Rule Modeling and Markup

Gerd Wagner

Institute of Informatics, Brandenburg University of Technology at Cottbus,
P.O.Box 10 13 44, 03013 Cottbus, Germany
G.Wagner@tu-cottbus.de

Abstract. In this paper we address several issues of rule modeling on the basis of UML. We discuss the relationship between UML class models and OWL vocabularies. We show how certain rules can be specified in a class diagram with the help of OCL. We also show how rule concepts can be described, and how the abstract syntax of RDF, OWL, SWRL and RuleML can be defined, by means of UML class diagrams in a concise way.

1 Introduction

Rules play an important role not only in everyday life but also in computational formalisms and information systems. They define derived concepts as elements of the information state structure and constrain or prescribe the behavior of people and IT systems. In particular, rules are being used to express privacy protection and access control policies, both of which are important issues on the Web.

As we model the state structure and behavior of a system to be analyzed or to be designed, we also have to model the rules defining the derived elements of its information base and governing its behavior. Therefore, rule modeling is part of a general *model-driven* approach to software and information systems engineering.

Rules always come on top of a vocabulary. There is no rule without an underlying vocabulary. Consequently, for being able to see how rules can be modeled and represented in formal languages, we also have to understand how vocabularies are being modeled and expressed. in formal languages.

1.1 Specifying Vocabularies

While the recommended method for specifying domain vocabularies, as part of systems analysis, in general software engineering is to use the *Unified Modeling Language*[1] *(UML)* for making a *class model* in the semi-visual form of a *class diagram*, the W3C has recommended to use the languages *RDF* and *OWL*[2] for specifying vocabularies as part of Web applications. In particular, OWL has a great overlap with UML class models. However, while UML class models have a visual

---

[1] See http://www.uml.org/.

[2] RDF is the *Resource Description Framework* (see http://www.w3.org/RDF/). OWL is the *Web Ontology Language* (see http://www.w3.org/2004/OWL).

N. Eisinger and J. Małuszyński (Eds.): Reasoning Web 2005, LNCS 3564, pp. 251 – 274, 2005.
© Springer-Verlag Berlin Heidelberg 2005

syntax and are widely used in academic and industrial software engineering activities, they don't have a formal logic semantics. OWL, on the other hand, has a formal logic semantics, but has no visual syntax and is not (yet?) widely used in industry.

Clearly, both languages can benefit from each other:

- OWL vocabularies can be captured in the user-friendly form of class diagrams. For this purpose the UML provides an extension mechanism that allows to use OWL-specific elements in a class diagram. Expressing OWL construct as elements of a UML class model gives OWL a kind of operational semantics and makes it accessible to software engineers who are not familiar with, and not willing to learn, the description logic semantics of OWL.
- UML class diagrams can be mapped to OWL vocabularies and, in this way, obtain a logical semantics.

There is yet another good reason to consider UML: UML class diagrams can also be used as a visual language to describe the vocabulary, and the abstract syntax, of all kinds of languages in a concise visual manner. The particular fragment of UML class modeling that has been proposed for this purpose by the OMG is called *Meta-Object Facility*[3] *(MOF)*; we call it *MOF/UML* in the sequel We use MOF/UML in this article for describing the abstract syntax, or the language model, of RDF, OWL, SWRL[4] and RuleML. This representation helps to identify commonalities and differences between these languages.

1.2 Modeling Rules

Since rules are based on vocabularies, it is natural to add rule constructs to the language of UML class models for obtaining a general rule modeling language. For this purpose, the UML has been supplemented by the *Object Constraint Language (OCL)*, which allows to add *integrity rules* (called *invariants*) and *derivation rules* to a class model in order to constrain or derive certain model elements. However, UML and OCL do not provide any visual syntax for rules, nor do they support other kinds of rules. In particular, the concept of *reaction* (or *event-condition-action*) rules is not supported at all in UML.

The *Model Driven Architecture*[5] *(MDA)* is a framework for software development defined by the *Object Management Group (OMG)*. It is based on a fundamental distinction between three different modeling levels:

1. the level of semi-formal business domain modeling, called *'computation-independent' modeling (CIM)*,
2. the level of platform-independent logical design modeling, in short: *platform-independent modeling (PIM)*, and
3. the level of platform-specific implementation modeling, in short: *platform-specific modeling (PIM).*

[3] See http://www.omg.org/mof.
[4] See the subsection on SWRL below.
[5] See http://www.omg.org/mda.

As illustrated in Fig. 1, we consider rules at these three different abstraction levels:

1. At the ***business domain (CIM) level***, rules are statements that express (certain parts of) a business/domain policy (e.g., defining terms of the domain language or defining/constraining domain operations) in a declarative manner, typically using a natural language or a visual language. Examples are:

   (R1) "The driver of a rental car must be at least 25 years old"

   (R2) "A gold customer is a customer with more than $1Million on deposit"

   (R3) "An investment is exempt from tax on profit if the stocks have been bought more than a year ago"

   (R4) "When a share price drops by more than 5% and the investment is exempt from tax on profit, then sell it"

   R1 is an *integrity rule*, R2 and R3 are *derivation rules*, and R4 is a *reaction rule* (see below for explanations of these rule categories).
2. At the platform-independent ***operational design (PIM) level***, rules are formal statements, expressed in some formalism or computational paradigm, which can be directly mapped to executable statements of a software system. Examples of rule languages at this level are SQL:1999, OCL 2.0, and DOM Level 3 Event Listeners. Remarkably, SQL provides operational constructs for all three business rule categories mentioned above: *checks/assertions* operationalize a notion of integrity rules, *views* operationalize a notion of derivation rules, and *triggers* operationalize a notion of reaction rules.
3. At the platform-specific ***implementation (PSM) level***, rules are statements in a language of a specific execution environment, such as Oracle 10g views, Jess 3.4, XSB 2.6 Prolog, or the Microsoft Outlook 6 Rule Wizard.

Generally, rules are self-contained knowledge units that typically involve some form of reasoning. They may, for instance, specify:

– static or dynamic integrity constraints (e.g. for constraining the state space or the execution histories of a system),
– derivations (e.g. for defining derived concepts),
– reactions (for specifying the reactive behavior of a system in response to events)

Given the linguistic richness and the complex dynamics of application domains, it should be clear that any specific mathematical account of rules, such as classical logic Horn clauses, must be viewed as a limited descriptive theory that captures just a certain fragment of the entire conceptual space of rules, and not as a definitive, normative account. Rather, we need a pluralistic approach to the heterogeneous conceptual space of rules. Therefore, the goal should be to define a family of rule languages capturing the most important types of rules. While these languages should come with a recommended standard semantics, their rule expressions may, in addition, allow alternate semantics, which are also considered acceptable. This will accommodate various formalisms based on non-standard logics, supporting temporal, fuzzy, defeasible, and other forms of reasoning.

