A Logic Based SLA Management Framework

Adrian Paschke[1], Jens Dietrich[2], Karsten Kuhla[3]

[1]Internet-based Information Systems, Technische Universität München
Paschke@in.tum.de
[2]Information Sciences & Technology, Massey University
J.B.Dietrich@massey.ac.nz
[3]FileAnts GmbH München
Kuhla@fileants.com

Abstract. Management, execution and maintenance of Service Level Agreements (SLAs) in the upcoming service oriented IT landscape need new levels of flexibility and automation not available with the current technology. In this paper we evolve a rule based approach to SLA representation and management which allows a clean separation of concerns, i.e. the contractual business logic are separated from the application logic. We make use of sophisticated, logic based knowledge representation (KR) concepts and combine adequate logical formalisms in one expressive logic based framework called **ContractLog.** ContractLog underpins a declarative rule based SLA (**RBSLA**) language with which to describe SLAs in a generic way as machine readable and executable contract specifications. Based on ContractLog and the RBSLA we have implemented a high level architecture for the automation of electronic contracts - a rule-based Service Level Management tool (**RBSLM**) capable of maintaining, monitoring and managing large amounts of complex contract rules.

1 Why declarative rule-based SLA representation?

Our studies of a vast number of SLAs currently used throughout the industry have revealed that today's prevailing contracts are plain natural language documents. Consequently, they must be manually provisioned and monitored, which is very expensive and slow, results in simplified SLA rules and is not applicable to a global distributed computing economy, where service providers will have to monitor and execute thousands of contracts. First basic automation approaches and recent commercial service management tools[1] directly encode the contractual rules in the application logic using standard programming languages such as Java or C++. The procedural control flow must be completely implemented and business logic, data access and computation are mixed together, i.e. the contract rules are buried implicitly in the application code. SLAs are therefore hard to maintain and can not be adapted to new requirements without extensive reimplementation efforts. Consequently, the upcoming service orientation based on services that are loosely coupled across heterogeneous, dynamic environments needs new ways of knowledge representation with a high degree of flexibility in order to efficiently manage, measure and continuously monitor and enforce complex SLAs. Examples which illustrate this assertion are *dynamic rules*, *dependent rules*, *graduated rules* or *normative rules with exceptions/violations*

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

of rights and obligations, in order to pick up a few examples which frequently occur in SLAs:

- **Graduated rules** are rule sets which e.g. specify graduated high/low ranges for certain SLA parameters so that it can be evaluated whether the measured values exceed, meet or fall below the defined service levels. They are often applied to derive graduate penalties or bonuses. Other examples are service intervals, e.g., **"***between 0 a.m. and 6 a.m. response time must be below 10 s, between 6 a.m. and 6 p.m. response time must be below 4 s …"* or exceptions such as *maintenance intervals*.
- **Dependent rules** are used to adapt the quality of service levels. For example an reparation service level must hold, if a primary service level was missed (for the first time / for the umpteen times), e.g.: *"If the **average availability** falls below 97 % then the **mean time to repair** the service must be less than 10 minutes."*
- **Dynamic rules** either already exist within the set of contract rules or are added dynamically at run time. They typically define special events or non regular changes in the contract environment, e.g. a rule which states that *there might be an unscheduled period of time which will be triggered by the customer. During this period the bandwidth must be doubled*.
- **Normative rules with violations and exceptions** define the rights and obligations each party has in the present contract state. Typically, these norms might change as a consequence of internal or external events / actions, e.g. an obligation which was not fulfilled in time and hence was violated, might raise another obligation or permission: "*A service provider is obliged to repair an unavailable service within 2 hours. If she fails to do so the customer is permitted to cancel the contract.*"

The code of pure procedural programming languages representing this type of rules would be cumbersome to write and maintain and would not be considered helpful in situations when flexibility and code economy are required to represent contractual logic. In this paper we describe a declarative approach to SLA representation and management using sophisticated, logic based knowledge representation (KR) concepts. We combine selected adequate logical formalisms in one expressive framework called **ContractLog.** It enables a clean separation of concerns by specifying contractual logic in a formal machine-readable and executable fashion and facilitates delegating service test details and computations to procedural object-oriented code (Java) and existing management and monitoring tools. ContractLog is extended by a declarative **r**ule **b**ased **SLA** language (**RBSLA**) in order to provide a compact and user-friendly SLA related rule syntax which facilitates rule interchange, serialization and tool support. Based on ContractLog and the higher-level RBSLA we have implemented a **r**ule **b**ased **s**ervice **l**evel **m**anagement (**RBSLM**) prototype as a test bed and proof of concept implementation. The essential advantages of our approach are:

