RBSLA
A declarative Rule-based Service Level Agreement Language based on RuleML

Adrian Paschke
Internet-based Information Systems, Technische Universität München
Adrian.Paschke@in.tum.de

Abstract

*This paper describes a **R**ule **B**ased **S**ervice **L**evel **A**greement language called **RBSLA** which is based on RuleML. With this language SLAs can be implemented in a machine readable syntax which can be fed into a rule engine in order to monitor the contract performance at runtime and automatically execute the contractual rules. The declarative logic based approach simplifies interchange, maintenance, management and execution of SLA rules and enables easy combination and revision of contractual rule sets and contract modules.*

1. Motivation

Our studies [1] of a vast number of Service Level Agreements (SLAs) currently used throughout the industry have revealed that today's prevailing contracts are plain natural language documents and almost all contracts focus on a single Quality of Service (QoS) parameter – namely availability. We believe this reflects the early stage in SLA representation and the lack of suitable automated management tools which efficiently measure and continuously monitor and enforce SLAs. Recent commercial SLA management tools[1] hide the contractual logic in their application code or database tiers, i.e. the SLA rules are expressed implicitly and often non-modifiable. This makes it complicated to interchange the distributed agreements, adapt them to new requirements without extensive re-implementation efforts and automatically maintain, monitor and execute large amount of SLAs. However, the upcoming service orientation ("semantic" service oriented computing) and new e-business models such as On-Demand or Utility Computing based on services that are loosely coupled across heterogeneous, distributed, dynamic environments - so called Service Oriented Architectures (SOA) - need new ways of representing SLAs. A suitable SLA language should *make the contractual rules explicit in a formal, machine-readable and interchangeable way* in order to transform them into executable code and exchange them between business partners and organizations. Furthermore, it should *not constrain the parties in the way they formulate the contract rules but instead allow for a high degree of expressiveness, flexibility and agility*. In contrast to conventional imperative programming languages such as Java or C++ *declarative rule languages based on logic programming (LP) techniques* provide a clean separation of concerns by explicitly expressing contractual logic in a formal machine-readable, interchangeable and executable fashion with a high degree of flexibility. But, real usage of a formal rule-based representation language which is usable by others than its inventors immediately makes rigorous demands on the syntax: *declarative machine-readable syntax, comprehension, usability of the language by human users, compact representation, exchangeability with other formats, means for serialization and persistence, tool support in writing and parsing rules (verification/validation)* etc.

In this paper we introduce a declarative **r**ule-**b**ased **S**ervice **L**evel **A**greement language (**RBSLA**) which addresses these requirements. Therefore, it adapts and extends the emerging Semantic Web rule standard **RuleML**[2] to the needs of the SLA domain in order to facilitate interoperability with other rule languages and tool support. The further paper is structured as follows. In section 2 we discuss related work. In section 3 we argue for using RuleML as basis of RBSLA. In section 4 we briefly describe the basic concepts of RuleML. In section 5 we present the RBSLA language and illustrate its usage in section 6. Finally, in section 7 we conclude this paper with a short summary.

2. Related Work

Recently, there are many efforts aiming on rule interchange and building an effective, practical, and deployable rule standard. This includes several important standardization or standards-proposing efforts including RuleML, SWRL, n3, Metalog, KIF and ISO Common Logic, ISO Prolog and others. Some of those have been aimed at more or less specialized purposes, e.g., in the domains of business rules (SRML, BRML), Web Service policies (WS-Policy, WSPL, Policy RuleML, SWSL, WSML) and other areas as well. There have been also several SLA related language approaches such as the widely known Web Service Level Agreements (WSLA) [2] and comparable approaches such as the SLA language (SLAng) or the Web Services Offering Language (WSOL). The semantics of these languages are not represented via LP model theory and inferences are not based on LP proof theory as in our approach but use pure procedural, imperative logic for interpretation and execution which has many disadvantages. In particular they do not provide expressive rule representations with quantifiers, rule chaining, negation as failure etc. to describe the contract logic and have no relations to any rule standard. Comparable to our approach the work of Grosof

[1] e.g. IBM Tivoli Service Level Advisor, HP OpenView, Remedy SLAs

[2] http://www.ruleml.org/

Published at International Conference on Intelligent Agents, Web Technology and Internet Commerce, Vienna, Austria, 2005.

and colleagues [3] must be mentioned. Their Semantic Web Enabling Technology (SWEET) toolkit comprises the CommonRules syntax and enables business rules to be represented in RuleML. Whilst their approach deals with contracts in a broader range namely e-commerce contracts and supports rule priorities via Generalized Courteous Logic Programs (GCLP) [4] and simple procedural attachments, our approach is more **focused on SLA representation** and incorporates further, important and needed logical concepts such as constructs to maintain live representations of contracts for monitoring and state tracking purposes (contract states ~ fluents), explicit rights and obligation management (deontic norms) supplemented with violations and exceptions of norms, complex event processing with active event monitoring and triggering of internal and external reactions as well as a multitude of SLA-related built-ins, full support of user-defined type systems and sophisticated procedural attachments in order to integrate existing (business) object implementations e.g., EJBs, databases and SLA-specific contract vocabularies (e.g. RDFS/OWL ontologies).