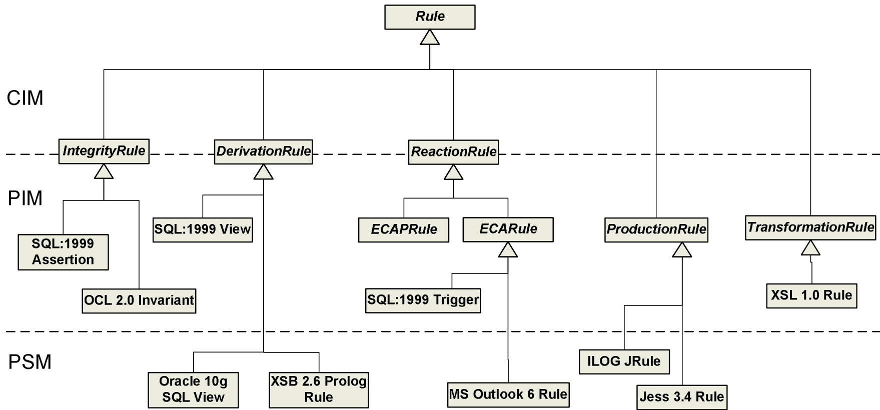


Fig. 1. Various rule concepts and rule languages at different levels of abstraction

We assume that the reader is familiar with the basic conceptual modeling constructs of UML class diagrams (types/classes, attributes, associations, role names, multiplicity constraints, aggregations, generalization) and to some degree also with OCL. We explain some of theses modeling constructs in the next section when discussing an example..

The structure of this article is as follows: after showing with an example ho to use UML class models for specifying vocabularies and rules in section 2, the foundational vocabularies of OWL/SWRL and RuleML are compared with each other in section 3. In section 4, different rule categories are discussed and modeled as class diagrams. In section 5, MOF/UML meta-models of OWL, SWRL and RuleML are presented. Finally, in section 6, the relationship between UML class models and OWL vocabularies is discussed.

2 Rule Modeling and Markup – An Example

An example, where a derived attribute in a UML class model is defined by a derivation rule, is the following:

> *A car is available for rental if it is not assigned to any rental contract and does not require service.*

This rule defines the derived Boolean-valued attribute isAvailable of the class RentalCar by means of an association isAssignedTo between cars and rental contracts and the stored Boolean-valued attribute requiresService, as shown in the UML class diagram in Fig. 2.

This class diagram specifies a vocabulary fragment consisting of

- two *basic entity types* (classes), `RentalCar` and `RentalContract`
- one *attribution fact type* that can be verbalized as: RentalCar has String as RentalCarID;

- two *subtypes* of `RentalCar`, `AvailableRentalCar` and the derived subtype `RentalCarRequiringService`, both being represented by Boolean-valued attributes
- an *association fact type*: RentalCar isAssignedTo RentalContract, which comes with three integrity rules and a clarification:

  1. *Functional*: It is necessary that each RentalCar isAssignedTo at most one RentalContract.
  2. *Inverse Total*: It is necessary that each RentalContract is assigned at least one RentalCar.
  3. *Inverse functional*: It is necessary that each RentalContract is assigned at least one RentalCar
  4. *Not total*: It is possible that a RentalCar isAssignedTo no RentalContract

An implicational OCL invariant, attached to the `RentalCar` class rectangle, is used to state that for a specific rental car whenever there is no rental contract associated with it, and it does not require service, then it must be available (for a new rental). In this OCL invariant expression, the condition `RentalContract->isEmpty()` means that the set of associated rental contracts must be empty.

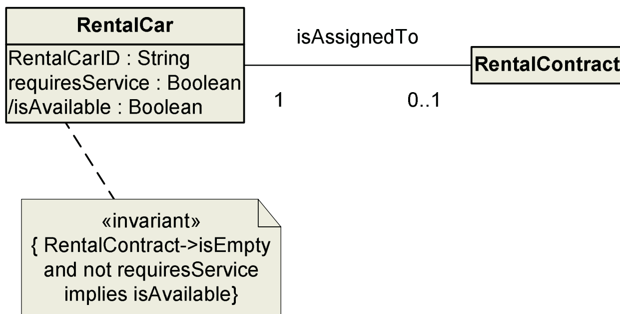


Fig. 2. An OCL invariant that constrains the derived attribute isAvailable

However, such an OCL invariant does not really define anything but rather puts a constraint on the model elements it refers to. OCL 2.0, in addition to expressing integrity rules ('invariants'), also allows to express derivation rules for defining derived elements of a class model. Using this possibility, we get the following OCL expression:

```
context RentalCar::isAvailable : Boolean derive:
RentalContract->isEmpty() and not requiresService
```

This OCL derivation rule assigns the truth value of the conjunction

```
RentalContract->isEmpty() and not requiresService
```

to the Boolean attribute `isAvailable` of the class `RentalCar`, and in this way it is a definition and not just a constraint.

We now present the concrete XML syntax of this rule according to the RuleML 0.88 syntax. Notice that the *head* element corresponds to the *Conclusion*, and the *body* element corresponds to the *Condition* of Fig. 6. It is assumed that the attribute `requiresService` is optional, that is it does not need to have a value (in case it is unknown whether a particular car requires service or not). By contrast, the attribute `isAvailable` is assumed to be mandatory.

The first condition of this rule, `RentalContract->isEmpty()`, corresponds to a negation-as-failure, which is expressed by the tag `<naf> in RuleML, while the second condition, not requiresService`, corresponds to a strong negation since it requires that the value of this Boolean attribute is explicitly `false`. If it would be unknown, its negation with `not` would result in unknown and not in true. So, this rule involves two kinds of negation, marked up with `<Naf>` and `<Neg>` in RuleML:

```
<Implies>
   <head>
      <Atom>
         <Rel>isAvailable</Rel>
         <Var>Car</Var>
      </Atom>
   </head>
   <body>
      <Atom>
         <Rel>RentalCar</Rel>
         <Var>Car</Var>
      </Atom>
      <Neg>
         <Atom>
            <Rel>requiresService</Rel>
            <Var>Car</Var>
         </Atom>
      </Neg>
      <Naf>
         <Atom>
         <Rel>isAssignedToRentalContract</Rel>
         <Var>Car</Var>
         </Atom>
      </Naf>
   </body>
</Implies>
```

Rule markup languages are a vehicle for using rules on the Web. They allow deploying, publishing and communicating rules on the Web. They are also converging towards a lingua franca for exchanging rules between different systems and tools.

In a narrow sense, a rule markup language is a concrete (XML-based) rule syntax for the Web. In a broader sense, it should be defined by an abstract syntax as a common basis for defining various concrete languages serving different purposes. The main purpose of a rule markup language is to permit reuse, interchange and publication of rules.

3 Foundational Concepts for Vocabularies and Rules

Rules are built on vocabularies, which include proper names designating individuals, type terms designating entity types (or classes) and fact types expressions designating fact types or predicates.

In this section, we discuss the *foundational* concepts (or *meta-concepts*) being used in this report and the terms we are using to designate them. These concepts, and their canonical designations, are described in a ***foundational vocabulary***, which is also called a *foundational* (or 'upper level') *ontology*. They define a range of top-level domain-independent ontological categories, which form a general foundation for more elaborated domain-specific vocabularies. Our foundational vocabulary is based on the Unified Foundational Ontology (UFO) proposed in [1,2].

Our analysis is focused on four languages for expressing vocabularies and rules:

1. SBVR – "Semantics of Business Vocabularies and Rules", the main submission to the OMG BSBR CFP [3]
2. UML – the *Unified Modeling Language* of the OMG [4]
3. RDF – the *Resource Description Framework* of the W3C [5]
4. OWL – the *Web Ontology Language* of the W3C [6]

All these languages come with their own foundational vocabulary, employing different (or the same) designations for the same (or different) concepts. We will therefore use our own 'unified' foundational vocabulary as defined in the first column, called REWERSE I1 (after the name of the REWERSE working group on rule markup), of the terminology tables below. The I1 foundational vocabulary helps to understand the differences and overlaps among these terminologies.

