How to Design a General Rule Markup Language?

Gerd Wagner
Eindhoven University of Technology, Faculty of Technology Management
`G.Wagner@tm.tue.nl`
`http://tmitwww.tm.tue.nl/staff/gwagner`

Abstract: A General Rule Markup Language has several purposes. It may serve as a *lingua franca* for different rule systems and rule components in application software to exchange rules between each other. It may be used to express derivation rules for enriching XML-based taxonomies (also called 'web ontologies') by adding definitions of derived concepts. It may be used to publish the reactive behavior of a system in the form of reaction rules. And it may be used to provide a complete XML-based specification of a software agent. Further uses may arise in novel web applications. In this paper, I consider the problem of how to design a General Rule Markup Language that can be used for these, and for future emerging purposes.

1 Introduction

Rule markup languages, that allow to express 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 for facilitating business-to-customer (B2C) and business-to-business (B2B) interactions over the Web.

This paper discusses the difficult question of how to design a general rule markup language, based on the initial experiences made with RuleML 0.8, a preliminary standards proposal by the RuleML initiative.

2 Rules in Today's IT Landscape

2.1 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 concept 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 operationalized by transforming them into executable rule expressions, preferably in a declarative rule language. In other cases they cannot or don't have to be operationalized and automated, but are only expressed in natural language for the purpose of communication and documentation.

It follows from these remarks that a GRML should allow to express rules both in natural languages and in declarative executable languages.

Three basic types of business rules have been identified in the literature (see [TW01]): *integrity constraints* (also called 'constraint rules' or 'integrity rules'), *derivation rules*, and *reaction rules* (also called 'stimulus response rules', 'action rules', 'event-condition-action (ECA) rules', or 'automation rules'). A fourth type, *deontic assignments*, has only been marginally discussed (in a proposal of considering 'authorizations' as business rules).

An ***integrity constraint*** is a an assertion that must be satisfied in all evolving states and state transition histories of an enterprise viewed as a discrete dynamic system. There are state constraints and process (or behavior) constraints. *State constraints* must hold at any point in time. An example of a state constraint applied by a car rental company is: *A customer 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 to capture terminological and heuristic domain knowledge about concepts 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 are concerned with the invocation of actions in response to events. They state the conditions under which actions must be taken; this includes triggering event conditions, pre-conditions, and post-conditions (effects). They define the behaviour of a system (or agent) in response to perceived environment events and to communication events (created by communication acts of other systems/agents). An example of a reaction rule from the domain of car rental is: *When a customer requests to make a car reservation, the car rental branch checks with the headquarter to make sure that the customer is not blacklisted.* In general, reaction rules consist of an event condition, a state condition (or precondition), an action term, and a state effect (or post-condition) formula. Thus, they can also be called Event-Condition-Action-Effect (ECAE) rules, subsuming Event-Condition-Action (ECA) and Condition-Action (or *production*) rules as special cases.

Business Rule Concept	*Implementation in SQL*	*Other Implementations*
Constraint	`DOMAIN`, `CHECK` and `CONSTRAINT` clauses in table definitions; `CREATE ASSERTION` statements in database schema definitions	if-then statements in programming languages
Derivation Rule	`CREATE VIEW` statements	'clauses' in deductive databases and Prolog
Reaction Rule	`CREATE TRIGGER` statements	production rules in CLIPS/Jess; if-then statements in programming languages

Table 1: Mapping of business rules from the business level to the information system level using currently available technology.

Deontic assignments of powers, rights and duties to (types of) internal agents define the deontic structure of an organization, guiding and constraining the actions of internal agents. An example of a deontic assignment rule is: *Only the branch manager has the right to grant special discounts to customers*.

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-oriented 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). Current software technologies fail to adequately support business rules.

2.2 Rules in UML/OCL, SQL, CLIPS/Jess and Prolog

UML/OCL, SQL, CLIPS/Jess and Prolog may be viewed as the paradigm-setting languages for *modeling*, *databases*, *production rules* and (logical) *derivation rules*. In each of them rules play an important role.

2.2.1 Rules in UML/OCL

The Unified Modeling Language (UML) may be viewed as the paradigm-setting language for software and information systems modeling. In the UML, one may include *integrity constraints* and *derived* attributes, classes or associations in a class diagram. These derived concepts are defined by means of derivation rules. Integrity constraints can be formally expressed as *invariants* in the Object Constraint Language (OCL) of the UML. 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, does not require service and is not scheduled for a maintenance check.*

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 attributes `requiresService` and `isSchedForMaint`, as shown in the UML class diagram in Figure 1, where the derivation rule is expressed as an implicational OCL invariant that states that for a specific rental car whenever there is no rental order associated with it, and it does not require service and is not scheduled for maintenance, then it has to be available for a new rental.

