Object Management Group

Production Rule Representation Proposal

Draft Response to OMG RFP br/2003-09-03

Version (draft): June 2006

Objective of RFP br/2003-09-03

The RFP addresses the representation of production rules in UML models. (Production rules, as considered in this RFP, should not be confused with XMI production rules as defined in XMI 1.1 specification (formal/2000-11-02) or other model or grammar transformation rules specified by the OMG standards.) With respect of production rules, the RFP solicits 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.

Contents

This draft constitutes a work in progress. To be discussed at ongoing OMG Technical Meetings and Conference Calls.

In particular, the following non-normative content is in progress:
1. Further explanations as to the semantics of inference rule engines
2. Examples of PRR OCL versus commercial rule engine languages~
3. Example of PRR extensions for other languages such as ECA rules
4. Completion and verification of PRR OCL.
5. Any revisions outstanding from the W3C RIF effort that affects rule modeling and production rules in particular.
6. PRR RuleML as an example PRR OCL implementation.

Sections 3.5 on will be completed later.

1.0 Introduction

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).

The vendors involved in this standard all provide their own production rule representations, and these have been used as the basis for this standard.

1.1 Goals of the Production Rule Representation Standard

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.

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.

There are currently 3 standards bodies involved with defining domain-independent production rule representations:

1. OMG – represented by the Business Modeling Integration group and developers of the RFP to which this proposal responds.
2. RuleML – http://www.ruleml.org/ - a family of related rule standards, with specific focus on W3C and the Semantic Web, and including PR-RuleML based on PRR - http://prruleml.ruleml.org/ . See Appendix C.
3. W3C - http://www.w3.org/2005/rules/ - has chartered a working group to define a rule interchange format for rule-driven systems. See also Appendix D.

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.

1.2 Organization of this document

This proposal is organized as follows:

Chapter 2 - *Context of the PRR* – relationships between the PRR and rule modeling, software modeling and the OMG UML, and OMG's Model Driven Architecture and Unified Modeling Language "languages".

Chapter 3 - *PRR Proposal* – definition, semantics and metamodel, scope, UML diagram notation examples, XMI specification, and compliance requirements for the proposed standard PRR Core with its proposed concrete syntax PRR OCL.

Chapter 4 – *Comparison with RFP* – showing compliance and deviations from the original RFP, with explanations.

Chapter 5 – *Comparison with other OMG Standards* – compare the relevancy and potential model interactions for PRR with UML and other standards related to rules such as SBVR, BPDM, ODM.

Chapter 6 – *References*

Appendix A – *Glossary* – definitions used in this document.

Appendix B – *Guidance for Users* – scenarios showing how the PRR OCL should be used in rule modeling and development.

Appendix C – *Example implementation: PR-RuleML* – PRR example using RuleML, with mapping examples.

Appendix D – *Relationship with: W3C Rule Interchange Format* – the relationship between PRR and the W3C RIF.

Appendix E – *PRR Syntax Examples* – provided examples of the usage of production rules in vendor-specific and PRR OCL formats.

Appendix F – *Other Rule Types* – provided example of how other rule types would relate to PRR as defined here.

1.3 Contact Information

The PRR standard is developed by the submitters (Fair Isaac, Ilog, and IBM) and a set of active supporters. Submitters' and supporters' team members are listed below.

Representing the RuleML standards body:
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
Paul Vincent of Fair Isaac: paulvincent@fairisaac.com
Gonzaques Jacques of ILOG: gjacques@ilog.fr
Christian de Sainte Marie of ILOG: csma@ilog.fr
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

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

1.4 RFP Credit

This proposal develops from and extends the principles defined in the original RFP.

1.5 IPR and Patents

Statements have been made by submitters IBM and Ilog on the PRR and are reproduced in Appendix G. A statement from Fair Isaac is in preparation.

2.0 Context of the PRR

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

(i) 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).

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

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

(iv) 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.

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

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

2.1 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, that 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.

2.1.1 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 "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.

RFP comment:
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 this 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.

2.1.2 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).

2.2 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 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 representing (modeling) 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.

Further comments on OCL and AS for PRR are provided in Chapter 5.

3.0 PRR Proposal

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 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
- 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.

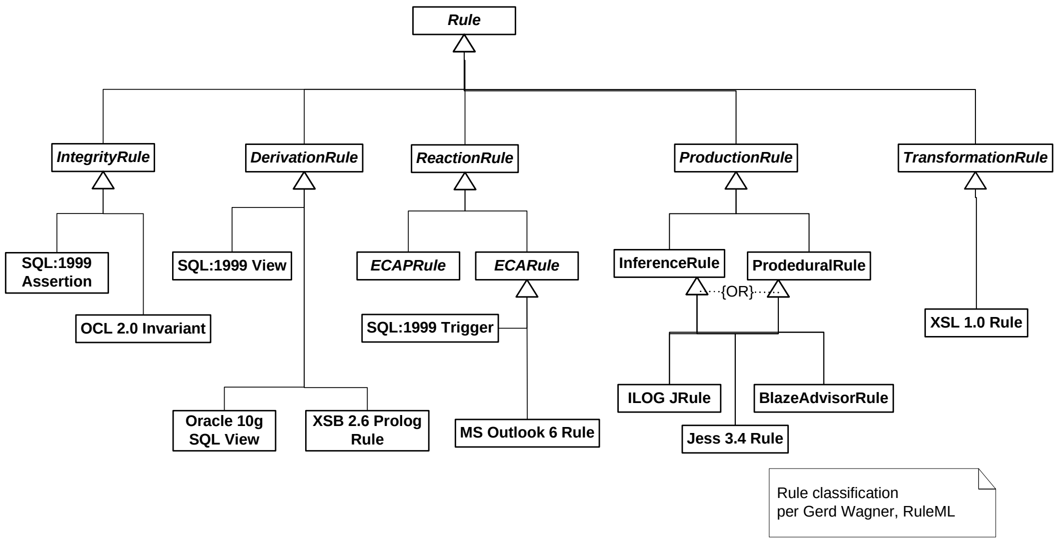

Figure 1: Rule classification scheme

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.

