# Towards a general web rule language

## Gerd Wagner*

Institute of Informatics,
Brandenburg University of Technology at Cottbus, Germany
E-mail: G.Wagner@tu-cottbus.de
*Corresponding author

## Carlos Viegas Damásio

Centro de Inteligência Artificial, Universidade Nova de Lisboa,
Caparica, Portugal
E-mail: cd@di.fct.unl.pt

## Grigoris Antoniou

Institute of Computer Science, FORTH-ICS, Greece
E-mail: antoniou@ics.forth.gr

**Abstract:** A general web rule (markup) language has several purposes. It may serve as a lingua franca to exchange rules between different systems and tools. It may be used to express derivation rules for enriching web ontologies by adding definitions of derived concepts or for defining data access permissions; to describe and publish the reactive behaviour of a system in the form of reaction rules; and to provide a complete XML-based specification of a software agent. Further uses may arise in novel web applications. In this paper, we consider the problem of how to design a general web rule language that can be used for these and for future emerging purposes. Given the great diversity of rule concepts and existing rule languages, such a language will consist of several overlapping sublanguages that share a common metamodel. The development of this rule metamodel is a difficult conceptualisation and integration problem.

**Keywords:** rule markup language; rule metamodel; rules; semantic web.

**Reference** to this paper should be made as follows: Wagner, G., Viegas Damásio, C., and Antioniou, G. (2005) 'Towards a general web rule language', *Int. J. Web Engineering and Technology*, Vol. 2, Nos. 2/3, pp.181–206.

**Biographical notes:** Gerd Wagner is Professor of Internet Technology at the Brandenburg University of Technology at Cottbus, Germany. His research interests include agent-orientated modelling and agent-based simulation, foundational ontologies, (business) rule technologies and the semantic web. He has published a book on *Foundations of Knowledge Systems* with Kluwer Academic Publishers in 1998. He is a steering committee member of the international rule markup language standardisation effort *RuleML.org* and the leader of a work package on rule markup in the European research network *REWERSE.net*. He is also participating in the (business) rule standardisation activities of the Object Management Group.

Carlos Viegas Damásio is an Assistant Professor at the Department of Informática of Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, Portugal. He is also a member of the *REWERSE* European network of Excellence and participates in the RuleML initiative. His research interests include design, semantics and implementation of rule languages for the semantic web. His main background is in logic programming, in particular the study of semantics for dealing with contradiction, imprecision and uncertainty.

Grigoris Antoniou is Professor of Computer Science at the University of Crete, Greece, and head of the Information Systems Laboratory at the Institute of Computer Science of FORTH in Greece. His main research interests are in the area of knowledge systems: how to represent knowledge using logic and how to automatically deduce conclusions from knowledge. He has specialised in nonmonotonic reasoning, which deals with incomplete and inconsistent information.

## 1 Introduction

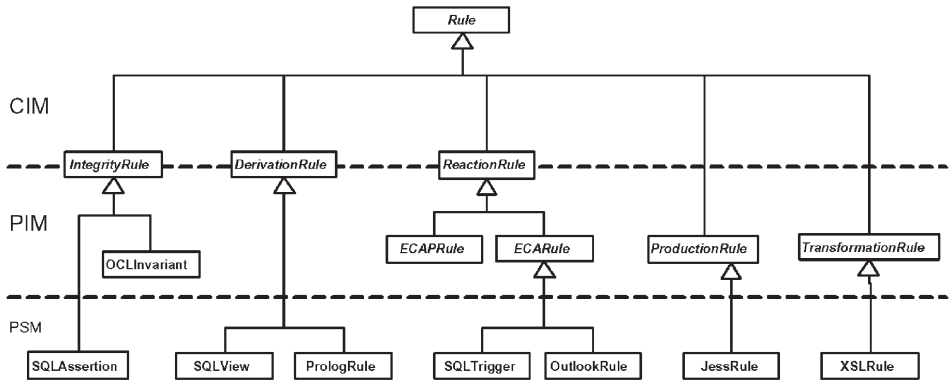
Rules are considered to be a design issue for the semantic web and have been a topic of discussion in the W3C Web Ontology Working Group, but have not been included in the web ontology language OWL.[1] It is expected that there will be a W3C Working Group for developing a W3C rule markup language, possibly starting in 2005.

Rule markup languages, that allow the expression of business rules as modular, stand-alone units in a declarative way, and to publish them and interchange them between different systems and tools, will play an important role in facilitating business-to-customer (B2C) and business-to-business (B2B) interactions over the web.

As depicted in Figure 1, we may consider rules at three different levels:[2]

- At the ('computation-independent') problem domain level, rules are statements that express (certain parts of) a business/domain policy (e.g. defining terms of the domain language, defining or constraining domain operations) in a declarative manner, typically in a natural language or a visual language. These statements are also called business rules. As shown in Figure 1 and discussed in Section 2, there are three important types of business rules: integrity rules, derivation rules and reaction rules. Examples of these rule types are:
  - the driver of a rental car must be at least 25 years old
  - an investment is exempt from profit taxes, if it has been in the portfolio for more than one year
  - if a share price drops by more than 5% and the corresponding investment is exempt from profit taxes, then sell the investment.
- At the (platform-independent) computational specification level, rules are statements in a computational formalism. Rules at this level may operationalise business rules and may be mapped into executable statements of a software system platform. Rule languages at this level are RuleML 0.89, SQL:1999, OCL 2.0, or ISO Prolog.
- At the (platform-specific) execution level, rules are statements in a specific executable language, such as Jess 6.1, XSB 2.6, or the Rule Wizard of Microsoft Outlook 9.

**Figure 1** Rule concepts at three different abstraction levels: computation independent (CIM), platform-independent (PIM) and platform-specific (PSM) modelling. Notice that integrity, derivation and reaction rules are both CIM and PIM concepts, while production and transformation rules are just PIM, but not CIM, concepts




In any case, rules are modular, stand-alone units, which directly support or involve some form of reasoning. They may, for instance, specify

- derivations (e.g. for defining derived concepts or for establishing permissions)
- static and dynamic integrity constraints (e.g. for constraining the state space or the execution histories of a system)
- reaction patterns (e.g. for specifying the reactive behaviour of a system in response to events)
- transformations.

Given the linguistic richness and the dynamics of natural problem domains, it should be clear that any specific account of rules, such as viewing rules as classical logic Horn clauses, must be considered as a limited descriptive theory that captures only a certain fragment of the entire conceptual space of rules, and not as a definitive normative account.[3]

What is needed is a pluralistic approach to the heterogeneous conceptual space of rules. Therefore, in the Rule Markup Language (RuleML) standardisation effort,[4] the goal is to define a family of rule languages capturing the most important types of rules. Each of these languages shall have a recommended formal semantics, but may have one or more alternate acceptable semantics. Such an approach accommodates various formalisms based on non-standard logics, supporting temporal, fuzzy, defeasible and other forms of reasoning.

This paper discusses the question of how to design a general rule markup language, taking into account the experiences made with the initial RuleML beta versions.[4] Given the great diversity of rule concepts and existing rule languages, such a language will consist of several overlapping sublanguages that share a common metamodel, the development of which is a difficult conceptualisation and integration problem.

The purpose of this paper is not to propose a solution to this problem but rather to discuss the problem.

The paper is organised as follows. In Section 2 the concept of business rules is discussed, Sections 3 and 4 contain a brief survey of rule languages, in Section 5 the problems of an abstract syntax and a semantics are discussed, and Section 6 presents use cases, design goals and requirements for a general rule markup language.