For simplicity, we will not always be consistent in distinguishing the conceptual from the terminological level; we will, for instance, often say "rule" instead of "rule expression", "fact" instead of "fact statement", and "fact type" instead of "fact type expression".

3.1 Things, Sets, Entities and Individuals

A ***thing*** is 'anything perceivable or conceivable'. This includes concrete entities and also abstract things such as sets A ***set*** is a *thing* that has other *things* as *members* (in the sense of set theory).

An ***entity*** is a *thing* that is not a *set*; neither the set-theoretic membership relation nor the subset relation can unfold the internal structure of an entity. An ***individual*** is an *entity* that does not have any *instances*, i.e., that is not an *entity type*. A ***data value*** is a member of a *datatype*, which is a particular kind of named *set*.

3.2 Entity Types and Datatypes

An ***entity type*** is an *entity* that has an *extension* (the set of entities that are instances of it) and an *intension*, which includes an applicability criterion for determining if an entity is an instance of it. A ***basic entity type*** is an entity type whose instances are individuals. A ***datatype*** is a *set* whose members are *data values*.

Table 1. Different kinds of things

<table>
<tr><th>REWERSE-I1</th><th>UML</th><th>SBVR</th><th>RDF</th><th>OWL</th></tr>
<tr><td>thing</td><td>n.a.</td><td rowspan="2">Thing</td><td>n.a.</td><td>n.a.</td></tr>
<tr><td>entity</td><td rowspan="2">object</td><td rowspan="2">resource (an instance of rdfs:Resource)</td><td>n.a.</td></tr>
<tr><td>individual</td><td rowspan="2">individual concept</td><td>individual (an instance of owl:Thing)</td></tr>
<tr><td>data value</td><td>data value</td><td>literal (an instance of rdfs:Literal)</td><td>data value</td></tr>
</table>

Table 2. Different kinds of entity types

<table>
<tr><th>REWERSE-I1</th><th>UML</th><th>SBVR</th><th>RDF</th><th>OWL</th></tr>
<tr><td>entity type</td><td rowspan="2">type / class</td><td rowspan="3">object type / general concept</td><td rowspan="2">class (an instance of rdfs:Class)</td><td>n.a.</td></tr>
<tr><td>basic (1st order) entity type</td><td>class (an instance of owl:Class, which is a subclass of rdfs:Class)</td></tr>
<tr><td>datatype</td><td>datatype</td><td colspan="2">datatype (an instance of rdfs:Datatype)</td></tr>
</table>

In Fig. 3, the foundational vocabulary about things, sets, entities and individuals adopted by I1 from UFO is described in the form of a UML class diagram.

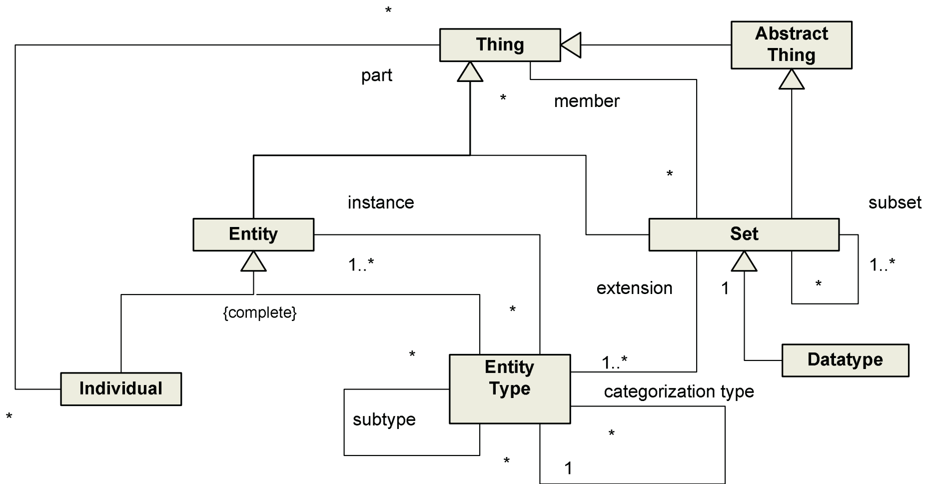


Fig. 3. The foundational vocabulary about things, sets, entities and individuals adopted by I1 from UFO

In Fig. 4 the foundational vocabulary supported by RDF(S) is summarized. Notice that rdfs:Class is an instance of itself. Fig. 5 describes the relationships between some basic RDF(S) concepts and their OWL counterparts.

3.3 Facts and Statements

We distinguish between 5 different kinds of *facts* (or atomic statements), as depicted by Table 3. In addition to the basic fact kinds of classification facts, association facts and attribution facts, we also consider categorization facts and aggregation facts. A categorization fact states that an entity, as an instance of a type, is an instance of a 'category', i.e. a subtype of that type. An aggregation fact is a part-whole statement.

3.4 Fact Types

A fact type corresponds to a predicate in predicate logic. But while there is no further distinction between different kinds of predicates in standard predicate logic, we distinguish between four different kinds of fact types as depicted in Table 4.

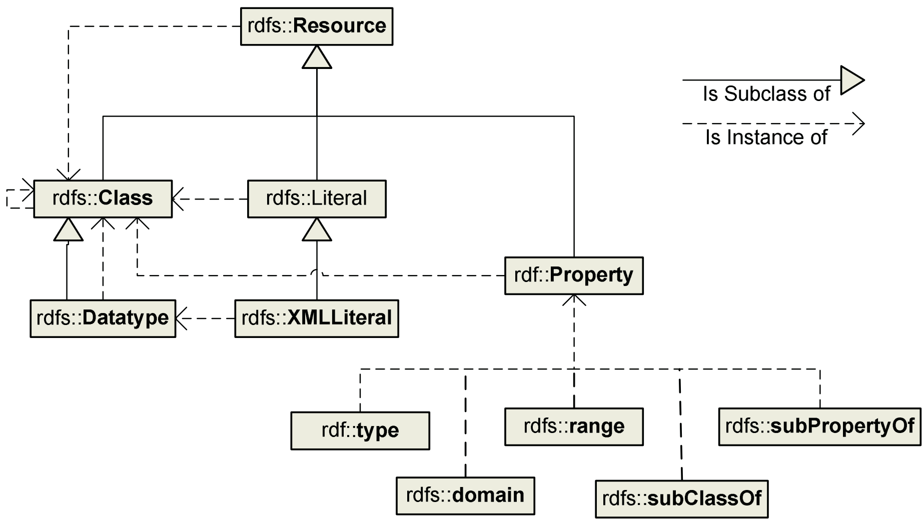


Fig. 4. The foundational vocabulary supported by RDF(S)