3.2.2 Production Ruleset definition

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

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

From an architecture and framework perspective, a ruleset is

- a Behavior in UML terms,
- a service implementation in SOA terminology.

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:
a. parameters
b. context at invocation time.

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

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:

- 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]

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:

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

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

3. *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**[1] consideration of the explicit ordering of the rules; execution ordering is under the control of the inference engine which maintains a stateful representation of rule bindings.

The operational semantics of forward-chaining production rules extends 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

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 are able to 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 impact 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.

[1] In this version of PRR we do not consider rule priorities, which in any case would form part of a conflict resolution strategy.

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

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 2-4 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:

- 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 but it may affect the satisfiability of their condition.

3.3 PRR Core Metamodel

This metamodel applies to PRR Core and PRR OCL.

3.3.1 Overview

This diagram applies to PRR Core and PRR OCL.

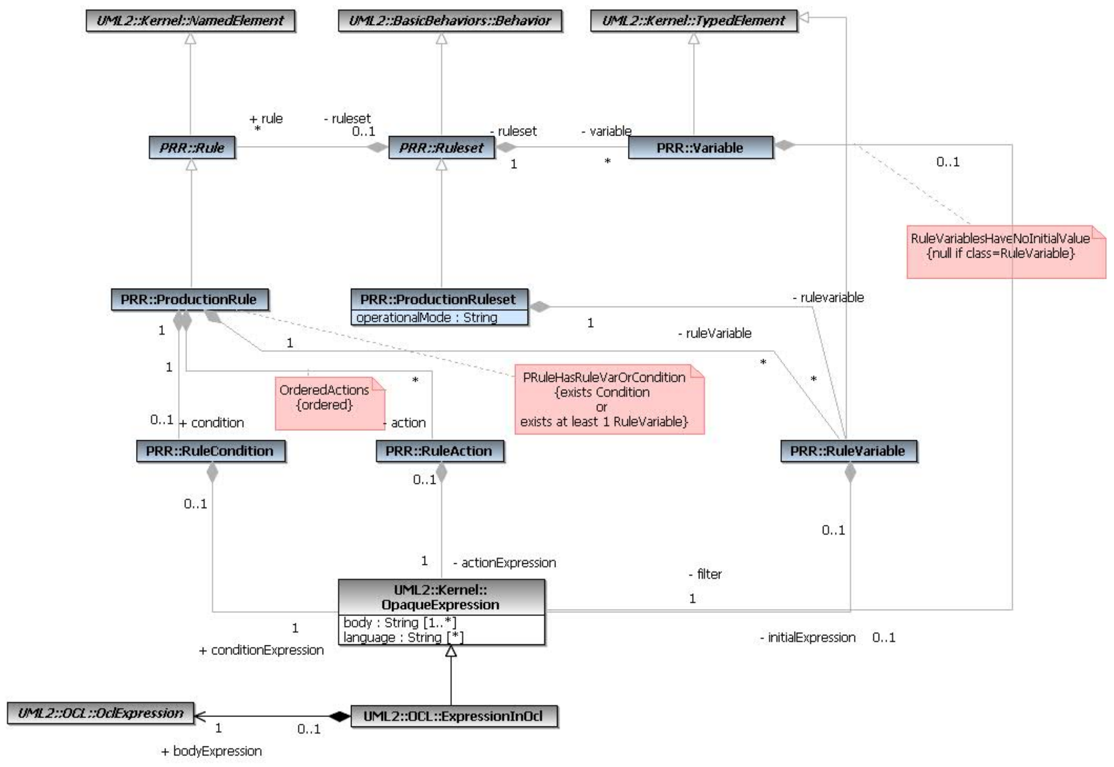

Figure 2: Production Rule Representation Overview

PRR includes the concepts of general rules and rulesets for future extensions to other rule types.

3.3.2 Class Rule

3.3.2.1 Description

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.

3.3.2.2 Attributes

No additional attributes defined.

3.3.2.3 Associations

A rule may be contained in a ruleset.

3.3.2.4 Constraints

No additional constraints.

3.3.2.5 Semantics

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

3.3.3 Class Ruleset

3.3.3.1 Description

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).

3.3.3.2 Attributes

No additional attributes defined.

3.3.3.3 Associations

• RuleContainment: Rule[*]	The rules contained in the ruleset.
• VariableDefinition: Variable[*]	The variables defined in the ruleset.

3.3.3.4 Constraints

None.

3.3.3.5 Semantics

The semantics for rulesets are determined by the rule subtypes such as production rules in production rulesets.

3.3.4 Class Variable

3.3.4.1 Description

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

3.3.4.2 Attributes

No additional attributes defined.

3.3.4.3 Associations

- InitialExpression: OpaqueExpression [1] The expression specifying an initialization on the variable.

3.3.4.4 Constraints

None.

3.3.4.5 Semantics

The variable, if not specialized as a rule variable, represents a typed element that is used in rule expressions as a substitute for an explicit object reference.

3.3.5 Production Ruleset Diagram

This diagram applies to PRR Core and PRR OCL.