## 2 Business rules

Business rules refer to the hundreds, if not thousands, of policies, procedures and definitions that govern how a company operates and interacts with its customers and partners. They represent the knowledge a company has about its business. Typically, a company does not maintain an explicit list of all its business rules. Rather these rules are implicitly expressed in documents about things such as corporate charters, marketing strategies, pricing policies, and customer relationship management practices, as well as in contracts and other legal documents. They are also implicitly expressed (or 'buried') in many pieces of program code distributed across the client, application logic and database tiers of modern business application systems. It is the task of requirements engineering and domain analysis to capture all these business rules that are at the core of functional requirements.

The term *business rule* can be understood both at the level of a business domain and at the operational level of an information system. The more fundamental concepts are business rules at the level of a business domain, captured in the form of natural language statements. In certain cases, these statements can be operationalised in a computational formalism and then implemented with some technology platform. In other cases they cannot or do not have to be operationalised and automated, but are only expressed in natural language for the purpose of communication and documentation.

It follows from these remarks that a general web rule language should allow the expression of rules both in natural languages and in declarative executable languages.

Three basic types of business rules have been identified in the literature (see, e.g. Taveter and Wagner (2001): *integrity rules* – also called 'integrity constraints', *derivation rules* – also called 'deduction rules', and *reaction rules* – also called 'stimulus response rules' or 'event-condition-action (ECA) rules'. A fourth type, *deontic assignments*, has only been marginally discussed.

An *integrity rule*, or constraint, specifies a condition that must hold in all evolving states and state transition histories of an enterprise viewed as a discrete dynamic system. There are *state constraints* and *process constraints*. State constraints must hold at any point in time. An example of a state constraint applied by a car rental company is: *The driver must be at least 25 years old*. Process constraints refer to the dynamic integrity of a system; they restrict the admissible transitions from one state of the system to another.

A *derivation rule* is a statement of knowledge that is derived from other knowledge by an inference or a mathematical calculation. Derivation rules allow the capture of mathematical, terminological and heuristic domain knowledge about predicates whose extension does not have to be stored explicitly because it can be derived from existing or other derived information on demand. An example of a derivation rule is the following definition of a customer category by an insurance company: *Our 'Gold' customers are those who own more than one policy and spend more than $1000 per year in total premiums.*

*Reaction rules* state under which conditions actions must be taken in response to events; this includes triggering event conditions, pre-conditions, and post-conditions (effects). An example of a reaction rule from the domain of car rental is:*When a customer requests a car reservation, the car rental branch checks with its headquarters to make sure that the customer is not blacklisted.* In general, reaction rules consist of an event condition formula, a state condition (or precondition) formula, an action term and a state effect (or postcondition) formula. Thus, they can also be called Event-Condition-Action-Postcondition (ECAP) rules, subsuming Event-Condition-Action (ECA) as a special case.

*Deontic assignments of powers*, rights and duties to (types of) internal agents define the deontic structure of an organisation, guiding and constraining the actions of internal agents. The corresponding rules are called empowerment, permission/prohibition and duty assignment rules. An example of a permission rule is: *Only the branch manager has the right to grant special discounts to customers*.

Ideally, business rules expressed at the business level can be operationalised by mapping them to executable statements at the software system level as shown in Table 1. This mapping is, however, not one-to-one, since programming languages and database management systems offer only limited support for it. While general purpose programming languages do not provide built-in support for any of the four types of business rules (with the exception of the object-orientated language Eiffel that supports integrity constraints in the form of 'invariants' for object classes), SQL has some built-in support for constraints, derivation rules (views), and limited forms of reaction rules (triggers). In general, however, current software technologies fail adequately to support business rules.

**Table 1** Mapping of business rules from the business level to the information system level referring to currently available technology

| *Business/CIM concept* | *SQL concepts* | *Other PIM concepts* |
|---|---|---|
| Constraint | `DOMAIN`, `CHECK` and `CONSTRAINT` clauses in table definitions; `CREATE ASSERTION` statements in database schema definitions | *Invariants* in OCL |
| Derivation Rule | `CREATE VIEW` statements | 'Clauses' in deductive databases and Prolog; *production rules* in CLIPS/Jess |
| Reaction Rule | `CREATE TRIGGER` statements | *Production rules* in CLIPS/Jess; ECA rules in 'active' databases |

## 3 Rules and rule languages in today's IT landscape

### *3.1 Rules in UML/OCL*

The Unified Modelling Language (UML) may be viewed as the paradigm-setting language for software and information systems modelling. In the UML, one may include integrity rules in a class diagram either graphically, or by attaching suitable statements in natural

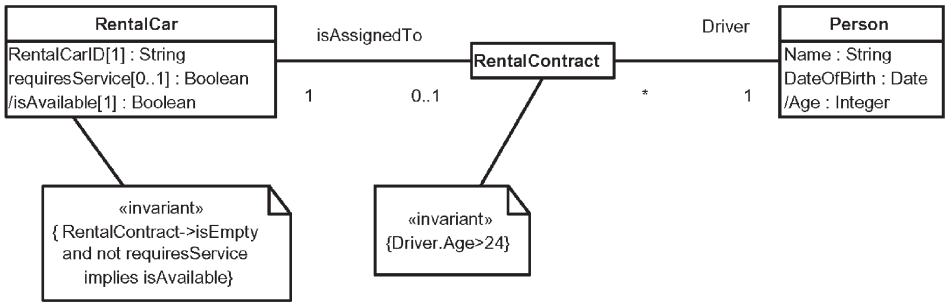
language or in the Object Constraint Language (OCL), to the modelling elements concerned. Integrity constraints can be formally expressed as *invariants* in OCL.

For example, the car rental company integrity rule

> *Drivers must be at least 25 years old,*

is expressed as the OCL invariant Driver. `Age>24` attached to the `RentalOrder` class in Figure 2.

**Figure 2** This UML class diagram shows three object classes, `RentalCar`, `RentalOrder` and `Person`, and two associations between them. The Boolean-valued attribute `isAvailable` of `RentalCar` is a derived attribute whose definition is expressed by the attached OCL invariant




OCL also allows the inclusion of *derived* attributes, classes or associations in a class diagram. These derived concepts are defined by means of derivation rules. Since OCL does not provide a genuine rule construct, derivation rules have to be expressed as *implicational* invariants.

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 order 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 orders and the stored Boolean-valued attribute `requiresService`, as shown in the UML class diagram in Figure 2, where the derivation rule is expressed as an implicational OCL invariant stating that for a specific rental car whenever there is no rental order associated with it `(RentalOrder->isEmpty)`, and it does not require service, then it is available for a new rental.

### *3.2 Rules in SQL*

SQL is the paradigm-setting language for databases. In SQL databases:

- integrity rules may occur in various places, most notably at the level of attribute definitions in the form of `CHECK`s, at the level of table definitions in the form of `CONSTRAINT`s, and at the database schema level in the form of `ASSERTION`s
- derivation rules may occur in the form of `VIEW`s that define a derived table by means of a query
- reaction rules may occur in the form of `TRIGGERS` that define a reaction in response to state change events of a certain type.