3. Why RuleML?

The XML-based Rule Markup language RuleML is a standardization initiative with the goal of creating an open, producer-independent, XML/RDF based web language for rules. It develops a modular, hierarchical specification for different types of rules comprising reaction rules (Event-Condition-Action rules), transformation rules (functional-equation rules), derivation rules (implicational-inference rules), facts (derivation rules with empty bodies), queries (derivation rules with empty heads) and integrity constraints (consistency-maintenance rules) as well as transformations via XSLT from and to other rule standards/systems. [5] It is expected that RuleML will be the declarative method to describe (business) rules on the Web and distributed systems. It allows the deployment, execution, and exchange of rules between different major commercial and non-commercial rules systems like e.g. Jess[3], Mandarax[4] or Prova[5] via XSLT transformations. RuleML is not intended to be executed directly, but transformed into the target language of an underlying rule-based systems (e.g. a Prolog interpreter) and then executed there. Therefore, it perfectly solves some of the requirements stated in section 1 – in particular machine-readability, interoperability with other rule languages and tool support (e.g. transformation into executable code and execution in rule engines). Since the object oriented RuleML (OO RuleML) specification 0.85 it adds further concepts from the object-oriented knowledge representation domain namely user-level roles, URI grounding and term typing and offers first ideas to prioritise rules with quantitative or qualitative priorities. Nevertheless, the latest version 0.88 is still mostly limited to derivation rules, facts and queries. Currently, reaction rules have not been specified in RuleML and other key components needed to efficiently represent SLAs such as procedural attachments on external programs and data sources, complex event processing and state changes as well as normative concepts and violations/exceptions to norms are missing - in order to pick up a few examples. As such improvements must be made. Therefore, RBSLA adds several new concepts to RuleML, e.g.:

- *Typed Logic with Types and Modes*
- *Procedural Attachments*
- *External Data Integration*
- *ECA Rules with Sensing, Monitoring and Effecting*
- *(Situated) Update Primitives*
- *Complex Event Processing and State Changes (Fluents)*
- *Deontic Norms including Violations and Exceptions*
- *Defeasible Rules and Rule Priorities*
- *Built-Ins, Aggregate and Compare Operators, Lists*
- *Additional compact If-Then-Else-Syntax*
- *SLA Domain Specific Elements such as Metrics, Escalation Levels and Ontology-based domain specific Contract Vocabularies*
- *Test Cases for verification and validation of SLA specifications*

Before we evolve our RBSLA language we briefly sketch the basics of RuleML in the following section. For the reason of brevity and compactness we use a DTD-like content model notation and skip definitions which are not relevant for understanding.

4. Fundamentals of RuleML

The building blocks of RuleML are [5]:

Predicates (atoms) are n-ary relations defined as an <Atom> element in RuleML:

Atom (Rel,(Ind|Var)*)

<Var> and <Ind> are **Var**iables to be instantiated by ground values when the rules are applied and **Ind**ividual constants. RuleML also supports **C**omplex **term**s <Cterm>. Constants are either simple names or URIs referring to the appropriate individuals. <Rel> is the predicate. Atoms can be negated with <Neg> or <Naf> which represents negation as failure.

Derivation Rules consist of one or more conditions (<body>) and a conclusion (<head>) which is derived from existing other rules (rule chaining) and facts via a logical conjunction of formulas. Typically, they are horn rules of the form: $H \leftarrow B_1 \wedge ... \wedge B_n$ supplemented with Naf (~) applied in a forward or backward reasoning manner. In RuleML they have the following syntax:

Implies ((head,body)|(body, head))
body (And) head (Atom)
And (Atom)+

Role tags such as <head> or <body> can be omitted. In RBSLA we additionally allow conjunctions "And" in the head and disjunctions "Or" in the body of a rule (cf. section 6).

Facts are derivation rules with empty bodies and are deemed to be always true. The syntax is Atom (Rel,(Ind|Var)*).

[3] http://herzberg.ca.sandia.gov/jess/
[4] http://mandarax.sourceforge.net/
[5] http://www.prova.ws/

Queries are derivation rules with empty heads. Queries can either be proved backward as top-down goals or forward via 'goal-directed' bottom-up processing. Queries have the following syntax: Query (Atom | And)

As mentioned before, improvements must be made to the core RuleML syntax in order to use it as an adequate and efficient representation language for SLAs. The following section introduces the main extensions of RBSLA to RuleML.