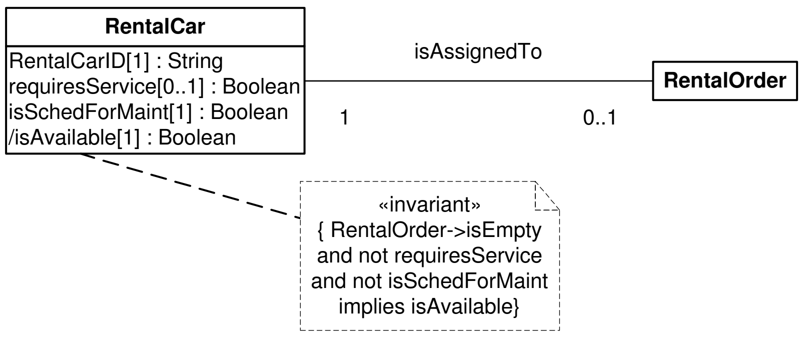

Figure 1: This UML class diagram shows two classes, `RentalCar` and `RentalOrder`, and the functional association `isAssignedTo` between them. The Boolean-valued attribute `isAvailable` is a derived attribute whose definition is expressed by the attached OCL constraint.

2.2.2 Rules in SQL

In SQL databases, rules occur in the form of *views* that define a derived table by means of a query. For instance, referring to Figure 1, the rule for the derived class of available cars is defined as the view

```
CREATE VIEW AvailableCar( CarID)
  SELECT CarID FROM RentalCar
  WHERE NOT requiresService
    AND NOT isSchedForMaintenance
    AND isAssignedTo IS NULL
```

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

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

2.2.3 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 computer science 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 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 effects 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 and may be considered a special case of the general concept of reaction rules (having an if-*Event, Condition*-then-*Action, Effect* format). To determine which rules are applicable in a given system state, conditions are evaluated against a fact base that is typically maintained in main memory.