1. Contract rules are separated from the service management application. This allows easier maintenance and management and facilitates contract arrangements which are adaptable to meet changes to service requirements dynamically with the indispensable minimum of service execution disruption at runtime, even possibly permitting coexistence of differentiated contract variants.
2. Rules can be automatically linked (rule chaining) and executed by rule engines in order to enforce complex business policies and individual contractual agreements.
3. Test-driven validation and verification methods can be applied to determine the correctness and completeness of contract specifications against user requirements [1] and large rule sets can be automatically checked for consistency. Additionally, explanatory reasoning chains provide means for debugging and explanation. [2]

4. Contract norms like rights and obligations can be enforced and contract/norm violations and exceptions can be (proactively) detected and treated via automated monitoring processes and hypothetical reasoning. [2]
5. Existing tools, secondary data storages and (business) object implementations might be (re)used by an intelligent combination of declarative and procedural programming.

The contribution of this paper is twofold. First it argues that in a dynamic service oriented business environment a declarative rule based representation of SLAs is superior to pure procedural implementations as it is used in common contract management tools and gives a proof of concept solution. Second it presents a multi-layered representation and management framework for SLAs based on adequate logical concepts and rule languages. The paper is organised as follows. We give an overview on our approach in section 2 and describe its main components, in particular the expressive logical ContractLog framework, the declarative XML based RBSLA language and the RBSLM tool in the sections 3 to 5. In section 6 we present a use case in order to illustrate the representation, monitoring and enforcement process. Finally, in section 7 we conclude with a short summary and some final remarks on the performance and usability of the presented rule-based SLA management approach.

2 Overview

Fig. 1 shows the general architecture of our **r**ule **b**ased **s**ervice **l**evel **m**anagement tool (**RBSLM**). The rule engine Mandarax (1) acts as execution component and inference engine for the formalized, logical contract rules. The rules are represented on basis of our expressive, logical **ContractLog** framework (2) and are imported over the Prova language extension into the internal knowledgebase of the rule engine. An additional declarative language format based on XML[2], called the **r**ule **b**ased **SLA** (**RBSLA**) language, is provided to facilitate machine-readability, reusability, rule interchange, serialisation, tool based editing and verification. A mapping is defined which transforms the RBSLA into executable ContractLog rules. The graphical user interface - the **Contract Manager** (4) - is used to write, edit and maintain the SLAs which are persistently stored in the contract base (3). The **repository** (5) con-

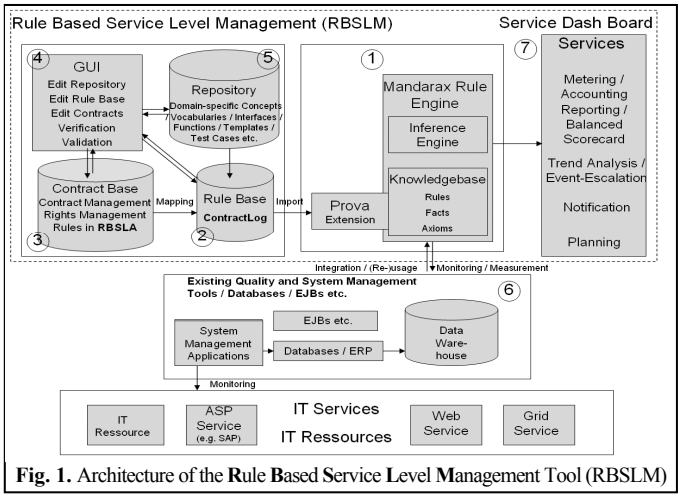

Fig. 1. Architecture of the **R**ule **B**ased **S**ervice **L**evel **M**anagement Tool (RBSLM)

[2] The semantic web rule standard RuleML (in particular object oriented RuleML)