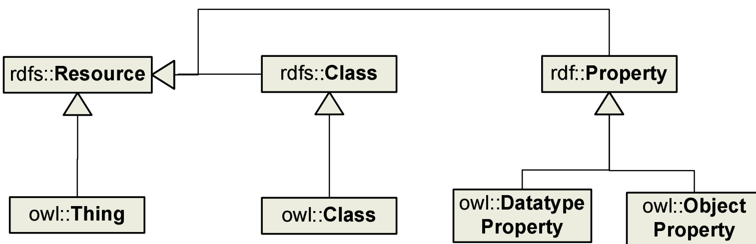


Fig. 5. The relationships between RDF(S) concepts and their OWL counterparts

Table 3. Different kinds of facts

REWERSE-I1	*UML*	*SBVR*	*RDF*	*OWL*
association fact	link	associative fact	n.a.	n.a.
binary association (reference property) fact	binary link	binary associative fact	triple, statement	individual-valued property fact
attribution fact	object-attribute-value triple	is-property-of fact		data-valued property fact
classification fact	instanceOf dependency	assortment fact	rdf:type statement	classification fact
categorization fact	n.a.	categorization fact	n.a.	n.a.
aggregation fact	aggregation link	partitive fact	n.a.	n.a.
generalization statement	generalization	specialization fact	subclassOf statement	subclass axiom

Table 4. Different kinds of fact types

REWERSE-I1	*UML*	*SBVR*	*RDF*	*OWL*
association fact type	association	fact type	n.a.	n.a.
binary association fact type	binary association	binary associative fact type	property (an instance of rdf:Property)	individual-valued property (an instance of owl:ObjectProperty)
attribution fact type	attribute	is-property-of fact type		data-valued property (an instance of owl:DatatypeProperty)
categorization fact type	n.a.	categorization fact type	n.a.	n.a.
aggregation fact type	aggregation	partitive fact type	n.a.	n.a.

4 Rule Categories

We briefly discuss the main categories of rules: integrity rules, derivation rules, reaction rules, production rules and transformation rules. The different parts of a rule expression can be any of the five semantic categories listed in Table 5.

Table 5. Semantic categories of rule expression parts

Type	*Semantic Category*
Logical Sentence	Truth value
Logical Formula	Function from variable bindings to truth values
Event Term	Event
Action Term	Action
Term	Can denote anything (an element from some term algebra)

4.1 Derivation Rules

Logical derivation rules (also called *deduction rules*), in general, consist of one or more *conditions* and one or more *conclusions*[6], which are both roles played by expressions of the type LogicalFormula.

For specific types of derivation rules, such as definite Horn clauses or normal logic programs, the types of condition and conclusion are specifically restricted.

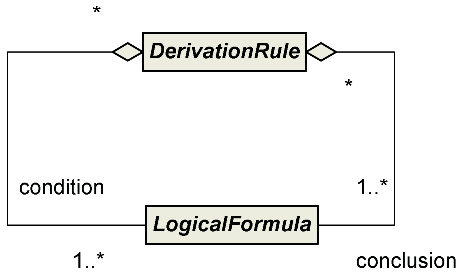


Fig. 6. The abstract concept of derivation rules

For instance, in RuleML 0.85, conditions are quantifier-free logical formulas with weak and strong negation, called *AndOrNafNeg-Formula* in Fig. 7. More precisely, they are quantifier-free predicate logic formulas with weak and strong negation, called *AndOrNafNeg-PL-Formula* (this formula class specializes the abstract class *AndOrNafNeg-Formula*, which admits also of other kinds of atoms such as OCL-like atoms, by restricting it to predicate logic atoms).

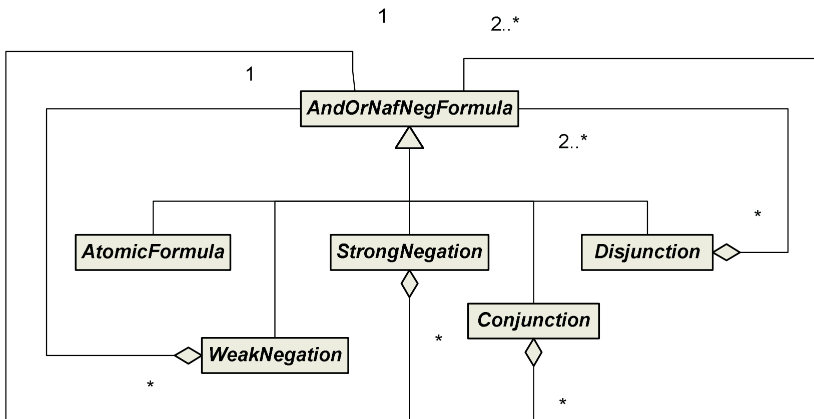


Fig. 7. Quantifier-free formulas with weak and strong negation

[6] Notice that we don't consider rules with no condition or no conclusion. These expressions are better not called "rules", but "facts" and "denial constraints".

The distinction between weak and strong negation is present in several computational languages: in *extended logic programs* it is present in explicit form, while it is only implicitly present in SQL and OCL. Intuitively speaking, weak negation captures the absence of positive information, while strong negation captures the presence of explicit negative information (in the sense of Kleene's 3-valued logic). Under the preferential model semantics of minimal/stable models, weak negation captures the computational concept of negation-as-failure (or closed-world negation).

There are three different kinds of atoms in RuleML, as depicted by Fig. 8.

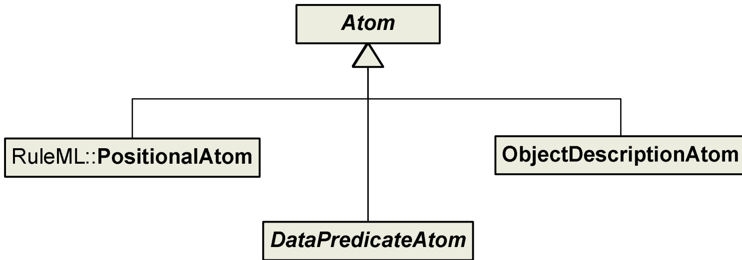


Fig. 8. Three kinds of atomic formulas in RuleML

A *positional atom* corresponds to an atomic formula in standard predicate logic. A *data predicate atom* (also called *built-in*) is formed with the help of a datatype predicate. An object description atom corresponds to an OWL individual description: it refers to an individual, classifies it, and makes a number of property-value-assertions about it, as depicted in Fig. 10.

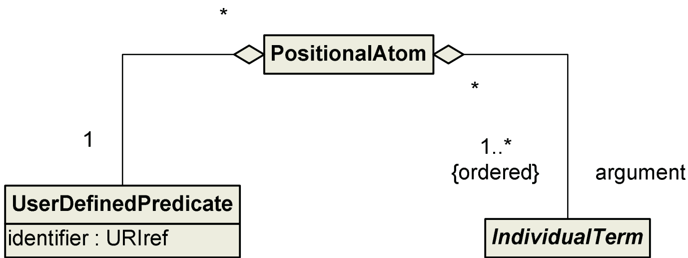


Fig. 9. A positional atom consists of a user-defined predicate and a sequence of one or more individual terms (as defined in Fig. 20) as arguments

In the example discussed in section 2 it may seem that the implicational invariant is equivalent to the corresponding derivation rule. However, there is an important conceptual difference between an implicational constraint $p \rightarrow q$ and the corresponding derivation rule *from p derive q*. While the former only constrains the logical state space (and is also satisfied by the truth of $\neg p$), it does not prescribe a derivation procedure to be applied for deriving the conclusion q. We may consider the rule *from p derive q* to be one of several possible derivation procedures that comply

with the constraint $p \rightarrow q$. Another one would be the derivation procedure consisting of the two rules *from p derive r* and *from r derive q*.