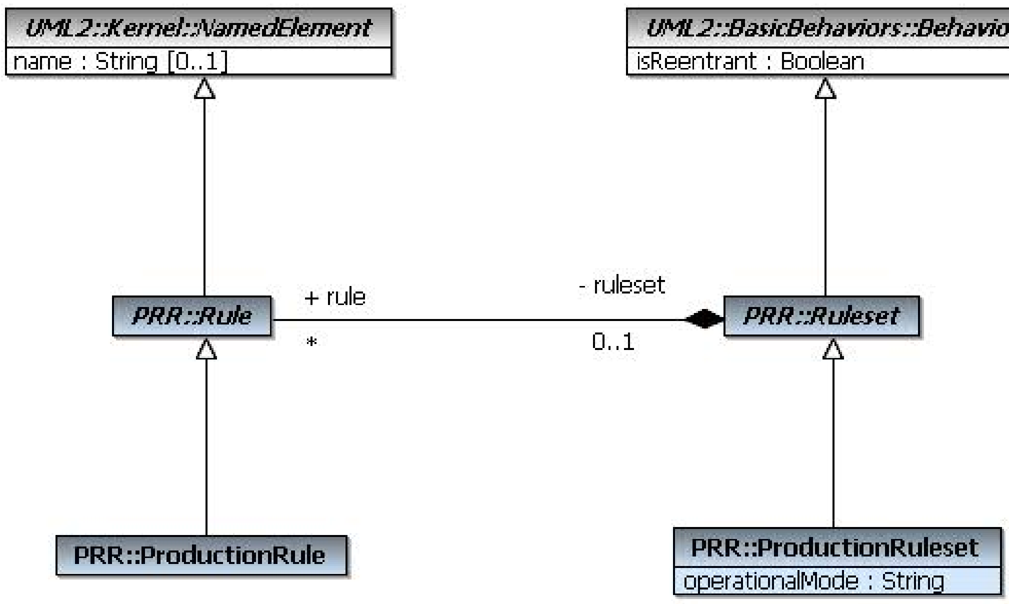

Figure 3: Production Ruleset Diagram

3.3.6 Class ProductionRuleset

3.3.6.1 Description

The ProductionRuleset represents a ruleset for production rules.

3.3.6.2 Attributes

• operationalMode: String	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 "Rete".

3.3.6.3 Associations

There are no additional associations.

3.3.6.4 Constraints

A ProductionRuleset may only contain ProductionRules.

3.3.6.5 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 14 for further detail.

3.3.7 Production Rule Diagram

This diagram applies to PRR Core and PRR OCL.

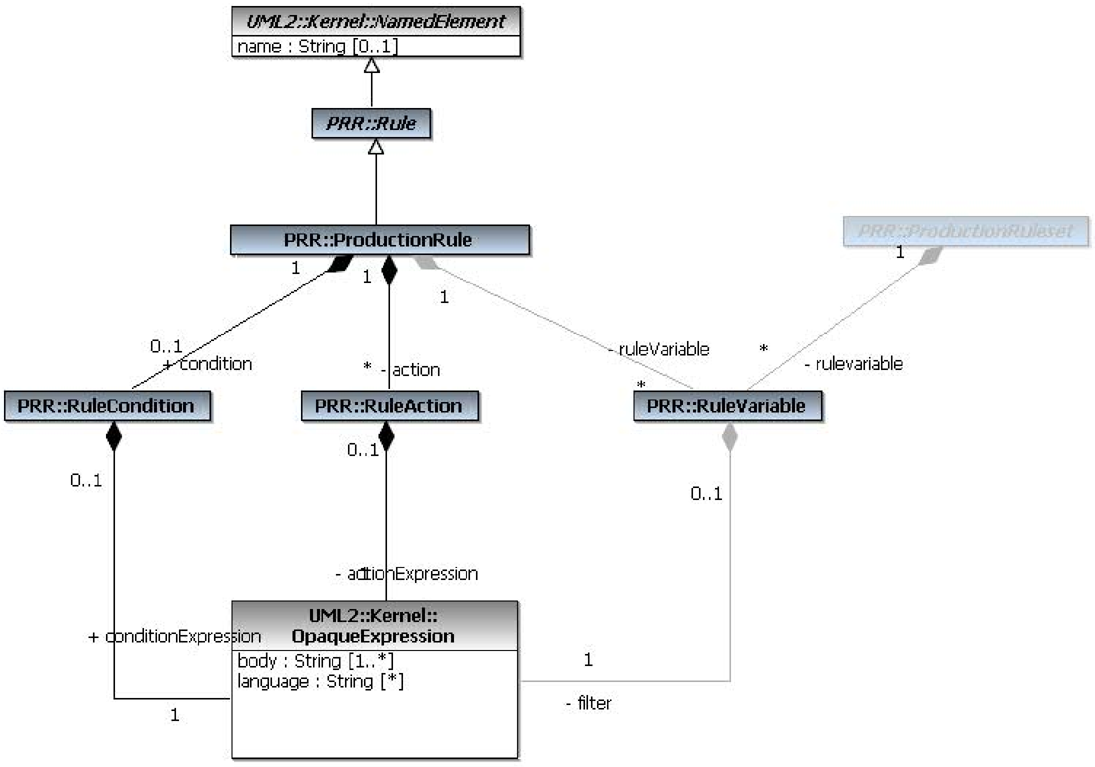

Figure 4: Production Rule Diagram

3.3.8 Class ProductionRule

3.3.8.1 Description

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]

and the RuleVariables may be locally defined or reference those defined in its owning ProductionRuleset.

3.3.8.2 Attributes

No additional attributes defined.

3.3.8.3 Associations

• RuleCondition [0..1]	The rule condition that is required to be satisfied for the rule to be triggered.
• RuleAction [*]	The ordered list of actions that are executed when the rule is fired.
• RuleVariable [*]	The list of RuleVariables that define the bindings in rule instantiation.

3.3.8.4 Constraints

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

3.3.8.5 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.

3.3.9 Class RuleCondition

3.3.9.1 Description

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.

3.3.9.2 Attributes

No additional attributes defined.

3.3.9.3 Associations

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

3.3.9.4 Constraints

[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.

The OpaqueExpression evaluates to a Boolean result.

3.3.9.5 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.

3.3.10 Class Action

3.3.10.1 Description

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.

3.3.10.2 Attributes

No additional attributes defined.

3.3.10.3 Associations

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

3.3.10.4 Constraints

The actions form an ordered list.

3.3.10.5 Semantics

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

3.3.11 Rule Variable Diagram

This diagram applies to PRR Core and PRR OCL.