5. Extended RBSLA Language

Typed Logic: Logical terms in RBSLA are either un-typed or typed. Types can be assigned to terms using a **type** attribute. RBSLA supports primitive data types and order-sorted type systems, i.e. the type information can be represented as a directed acyclic graph with class – subclass relationships, e.g. Java class hierarchies (fully qualified Java class names) or Semantic Web taxonomies based on RDFS or OWL. Example:

```
<Var type="java.lang.Integer">1234</Var>
<Var type="rbsla:Provider">Service Provider</Var>
```

Values of primitive data types such as integer, string, decimal, float, date, time etc. can be interchanged between the different type systems (e.g. SQL, Java, XML Schema data types etc.), i.e. they are unified and evaluated against each other.

Note: In a rule engine the following rules should apply:

Untyped-Typed Unification:
1. The un-typed query variable assumes the type of the typed target variable or constant (individual)

Variable-Variable Unification:
2. If the query and the target variable are not assignable, the unification fails otherwise it succeeds
3. If the query variable belongs to a subclass of the class of the target variable, the query variable assumes the type of the target variable.
4. If the query variable belongs to a super-class of the class of the target variable or is of the same class, the query variable retains its class

Variable-Constant Unification:
5. If a variable is unified with a constant (individual) of its super-class, the unification fails otherwise if the type of the constant is the same or a sub-type of the variable it succeeds and the variable becomes instantiated.

Constant-Constant Unification:
6. The type of term from the head of the fact or rule is the same as or inherits from the type of term from the body of the rule or query

Modes

In addition to types, which define the domain of the arguments of a predicate, RBSLA supports so call **Modes**. Modes are states of instantiation of the predicate described by mode declarations (mode attributes), i.e. declarations of the intended input-output constellations of the predicate terms with the following semantics:

"+" The term is intended to be *input*
"-" The term is intended to be *output*
"?" The term is *undefined/arbitrary* (input or output)

```
<Atom>                  // add(2,3,Result)
        <Rel>add</Rel>
        <Ind mode="+">2</Ind>
        <Ind mode="+">3</Ind>
        <Var mode="-">Result</Var>
</Atom>
```

The example defines a predicate "*add*" with "*2*" and "*3*" as input and one output variable "*Result*". The default mode is "*?*". During unification the following rules should apply:

+: Goal must be either a constant term or a bound variable
-: Goal must be a free variable
?: Goal assumes the mode of the target

Note: Types and modes can be considered as an *approximation of the intended interpretation*, i.e. as an instrumentation of a logic program. Types and modes help to safeguard the authoring process of SLA rules using RBSLA as an "implementation" language which enables fault analysis via *static and dynamic (test-case based) testing of type and mode correctness*. In addition, they restrict the search space to clauses where the type and mode restrictions are fulfilled and therefore make the unification process more efficiently.

Procedural Attachments are predicates (Boolean valued attachments) or methods (object valued attachments) that are implemented by an external procedure (e.g. a Java method), i.e. they are procedure calls on an external computational model of a standard programming language. They are used to *delegate computation-intensive tasks to optimized procedural code* (e.g. Java), *effect the outside environment* (e.g. sending an eMail via a Java method) or they can *receive information from the outside world* (e.g. via JDBC or Web Service interfaces). Therefore, procedural attachments are a crucial extension of the pure logic inferences used in logic programming. They allow a combination of the benefits of declarative (rule-based) and procedural (imperative) languages such as Java. In RBSLA a procedural attachment is defined as: $O_m[p^1..p^n] \rightarrow [r^1..r^m]$. *O* denotes an object or a class, *m* denotes a method invocation, $p^1..p^n$ the parameters and $r^1..r^m$ the list of one or more result objects which might also be a Boolean true or false value. The serialization in RuleML extends complex terms:

Cterm(Ctor | Attachment) Attachment((Ind|Var|Cterm), Ind)

The first element of an **<*Attachment*>** is either a link on a qualified (Java) class, a variable bound to an object/class instance or a nested complex term which itself constructs/returns an object. The second argument references the called method/constructor. The parameters for the method invocation are the subsequent elements under the Cterm element, which might be empty, e.g.: **java.lang.Integer$_{parseInt}$[1234]→Integer(1234)** is serialized to:

```
<Cterm type="java.lang.Integer">   // (types are optional; here result type)
  <Attachment>
        <Ind>java.lang.Integer</Ind>
        <Ind>parseInt</Ind>
  </Attachment>
  <Ind type="java.lang.String">1234</Ind>        </Cterm>
```

