.CD ; shoppingCart == ?shoppingCart1);
}
then
{
  modify ?customer1{ hyperlinkToCD = true; }
}
}
```

PRR OCL:

Rule *noCDItem*

ruleVariable:

```
      ?customer: Customer = Customer->any()
      ?sCart: ShoppingCart = ShoppingCart-
>any(c:customer|c=?customer)
      ?cdItems: Set = ?sCart.items-
>select(e:items|e.type=ItemType.CD)
```

Condition:

```
      ?cdItems.isEmpty()
```

Action:

```
      ?customer.hyperlinkToCD = true
```

Example 3: atLeastOneBook rule

English text:

If there is at least one book item in the customer shopping cart and this book is a bestseller then add a hyperlink to the bestsellers page in the customer web page.

Proprietary rule language:

```
rule atLeastOneBook {
when
{
  ?customer1: Customer();
  ?shoppingCart1: ShoppingCart(customer == ?customer1);
  exists Item(shoppingCart == ?shoppingCart1 ; isBestseller());
}
then
{
  modify ?customer1 { hyperlinkToBestseller = true; }
}
}
```

PRR OCL:

Rule *atLeastOneBook*

ruleVariable:

```
      ?customer: Customer = Customer->any()
      ?sCart: ShoppingCart = ShoppingCart-
>any(c:customer|c=?customer)
      ?bookItems: Set = ?sCart.items-
>select(e:items|e.isBestseller())
```

Condition:

```
      ?bookItems.size() > 0
```

Action:

```
      ?customer.hyperlinkToBestseller = true
```

Example 4: atLeast3Items rule

English text:

If there are at least 3 items of the same type in the customer shopping cart and each item's value is greater than $30 then give to the customer a voucher whose value is 10% of the cheapest item.

Proprietary rule language:

```
rule atLeast3Items{
when
{
  ?customer1: Customer();
  ?shoppingCart1: ShoppingCart(customer == ?customer1);
  ?itemType1: ItemType();
  ?items: collect Item(type == ?itemType1 ; value > 30)
                          in ?shoppingCart1.getItems()
      where (size()>3);
}
then
{
  bind ?var1 = ?items.elements();
  bind ?min = 0;
  while (?var1.hasMoreElements())
  {
    bind ?elt = (Item)?var1.nextElement();
    if (?elt.value < ?min)
    {
       ?min = ?elt.value;
    }
  }
  assert Voucher
  {
    value = .1 * ?min;
    customer = ?customer1;
  }
}
```

PRR OCL:

That rule can be represented by a ruleset in PRR OCL (assuming forward chaining):

Ruleset *atLeast3Items (in scart : ShoppingCart)*

Variable:

```
      low : Real = -1
```

Rule *initializeCheapestPrize (priority = 1)*

ruleVariable:

```
      ?itemType: ItemType = ItemType->any()
      ?items30: Set = sCart.items->select(e:items|e.type=?itemType
&&
                                          e.value() > 30)
```

Condition:

```
      ?item30.size() >= 3 and low < 0
```

Action:

```
      Low = ?item30.at(1).value()

Rule cheapestPrize (priority = 1)
ruleVariable:
      ?itemType: ItemType = ItemType->any()
      ?items30: Set = sCart.items->select(e:items|e.type=?itemType
&&
                                          e.value() > 30)
      ?cheaperItem: Item = sCart.items-
>any(e:items|e.type=?itemType &&
                                        e.value() > 30 &&
                                      e.value() < low)
Condition:
      ?item30.size() >= 3
Action:
      Low = ?cheaperItem.value()


Rule awardVoucher (priority = 0)
Condition:
      Low > 0
Action:
      assert Voucher(value = .1 * low; customer = sCart.customer)
```

Example 5: twoDifferentItems rule

English text:

If the shopping cart contains 2 items related but having different type then give to the customer a voucher of $1.

Proprietary rule language:

```
rule twoDifferentItems
when
{
  ?customer1: Customer();
  ?shoppingCart1: ShoppingCart(customer == ?customer1);
  ?item1: Item(shoppingCart == ?shoppingCart1 );
  Item(shoppingCart == ?shoppingCart1 ; type!=?item1.type)
        in getRelatedItems();
}
then
{
  assert Voucher
  {
    value = 1;
    customer = ?customer1;
  }
}
```

PRR OCL:

```
Rule twoDifferentItems
ruleVariable:
      ?sCart: ShoppingCart = ShoppingCart->any()
      ?item1: Item = ?sCart.items->any()
      ?item2: Set =
                ?sCart.items->select(e:items|e.type=?item1.type &&
                                     e.relatedItems-
>includes(?item1))
Condition:
      ?item2.size() > 0
Action:
      assert Voucher(value = 1; customer = ?sCart.customer)
```

Example 6: removeVoucher rule

English text:

It the shopping cart discount value is greater than 10% and a voucher has a value greater than $4 then remove the voucher.

Proprietary rule language:

```
rule removeVoucher {
when
```

```
{
  ?customer1: Customer();
  ?shoppingCart1: ShoppingCart(customer == ?customer1 ;
                               discountValue>10);
  ?voucher: Voucher(customer == ? customer1 ; value >4);
}
then
{
 retract voucher;
}
}
```

PRR OCL:

Rule *removeVoucher*

ruleVariable:

```
?sCart: ShoppingCart = ShoppingCart->any(s: ShoppingCart |
                                                s.discountValue
                                        > 10)
?voucher: Voucher = ?sCart. customer.vouchers(v: Voucher |
                                                v.value >
                                                4)
```

Action:

retract ?voucher

F Other Rule Types

The development of the Production Rule Representation represents the first rule modeling standard by OMG for end-user UML-type executable rules. To support this, PRR includes constructs for ComputerExecutableRule and ComputerExecutableRuleset with an associated Variable (see 3.4.1). These constructs also open the possibility of modeling other rule types in UML, such as:

- Event Condition Action / Reaction Rules.
  These rules are similar to production rules but include an event condition; their semantics are usually such that an explicit invocation event is detected by the rule causing rule execution, which simplifies the model but excludes explicit inferencing unless rules also generate events for other rules to detect.
- Constraint Rules.
  These rules define constraints or cost functions on data models, allowing constraint-based reasoning engines to maximize some overall cost function based on constraint expressions. Such rules would be modeled differently from PRR, as they do not share the same “if… then…” structure.