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

```
availableCar(X) :-
  rentalCar(X),
  not requiresService(X),
  not isSchedForMaint(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 classical logic model theory). For rules without negation, there is exactly one intended model: the unique *minimal* model. The intended models of a set of rules with negation are its *stable* models.

Production rules do not explicitly refer to events, but events can be simulated 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 (isSchedForMaint ?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.

2.3 Rule Software Components for Rule-Based Application Development

There are a number of commercially available rule software components and tools for developing rule-based applications, mostly in the Java programming language. I mention only two of them: the production rule system *iLOG Rules/JRules* and the ECA rule system *Savvion BizPulse*.

2.3.1 iLOG Rules/JRules

In the iLOG Rules/JRules system, rules have a 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 customizable by means of a rule 'token model'.

2.3.2 Savvion BizPulse

Savvion considers business rules in the context of extended enterprise processes and calls them 'process rules'. With respect to Savvion's business process modeling 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 the Event-Condition-Action (ECA) form of reaction rules. Savvion's *BizPulse Server* is an ECA rule engine that collaborates closely with the same vendor's workflow engine *BizLogic Server*, thereby allowing to process workflow events as well as other events.

2.4 Rule Modules in Standard Application Software

Rules may be used for special purposes in standard application software. The most important case seems to be a built-in rule module that allows users to define reaction rules for handling application events. This is well-known from email 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 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 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 Abstract versus Concrete Syntax

A General Rule Markup Language (GRML) should be defined by an abstract syntax, which specifies the major categories, such as Condition, Conclusion, Event and Action, and their subcategories, such as Negation, Conjunction, Sequence and Fork, without specifying any concrete symbols for writing them. At the abstract level, even the ordering is left undefined so that there is no bias toward a prefix notation such as Jess or an infix notation such as Prolog.

See Figure 2 for an overview of the abstract syntax of a GRML.

There should be an XML-based syntax that corresponds directly to the abstract syntax. For example, the abstract category Fork could be expressed as the XML element `<Fork>` ... `</Fork>`.

A GRML standard should contain mappings for the most important concrete rule syntaxes: OCL implications, SQL views, 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.

The GRML standard should also contain

- a ***static-declarative semantics*** for abstract statements that may play the role of integrity constraints, conditions, conclusions and effect statements, and
- a ***dynamic-declarative semantics*** for events, actions and reaction rules in the abstract syntax.

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

Concerning the abstract syntax for logical statements, a GRML should follow the forthcoming *Common Logic* standard (see [CL]) as much as possible.

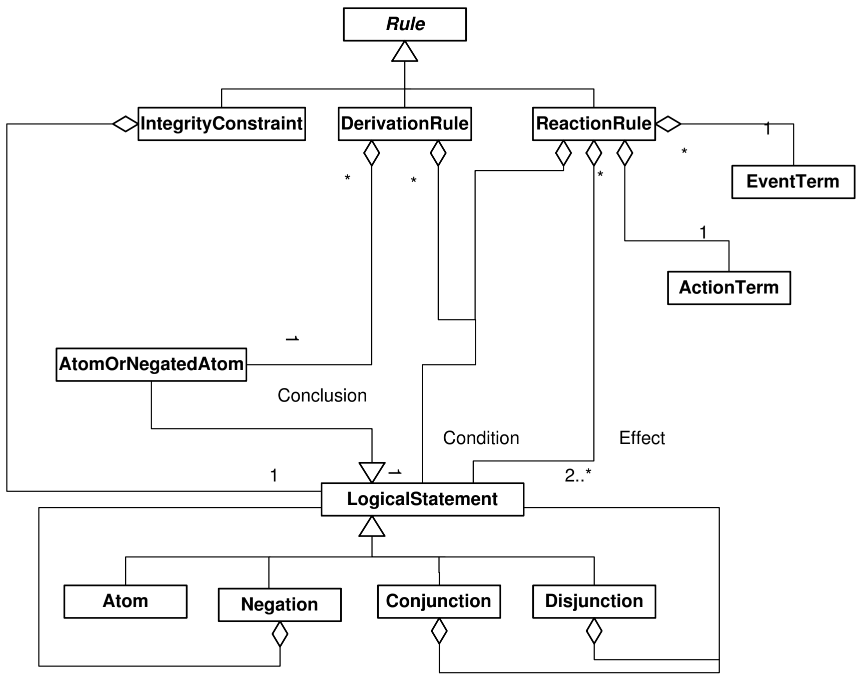

Figure 2: The major categories of the abstract syntax of a GRML.

4 From Use Cases and Design Goals to Requirements

4.1 Use Cases

A GRML can be used to improve existing rule-based applications and may enable new uses of rules. In this section I describe six representative use cases of a GRML.

4.1.1 Business Rule Documentation

A GRML could be used to create an XML-based documentation of the business rules of an organization. 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 modeling tools that are primarily intended for business process documentation, such as ARIS.

4.1.2 Enriching Web Ontologies

For capturing more of the semantics of an application domain, an existing Web ontology, that may have been defined using RDFS, DAML+OIL or the forthcoming W3C *Ontology Web Language (OWL)*, can be enriched by adding suitable derivation rules and integrity constraints using a GRML. For instance, neither RDFS nor DAML+OIL can express 'default properties' and 'closed world statements' (and presumably neither will OWL, as discussed in [Web], while this is straightforward using rules of a suitable GRML (see 4.3.8).

4.1.3 Email Rule Interchange

A GRML can be used to represent email processing rules in export/import files and to interchange these rule files between different email programs, such as Microsoft Outlook and UNIX sendmail. Another use of the ability of an email 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 4.1.1.

4.1.4 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.

4.1.5 Knowledge Interchange

Since rules are the central element in knowledge-based systems implemented either with production rules (such as CLIPS/Jess) or with derivation rules (such as Prolog and deductive databases, respectively SQL), a GRML would be essential to allow knowledge interchange between different knowledge-based systems implemented with different rule technologies.

4.1.6 Declarative Specification of Artificial Agents

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

- an RDFS-based taxononmy 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, and

- a set of GRML reaction rules for specifying its behavior in response to communication and environment events.

Thus, it will 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.

4.2 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.

4.2.1 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 views, Jess and Prolog) than to support languages that 'live' only in single academic projects.

4.2.2 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.

4.2.3 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.

4.2.4 Support distributed information

Information is often distributed, but available via network links. In the evaluation of rules, it should be possible to include various forms of distributed information.

4.2.5 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).

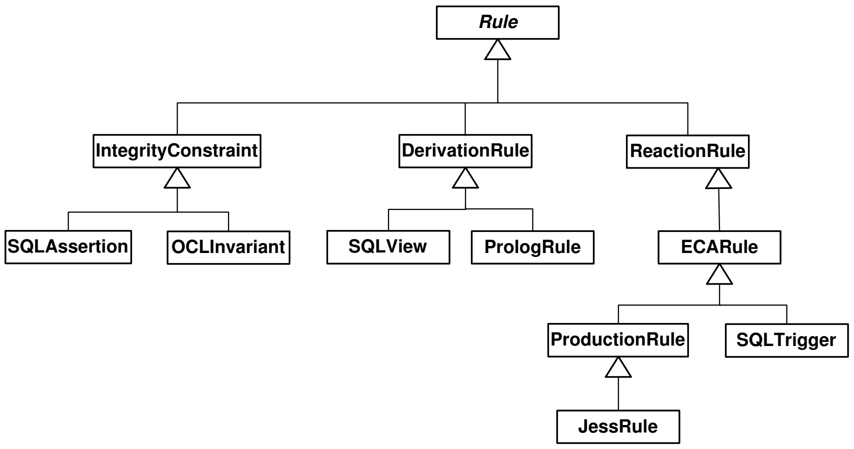

Figure 3: The rule type hierarchy with the most important rule types.

But in other domains, such as in medicine and criminology, most information is incomplete. A GRML should allow to express rules for both types of information.

4.2.6 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.

4.3 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 list is just a preliminary, necessarily incomplete first attempt to form a list of requirements.

4.3.1 Rules and rule sets as distinct objects

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

4.3.2 Include integrity constraints, derivation rules and reaction rules

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

Motivation: these are the three rule types needed for dealing with the use cases 4.1.1 to 4.1.6.

4.3.3 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 4.1.4 and 4.1.5, design goal 4.2.2.

4.3.4 Support references to and the interoperation with 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 forthcoming W3C *Ontology Web Language (OWL)*, and
- that they support W3C *namespaces*.

Motivation: use case 4.1.2, design goal 4.2.2.

4.3.5 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 4.2.3.

4.3.6 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 4.2.4.

4.3.7 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 4.2.4.

4.3.8 Support complete 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 generalization 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 3-valued and 4-valued logics (see [HJW99]). For instance, SQL, which may be considered a knowledge representation and query language for simple knowledge bases, adopts a partial (3-valued) logic with the truth values {*false, unknown, true*}.

A GRML should allow to distinguish between predicates that are completely represented in a database (or knowledge base) and those that are not (such a completeness assumption corresponds to a predicate-specific *Closed-World Assumption*). The classification if a predicate is completely represented or not is up to the owner of the database: the owner must know for which predicates there is complete information and for which there is not. In the case of a completely represented predicate, negation-as-failure reflects falsity, and negation-as-failure and strong negation collapse into classical negation. In the case of an incompletely represented predicate, negation-as-failure only reflects non-provability, but does not allow to infer the classical 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 three negations:

- 'natural' negation (alias negation in natural language)

- negation-as-failure (alias weak negation under the preferential semantics of stable models)
- strong negation (alias Kleene negation in 3-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). Using the negation-as-failure tag `<naf>` and the strong negation tag `<neg>`, it may be expressed in RuleML as in Figure 4. For more on the semantics of negation see [Wag02].

Motivation: design goal 4.2.5.