The result of a method invocation finally replaces the complex term and is used in the derivation process. Results can be bound to variables via <*Equal*>: **X = java.lang.Integer**$_{parseInt}$**[1234]**

```
<Equal>    <Var type="java.lang.Integer"> X </Var>
           <Cterm> … [Attachment] … </Cterm>
</Equal>
```

External Data Integration: RBSLA supports language constructs to integrate facts managed by external databases in particular *SQL databases* and *XML/RDF files*. It therefore provides different predefined predicates such as **<Location>** (source location), **<Select>** (SQL select query), **<XML>** (construct a DOM tree from a XML file / select XPath), **<RDF>** (query RDF file) etc. to access and query an external data source such as relational databases, XML files, RDF files in order to reuse the query result as facts for the rule execution. The language constructs are very general and can be given a specialized implementation in the underlying rule engine, e.g. built-in predicates using JDBC, XML APIs, XPath/Xquery engines or Semantic Web engines (e.g. Jena) and description logic engines (e.g. Pellet).

Event Condition Action Rules are rules which autonomously react to events occurring in the internal data (e.g. internal knowledge updates via update primitives) or in the external world (e.g. monitored via procedural attachments), by evaluating a data dependent condition and executing a reaction whenever the event occurred and the condition is true. Accordingly, they are a key feature of a SLA monitoring system and many rules in SLAs are actually ECA rules. In RBSLA an ECA rule **<ECA>** is defined as:

ECA(**time**?, **event**?, **condition**?, **action**, **postcondition**?, **else**?)

time(Naf |Neg | Cterm); **event**(Naf |Neg | Cterm); **condition**(Naf |Neg | Cterm); **action**(Naf |Neg | Cterm); **postcondition** (Naf |Neg | Cterm); **else**(Naf |Neg | Cterm)

Naf(Cterm) **Neg**(Cterm)

The syntax is a striped syntax with two kinds of tags: method-like role tags which start with a lower-case letter and class-like type tags which start with an upper-case letter in order to be compatible with RDF. The respective parts of an ECA rule are defined as <*Cterm*> elements (complex terms)[6] which might be negated[7]. The *Cterms* are interpreted as goals on associated derivation rules which implement the internal functionalities of each ECA part, i.e. the actual logic is decoupled from the ECA rule and represented in terms of derivation rules which are evaluated via querying the rule base with the ECA Cterms. The semantics follows the ECA paradigm, which executes an ECA rule in an active, forward directional manner, i.e., it proceeds with the next ECA part if the query on the currently referenced derivation rule succeeds. The advantages are: First once defined functionalities (derivation rules) can be easily referenced in several ECA rules leading to a compact program description. Second the rich expressiveness of derivation rules and the underlying inference capabilities of logic programs with resolution and unification of variables can be used, i.e. free variables in the ECA Cterms which are bound during unification can be dynamically handed to succeeding ECA parts (Cterms) (including backtracking), e.g. from the event part to the action part. Furthermore, complex events, conditions or actions can be easily represented via logical connectives (and/or) and via chaining derivation rules, e.g. combining several atomic events into a complex event via "and" conjunctions. The optional <*time*> term is introduce to define monitoring intervals in order to control the monitoring/evaluation costs of each ECA rule and define its validity period, e.g. "*every 10 minutes between 9a.m. and 18p.m*". The optional <*postcondition*> term defines a post-condition which should hold after execution of the action. If the post-condition fails, a rollback might be applied, e.g. rollback internal knowledge updates. Additionally, the post-condition can be used to set a cut, so that there is no backtracking, i.e. the ECA is successfully executed only once for one variable binding. The optional <*else*> term defines an alternative action which is triggered if the event part fails, e.g. in order to combine two rules "*if event then do action1*" and "*if not event then do action2*" into one rule "*if event then do action1 else action2*; here "*not event*" is a negative event test, e.g. *not(available(service))*. If a complex term is empty, i.e. without a value, it is simply omitted which leads to different ECA rule variants such as CA (*ECA(condition,action)*) rules a.k.a. production rules.

Example:

```
eca(everyMiinute(T),available("http://www.google.de"),, sendNotification(T)).

<ECA>
 <time><Cterm><Ctor>everyMinute</Ctor> <Var>T</Var> </Cterm></time>
 <event><Cterm><Ctor>available</><Ind>"http://www.google.de"</></></></>
  <action><Cterm><Ctor>sendNotification</ > <Var>T</></></></action>
</ECA>
```

Note: If the underlying rule engine is based on backward reasoning, the active behaviour of ECA rules must be simulated. This can be done by frequently querying the rule engine. Threads might be used to apply concurrent execution of ECA rules.