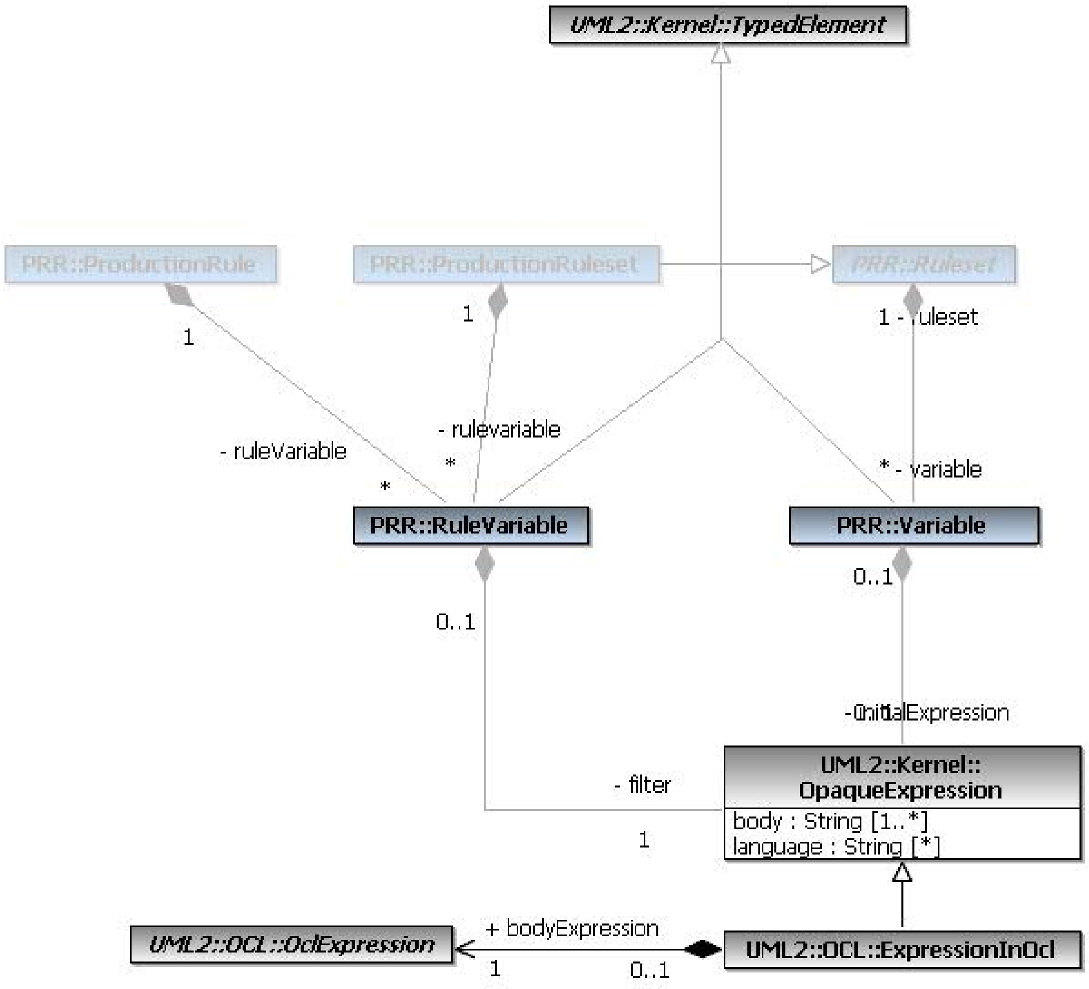

Figure 5 : Rule Variable Diagram

3.3.12 Class RuleVariable

3.3.12.1 Description

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.

3.3.12.2 Attributes

No additional attributes defined.

3.3.12.3 Associations

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

3.3.12.4 Constraints

The InitialExpression association from the parent Variable class is not available to any RuleVariable.

3.3.12.5 Semantics

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

3.4 PRR OCL Metamodel

This metamodel applies to PRR OCL only.

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.

3.4.1 PRR OCL Metamodel

3.4.1.1 OCL for PRR OCL

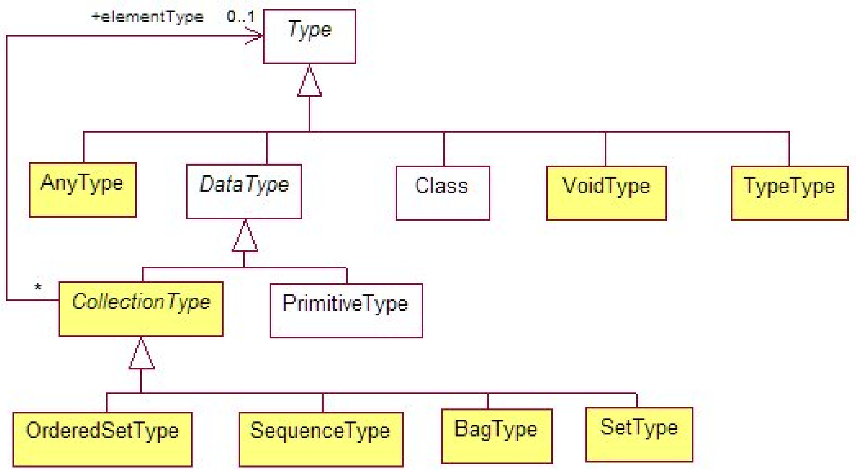

Figure 6, Figure 7 and Figure 8 show the subset of BasicOCL package that is used by PRR OCL.

The classes that are not part of OCL are shown with a transparent fill color.

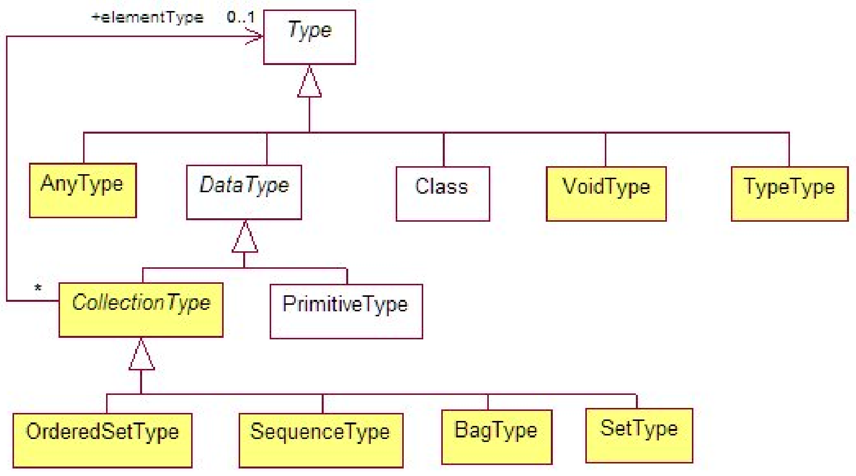

Figure 6: Types

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.