```
<imp>
  <_head>
    <atom>
      <_opr><rel>requiresService</rel></_opr>
      <var>Car</var>
    </atom>
  </_head>
  <_body>
    <and>
      <atom>
        <_opr><rel>mileageSinceLastService</rel></_opr>
        <var>Car</var>
        <var>M</var>
      </atom>
      <atom>
        <_opr>GTE</_opr>
        <var>M</var>
        <const>10000</const>
      </atom>
      <naf>
        <neg>
          <atom>
            <_opr><rel>requiresService</rel></_opr>
            <var>Car</var>
          </atom>
        </neg>
      </naf>
    </and>
  </_body>
</imp>
```

Figure 4: The default rule *Normally, a car requires service after every 10,000 miles* in RuleML 0.8 with `<naf>` and `<neg>`.

4.3.9 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 to express 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*, *lineage/source* for facts and rules, and in addition *priority* for rules.

Motivation: design goal 4.2.6.

5 RuleML

The RuleML standardization initiative has been started in August 2000 by Harold Boley (DFKI, Kaiserslautern) and Said Tabet (Nisus Inc., Boston) 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 *RuleML steering committee*, and a 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 intitiative is `http://www.dfki.de/ruleml` and my personal RuleML website is `http://myRuleML.rezearch.info`.

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 as a design goal for OWL. It is expected that there will be a W3C Working Group for developing a W3C rule markup language, possibly starting in 2003. The work on RuleML will then be a valuable input for such a working group.

The current version of RuleML (in May 2002) has the version number 0.8 and does only cover a very limited form of derivation rules (similar to Datalog). The paper [BTW01] discusses the rationale behind RuleML 0.8 and some future extensions of it.

In the sequel, I give a brief introduction into RuleML 0.8 and discuss some of its weaknesses.

5.1 Main Features of RuleML 0.8

Rather than leaving conjunction implicit, as in Prolog rules, an explicit tag pair `<and>` ...`</and>` marks up a sequence of several conjuncts, preparing for the explicit markup of other connectives, such as `<or>`...`</or>`, and their nesting.

The top-level components of a rule, which are called 'roles', are treated like RDF predicates and marked up by "_"-prefixed tags. A derivation rule is marked up as