Update Primitives: RBSLA supports primitives to *add* (**<Assert>**) and *delete* (**<Retract>**; **<RetractAll>**) knowledge. These primitives might be applied in ECA rules, e.g. internal update actions which assert new information or active rules where knowledge updates are itself events which trigger further update actions.

---

[6] We use complex terms (<Cterm>) and not e.g. <Atom> or <Query> because the intended interpretation in an ECA rule is that the term constructs a true/false value which can be derived by querying the knowledge base but also via procedural attachments, i.e. calls on external method/functions (within a <Cterm> element) which return Boolean.

[7] RBSLA supports negation as failure and classical negation. Note: free Variables in negated terms are not allowed.

Note: The execution model of active rules might result in non-terminating computations (*termination*) and unclear behaviour (*confluence*) due to loops, procedural interpretation or multi-threaded, concurrent execution of ECA rules. An underlying logic system therefore should implement techniques for checking termination and confluence properties and support transactional updates where a sequence of atomic updates must be completely executed or not at all.

Complex Event Processing and State Changes (Fluents): RBSLA supports complex event processing and temporal reasoning about events/actions and their effects on the internal state of the knowledge system. Compound events can be easily represented via conjunctions ("and") of several events in a referenced (ECA event derivation) rule. In addition, RBSLA supports a model of state changes a la event calculus in which *events* happen at *time-points* and *initiate* and/or *terminate time-intervals* over which some *properties* (time-varying **fluents**) of the world hold. Therefore, it defines the following constructs (a la event calculus):

Fluent: fluent(Ind|Var|Cterm)
Parameters: parameter(Ind|Var|Cterm)

The EC axioms are serialized on the level of <Atoms>, i.e. they can be directly used in rules and as facts:

Persistent Events: Happens((event | action | Ind | Var | Cterm) , (time | Ind | Var | Cterm))
Believed Events: Planned((event | action | Ind | Var | Cterm) , (time | Ind | Var | Cterm))
Effects of events:
- Initially(fluent | Ind | Var | Cterm)
- Initiates((event | action | Ind | Var | Cterm), (fluent | Ind | Var | Cterm), (time | Ind | Var | Cterm))
- Terminates((event | action | Ind | Var | Cterm), (fluent | Ind | Var | Cterm), (time | Ind | Var | Cterm))

Queries:
- HoldsAt((fluent | Ind | Var | Cterm), (time | Ind | Var | Cterm))
- ValueAt((parameter | Ind | Var | Cterm), (time | Ind | Var | Cterm), (Ind|Var|Cterm))

This enables to model the effects of events on changeable SLA properties (e.g. rights or obligations) and to reason about the contract state at certain time points (a.k.a. contract tracking).

Deontic Norms and Norm Violations: The main aim for concluding a contract is to arrange the normative relationships relating to permissions, obligations and prohibitions between contract partners. RBSLA provides constructs to serialize such deontic norms as *personalized, time-varying* fluents of the form: *norm(S,O,A)* *(S=Subject, O=Object, A=Action)*

<Oblige><Ind>*S*</><Ind>*O*</><action><Ind>*A*</></action>**</Oblige>**
<Permit><Ind>*S*</><Ind>*O*</><action><Ind>*A*</></action> **</Permit>**
<Forbid><Ind>*S*</><Ind>*O*</><action><Ind>*A*</></action>**</Forbid>**
<Waived><Ind>*S*</><Ind>*O*</><action><Ind>*A*</></action>**</Waived>**

Example: *oblige(provider, consumer, pay(Penalty))*

```
<Oblige><Ind>provider</Ind><Ind>consumer</Ind>
  <action><Cterm><Ctor>pay</Ctor><Var>Penalty</Var></Cterm></action>
</Oblige>
```

To some extend the *subject* and *object* and the fluent-based normative proposition which determinates a truth value (holdAt) can be seen as the "*context*" under which the *action* is obligatory. This means *oblige(S_1,O,a)* and *oblige(S_2,O,¬a)* can both hold at the same time. Using the previously defined event processing constructs (initiates, terminates), **deontic rules** a.k.a. conditional norms (this differs from the "context" notion of dyadic deontic logic) can be defined, e.g., $a_2 \rightarrow O(a_1)$ which states that if a_2 (the condition) is the case then a_1 (the conclusion) should be the case:

```
<Initiates>  <action><Ind>a2</Ind></action>
             <Oblige> …a1 …</Oblige>
             <Time><Var>T</Var></Time>          </Initiates>
```

Therewith, **Contrary-to-duty (CTD) obligations**, obligations which are conditional to a violation, and **defeasible obligations** (incl. prima facie obligations[8]), obligations which are subject to exceptions (exceptional circumstances) can be represented. RBSLA defines two explicit events for violations (e.g., *O(a)* ∧ ¬a) **<Violation>** and exceptions (e.g., *O(a); e* → *O(¬a)*) **<Exception>**.