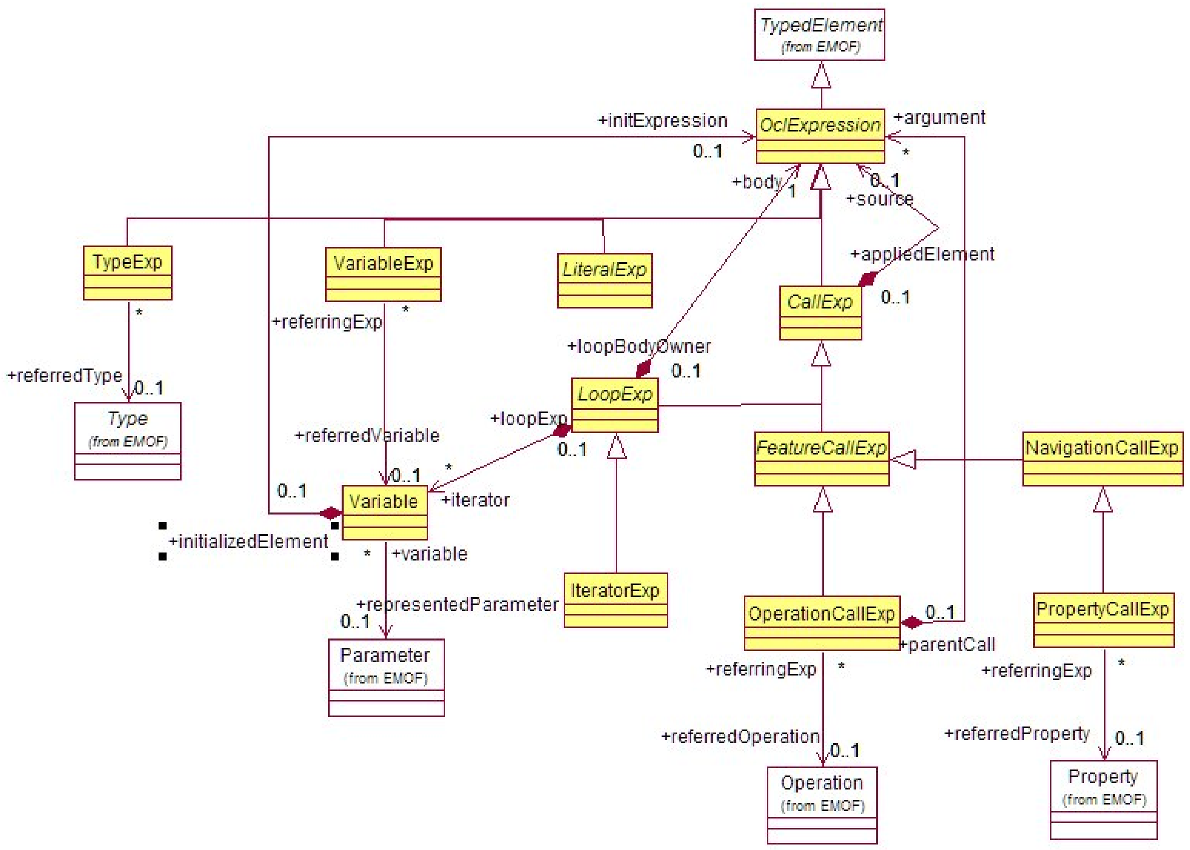

Figure 7: OCL Expressions

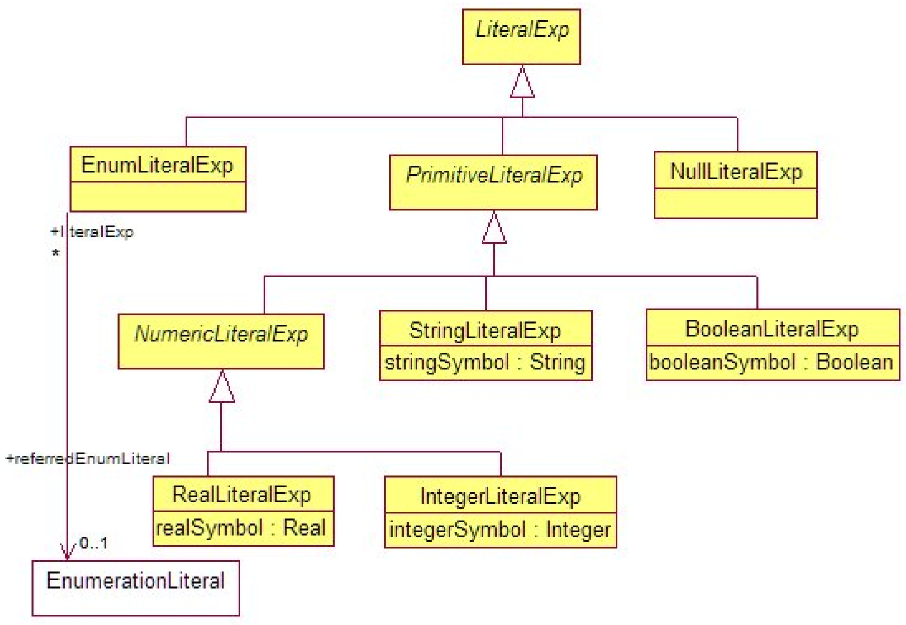

Figure 8: Literals

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.