```
<imp> <_head> ... </_head> <_body> ... </_body> </imp>
```

or, equivalently, as

```
<imp> <_body> ... </_body> <_head> ... </_head> </imp>
```

For instance, the rule for available cars discussed above is marked up in RuleML 0.8 as in Figure 5.

```
<imp>
  <_head>
    <atom>
      <_opr><rel>isAvailable</rel></_opr>
      <var>Car</var>
    </atom>
  </_head>
  <_body>
     <and>
      <atom>
        <_opr><rel>RentalCar</rel></_opr>
        <var>Car</var>
      </atom>
      <not>
        <atom>
          <_opr><rel>requiresService</rel></_opr>
          <var>Car</var>
        </atom>
      </not>
      <not>
        <atom>
          <_opr><rel>isSchedForMaint</rel></_opr>
          <var>Car</var>
        </atom>
      </not>
      <not>
        <atom>
          <_opr><rel>isAssToRentalOrder</rel></_opr>
          <var>Car</var>
        </atom>
      </not>
    </and>
  </_body>
</imp>
```

Figure 5: The rule for available cars marked up in RulML 0.8.

5.2 Weaknesses of RuleML 0.8

5.2.1 There is no reaction rule format

The main weakness of RuleML 0.8 is its lack of a markup definition for reaction rules. A natural markup for these rules, considered as quadruples

$\langle$ *Event, Condition, Action, Effect* $\rangle$

with an optional *Effect* component, would be

```
<ReactionRule>
  <_event>
    ...
  </_event>
  <_condition>
    ...
  </_condition>
  <_action>
    ...
  </_action>
  <_effect>
    ...
  </_effect>
</ReactionRule>
```

5.2.2 A derivation rule is not an implication

Unfortunately, although logically and conceptually a derivation rule is not an implication, RuleML 0.8 uses the tag `<imp>` for derivation rules.

While an implication is an expression of a logical object language possessing a truth value, a derivation rule is a meta-logical expression that does not possess a truth value but whose role is to generate derived information (like a function from a set of logical sentences to a set of logical sentences). In the knowledge representation (and interchange) language KIF, this distinction between implications and derivation rules has therefore been taken into consideration.

In standard logics, there is a close relationship between a derivation rule and the corresponding implicational formula: they have the same models. For nonmonotonic rules (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.

5.2.3 Too many features are still missing

Too many of the features discussed in section 4.3 are still missing. But this may change soon, as I hope `:-)`.

6 Conclusion

To design a general rule markup language is not an easy exercise. It involves many advanced concepts from the AI and Semantic Web research areas. And in order to be successful it requires a strong focus on practically relevant languages and technologies, and not on the proposals and systems of individual academic researchers (their 'babies'). RuleML 0.8 is a small first step in the direction of such a language standard.

References

[BTW01] Harold Boley, Said Tabet, and Gerd Wagner. Design Rationale of RuleML: A Markup Language for Semantic Web Rules. In *Proc. Semantic Web Working Symposium (SWWS'01)*. Stanford University, July/August 2001.

[CL] Common Logic. Standardization project. http://suo.ieee.org/email/msg08241.html.

[HJW99] H. Herre, J. Jaspars, and G. Wagner. Partial Logics with Two Kinds of Negation as a Foundation of Knowledge-Based Reasoning. In D.M. Gabbay and H. Wansing, editors, *What Is Negation?* Oxford University Press, 1999.

[TW01] K. Taveter and G. Wagner. Agent-Oriented Enterprise Modeling Based on Business Rules. In *Proc. of 20th Int. Conf. on Conceptual Modeling (ER2001)*, pages 527–540, Yokohama, Japan, November 2001. Springer-Verlag. LNCS 2224.

[Wag02] G. Wagner. The Semantic Web Needs Two Kinds of Negation. Mansucript, 2002. available from `myRuleML.rezearch.info`.

[Web] Requirements for a Web Ontology Language. W3C Working Draft 07 March 2002. http://www.w3.org/TR/2002/WD-webont-req-20020307/.