Note: Deontic Logic is plagued by a large number of so called "*paradoxes*", sets of sentences that derive sentences with a counterintuitive reading. A logic system must provide mechanisms to deal with these problems. Because, in RBSLA norms are embedded in temporal constructs an elegant way to deal with the formalization problem of contradicting norms is to terminate the normal, default or *primary norm* and instantiate the *secondary CTD or exceptional norm* so that there is never a situation where contradicting norms are true at the same time. This works for time-based norms (luckily most norm regulations in SLAs are time based) but not for time-less paradoxes (such as the well-know "gentle murder" paradox), e.g.:

(1)The service provider must not violate an agreed service level.
(2) But, if a service level is violated, the violation should be as small as possible.

In such cases defeasibility might be used in such way that the counterintuitive sentences are no longer derived, i.e. the primary obligation might be defined with higher priority than the secondary obligation so that the primary norm is blocked by the derivation of the secondary obligation using defeasible logic formalisms.

Defeasible Rules and Rule Priorities: In order to deal with conflicts (e.g. positive and negative information) and rules of precedence (rule priorities) RBSLA supports defeasible rules "*body => head*" in addition to strict rules (derivation rules of the form "*head* → *body*" *(<Implies>)*). Therefore, in RBSLA a new rule element **<Defeasible>** is introduced. To express incompatible and conflicting literals between rules we use an *<Integrity>* constraint:

```
<Integrity>
  <Atom>  <Rel>discount</Rel> <Var>X</Var></Atom>
  <Atom>  <Rel>discount</Rel> <Var>Y</Var></Atom>
```

[8] Promised prima facie duties formalized as defeasible conditionals

```
  <Atom><Cond><Neg><Equal>  <Var>X</Var>  <Var>Y</Var>
    </Equal></Neg></Cond></Atom>
</Integrity>
```

An **<Overrides>** element defines the priority of rules or rule sets / modules:

```
<Overrides> <Ref><Ind>rule1</Ind></Ref><Ref><Ind>rule2</Ind></Ref></>
```

Built-Ins, Aggregate and Compare Operators, Lists: RBSLA provides different useful built-in functions and predicates to effectively work with variables, numbers, strings, date and time values, lists etc. Here we can only list the interesting ones:

Variables:
- **<Bound>, <Free>**: Test whether variable is instantiated or not

Numbers:
- **<Add|Sub|Mult|Div|Mod|Max|Min|Abs>**: Compute numeric values.

Strings:
- **"\"** : separator "\" for special characters in string such as \n \r \t etc.
- **<Concat>, <Parse>, <Tokenize>** etc.

Date and Time:
- **<SysTime>** Current system time.
- **<Date> <Ind>***Year*</><Ind>*Month*</><Ind>*Day*</> </Date>**
- **<Time><Ind>***Hour*</>, <Ind>*Min*</>, <Ind>*Sec*</> </Time>**
- **<DateTime>** Combined Date/Time or Epoch value (millisecond)
- **<TimeSpan>, <Intervall>**
- **<Compare>, <Less>, <Equal>, <More>, <Add>, <Sub>** etc.

Lists:
- **<Plex><Var1/><Var2/>**...**<VarN/></Plex>**: List [Var1 .. VarN]
- **<Plex><Var>***Head*</><*repo*><Var>*Rest*</></></Plex>:** [Head|Rest]
- **<Member>**: Test whether an object is member of a list
- **<Element_At>**: Return the object at position X in a list
- **<Append>, <Delete>**: Add/Delete an element/list.
- **<Head>, <Tail>**: Return the head / the tail of a list
- **<First>, <Last>**: Return the first / last element of a list
- **<Size>**: Return the size of a list

Aggregations:
- **<Sum>,<Max>,<Min>,<Mean>**: Calculate the aggregation of a list

Comparison:
- **<Equal>, <LessEqual>, <Less>, <More>, <MoreEqual>, <Between>**

Test Cases: Engineering of SLA rule sets is complicated and error-prone. Test Cases can be used to validate and verify the correctness of the SLA specifications. They can be used to safeguard the authoring, interchange and execution process of rule sets. A test case consists of a set of test goals *{"?p1(a) ⇒ true", "?p1(b) ⇒ false"}* and a set of test input facts *{p1(a), p2(a)}*; the test goals should be interpreted as follows "*querying the rule engine with p1(a) should yield true and with p1(b) false*". In RBSLA a test case <TestCase> is serialized as set of tests <*Test*> consisting of a RuleML <*Query*> (with one or more goals connected via implicitly assumed "and") and an intended result <*Result*> and a set of test input facts stated as <*Atom*>s:

```
TestCase(Test*, Atoms*)
Test(Query,Result)
```

A test case can be annotated with meta data about its semantics with an optional *@semantic* attribute in order to support distributed scenarios where rules are exchanged and executed in different inference engines with different semantics.