Derivation rules should be semantically distinguished from implications. While an implication is an expression of a logical formula language (such as classical predicate logic or OCL), typically possessing a truth-value, a derivation rule is a meta-logical expression, which does not possess a truth-value, but has the function to generate derived sentences. There are logics, which do not have an implication connective, but which have a derivation rule concept. In standard logics (such as classical and intuitionistic logic), there is a close relationship between a derivation rule (also called "sequent") and the corresponding implicational formula: they both have the same models. For nonmonotonic rules (e.g. with negation-as-failure) this is no longer the case: the intended models of such a rule are, in general, not the same as the intended models of the corresponding implication.

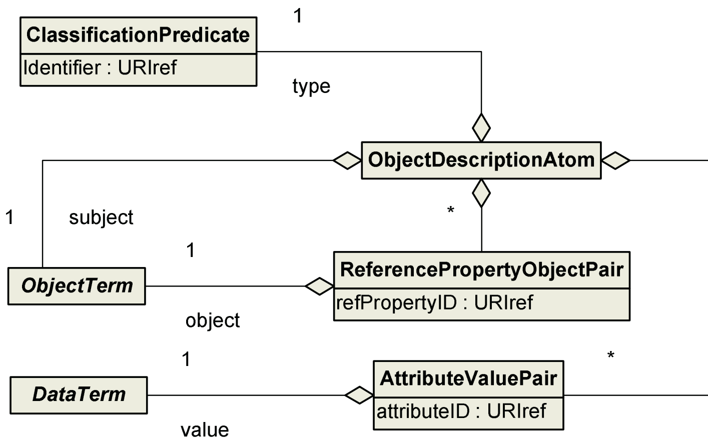


Fig. 10. An object description atom refers to an object (its 'subject'), classifies it, and makes a number of property-value-assertions about it

4.2 Integrity Rules (Constraints)

Integrity rules, also known as (integrity) constraints, consist of a constraint modality and a constraint assertion, which is a sentence in some logical language such as first-order predicate logic or OCL. This is depicted in Fig. 11. We consider two constraint modalities: the ***alethic*** and the ***deontic*** one. The alethic constraint modality can be expressed by a phrase such as "it is necessarily the case that". The deontic constraint modality can be expressed by phrases such as "it is obligatory that" or "it should be the case that". Notice that in English the phrase "it must be the case that" is ambiguous: it can denote either the alethic or the deontic modality.

The constraint assertion is a logical sentence that *must necessarily*, or that *should*, hold in all evolving states and state transition histories of the discrete dynamic system to which it applies. Notice that not only software systems, but also physical,

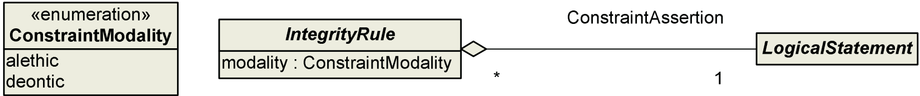


Fig. 11. The abstract concept of integrity rules

biological and social systems, such as organizations, can be viewed as discrete dynamic systems. Typically, we describe the natural and social laws that govern material (i.e. physical, biological and social) systems in the form of CIM integrity rules (at the domain modeling level). Then, when we transform the domain model into an operational design, we formalize these rules in the chosen PIM language, after which they no longer refer to the material system itself but to its computational model. So, a PIM constraint refers to the state (and execution histories) of the software system that models (or represents) the material system under consideration.

Rule R1 is an example of a (deontic) static CIM constraint. An example of a (deontic) dynamic CIM constraint is: "The confirmation of a rental reservation must lead to an allocation of a car of the requested car group for the requested date prior to that date". Well-known languages for expressing PIM constraints are SQL and OCL. In logic programming, rules with empty heads (also called "denials") corresponding to the negation of the conjunction of all body atoms are sometimes used as constraints.

4.3 Reaction Rules

Reaction rules are the second important type of rule in RuleML. Integrity and transformation rules have not received as much attention as derivation and reaction rules. Reaction rules are considered to be the most important type of business rule in [7].

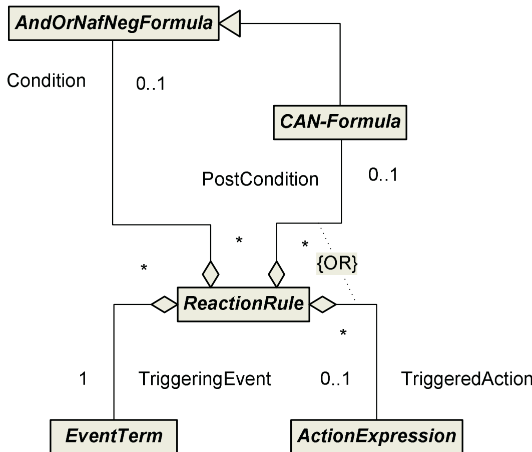


Fig. 12. The abstract concept of reaction rules

Reaction rules consist of a mandatory triggering event term, an optional condition, and a triggered action term or a post-condition (or both), which are roles of type EventTerm, LogicalFormula, ActionTerm, and LogicalFormula, respectively, as shown in Fig. 12. While the condition of a reaction rule is, exactly like the condition of a derivation rule, a quantifier-free formula, the post-condition is restricted to a conjunction of possibly negated atoms (called CAN-formula)..

Action and event terms may be composite and specified in different ways. For instance, the UML Action Semantics could be used to specify triggered actions in a platform-independent manner.

There is a little known parallel between derivation rules and reaction rules. Reaction rules are to dynamic (temporal logic) implication constraints what derivation rules are to static implication constraints.

There are basically two types of reaction rules: those that do not have a post-condition, which are the well-known *Event-Condition-Action (ECA)* rules, and those that do have a post-condition, which we call *ECAP rules*.

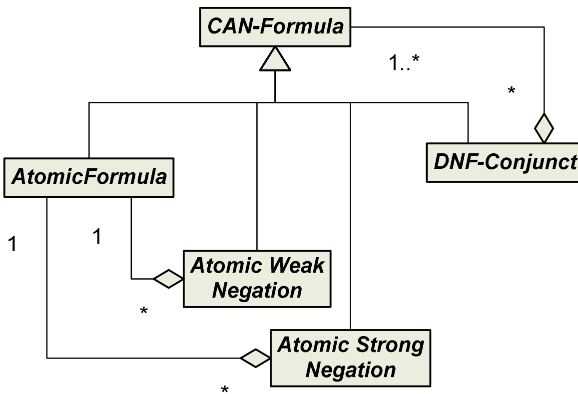


Fig. 13. The post-condition in a reaction rules is a conjunction of possibly negated atoms, also called CAN-formula

The post-condition of a reaction rule is either an atomic formula, a negation of an atomic formula or a conjunction of these (thus corresponding to a disjunctive normal form conjunct). This is called a CAN-Formula in Fig. 13. Such a definite formula specifies an update in a declarative way.

Event-Condition-Action-Postcondition (ECAP) rules extend ECA rules by adding a postcondition that accompanies the triggered action. ECAP rules allow specifying the effect of a triggered action on the system state in a declarative manner, instead of specifying this state change procedurally by means of corresponding state change operations (like SQL UPDATEs).

An application-specific ECA rule language may be used in software applications for handling application events in an automated fashion. A prominent example of this is the Microsoft Outlook rule wizard, which allows specifying email handling rules referring to incoming (or outgoing) message events.

4.4 Production Rules

Production rules consist of a condition and a produced action, which are roles of the type LogicalFormula and ActionTerm, respectively, as shown in Fig. 14. While OCL could be used in a platform-independent production rule language to specify conditions on an object-oriented system state, the UML Action Semantics could be used to specify produced actions.