#### *3.2.1 Integrity rules in SQL*

SQL provides several constructs for expressing various kinds of integrity rules (constraints). The simplest one is the `CHECK` constraint which allows the specification of a wide range of integrity rules for tables, such as range of values and list of values. For instance, the OCL invariant in Figure 2, expressing the rule that drivers must be at least 25 years old, is expressed as a `CHECK` constraint within the definition of the `RentalOrder` table in the following way:

```
CREATE TABLE  RentalOrder(
OrderNo   INTEGER PRIMARY KEY,
Driver    CHAR(10) REFERENCES Person
          CHECK( 24 < (SELECT Age FROM Person WHERE PersID = Driver))
...       ...,
)
```

#### *3.2.2 Derivation rules in SQL*

Referring to Figure 2, the rule for the derived class of available cars is defined as the view

```
CREATE VIEW AvailableCar
  SELECT RentalCarID FROM RentalCar
  WHERE NOT requiresService
    AND isAssignedTo IS NULL
```

on the basis of the following implementation of the class `RentalCar`:

```
CREATE TABLE  RentalCar(
RentalCarID       CHAR(20) NOT NULL,
requiresService   BOOLEAN,
isAvailable       BOOLEAN,
isAssignedTo      INTEGER REFERENCES RentalOrder
)
```

### 3.2.3 Reaction rules in SQL

A limited form of reaction rules can be expressed in SQL in the form of triggers that define a reaction to a database change event. The triggered action can be either a database change action (expressed by means of `INSERT`, `UPDATE`, `DELETE`) or any procedure call that can be embedded in SQL, such as sending an e-mail message.

An example of a trigger would be the following

```
CREATE TRIGGER  alert_service_station
AFTER UPDATE ON RentalCar( requiresService)
FOR EACH ROW
WHEN requiresService
CALL send_email('ServiceStation','New service required')
```

This trigger creates an e-mail alert sent to the service station whenever the `requiresService` Attribute is set to *true*.

## 3.3 Rules in CLIPS/Jess and Prolog

CLIPS/Jess and Prolog may be viewed as the paradigm-setting languages for *production rules* and (computational logic) *derivation rules*. Both languages have been quite successful in the Artificial Intelligence research community and have been used for many AI software projects. However, both languages also have difficulties to reach out into, and integrate with, mainstream information technologies and live rather in a niche. Moreover, while Prolog has a strong theoretical foundation (in computational logic), CLIPS/Jess and the entire production rule paradigm lack any such foundation and do not have a well-defined formal semantics. This problem is partly due to the fact that in production rules, the semantic categories of events and conditions in the left-hand-side, and of actions and postconditions in the right-hand-side of a rule are mixed up.

While derivation rules have an if-*Condition*-then-*Conclusion* format, production rules have an if-*Condition*-then-*Action* format.

In Prolog, the rule for available cars is defined by means of the following two rules:

```
availableCar(X) :-
  rentalCar(X),
  not requiresService(X),
  not isAssignedToSomeRental(X).
isAssignedToSomeRental(X) :-
  isAssignedTo(X,Y).
```

The second of these rules is needed to define the auxiliary predicate `isAssignedToSomeRental` because Prolog does not provide an existential quantifier in rule conditions. Prolog rules and deductive database rules (including SQL views) have a purely declarative semantics in terms of their intended models (in the sense of formal logic model theory). For rules without negation, there is exactly one intended model: the unique minimal model. The intended models of a stable set of rules with negation[5] are its stable models (see Gelfond and Lifschitz, 1988).

Production rules do not refer explicitly to events, but events can be simulated in a production rule system by asserting corresponding objects into working memory. A derivation rule can be simulated 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.

The production rule system *Jess*, developed by Ernest Friedman-Hill at Sandia National Laboratories, is a Java successor of the classical LISP-based production rule system *CLIPS*. Jess supports the development of rule-based systems which can be tightly coupled to code written in the Java programming language. As in LISP, all code in Jess (control structures, assignments, procedure calls) takes the form of a function call. Conditions are formed with conjunction, disjunction and negation-as-failure. Actions consist of function calls, including the assertion of new facts and the retraction of existing facts.

In Jess, the rule for available cars is defined as

```
(defrule availableCar
  (and (RentalCar ?x)
       (not (requiresService ?x))
       (not (isAssignedToSomeRental ?x)))
  =>
  (assert (availableCar ?x))
```

Production rule systems such as Jess do not have a purely declarative semantics. In Jess, a rule will be activated only once for a given set of facts; once it has fired, that rule will not fire again for the same list of facts. Each rule has a property called *salience* that determines the order in which applicable rules are fired. Activated rules of the highest salience will fire first, followed by rules of lower salience. The default salience value is zero. Salience values can be integers, global variables, or function calls. The order in which multiple rules of the same salience are fired is determined by the active conflict resolution strategy. However the Jess user manual states that the use of salience is generally discouraged because it is considered bad style in rule-based programming to try to force rules to fire in a particular order.

### *3.4 Rule software components for rule-based applications*

There are a number of commercial rule software components and tools for developing rule-based applications, most of them for the Java platform. We mention only two of them: the production rule system *iLOG Rules/JRules* and the ECA rule system *Savvion BPM Server*.

#### *3.4.1 iLOG Rules/JRules*

In the iLOG Rules/JRules system,[6] rules have an if-*condition*-then-*action* format and are evaluated against a view of the application state created in working memory. All application objects that are relevant for rule evaluation have to be fetched from their home database and asserted into working memory before the evaluation can start.

In addition to the standard way of evaluating rule conditions at certain points in time, a rule may be specified to be evaluated during a certain period of time (either having to hold at all moments or at some moment within that period).

iLOG allows non-technical users to specify business rules in a kind of natural language, called the iLOG *Business Action Language*, using a syntax-driven point-and-click editor that provides condition and action terms based on an application-specific business object model. The syntax of this language is customisable by means of a rule 'token model'.

### *3.4.2 Savvion's process rules*

Savvion's *BPM Server*[7] is a workflow engine that includes an ECA rule engine component. Savvion considers business rules in the context of extended enterprise processes and calls them 'process rules'. With respect to Savvion's business process modelling and management architecture, rules are classified into three categories: 'process flow rules', 'process management rules' and 'process event-action inference rules'. While the first two of these categories refer to particular application functions (process flow definition and process management), and thus do not correspond to semantic categories, the latter category corresponds to ECA rules.

## *3.5 Rule modules in standard application software*

Rules may be used for special purposes in standard application software. An important case is represented by built-in rule modules that allow users to define rules for handling application events. This is well-known from e-mail applications such as the UNIX sendmail program or Microsoft Outlook.

In Microsoft Outlook, rules are used for automated message processing. The rule module is called 'Rule Wizard'. A Microsoft Outlook rule can be specified for incoming or for outgoing messages. It consists of a set of conditions referring to the message and its parameters, and of a set of actions. The specified conditions determine the messages the rule applies to. Negative conditions are called 'exceptions'. The specified actions may do something with a qualifying message, such as moving it to a specific folder or deleting or printing it, or they may do things like playing a specific sound, starting a specific application or sending a reply message.

So, there are two basic event types (InMsg and OutMsg) and many action types. A condition is a conjunction of atomic and negated ('except') atomic conditions, where an atomic condition is a substring relation or a string equality involving string constants and the parameters of a message (such as ?To, ?From, ?Cc, ?Body).

Thus, Outlook rules are ECA rules, having one of the following forms:

```
ON InMsg(?To, ?From, ?Cc, ?MsgBody)
IF some condition involving ?To, ?From, ?Cc, ?MsgBody holds
DO some action


ON OutMsg(?To, ?Cc, ?MsgBody)
IF some condition involving ?To, ?Cc, ?MsgBody holds
DO some action
```

To ease the use of rules, Microsoft Outlook provides a natural language template interface for specifying rules.