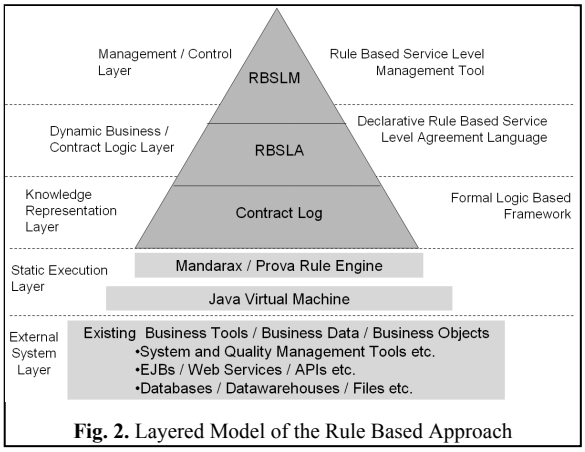

Fig. 2. Layered Model of the Rule Based Approach

tains typical rule templates and predefined SLA domain specific objects, built-in metrics and contract vocabularies (ontologies) which can be reused in the SLA specifications. During the enforcement and monitoring of the SLAs existing external business object implementations, quality and system management tools and external databases can be integrated (6). Finally, the **Service Dash Board** (7) visualizes the monitoring results and supports further SLM processes, e.g. reports on violated services, metering and accounting functionalities, notification services etc. Figure 2 summarizes the components of our rule based SLM approach in a layered model. In the following three sections we give a more detailed insight into the different layers described in figure 2 with a particular focus on the RBSLA serialization syntax.

3 ContractLog Framework

Table 1. Main logic concepts of ContractLog

Logic	Usage
Derivation rules (horn rules with NaF)	Deductive reasoning on contract rules.
Event-Condition-Action rules (ECA)	Active sensing and monitoring, event detection and "situative" behaviour by event-triggered actions.
Event Calculus (temporal reasoning a la "transaction logic")	Temporal reasoning about dynamic systems, e.g. effects of events on the contract state (fluents).
Defeasible / Courteous logic (priorities and conflicts)	Default rules and priority relations of rules. Facilitates conflict detection and resolution as well as revision/updating and modularity of rules.
Deontic Logic (contract norms) with norm violations and exceptions	Rights and obligations modelled as deontic contract norms and norm violations (contrary-to-duty obligations) and exceptions (condit. defeasible obligations).
Description logic (domain descriptions) / Description Logic Programs	Semantic domain descriptions (e.g. contract ontologies) in order to hold rules domain independent. Facilitates exchangeability and interpretation.
Object-oriented typed logic and procedural attachments	Typed terms restrict the search space. Procedural attachments integrate object oriented programming into declarative rules.

Table 1 summarizes the main concepts used in ContractLog. In the following we sketch the basic logical components. More details can be found in [2-4] and on our project site[5].

Typed Derivation Rules, Procedural Attachments and external Data Integration: Derivation rules based on horn logic supplemented with negation as failure (NaF) and

rule chaining enable a compact representation and a high degree of flexibility in automatically combining rules to form complex business policies and graduated dynamic contract rules. [6] On the other hand procedural logic as used in programming languages is highly optimized in solving computational problems and many existing business object implementations such as EJBs as well as existing system management and monitoring tools already provide useful functionalities which should be reused. Procedural attachments and the typed logic[3] used in ContractLog offer a clean way of integrating external programs into logic based rule execution paving the way for intelligently accessing or generating data for which the highest level of performance is needed and the logical component is minimal. This supports a smooth integration of facts managed by external systems (databases accessed via optimized query languages like SQL; systems, accessed using web services etc.) and avoids replication, because references are resolved at query time - which is crucial, as in SLA management we are facing a knowledge intensive domain which needs flexible data integration from multiple rapidly changing data sources, e.g. business data from data warehouses, system data from system management tools, process data from work flows, domain data from ontologies etc. Additionally, the tight integration with Java enables (re-)using existing business objects implementations such as EJBs and system management tools and allows for active sensing/monitoring and effecting via triggered actions in ECA rules.

ECA Rules: A key feature of a SLA monitoring system is its ability to actively detect and react to events. We implemented support for active ECA rules: *eca(T,E,C,A)*. Each term *T* (time), *E* (event), *C* (condition) and *A* (action) references to a derivation rule which implements the respective functionality of the term. The additional term *T* (time) is introduced to define monitoring intervals or schedules in order to control monitoring costs for each rule and to define the validity periods. Example:

eca(everyMinute, serviceUnavailable, notScheduledMaintanance , sendNotification)			
everyMinute(DT) ←	*serviceUnavailable(DT)* ←	*notScheduledMaintanance(DT)* ←	*endNotification(DT)* ←

Rule chaining combining derivation rules offers maximum flexibility to build complex functionalities, which can be referenced and reused in several ECA rules. More details on the ECA implementation can be found in [2-4].