These rules have become popular as a widely used technique to implement ‘expert systems’ in the 1980s. However, in contrast to (e.g. Prolog) derivation rules, the production rule paradigm lacks a precise theoretical foundation and does not have a formal semantics. This problem is partly due to the fact that early systems used production/ECA-like rules, where the semantic categories of a rule’s events and conditions in the left-hand-side, and of its actions and effects in the right-hand-side, were mixed up.

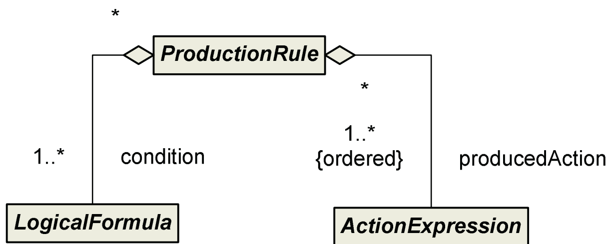


Fig. 14. The abstract concept of production rules

Production rules do not explicitly refer to events, but events can be simulated in a production rule system by externally asserting corresponding facts into the working memory. In this way, production rules can implement reaction rules.

A derivation rule can be implemented by a production rule of the form if-*Condition*-then-*assert*-*Conclusion* using the special action *assert* that changes the state of a production rule system by adding a new fact to the set of available facts.

Production rule platforms are the rule technology that is most widely used in the business rules industry. Well-known examples of production rule systems are JESS, Fair Isaac/Blaze Advisor, iLOG Rules/JRules, CA Aion, ART*Enterprise, Haley, and ESI Logist.

5 Semantics of Business Vocabularies and Rules

The *Semantics of Business Vocabularies and Rules (SBVR)* is an OMG proposal [3] for developing and structuring business vocabularies suited for business people to express business rules. A business vocabulary contains 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.

The SBVR follows a common-sense definition of ‘business rule’ as a *rule that is under business jurisdiction*. ‘Under business jurisdiction’ is taken to mean that the business can enact, revise and discontinue their business rules as they see fit.

All business rules need to be *actionable*. This means that a person who knows about a business rule could observe a relevant situation (including his or her own behavior) and decide directly whether or not the business was complying with the rule. Just because business rules are actionable, this does *not* imply they are always automatable. Many business rules, especially operative business rules, are *not* automatable in IT systems.

In SBVR, a rule is “an element of guidance that introduces an obligation or a necessity”. The two fundamental categories of rule are:

– **Structural Rule** (necessities): These are rules about how the business chooses to organize (i.e., ‘structure’) the things it deals with. Structural Rules supplement definitions.:
– **Operative Rules** (obligations): These are rules that govern the conduct of business activity. In contrast to Structural Rules, Operative Rules are ones that can be *directly* violated by people involved in the affairs of the business.

The preferred mode of expression for vocabularies and rules is *SBVR Structured English*, a controlled English that works with verbalization patterns and font markup.

The SBVR Structured English is not meant to offer all of the variety of common English, but rather, it uses a small number of English structures and common words to provide a simple and straightforward mapping.

The following keywords are used in SBVR Structured English:

- *IF, THEN, OR, AND, NOT* – designate logical connectives
- The keyword "the": 1. Used with a designation to make a pronominal reference to a previous use of the same designation; this is formally a binding to a variable of a quantification. 2. Introduction of a name of an individual thing or of a definite description.
- The keywords "a, an": Universal or existential quantification, depending on context based on English rules.
- The keyword "that": 1. When preceding a designation for a type or role, this is a binding to a variable (as with ‘the’). 2. When after a designation for a type or role and before a designation for a fact type, this is used to introduce a restriction on things denoted by the previous designation based on facts about them

Below, we use the following font types markup for the different parts of a SBVR Structured English expression:

- **type term –** designates a type (that is part of a vocabulary being used or defined)
- *type term* **–** This markup is applied to a type term in the special case where the term is used to name the represented concept rather than to refer to things denoted by the term. This is a reference to the concept itself.
- connecting verb phrase – designates a (user-defined) domain predicate symbol
- *predefined connecting verb phrase* – designates a predefined predicate symbol
- name – designates an individual or data value

This markup differs from the original SBVR markup, but is equivalent. The description of the SBVR Structured English is divided into sections:

– Expressions in SBVR Structured English
– Describing a Vocabulary
– Vocabulary Entries
– Specifying a Rule Set
– Rule and Clarification Entries

There are two styles of SBVR Structured English:

1. Prefixed Rule Keyword Style
2. Embedded (Mixfix) Rule Keyword Style

The Prefix Style introduces rules by prefixing a statement with keywords that convey a modality

Operative Business Rules and Clarifications	***Structural Rules and Clarifications***
It is obligatory that	It is necessary that
It is prohibited that	It is impossible that
It is permitted that	It is possible that

The Embedded Style features the use of rule keywords embedded (usually in front of verbs) within rules statements of appropriate kind. The following key words are used within expressions having a verb (often modified to be infinitive) to form verb complexes that add a modal operation.

Operative Business Rules and Clarifications	***Structural Rules and Clarifications***
… must …	… always …
… must not …	… never …
… may …	… sometimes …

5.1 Examples of SBVR-Style Rule Expressions

UML associations are verbalized as ***association fact type expressions***. For instance, consider the binary association between the classes rental car and rental in Fig. 15. It can be verbalized by the following fact type expressions:

rental car <u>is assigned to</u> *a* **rental**

Rules can be verbalized on the basis of fact type expressions. For instance, the rule that defines the derived association fact type

rental car <u>is available at</u> **branch**

can be expressed in the following way:

IF **rental car** <u>is stored at</u> *the* **branch** *AND* **rental car** <u>is</u> *NOT* <u>a *rental car scheduled for service*</u> *AND* **rental car** <u>is</u> *NOT* <u>assigned to</u> *a* **rental** *THEN* **rental car** <u>is available at</u> *the* **branch.**

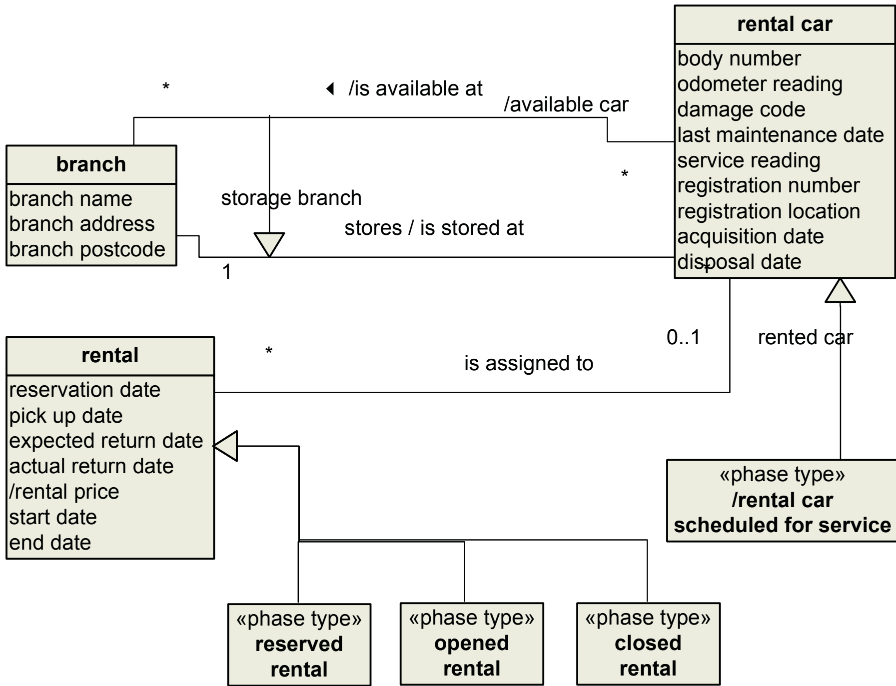


Fig. 15. A business vocabulary fragment for the domain of car rental

In OCL, this derivation rule can be expressed as:

```
context Branch::availableCar: RentalCar derive:
self.storedCar->select( c |
   not oclIsKindOf( RentalCarScheduledForService)
   and c.Rental->isEmpty())
```