If-Then-Else-Syntax: In order to make programming in RBSLA and specification of SLAs more efficient and easier, RBSLA provides an additional human-oriented *If-Then-Else syntax* for rules:

```
<If>     Body      </If>
<Then>   Head      </Then>
<Else>   Else Head <Else>
```

Whilst the if-then part of such a rule maps to a normal RuleML derivation rule (*<Implies>*) the else part maps to a corresponding negated (with Negation as Failure NaF) rule. ECA rules can be written as *If (time) and (event) and (condition) then (action)*. For defeasible rules a special attribute *@defeasible* must be set in the <If> element.

SLA Domain Specific Vocabularies and Elements: For typical components which frequently occur in SLAs such as *(SLA) metrics*, *escalation levels*, *rights and obligations*, *pricing policies* RBSLA provides direct serializations in order to support the design and building process of SLAs in RBSLA. As mentioned before RBSLA supports the *integration of external (contract) vocabularies* (e.g. RDFS/OWL ontologies, Java class hierachies) including representation of business objects into domain-independent rules via typed rule terms. In particular this enables reusing rules in different contexts and facilitates rule interchange between domain boundaries, e.g., between organizations. We have implemented a first *ontology of useful and typical concepts in the SLA domain*, such as service provider/consumer (role models which can be used in personalized deontic norm), metrics (direct resource metrics, compound SLA metrics, process metrics, and business metrics), SLA parameter etc. The defined concept classes can be used to type terms in SLAs: <Ind type="**rbsla:provider**">*provider name*</Ind>
Mappings between different vocabularies, e.g. from different organizations, can be defined (in OWL – note: this slows down performance because of the complex equivalence inference processes).

Remarks about RBSLA: RBSLA follows the design principle of RuleML which means the new concepts come in a layered structure where each layer adds different modelling expressiveness to the RuleML core, which is the *horn logic layer extended with negation as failure and disjunctions*. The layers are not organized around complexity, but are based on the different underlying logical formalism such as deontic logic (*deontic layer*), event calculus (*event/action layer*), logic of events/actions (*ECA active rules layer*) etc. and on the level of abstraction: *core RuleML syntax, abbreviated RBSLA syntax, compact if-then-else syntax*. It is very important to note, that serialization of RBSLA does not require any new constructs, i.e., it can be done by using the existing RuleML features via normalizing (XSLT transformation) the special RBSLA constructs into the usual RuleML syntax. For example:

```
<Oblige>
        <Ind>service provider</Ind>
        <Ind>service consumer</Ind>
```

```
            <action><Ind>payPenalty</Ind></action>
</Oblige>
```
Maps to RuleML: oblige("service provider","service consumer","payPenalty")
```
<Cterm>
            <Ctor>oblige</Ctor>
            <Ind >service provider</Ind>
            <Ind >service consumer</Ind>
            <Ind >payPenalty</Ind>
</Cterm>
```

Another important point is the transformation of RBSLA into an underlying logic system. During the transformation automated "*refactorings*" can be applied in order to improve the execution efficiency in the underlying logic system and reducing extended logic programs to general logic programs, e.g.:

- ***Narrowing:*** $A_{1,..,}A_N \rightarrow B$ and $A_{1,..,}A_N \rightarrow C$ becomes $A_{1,..,}A_N \rightarrow A$; $A \rightarrow B$; $A \rightarrow C$ (**eliminates redundancies**)
- ***Removing Disjunctions*****:** $A_1 \wedge A_n \wedge (B_1 \vee B_2) \rightarrow C$ becomes $A_1 \wedge A_n \wedge B' \rightarrow C$ and $B_1 \rightarrow B'$ and $B_2 \rightarrow B'$ (**clausal normal form**)
- ***Removing conjunctions from rule heads:*** $B \rightarrow (H \wedge H')$ via Lloyd-Topor transformation into $B \rightarrow H$ and $B \rightarrow H'$)

6 Writing SLAs using RBSLA

In this section we illustrate the logical representation of SLAs using RBSLA. The SLA defines **three monitoring schedules** *prime* (8a.m.-18p.m. where service availability is tested every minute), *standard* (18p.m. - 8a.m. where service availability is tested every 10 minutes):

```
<Implies>                                   // defines the monitoring schedule
    <body>[ time schedule definition]</body>
    <head><Atom><Rel>schedule</Rel><Ind>[name]</Ind></Atom> </head>
</Implies>
```