3.4.1.2 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);
```

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 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).

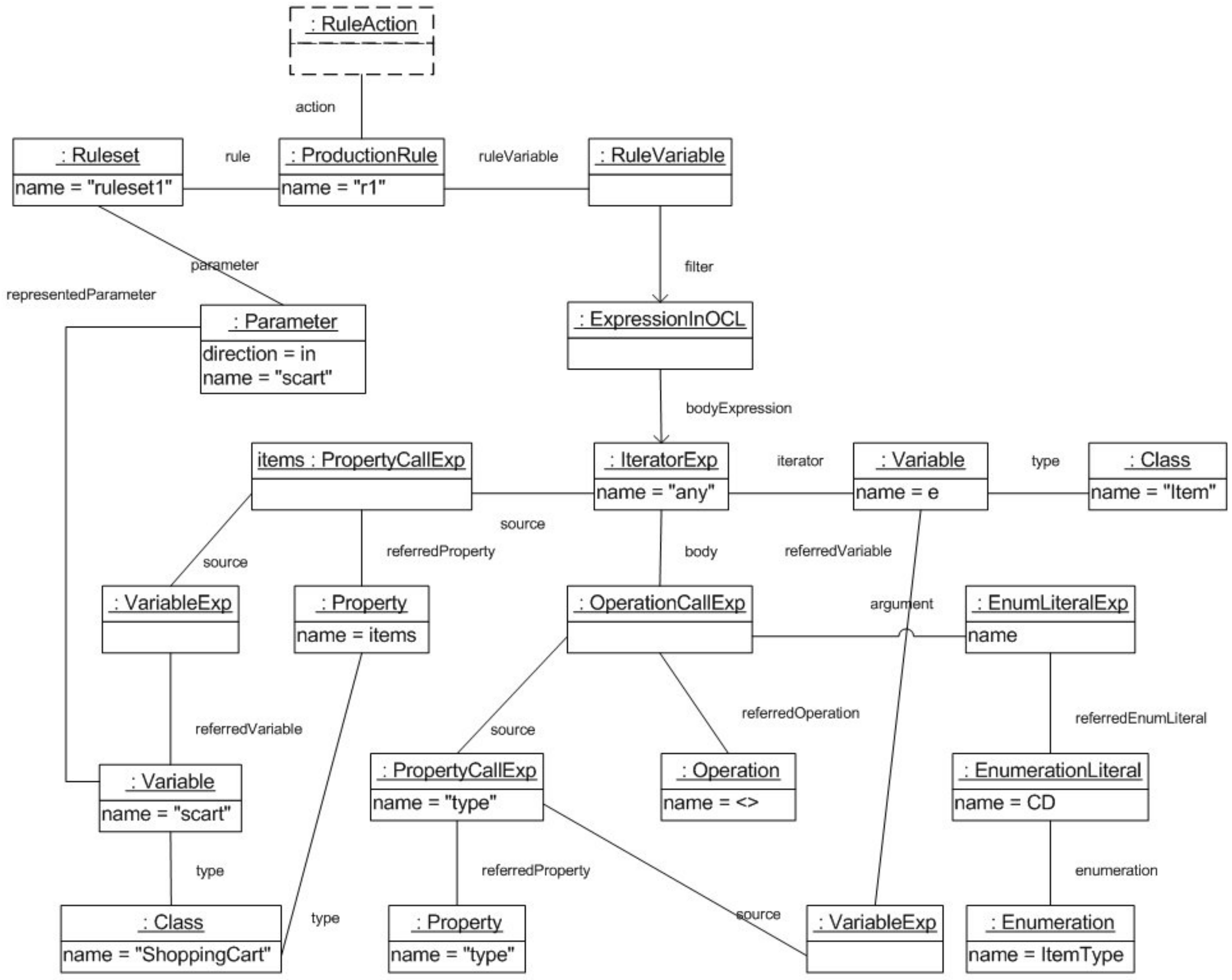

Figure 9 An example of Rule Variable abstract syntax

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 explain why the “e.type” expression is used as source in the Figure 9.

3.4.1.3 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. The only exceptions are the 2 IteratorExp instances (`any` and `select`) that are used to define rule variables but with a very restricted usage.
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:

```
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.

3.4.1.4 RuleAction

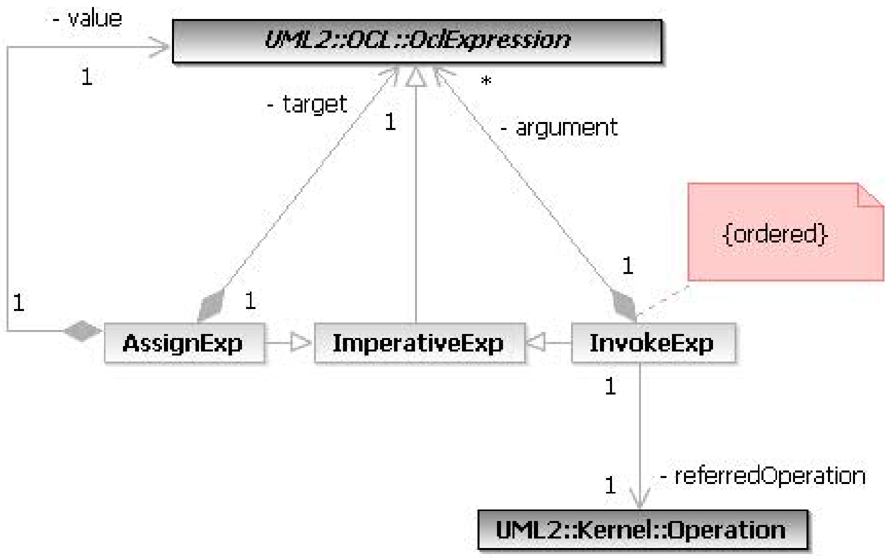

Figure 10: PRR OCL Actions

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. Four different actions have been selected:

- Add: make an object visible to 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.
- Remove: 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.

We can consider at least two possible designs to create the action metamodel:

- Create RuleAction subclasses for the various types of action.
- Extend BasicOCL to provide new types of expression
  - This solution is consistent with the way conditions are defined
  - It is the solution that has been chosen in QVT Specification (ptc/05-11-01). The QVT ImperativeOCL package defines an `ImperativeExpression` class that inherits from `OCLExpression` and from which derives classes like `AssignExp`, `InstantiationExp`, `ForExp`, and so on.
    - The design is already done. PRR can use it directly
    - Since there is an existing model, what justifies doing otherwise?
    - Later we can extend the action part by supporting other operations as required.