### *3.6 Rules in academic research prototypes*

There are a number of academic research prototypes implementing certain forms of rules, mainly in a database context. Examples of ECA rule ('active database') systems are Ariel (Hanson, 1996), Starburst (Widom, 1996) and HiPAC (McCarthy and Dayal, 1989). Examples of derivation rule ('deductive database') systems are ConceptBase[8] and FLORA-2 (http://flora.sourceforge.net/).

## 4 Rule languages in tomorrow's web-based IT landscape

There are a number of projects for developing new rule languages for the web. We briefly discuss some of them.

### *4.1 P3P APPEL*

The Platform for Privacy Preferences Project (P3P), developed by the World Wide Web Consortium, is an attempt to provide a mechanism for users to protect personal information from information-gathering programs running on web servers they interact with when visiting certain websites.

P3P APPEL[9] is a language that allows the expression of both the privacy policies used by a website and the privacy protection preferences of users. A P3P-enabled user agent (e.g. a web browser) can use the privacy preferences specified by the user and the privacy policies specified by the visited website in order to semi-automatically determine, if the user should read a page of that side or not.

### *4.2 RDF/N3*

Notation 3 (N3) is an alternative syntax for expressing RDF statements.[10] It is more human-readable than RDF's XML syntax and also provides special notation for logical concepts such as variables, quantifiers, and the implication connective. As in RDF, there is no way in N3 to express negative statements, as there is no negation.

N3 allows to express derivation rules whose antecedents and consequents are conjunctively interpreted sets of RDF/N3 statements. The following example shows a N3 rule stating that the brother of the father of a person is an uncle of that person:

```
{ ?p1 :hasFather ?p2.  ?p2 :hasBrother ?p3 } ⇒ { ?p1 :hasUncle
?p3 }.
```

The N3 project also experiments with a number of special predefined language constructs in order to make N3 sufficiently expressive for practical purposes. For instance, `log:notIncludes` is a built-in predicate that allows the expression of certain forms of default rules, and `log:definitiveDocument` allows the representation of closed predicates, which are discussed in Section 6.3.8.

A general problem with N3 rules, and any similar extension of RDF, where the derivation rule operator (called 'log:implies') is introduced as a RDF predicate, is that, as reified constructs, they do not have a direct model-theoretic semantics. The RDF model theory, which defines the semantics of RDF triples, does not provide any semantics for such reified logical constructs.

### 4.3 OWL

The web ontology language OWL[1] is an extension of RDF/RDFS, essentially by adding a number of elementary constructs for expressing basic statements such as class formation (`Class`), classification (`Individual`), attribution (`DatatypeProperty`), association (`ObjectProperty`), equality (`EquivalentClasses` and `SameIndividual`) and others. OWL allows the expression of subsumption rules such as

$$\text{isRentalCar}(X) \Rightarrow \text{isCar(X)}$$

in the form of `SubClassOf` statements, such as

SubClassOf(RentalCar, Car)

but it does not allow the expression of more complex derivation rules, e.g. for defining derived property predicates such as

$$\text{hasFather}(X, Y), \text{hasBrother}(Y, Z) \Rightarrow \text{hasUncle}(X, Z)$$

Positive logic programs without negation and without function symbols (also called 'Datalog'), corresponding to Definite Horn Logic, and OWL DL, corresponding to a two-valued Description Logic, are not reducible to each other (Grosof et al., 2003).

### 4.4 SWRL

Recently, the Joint US/EU ad hoc Agent Markup Language Committee has proposed the *Semantic Web Rule Language (SWRL)*, whose syntax is based on a combination of OWL DL and the Datalog sublanguage of RuleML.[11] The semantics of SWRL is a straightforward extension of the firstorder model-theoretic semantics of OWL DL.

Since, like RDF and OWL, SWRL is based on classical two-valued logic, it cannot deal with partial information and with closed predicates. Therefore, it cannot encode the rule for available rental cars shown in Figure 3. But it can encode property chaining rules like the rule for *hasUncle*:

```
<ruleml:Implies>
  <ruleml:body>
    <swrlx:individualPropertyAtom swrlx:property="hasFather">
      <ruleml:Var>x1</ruleml:Var>
      <ruleml:Var>x2</ruleml:Var>
    </swrlx:individualPropertyAtom>
    <swrlx:individualPropertyAtom swrlx:property="hasBrother">
      <ruleml:Var>x2</ruleml:Var>
      <ruleml:Var>x3</ruleml:Var>
    </swrlx:individualPropertyAtom>
  </ruleml:body>
  <ruleml:head>
    <swrlx:individualPropertyAtom swrlx:property="hasUncle">
      <ruleml:Var>x1</ruleml:Var>
      <ruleml:Var>x3</ruleml:Var>
    </swrlx:individualPropertyAtom>
  </ruleml:head>
</ruleml:Implies>
```

SWRL is restricted to OWL DL expressions, so it does not support higherorder statements as they are allowed in RDF and OWL Full. Since SWRL allows for disjunctive and existential statements in the consequent of a rule, there are SWRL rules that cannot be expressed in RuleML 0.89.

### *4.5 RuleML*

The RuleML standardisation initiative was founded in August 2000 by Harold Boley and Said Tabet with the goal of establishing an open, vendor neutral XML-based rule language standard. Further experts in rule theory and technology have subsequently joined the R*uleML steering group*, and a larger group of *RuleML participants* has been formed to get the expertise and involvement of a large number of academic researchers and industry experts from rule software vendors. The official website of the RuleML initiative is http://www.ruleml.org.

The RuleML effort aims to establish a general rule language framework that supports different kinds of rules and and various semantics. The current version of RuleML (in August 2005) has the version number 0.89 and does only cover certain forms of derivation rules (similar to extended logic programming rules with two kinds of negation). In Boley et al. (2001) the rationale behind RuleML 0.8 is discussed.

RuleML builds upon the paradigm of logic programming, while OWL/SWRL builds upon the paradigm of classical first-order logic. The difference between RuleML and OWL/SWRL reflects the difference between two academic research cultures. RuleML is rooted in the logic programming tradition, in which researchers have focused on computational interpretations of predicate logic, investigating a great number of syntactic and semantic extensions and variations. OWL/SWRL is rooted in the tradition of logic-based Artificial Intelligence, in which researchers have been remarkably faithful to classical (two-valued) predicate logic, considering it as the one and only true logic.

The antecedent ('body') of a RuleML 0.89 rule is not restricted to a conjunction of atoms. Rather, the antecedent can be a nested formula, formed with the help of the n-ary connective tags `<and>` and `<or>`, expressing conjunction and disjunction, and with the help of the unary connective tags `<naf>` and `<neg>`, expressing negation-as-failure and strong negation.

A RuleML 0.89 derivation rule is marked up as

```
<Implies> <head> ... </head> <body> ... </body> </Implies>
```

or, equivalently, as

```
<Implies> <body> ... </body> <head> ... </head> </Implies>
```

For instance, the rule for available cars discussed above is marked up in RuleML 0.89 as in Figure 3. Notice the use of two kinds of negation: strong negation (Neg) and negation-as-failure (Naf), which are necessary to deal with both open and closed predicates.

RuleML 0.89 also allows the expression of an atomic formula with the help of attribute (or role) names. So, instead of using the standard logical syntax with positional arguments like

**Figure 3** The rule for available cars marked up in RuleML 0.89

```
<Atom>
  <Rel>isAssignedTo</Rel>
  <Var>x</Var>
  <Var>y</Var>
</Atom>
<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>isAssToRentalOrder</Rel>
      <Var>Car</Var>
    </Atom>
    </Naf>
  </body>
</Implies>
```

one can also use a more readable attribute-value syntax like

```
<Atom>
  <oid><Var>x</Var></oid>
  <Rel>RentalCar</Rel>
  <slot><Ind>isAssignedTo</Ind><Var>y</Var></slot>
</Atom>
```

RuleML 0.89 does not yet contain a general syntax for integrity rules, nor for reaction rules and production rules. But the support of these other rule types is planned for the future.