Further it defines **three escalation levels** with different roles: *process officer*, *chief quality manager* and *control committee*. The process officer is informed in escalation level 1:

```
<Eca>       // trigger escl. level 1 if service unavailable and not maintenance
  <time><Cterm><Ctor>schedule</Ctor><Var>Name</Var></Cterm><time>
  <event><Attachment><Ind>pingService</Ind></Attachment></event>
   <action><Cterm><Ctor>escl_lv1</Ctor></Cterm><Action>
</Eca>
<Implies>          //defines escl. lv.1 (inform + add unavail. event)
  <And>     <SysTime>T</SysTime>
 <Atom><Cterm>  <Attachment>  …  inform  process  officer  …</
></Cterm></Atom>
 <Assert><Happens><event><Ind>unavail</></></></><time>T</time></Assert>
  </And>
 <Atom><Rel>escl_lv1</Rel></Atom> //head
</Implies>
```

If the service is unavailable, then the process manager is obliged to restart the service within time $t_{time\text{-}to\text{-}repair}$.

```
<Initiates>                 // Initiate obligation if service is unavailable.
   <event><Ind>unavail</Ind></event> // service unavail.
   <fluent><Oblige> // oblige process manager to restart the service
       <Ind>process manager</Ind><Ind>service</Ind>
       <action>.. restartSerivce … </action>
    </Oblige></fluent> <time><Var>T</Var></time>
</Initiates>
```

If the process manager fails to restart the service within $t_{time\text{-}to\text{-}repair}$ (violates obligation) than escalation level 2 is triggered, which informs the chief quality manager.

```
<Eca>       // trigger escl. level 2 if primary obligation is violated
  <event><Cterm><Ctor>violated</Ctor></Cterm></event>
  <action><Cterm><Ctor>escl_lv2</Ctor></Cterm><action>
</Eca>
<Implies>    // define violation condition with parameter "ttr"
  <And> <SysTime>T</SysTime>
       <ValueAt><parameter><Ind>ttr</Ind></arameter>
                 <time><Var>T</Var></time>
                  <Ind> t time-to-repair</Ind>
       </ValueAt>   </And>
  <Atom><Rel>violated</Rel></Atom></Implies> // head
```

The chief quality manager typically has a higher scope with more rights, e.g. the right (*permission*) to adapt the quality management systems respectively the service levels. For example the chief quality manager might discuss the needed time to repair with the process manager and extend it up to an agreed *maximum time to repair* level. If the service is still unavailable after this maximum threshold value then escalation level 3 is reached which permits the service consumer to cancel the contract:

```
<Initiates>       // initiates "reparational" permission norm
            <Violation></Ind>[violated obligation]</Ind></Violation>
            <Permit> … [cancel contract]  </Permit>
          <Var>T</Var>          </Initiates>
```

The informed control committee which typically consists of high-ranked persons (with authorities to decide) from both parties might discuss the situation and contract additional penalty payments in order to prevent the contract termination.

The code of an imperative programming language (Java, C++ etc.) implementing the same SLA logic would be much more cumbersome and difficult to write, maintain and update. The declarative rule approach based on RBSLA allows a more compact and intuitive representation. RBSLA can be translated via a compiler (e.g. XSLT RuleML transformation) into an underlying logical system and executed by a rule engine.

7 Conclusion

In this paper we have presented a declarative rule based SLA language called RBSLA which extends RuleML with useful and needed constructs for SLA representation. The language can be easily interchanged and fed in a suitable rule engine to execute and monitor the contract performance at run time. Wit RBSLA SLAs can be implemented in a machine readable, interchangeable and executable syntax based on RuleML.

References

1. Paschke, A., Schnappinger-Gerull, E.. *A Categorization Scheme for SLA Metrics. Multi-Conference Information Systems (MKWI)*. 2006. Passau, Germany.
2. Dan, A., et al., *Web Services on demand: WSLA-driven Automated Management.* IBM Systems Journal, Special Issue on Utility Computing, 2004. 43(1).
3. Grosof, B.N., Y. Labrou, and H.Y. Chan. *A Declarative Approach to Business Rules in Contracts: Courteous Logic Programms in XML*. in *Conf. on Electronic Commerce (EC-99)*. 1999. Denver UK: ACM Press.
4. Grosof, B.N., *A Courteous Compiler From Generalized Courteous Logic Programs To Ordinary Logic Progams.* 1999, IBM T.J. Watson Research Center.
5. Wagner, G., S. Tabet, and H. Boley, *MOF-RuleML: The Abstract Syntax of RuleML as a MOF Model*. 2003, OMG Meeting: Boston.