Note: Some BasicOCL extensions used below are based on the extensions of the same name specified in the OMG MOF QVT specification – see
Meta Object Facility (MOF) 2.0 Query/View/Transformation
Specification, Final Adopted Specification, ptc/05-11-01

3.4.1.4.1 Class ImperativeExp

Note: Based on the BasicOCL extension of the same name specified in the OMG MOF QVT specification.

3.4.1.4.1.1 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.

3.4.1.4.1.2 Attributes

No additional attributes defined.

3.4.1.4.1.3 Associations

None.

3.4.1.4.1.4 Constraints

No additional constraints defined.

3.4.1.4.1.5 Semantics

None.

3.4.1.4.2 Class AssignExp

Note: Based on the BasicOCL extension of the same name specified in the OMG MOF QVT specification.

3.4.1.4.2.1 Description

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

3.4.1.4.2.2 Attributes

None.

3.4.1.4.2.3 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.

3.4.1.4.2.4 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.

3.4.1.4.2.5 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.

3.4.1.4.3 Class InvokeExp

Note: Based on the OCL2 specification for OperationCallExp.

3.4.1.4.3.1 Description

A 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.

3.4.1.4.3.2 Attributes

None

3.4.1.4.3.3 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.

3.4.1.4.3.4 Constraints

No additional constraints defined.

3.4.1.4.3.5 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.

3.4.2 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.

3.4.2.1 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.

3.4.2.2 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)

3.4.2.3 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.

3.4.2.4 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.

3.4.2.5 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

3.4.2.6 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.

3.4.2.7 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.

3.4.2.8 Collection-Related Types

3.4.2.8.1 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))

3.4.2.8.2 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)))

3.4.2.8.3 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

3.4.2.8.4 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))

3.4.2.8.5 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

3.4.2.9 The PRRRuleEngine class

The PRRRuleEngine class defines the additional helper functions that may be required in specific rules to notify the rule engine of state change side effects during action execution.

Assert(obj : T)
Notifies the rule engine that the object obj is in scope.

Retract(obj : T)
Notifies the rule engine that the object obj is removed from scope.

Update(obj : T)
Notifies the rule engine that the state of the object obj has changed.

3.5 XMI W3C XML schema

The PRR Core and PRR OCL XMI schema will be detailed in a future submission.

3.6 PRR Compliance

The PRR Core and PRR OCL Compliance states will be detailed in a future submission.

4.0 Comparison with RFP

RFP compliance can be viewed from the following:

The PRR comparison with the RFP requirements will be detailed in a future submission.

5.0 Comparison with other OMG Standards

5.1 UML

5.1.1 UML Activities

The PRR comparison with UML Activities will be detailed in a future submission.

5.1.2 UML Events

The PRR comparison with UML Events will be detailed in a future submission.

5.2 Alignment with MDA - Model Driven Architecture

The PRR alignment with MDA will be detailed in a future submission.

5.3 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.

5.4 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.

5.5 Comparison with other Standards

5.5.1 Semantics for Business Vocabularies and Rules (SBVR)

The PRR comparison with SBVR will be detailed in a future submission.

5.5.2 Business Process Definition Metamodel (BPDM)

The PRR comparison with BPDM will be detailed in a future submission.

5.5.3 Ontology Definition Metamodel (ODM)

The PRR comparison with ODM will be detailed in a future submission.

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

The PRR comparison with EDOC/ECA will be detailed in a future submission.

5.6 Possible Transformations

5.6.1 Production Rule Transformation from the SBVR

The PRR transformation possibilities from SBVR will be detailed in a future submission.

6.0 References

6.1 References Specific to the PRR

[GUI] GUIDE project, "Defining Business Rules ~ What are they really?", *http://www.businessrulesgroup.org/first_paper/br01c0.htm*

[INF1] General discussion of inferencing technology, *http://www.cs.cornell.edu/Info/Projects/NuPrl/manual.with.index/node127.html*

[INF2] Stefik, Mark J., Introduction to Knowledge Systems. Morgan Kaufmann, Los Altos, CA, 1995.

[MDA1] OMG Architectural Board, "Model Driven Architecture – A Technical Perspective", http://www.omg.org/mda/papers.htm.

[MDA2] "Developing in OMG's Model Driven Architecture (MDA)," http://www.omg.org/cgi-bin/doc?omg/2001-12-01.

[MDA3] "MDA Guide" http://www.omg.org/doc/omg/03-06-01.pdf.

[MDA4] "MDA: The Architecture of Choice for a Changing World", http://www.omg.org/mda.

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

[RETE1] 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.

[RETE2] Explanation of the Rete algorithm, *http://www.csee.monash.edu.au/~dmoulder/thesis_draft/node71.html*

[RETE3] Explanation of the Rete algorithm, *http://cis.temple.edu/~ingargio/cis587/readings/rete.html#6*

[RETE4] Explanation of the Rete algorithm, *http://www.cse.unsw.edu.au/~cs9416/04-RETE/rete.html*

6.2 General References

The following documents are referenced in this document:

[ATC] Air Traffic Control Specification, *http://www.omg.org/technology/documents/formal/air_traffic_control.htm*

[BCQ] OMG Board of Directors Business Committee Questionnaire, *http://www.omg.org/cgi-bin/doc?bc/02-02-01*

[CCM] CORBA Core Components Specification, *http://www.omg.org/techprocess/meetings/schedule/Components_December_2000_FTF.html*

[CORBA] Common Object Request Broker Architecture (CORBA/IIOP), *http://www.omg.org/technology/documents/formal/corba_iiop.htm*

[CSIV2] [CORBA] Chapter 26

[CWM] Common Warehouse Metamodel Specification, *http://www.omg.org/technology/documents/formal/cwm.htm*

[DAIS] Data Acquisition from Industrial Systems, *http://www.omg.org/cgi-bin/doc?dtc/2001-07-03*

[EDOC] UML Profile for EDOC Specification, *http://www.omg.org/techprocess/meetings/schedule/UML_Profile_for_EDOC_FTF.html*

[EJB] "Enterprise JavaBeans™", *http://java.sun.com/products/ejb/docs.html*

[FORMS] Download of OMG templates and forms, *http://www.omg.org/technology/template_download.htm*

[GE] Gene Expression, *http://www.omg.org/cgi-bin/doc?dtc/2002-02-04*

[GLS] General Ledger Specification 1.0, *http://www.omg.org/technology/documents/formal/gen_ledger.htm*

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

[IDL] ISO/IEC 14750 also see [CORBA] Chapter 3.