### *4.6 Buchingae*

*Buchingae*[12] is a non-XML rule language used by the production rule engine *Bossam* with special support for processing RDF/S and OWL ontologies. It is inspired by RuleML and supports URIs, higher-order expressions and RDF/S and OWL primitives. As in RuleML, rules can have two kinds of negation. Buchingae comes with an XML exchange syntax, called 'LogicML'.

### *4.7 Mandarax*

Mandarax[13] is an open source rule platform consisting of a Java class library for defining and processing Prolog-like derivation rules within Java programs. Like Prolog systems, the Mandarax inference engine is based on top-down reasoning and term unification. Mandarax supports the export and import of RuleML 0.8 rules, but provides in addition a specific XML language for representing Mandarax knowledge bases.

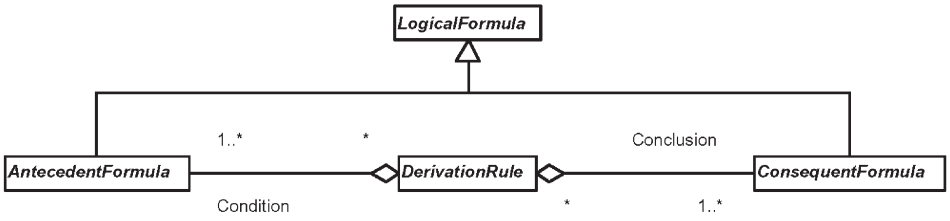
## 5 Abstract syntax and semantics

A *general rule markup language* – we abbreviate this term by GRML in the sequel – should be defined by an abstract syntax and a formal semantics. The abstract syntax specifies the major categories, such as Condition, Conclusion, Event, Action, and Postcondition, as well as their subcategories, such as Negation, Conjunction, Sequence and ParallelSplit, without specifying any concrete symbols for writing them. At the abstract level, also the ordering is left undefined so that there is no bias toward a prefix notation such as required in Jess or an infix notation such as allowed in Prolog.

### *5.1 Defining the abstract syntax of rules with the help of UML*

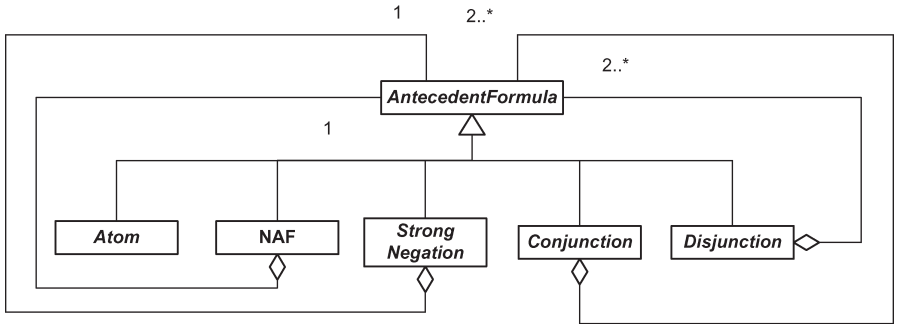
The abstract syntax of a language can be defined with the help of a MOF/UML model, as recommended by the Object Management Group, or it can be defined with the help of a suitably general grammar definition language such as the EBNF formalism used in the definition of the abstract syntax of OWL and SWRL. Figure 4 shows the syntax of abstract derivation rules in the form of a MOF/UML model. The abstract categories *AntecedentFormula* and *ConsequentFormula* used in this model have to be further specialised down to concrete language elements, such as *swrlx:individualPropertyAtom* or *ruleml:atom.*

**Figure 4** The abstract syntax of derivation rules. Notice that class names in italics indicate abstract classes



The abstract top-level type *AntecedentFormula* subsumes the abstract types of atoms, negations, conjunctions and disjunctions as shown in Figure 5.

**Figure 5** The abstract syntax of antecedent formulae




The abstract type *Atom* in a GRML should be specialised by all important types of atoms. In Figure 6, we have included in addition to the atom types of SWRL and the Prolog atom type of RuleML 0.89, two not yet existing types of atoms: RDF triples and OCL atoms. In all these expressions, logical variables are marked up with `<ruleml:Var>`, allowing variable bindings across different sublanguages.

**Figure 6** The syntax of atomic formulas of a GRML

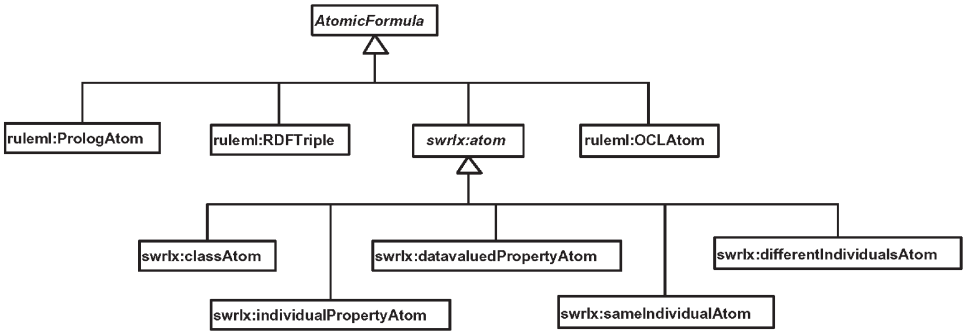
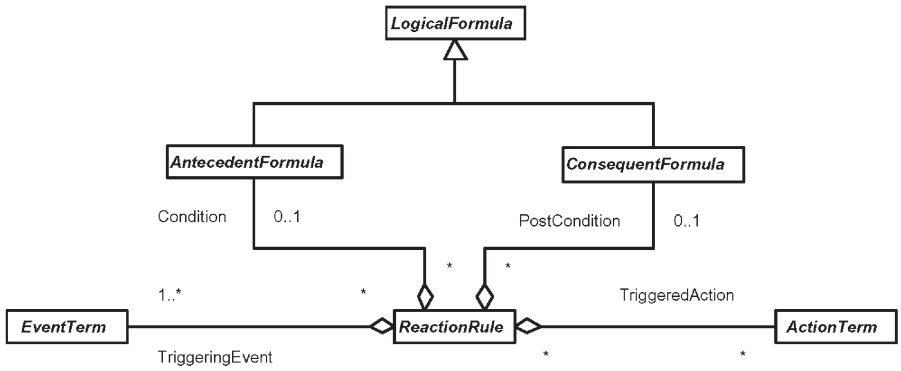



Figure 7 shows the syntax of abstract reaction rules. Also, the abstract categories *EventTerm* and *ActionTerm* have to be further specialised down to concrete language elements (possibly allowing for complex event and action terms).

Concerning the concrete XML syntax for logical expressions, a GRML should follow or integrate other standards such as MathML or the forthcoming *Common Logic* standard[14] as much as possible, in order to maximise exchangeability and tool reuse.

A GRML standard should contain mappings for the most important concrete rule syntaxes: OCL statements, SQL views, SQL triggers, Jess, and Prolog. Each of those mappings will specify how the abstract GRML categories are mapped to the symbols of the concrete notations. Any of the concrete notations can then be mapped into any of the others by reversing the mapping from concrete to abstract in one notation and then mapping from abstract to concrete in the other notation.