6 MOF/UML Metamodels as Language Definitions

UML class models also allow to specify the abstract syntax of a language. The set of class modeling core constructs needed for this purpose is called *Meta Object Facility (MOF)*. The MOF/UML language models are also called *metamodels*. They allow a concise definition of a language in a graphical notation.

We briefly show how the abstract syntax of OWL, SWRL and RuleML can be defined by means of MOF/UML language models.

6.1 OWL

The W3C *Web Ontology Language* OWL defines an ontology as a set of axioms and a set of facts, as shown in Fig. 16.

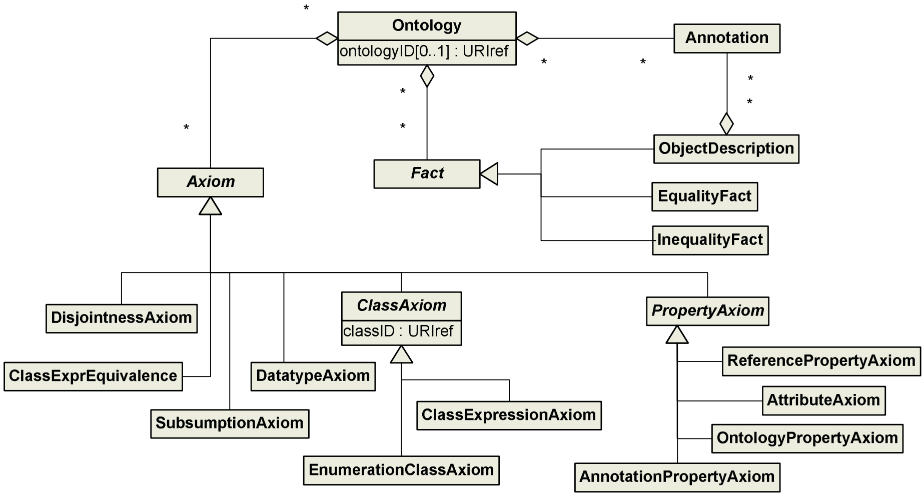


Fig. 16. An OWL ontology consists mainly of 'axioms' and 'facts'

There are six kinds of axioms. Three kinds of axioms allow to state conceptual relationships: disjointness axioms, class expression equivalences and subsumption atoms; whereas the remaining kinds of axioms allow to 'define' datatypes, classes and (various kinds of) properties.[7] A fact is either an equality/inequality assertion, or an 'individual description', which refers to an individual term and aggregates a number of classification facts and property-value-facts about it, as depicted in Fig. 17.

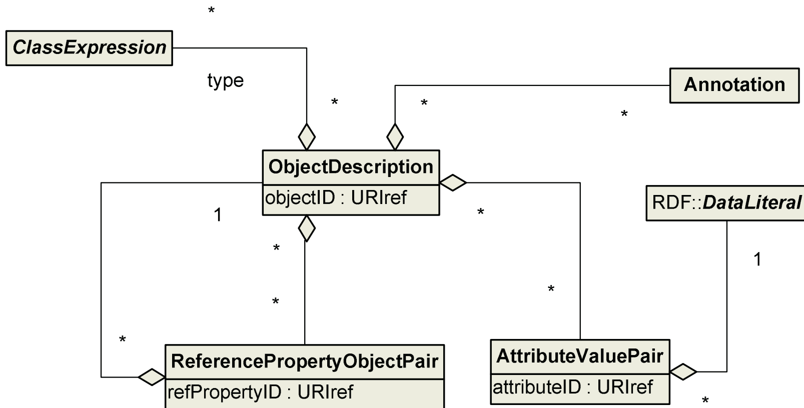


Fig. 17. An OWL 'individual description fact' is a collection of classification facts, attribution facts and binary association facts, all concerning one particular individual

[7] Notice that, strictly speaking, the semantics of OWL does not support the computational distinction between definition and constraint, which is reflected by the distinction between invariants and derivation rules in OCL, and which is also an essential part of the SBVR approach. Class 'definitions' in OWL are typically expressed by means of equivalence axioms.

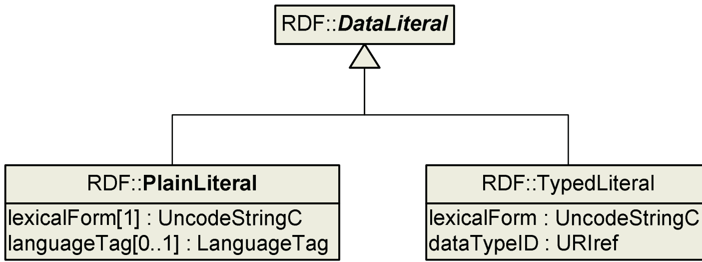


Fig. 18. An RDF data literal is either a plain literal, possibly associated with a language, or a typed literal

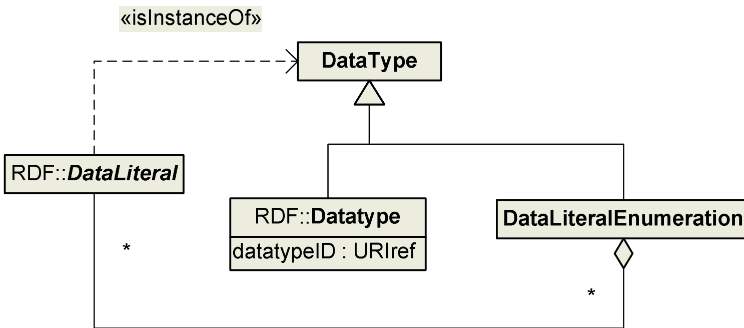


Fig. 19. A datatype in OWL is an RDF datatype or a data literal enumeration, or it is equal to the set of RDF literals

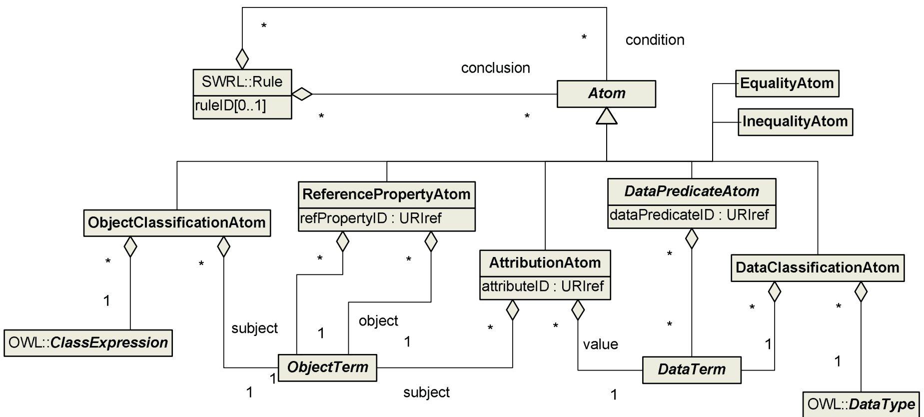


Fig. 20. The abstract syntax of SWRL rules

6.2 SWRL

The *Semantic Web Rule Language (SWRL)* extends the concept of an OWL ontology by adding a notion of logical variables and terms as well as a seventh kind of axiom,

called 'rule', which is a kind of implication that allows to include property atoms and built-in atoms in the condition and conclusion part of a rule.