[IDLC++] IDL to C++ Language Mapping, *http://www.omg.org/technology/documents/formal/c++.htm*

[MDAa] OMG Architecture Board, "Model Driven Architecture - A Technical Perspective", *http://www.omg.org/mda/papers.htm*

[MDAb] “Developing in OMG's Model Driven Architecture (MDA),” *http://www.omg.org/cgi-bin/doc?omg/2001-12-01*

[MDAc] “MDA Guide” (to be published)

[MDAd] “MDA "The Architecture of Choice for a Changing World™"”, *http://www.omg.org/mda*

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

[MQS] “MQSeries Primer”, *http://www.redbooks.ibm.com/redpapers/pdfs/redp0021.pdf*

[NS] Naming Service, *http://www.omg.org/technology/documents/formal/naming_service.htm*

[OMA] “Object Management Architecture™”, *http://www.omg.org/oma/*

[OTS] Transaction Service, *http://www.omg.org/technology/documents/formal/transaction_service.htm*

[P&P] Policies and Procedures of the OMG Technical Process, *http://www.omg.org/cgi-bin/doc?pp*

[PIDS] Personal Identification Service, *http://www.omg.org/technology/documents/formal/person_identification_service.htm*

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

[RAD] Resource Access Decision Facility, *http://www.omg.org/cgi-bin/doc?formal/01-04-01*

[RM-ODP] ISO/IEC 10746

[RuleML1] RuleML Draft Metamodels (unpublished)

[SEC] CORBA Security Service, *http://www.omg.org/technology/documents/formal/security_service.htm*

[TOS] Trading Object Service, *http://www.omg.org/technology/documents/formal/trading_object_service.htm*

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

[UMLC] UML Profile for CORBA, *http://www.omg.org/cgi-bin/doc?ptc/01-01-06*

[UMLM] Chapter 6 of UML Profile for EDOC, *http://www.omg.org/cgi-bin/doc?ptc/02-02-05*

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

[XML/Value] XML Value Type Specification, *http://www.omg.org/cgi-bin/doc?ptc/2001-04-04*

Appendix A Glossary

The PRR Glossary will be completed in a future submission.

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.

Scope of a Rule **–** Equivalent to “context” as defined for the Constraints Package in the UML 2.0 Infrastructure specification (ad/03-01-01), as applied to production rules. The scope of a rule is an optional structural feature of a production rule that specifies the namespace (specifically the Class) that provides the context for evaluating the rule. Scoping a rule to a class supports the use of ***self*** in the rule.

Appendix B Guidance for Users

The PRR guidance section will be detailed in a future submission.

Appendix C Example implementation: PRR RuleML

The PRR implementation in RuleML will be detailed in a future submission.

Appendix D Relationship with: W3C Rule Interchange Format

The PRR positioning with W3C RIF will be detailed in a future submission.

Appendix E Example Rule Mappings from other formats

The PRR OCL example mappings will be detailed in a future submission.

Appendix F Other Rule Types: ECA Rule example

A sample PRR extension for ECA type rules will be detailed in a future submission.

Appendix G IPR Statements for OMG Business Committee

G.1 Ilog

- Will you assign or license to OMG a worldwide, royalty free right to distribute the Specification and/or support measures document, if adopted?

  YES

- Will you grant OMG the right to develop and distribute derivative works based on the Specification documents, if adopted?

  YES

- Are there any patent, copyright, trademark or other intellectual property rights, owned by you or others, that are required in order to implement or use the Specification? If your answer is "Yes", please provide details.

  ILOG is not currently aware of any ILOG or other party patents that would necessarily be infringed by an implementation of the current specification. ILOG has not conducted a patent search and makes no representation as to the existence of any such patents. If ILOG owns any patents necessary to implement this specification, ILOG will make available licenses to those patents on reasonable and nondiscriminatory terms. If any third party owns any patents necessary to implement this specification ILOG does not know whether that third party will license its patents.

- If such rights are required, do you believe that an appropriate non-discriminatory license is available for these rights?

  NOT APPLICABLE

- If you own the required rights, will you license these rights on non-discriminatory and commercially reasonable terms?

  YES

- If you own the required rights, will you license these rights on a royalty-free basis?

  YES, for rule interchange and interoperability specifications

G.2 IBM

- Will you assign or license to OMG a worldwide, royalty free right to distribute the Specification and/or support measures document, if adopted? ***YES***
- Will you grant OMG the right to develop and distribute derivative works based on the Specification documents, if adopted? ***YES***
- Are there any patent, copyright, trademark or other intellectual property rights, owned by you or others, that are required in order to implement or use the Specification? If your answer is “Yes”, please provide details.
  IBM is not currently aware of any IBM or other party patents that would necessarily be infringed by an implementation of the current specification. IBM has not conducted a patent search and makes no representation as to the existence of any such patents. If IBM owns any patents necessary to implement this specification, IBM will make available licenses to those patents on reasonable and nondiscriminatory terms. If any third party owns any patents necessary to implement this specification IBM does not know whether that third party will license its patents.

- If such rights are required, do you believe that an appropriate non-discriminatory license is available for these rights? ***Not Applicable***
- If you own the required rights, will you license these rights on non-discriminatory and commercially reasonable terms? ***YES***
- If you own the required rights, will you license these rights on a royalty-free basis? ***YES (for Rule interchange and interoperability specs)***