**Figure 7** The abstract syntax of reaction rules




## *5.2 Defining the formal semantics of rules*

The GRML standard should also come with a formal semantics

- for logical statements that may play the role of integrity constraints, preconditions, conclusions and postconditions
- for derivation rules
- for event and action terms
- for production rules and reaction rules.

Such a semantics would help to assess the degree of compatibility between different rule systems.

While a formal semantics for logical statements and derivation rules should be based on a Tarski-style model theory, a semantics for production rules and reaction rules, and event and action terms, needs to account for state changes and execution histories and should, therefore, be based on a transition system formalism.

Tarski-style model theory is not limited to classical first-order models, as employed in the semantics of OWL. It allows various extensions, such as relaxing the bivalence assumption (e.g. allowing for partial models) or allowing higher-order models. It is also compatible with the idea of nonmonotonic inference, simply by not considering all models of a rule as being intended, but only those models that satisfy a certain criterion. Thus, the stable model semantics for normal and (generalised) extended logic programs, as defined in Gelfond and Lifschitz (1988) and Herre et al. (1999), can be viewed as a Tarski-style model-theoretic semantics for nonmonotonic derivation rules.

### *5.2.1 Formal semantics of derivation rules*

In this subsection, we briefly sketch the formal semantics of derivation rules.

A Tarski-style model theory is a triple $\langle L, \mathbf{I}, |= \rangle$, such that

- $L$ is a set of formulae, called *language*
- $\mathbf{I}$ is a set of interpretations
- $|=$ is a relation between interpretations and formulae, called *model relation*.

For each Tarski-style model theory $\langle L,\mathbf{I}, \models\rangle$, we can define

- a notion of a derivation rule $F \rightarrow G$ where $F \in L$ is called 'condition' and $G \in L$ is called 'conclusion'
- $\mathrm{DR}_L = \{F \rightarrow G : F, G \in L\}$, the set of derivation rules of $L$
- a standard model operator

$$\mathrm{Mod}(X) = \{I \in \mathbf{I} : I \models Q \text{ for all } Q \in X\}$$

where $X \subseteq L \cup \mathrm{DR}_L$, as a set of formulae and/or derivation rules, is called a *knowledge base*.

Notice that in this way we can also define rules for logics which do not contain an implication connective. This shows that the concept of a rule is more fundamental than, and independent of, the concept of implication.

Typically, in knowledge representation theories not all models of a knowledge base are *intended* models. Except from the standard model operator *Mod* there are also non-standard model operators, which do not provide all models of a knowledge base, but only a special subset that is supposed to capture its intended models according to some semantics.

A particularly important type of such an 'intended model semantics' is obtained on the basis of some *information ordering* $\leq$, which allows comparison of the information content of two fact sets $X_1, X_2 \subseteq L$: whenever $X_1 \leq X_2$, we say that $X_2$ *is more informative* than $X_1$. We define a Tarski-style model theory extended by an information ordering as a quadruple $\langle L, \leq, \mathbf{I}, \models\rangle$, and call it an *information model theory*.

For any information model theory, we can define a number of natural nonstandard model operators, such as the *minimal* model operator

$$\mathrm{Mod}_m(X) = \{I \in \mathrm{Mod}(X) : I \leq I' \text{ for all } I' \in \mathrm{Mod}(X)\}$$

and various refinements of it, like the *stable generated* models defined in Herre and Wagner (1997) and Herre et al. (1999).

For any given model operator $\mathbf{M} : L \cup \mathrm{DR}_L \rightarrow \mathbf{I}$, and knowledge base $X \subseteq L \cup \mathrm{DR}_L$, we can define an entailment relation

$$X \models_{\mathbf{M}} Q \text{ iff } \mathbf{M}(X) \subseteq \mathbf{M}(\{Q\})$$

For non-standard model operators, like minimal and stable models, this entailment relation is typically *nonmonotonic* in the sense that for an extension $X' \geq X$ it may be the case that $X$ entails $Q$, but $X'$ does not entail $Q$.

Another approach to the semantics of derivation rules, which can be characterised as *proof-theoretic*, is lattice-theoretic *fixpoint semantics*, as proposed in the logic programming literature (see e.g. van Gelder, 1989).

### *5.2.2 Formal semantics of reaction rules*

In this subsection, we briefly sketch the formal semantics of reaction rules on the basis of a high-level transition system formalism.

When reaction rules are fired in a system, they typically change its state, which may be given by a fact set $X \subseteq L$ in the context of an information model theory $\langle L, \leq, \mathbf{I}, \models\rangle$.

We assume that such a system possesses a state change function

$$\texttt{Upd} : 2^L \times L \rightarrow 2^L$$

such that $\texttt{Upd}(X, F)$ provides the updated knowledge state obtained from $X$ by assimilating the input formula $F$, and it holds that

$$\texttt{Upd}(X, F) \models_* F$$

where $\models_*$ denotes the entailment relationship based on the underlying non-standard model operator $\text{Mod}_*$.

A system that is processing reaction rules must also possess a state component that represents something like an *event perception state* in addition to its information state. We may assume that this state component has the form of a first-in-first-out queue (called *event queue*) buffering atomic event terms from an event language $L_E$.

For any given information model theory $\langle L, \leq, \mathbf{I}, |=\rangle$, and event and action languages $L_E$ and $L_A$, we can define a notion of reaction rule as a quadruple

$$E, C \Rightarrow A, P,$$

where $E \in L_E$ is called the *triggering event* term, $A \in L_A$ is called the *triggered action* term, and $C, P \in L$ are called (pre-)condition and postcondition.

A reaction rule processing system given by a knowledge base $X \subseteq L \cup \text{DR}_L$, an event queue $EQ \subseteq L_E$, and set of reaction rules $RR \subseteq 2^{L_E \times L \times L_A \times L}$, then, is a high-level state transition system with two basic transition types:

- *Event perception:* when a new event is perceived by the system, it is appended to its event queue.
- *Reaction*: whenever there is an event term in the event queue, fetch it and match it against the triggering event terms of all reaction rules; if they match (can be unified), the rule is triggered and its condition C is evaluated against the current information state $X$: if $X \models_* C$, then the free variables of the postcondition (or effect) $P$ are instantiated accordingly, and the resulting sentence $P'$ is assimilated into $X$ by forming the update
$$X' = \texttt{Upd}(X, P').$$

### *5.3 Derivation rules distinguished from implications*

Derivation rules are often identified with logical implications. But, in general, these are two different concepts. While an implication is an expression of a logical formula language, typically possessing a truth-value, a derivation rule is rather a meta-logical expression that does not possess a truth-value. 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. This is easy to see with help of an example. Consider the rule

$$\neg q \rightarrow p$$

whose model set, according to the minimal/stable model semantics, is $\{\{p\}\}$, that is, it entails $p$. On the other hand, the model set of the corresponding implication

$$\neg q \supset p$$

which is equivalent to the disjunction $p \vee q$, is $\{\{p\}, \{q\}, \{p, q\}\}$; consequently, it does not entail $p$.

## 6 From use cases and design goals to requirements

### 6.1 Use cases

A GRML can be used to improve existing rule-based applications and may enable new uses of rules. In this section six representative use cases of web rules are described. In some of these cases (6.1.2, 6.1.4, 6.1.5), purpose-specific rule markup languages have been defined already. But there has been no attempt yet to harmonise and unify these different web rule languages.