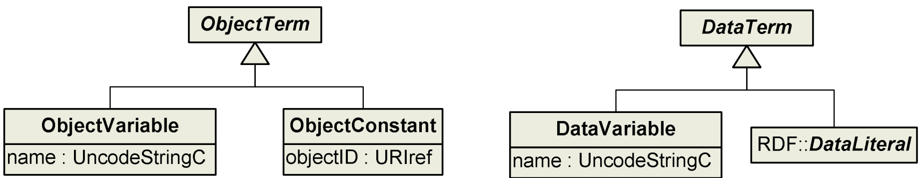


Fig. 21. There are two kinds of logical terms: object terms and data terms

6.3 RuleML

In RuleML 0.88, a rulebase or knowledge base (KB) consists of universally quantified atoms, implications and atom equivalences, as depicted in Fig. 23. RuleML atoms have been defined in Fig. 8.

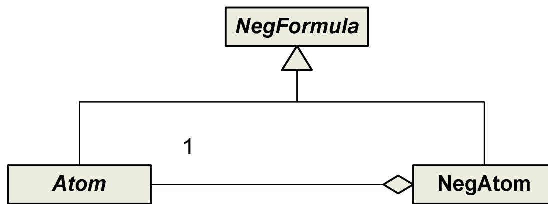


Fig. 22. The conclusion of a RuleML derivation rule is either an atom or a strongly negated atom

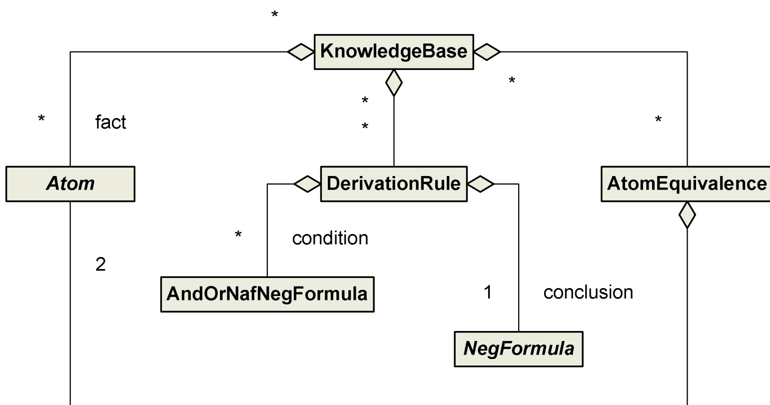


Fig. 23. A RuleML knowledge base consists of universally quantified atoms, implications and atom equivalences

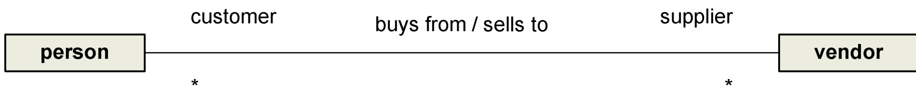
7 Correspondence Between OWL and UML

As already described in Table 1, there is a close correspondence between OWL and UML. We summarize the commonalities and differences between UML and OWL in Table 6.

Table 6. Commonalities and differences between UML and OWL

UML	*OWL*
datatype	datatype
class	class
n.a.	class description
association	n.a.
binary association	individual-valued property (an instance of owl:ObjectProperty)
attribute	data-valued property (an instance of owl:DatatypeProperty)
aggregation	n.a.
multiplicity constraint	cardinality restriction
generalization	subsumption axiom
n.a.	class expression equivalence
n.a.	anonymous class expression

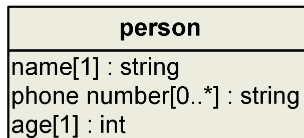
UML binary associations correspond to OWL object properties. For instance, the association



corresponds to the following OWL property axiom (expressed in the abstract syntax of OWL):

```
ObjectProperty( supplier
   domain(person) range(vendor) inverseOf(customer))
```

UML attributes correspond to OWL datatype property axioms. For instance, consider the following attributes of a class **person**:



The attribute *phone number* corresponds to the following OWL datatype property axiom:

```
DatatypeProperty( phone_number
   domain(person) range(xsd:string))
```

UML multiplicity constraints correspond to OWL cardinality restrictions. But while the graphical notation for multiplicity constraints in UML is simple and elegant, the

OWL syntax for cardinality restrictions is rather cumbersome and hard to read. A related issue is the lack of a shorthand for total properties. While properties can be declared to be functional and inverse functional, there is no corresponding shorthand construct for declaring a property to be total, resp. inverse total.

Another usability issue is the lack of a convenient mechanism in OWL to declare classes as mutually disjoint, which is the default assumption in UML

Since many core constructs of UML class models can be mapped to OWL, such a mapping provides a logical semantics for UML class models Exploiting this mapping possibility and the inference tools available for OWL, UML tools could e.g. check the consistency of a class diagram by running an OWL. inference engine.

8 Conclusions

We have shown that there is a close correspondence between the Web ontology language OWL and the vocabulary language of UML class diagrams, which can be exploited for capturing OWL ontologies with the more user-friendly graphical notation of UML. UML class diagrams, in the form of MOF/UML metamodels, can also be used to define the abstract syntax of OWL, SWRL and RuleML. These language metamodels provide a level of abstraction that allows to unify apparently distinct constructs. For instance, the metamodel for RuleML 'slot atoms' (better called *object description atoms*) shown in Fig. 10, reveals that RDF descriptions and OWL individual descriptions can be mapped to this RuleML construct.

References

[1] G. Guizzardi & G. Wagner. A Unified Foundational Ontology and some Applications of it in Business Modeling. In P. Green and M. Rosemann (Eds.), Business Systems Analysis with Ontologies, IDEA Publishing, 2005.

[2] G. Guizzardi & G. Wagner. Towards Ontological Foundations for Agent Modelling Concepts Using the Unified Foundational Ontology. In P. Bresciani et al. (Eds.): AOIS 2004, LNAI 3508, pp. 110 – 124, Springer-Verlag, 2005.

[3] Semantic of Business Vocabulary and Business Rules (SBVR). Revised Submission to OMG BEI RFP br/2003/06/03, http://www.omg.org/cgi-bin/doc?bei/2005-03-01.

[4] UML, http://www.uml.org/.

[5] G.Klyne and J.J.Caroll (Eds.), Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C, 2004.

[6] OWL Web Ontology Language, http://www.w3.org/2004/OWL.

[7] Taveter K., Wagner, G.: Agent-Oriented Enterprise Modeling Based on Business Rules. In Proc. of 20th Int. Conf. on Conceptual Modeling (ER2001), Springer-Verlag, LNCS 2224, pp. 527–540, 2001.