#### 6.1.1 Business rule documentation

A GRML could be used to create an XML-based documentation of the business rules of an organisation. Such a documentation would typically be only weakly structured: only the top-level GRML categories (Condition, Conclusion, Event, Action, etc.) are used for structuring a rule, but the content of each of these categories would be in natural language without any further structural breakdown.

This use of a GRML would also apply to business process modelling tools that are primarily intended for business process documentation, such as ARIS.[15]

#### 6.1.2 Enriching web ontologies

For capturing more of the semantics of an application domain, an existing web ontology, that may have been defined using RDFS or the *Ontology Web Language (OWL)*, can be enriched by adding suitable derivation rules and integrity constraints using a GRML. For instance, neither RDFS nor OWL can express 'default properties' and 'closed world statements', while this is straightforward, using the rules of a suitable GRML (see Section 6.3.8).

#### 6.1.3 Web forms

Integrity constraints expressed in a GRML may be used to include validation rules for specific fields in a web form. Likewise, derivation rules expressed in a GRML may be used to define derived fields in web forms.

#### 6.1.4 Disclosure permission and prohibition rules

As in the Platform for Privacy Preferences Project (P3P) of the W3C,[9] a GRML may allow to define disclosure permission and prohibition rules for users to protect personal information from information-gathering programs running on websites they visit.

#### 6.1.5 Access control rules

As in the *Extended Access Control Markup Language (XACML)* of OASIS,[16] a GRML may allow the specification of data access control policies, as required, e.g. in healthcare information systems for protecting sensitive information against unauthorised access.

### *6.1.6 E-mail rule interchange*

A GRML can be used to represent e-mail processing rules in export/import files for interchanging these rule files between different e-mail programs, such as Microsoft Outlook and UNIX sendmail.[17] Another use of the ability of an e-mail program to export its rules in a GRML format would be to import these rules in a GRML-enabled business process documentation tool, see Section 6.1.1.

### *6.1.7 Knowledge interchange*

In many cases, knowledge-based systems are implemented with the help of production rules or derivation rules. For reusing the rules of such a system easily, e.g. when migrating to another system, a GRML could play the role of a vendorneutral, platform-independent interchange language to which these rules could be mapped.

### *6.1.8 Declarative specification of artificial agents*

A more futuristic use of a GRML would be in the XML-based specification of an artificial agent by means of

- an RDFS-based taxonomy for defining the schema of its mental state
- a set of RDF facts for specifying its factual (extensional) knowledge
- a set of GRML integrity constraints for excluding non-admissible mental states
- a set of GRML derivation rules for specifying its terminological and heuristic (intensional) knowledge
- a set of GRML reaction rules for specifying its behaviour in response to communication and environmental events.

It will thus be possible to completely specify a software agent using RDF/RDFS and a GRML. Executing such an agent specification requires a combination of a knowledge subsystem (including an inference and an update operation), a perception (or incoming message handling) subsystem and an action (or outgoing message handling) subsystem.

## *6.2 Design goals*

Design goals describe general motivations for the language that do not necessarily result from any single use case. In this section, we describe some important design goals for a GRML.

### *6.2.1 Balance expressivity and practical relevance*

A GRML should be able to express a wide variety of rule knowledge, but should also allow tools to process it efficiently. In terms of practical relevance, it is much more important for a GRML to support rule languages that are widely used in practical applications (such as OCL, SQL, Jess and Prolog) than to support languages that 'live' only in single academic projects.

### *6.2.2 Integrate all relevant related standards*

A GRML should be able to express rules from all relevant rule standards and to interface, or be compatible, with all relevant web standards.

### 6.2.3 Support very large sets of facts

When evaluating rules, it should be possible to process very large sets of facts, as they are common, for instance, in enterprise information systems.

### 6.2.4 Support distributed heterogeneous information

Information is often distributed and heterogeneous. It should be possible therefore to refer to various forms of distributed information in the antecedent of a rule.

### 6.2.5 Support both complete and incomplete information

In business and administrative domains, most of the information to be processed is assumed to be complete (this tacit assumption is called *Closed-World Assumption* in AI). But in other domains, such as in medicine and criminology, most information is incomplete. A GRML should allow the expression of rules for both types of information.

### 6.2.6 Support various kinds of facts and rules

Depending on the domain of application, rules have to able to operate with various kinds of information. Facts may be:

- temporally qualified (by valid time and belief time)
- uncertain
- qualified by lineage and reliability
- qualified in some other way.

Also, rules may be qualified in the same way (by valid time, belief time, uncertainty, lineage and reliability). In addition, a rule may be qualified by a priority value that determines how it is treated in a conflict with another, competing rule.

## 6.3 Requirements

The use cases and design goals motivate a number of requirements for a GRML.
Each requirement is motivated by one or more use cases or design goals from the previous sections. Notice that the following collection of requirements is necessarily incomplete, since it is the result of an analysis carried out by only one researcher instead of a more comprehensive community process.

### 6.3.1 Rules and rule sets as distinct objects

In a GRML, rules and rule sets should have their own unique identifiers, which may be URIs.

### 6.3.2 Include integrity constraints, derivation rules, production rules and reaction rules

Provide a coherent markup for the four basic rule types Integrity Constraints, Derivation Rules, Production Rules and Reaction Rules.

*Motivation*: these are the rule types needed for dealing with the use cases outlined in Sections 6.1.1 to 6.1.8.

### *6.3.3 Provide mappings and embeddings for OCL invariants, SQL assertions, SQL views, SQL triggers, Jess rules and Prolog rules*

A GRML should be able to express rules from all these rule standards by either mapping them to its abstract syntax or by embedding them in the form of inline expressions.

*Motivation*: use cases outlined in Sections 6.1.3 and 6.1.7; design goal in Section 6.2.2.

### *6.3.4 Support web ontologies*

A GRML rule set should be able to refer to existing web ontologies by means of URLs in order to import their definitions. This requires that the content languages for the GRML top-level categories

- allow the use of terms from ontologies specified in the W3C *web ontology language (OWL)*
- that they support *W3C namespaces*.

*Motivation*: use case outlined in Section 6.1.2; design goal in Section 6.2.2.

### *6.3.5 Allow facts to be retrieved from secondary storage data sources*

Large sets of facts to be considered in the evaluation of a rule condition cannot be stored in main memory. Typical secondary storage data sources include SQL databases and XML files. This implies that a GRML should provide a construct to specify predicates whose extension is retrieved from a database table or XML file at runtime.

*Motivation*: design goal in Section 6.2.3.

### *6.3.6 Allow facts to be retrieved from multiple, possibly remote sources*

There may be several, possibly remote, information sources providing facts to be considered in the evaluation of a rule condition. These sources include SQL databases and XML files. A GRML should provide constructs to specify predicates whose extension is retrieved from remote database tables or XML files at runtime.

*Motivation*: design goal in Section 6.2.4.

### *6.3.7 Allow events/messages to be perceived/received from multiple external sources*

There may be several external sources providing events/messages to be considered in the evaluation of a reaction rule event term. These sources include SMTP mailboxes, SOAP message queues, and transactional message queues like Java Message Service and IBM's MQSeries. A GRML should provide constructs to specify event/message types whose instances are perceived/received from specific sources.

*Motivation*: design goal in Section 6.2.4.

### *6.3.8 Support closed predicates, negation-as-failure and strong negation*

In natural language (and in the underlying 'real world' of cognition) there are two kinds of negation: a weak negation expressing non-truth, and a strong negation expressing explicit falsity. Likewise, a practical knowledge representation system needs these two negations for handling both complete and incomplete information. Weak negation is turned into negation-as-failure by adopting the preferential semantics of minimal, respectively stable models.

Unlike negation in classical logic, real-world negation is not a simple two-valued truth function. The simplest generalisation of classical logic that is able to account for two kinds of negation is *partial logic* giving up the classical bivalence principle and subsuming a number of three-valued and four-valued logics (Herre et al., 1999). For instance, SQL, which may be considered a knowledge representation and query language for simple knowledge bases, adopts a partial (three-valued) logic with the truth values {*false*, *unknown*, *true*}.

A GRML should allow to distinguish between closed predicates (that are completely represented in a database or knowledge base) and *open* or *partial* predicates. Such a distinction allows to make predicate-specific *Closed-World Assumptions*. The classification if a predicate is closed or not is up to the owner of the database: the owner must know for which predicates there is complete information in the database and for which there is not. In the case of a closed predicate, negation-as-failure expresses falsity, and both negation-as-failure and strong negation collapse into classical negation. In the case of an open predicate, negation-as-failure only reflects non-provability, but does not allow one infer the classical or strong negation.

In order to accommodate negation in natural language business rules and the two negations needed to handle complete and incomplete/partial information, a GRML should provide two kinds of negation:

- negation-as-failure (alias *weak negation* under the preferential semantics of stable models)
- strong negation (alias Kleene negation in three-valued logic).

By combining negation-as-failure and strong negation, one can express *default rules* in a natural way. An example of a default rule is the rule *Normally, a car requires service after every 10,000 miles.* This is not a strict rule, since it allows for an open number of exceptions (for instance, if the car is a Mercedes, it requires service only every 20,000 miles).

These issues are discussed at greater length in Wagner (2003).

*Motivation*: design goal in Section 6.2.5.

### *6.3.9 Allow facts and rules to be qualified in an extensible way*

Both for facts and for rules there should be a qualification construct that allows the expression of facts and rules which are qualified in some (possibly user-defined) way. Such a construct needs two meta-attributes: *qualification type* and *qualification value*. A GRML should support the predefined qualification types *valid time*, *belief time*, *degree of uncertainty*, and *lineage/provenance* both for facts and rules, and in addition priority for rules.

*Motivation*: design goal in Section 6.2.6.

## 7 Conclusion

The design of a general rule markup language is a difficult conceptualisation and integration problem, involving many advanced concepts from the knowledge representation and semantic web research areas. In order to be successful it requires a strong focus on practically relevant languages and technologies.

RuleML 0.89 is a small first step in the direction of such a language standard. It does not yet satisfy many of the requirements for a general rule markup language we have identified in this paper. In particular, it does not yet have a layered architecture that allows to combine atomic formulas from different languages in one rule expression.

## Acknowledgements

We are grateful to Harold Boley and Michael Schroeder for providing valuable comments on preliminary versions of this article. A preliminary version of this paper was presented as an invited paper by Gerd Wagner at the workshop XML Technologies for the Semantic Web (XSW2002) held at the Humboldt University of Berlin, Germany, in June 2002. This research has been partially funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (www.rewerse.net).

## References

Boley, H., Tabet, S. and Wagner, G. (2001) 'Design rationale of RuleML: a markup language for semantic web rules,' in *Proc. Semantic Web Working Symposium (SWWS '01)*, Stanford University, July/August 2001.

Gelfond, M. and Lifschitz, V. (1998) 'The stable model semantics for logic programming', in R.A. Kowalski and K A. Bowen (Eds) *Proc. of ICLP*, MIT Press, pp.1070–1080.

Grosof, B., Horrocks, I., Volz, R. and Decker, S.(2003) 'Description logic programs: Combining logic programs with description logic,' in *Proceedings of WWW* 2003, Budapest, Hungary, May 2003, pp.117–132.

Hanson, E. (1996) 'The design and implementation of the Ariel active database rule system,' *IEEE Transactions on Knowledge and Data Engineering*, Vol. 8, No. 1, pp.157–172.

Herre, H. and Wagner, G. (1997) 'Stable models are generated by a stable chain', *J. of Logic Programming*, Vol. 30, No. 2, pp.165–177.

Herre, H., Jaspars, J. and Wagner, G. (1999) 'Partial logics with two kinds of negation as a foundation of knowledgebased reasoning', in D.M. Gabbay and H. Wansing (Eds) *What Is Negation?*, Oxford University Press.

McCarthy, D. and Dayal, U. (1989) 'The architecture of an active database management system', in *Proceedings of ACM SIGMOD*, May, pp.215–223.

Taveter, K. and Wagner, G. (2001) 'Agent-oriented enterprise modelling based on business rules', in H.S. Kunii, S. Jajodia and A. Solvberg (Eds) *Proc. of 20th Int. Conf. on Conceptual modelling (ER2001)*, Yokohama, Japan, November 2001, Springer-Verlag, LNCS 2224, pp.527–540.

van Gelder, A. (1988) 'Negation as failure using tight derivations for general logic programs', in *Foundations of Deductive Databases and Logic Programming*, San Mateo (CA): Morgan Kaufmann, pp.149–176.

van Gelder, A. (1989) 'The alternating fixpoint of logic programs with negation', in *Proceedings of the ACM Symposium on Principles of Database Systems (PODS-89)*, pp.1–10.

Wagner, G. (2003) 'Web rules need two kinds of negation', in F. Bry, N. Henze, and J. Maluszynski (Eds) *Proceedings of Principles and Practice of Semantic Web Reasoning PPSWR 2003*. Springer-Verlag LNCS 2901, December 2003.

Widom, J. (1996)'The Starburst active database rule system', *IEEE Transactions on Knowledge and Data Engineering*, Vol. 8, No. 4, pp.583–595.

## Notes

[1] W3c web ontology language owl, http://www.w3.org/TR/owl-ref.

[2] This three-level distinction follows the Model-Driven Architecture (MDA) of the Object Management Group (see http://www.omg.org/cgi-bin/doc?mda-guide).

[3] What is a rule. Discussion thread in the archive of the WWW RDF rules e-mail list, http://lists.w3.org/Archives/Public/wwwrdf-rules/2001Sep/0079.html.

[4] RuleML initiative, http://www.ruleml.org.

[5] A rule set may be called stable, if it does not contain any odd loop (van Gelder, 1988).

[6] ilog rules/jrules, vendor website, http://www.ilog.com/.

[7] Savvion bpm server, vendor web page, December 2003, http://www.savvion.com/products/.

[8] ConceptBase, project website, http://www-i5.informatik.rwthaachen. de/CBdoc/cbflyer.html.

[9] W3c a p3p preference exchange language 1.0 (appel1.0). http://www.w3.org/TR/P3P-preferences/.

[10] Notation 3, http://www.w3.org/2000/10/swap/doc/.

[11] Swrl: a semantic web rule language combining owl and ruleml, http://www.daml.org/2003/11/swrl.

[12] Minsu Jang, Buchingae – a rule language for the web, web document, http://machine-knows.etri.re.kr/bossam/index.html.

[13] Mandarax, project website, http://www.mandarax.org/.

[14] Common logic, standardisation project, http://cl.tamu.edu.

[15] ARIS, vendor website, http://www.idsscheer. com/countries/corporate/english/.

[16] Oasis extensible access control markup language, http://www.oasisopen. org/committees/tc home.php?wg abbrev=xacml.

[17] An IETF standard for rules that process incoming e-mail messages, called Sieve, is under development. See http://www.cyrusoft.com/sieve.