Event Calculus: The Event Calculus (EC) [7] defines a model of change 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. We implemented the classical logic formulations using horn clauses and made some extensions to the core set of axioms to represent derived fluents, delayed effects (e.g. validity periods and deadlines of norms), continuous changes (e.g. time-based counter) and epistemic knowledge (planned events for hypothetical reasoning) [2, 4]:

Classical Domain independent predicates/axioms	**ContractLog Extensions**
happens(E,T) event E happens at time point T	*valueAt(P,T,X)* parameter P has value X at time point T
initiates/terminates(E,F,T) E initiates/terminates fluent F	*planned(E,T)* event E is believed to happen at time point T
holdsAt(F,T) fluent F holds at time point T	*derivedFluent(F)* derived fluent F

[3] ContractLog supports typed variables and constants based on the object-oriented type system of Java or other OO typed systems.

The EC and ECA rules might be combined and used vice versa, e.g. fluents might be used in the condition parts of ECA rules or ECA rules might assert detected events to the EC knowledgebase. The EC models the effects of events on changeable SLA properties (e.g. deontic contract norms such as rights and obligations) and allows reasoning about the contract state at certain time points. Its rules define complex transaction logics with state changes similar to workflows. This is very useful for the representation of deontic contract norms and exceptions or violations of norms.

Deontic Logic with Norm Violations and Exceptions: Deontic Logic (DL) studies the logic of normative concepts such as obligation (O), permission (P) and prohibition (F). However, classical standard deontic logic (SDL) offers only a static picture of the relationships between co-existing norms and does not take into account the *effects of events on the given norms, temporal notions* and *dependencies between norms*, e.g. violations of norms or exceptions. Another limitation is the inability to express *personalized statements*. In real world applications deontic norms refer to an explicit concept of an agent. We extended the concepts of DL with a role-based model and integrated it in the Event Calculus implementation in order to model the effects of events/actions on deontic norms [2]. This enables the definition of institutional power assignment rules (e.g. empowerment rules) for creating institutional facts which are initiated by a certain event and hold until another event terminates them. Further, we can define complex dependencies between norms in workflow like settings which exactly define the actual contract state and all possible state transitions and which allow representing norm violations and their conditional secondary norms, e.g. contry-to-duty (CTD) obligations as well as exceptions of (defeasible prima facie) norms.. A deontic norm consists of the normative concept (*norm N*), the subject (*S*) to which the norm pertains, the object (*O*) on which the action is performed and the action (*A*) itself. We represent a role based deontic norm $N_{s,o}A$ as an EC fluent: *norm(S, O, A)*, e.g. *inititates(e1,* ***permit(s,o,a)****, t1)*. We implemented typical DL inference axioms in ContractLog such as $O_{s,o}A \rightarrow P_{s,o}A$ or $F_{s,o}A \rightarrow W_{s,o}A$ etc. and further rules to deal with deontic conflicts (e.g. $P_{s,o}A \wedge F_{s,o}A$), exceptions (E) ($E \rightarrow O_{s,o}\neg A$), violations (V) of deontic norms (e.g. $O_{s,o}A \wedge \neg A$) and contrary-to-duty (CTD) obligations ($V \rightarrow O_{s,o}CTD$) or other "reparational" norms. In particular *derived fluents* and *delayed effects* (with *trajectories and parameters* [2]) offer the possibility to define norms with deadline-based validity periods, (time-based) violations of contract norms and conditional secondary norms e.g., contrary-to-duty (CTD) obligations. A typical example which can be found in many SLAs is a primary obligation which must be fulfilled in a certain period, but if it is not fulfilled in time, then the norm is violated and a certain "reparational" norm is in force, e.g., a secondary obligation to pay a penalty or a permission to cancel the contract etc. [2-4]

Remark. DL is plagued by a large number of paradoxes. We are aware of this. However, because our solution is based on temporal event logic we often can avoid such conflicts, e.g. a situation where a violated obligation and a CTD obligation of the violated obligation are true at the same time is avoided by terminating the violated obligation so that only the consequences of the violation (CTD obligation) are in effect. Other examples are **defeasible prima facie obligations** ($O_{s,o}A$) which are subject to exceptions ($E \rightarrow O_{s,o}\neg A$) and lead to contradictions, i.e. $O_{s,o}\neg A$ and $O_{s,o}A$ can be derived at the same time. We terminate the general obligations in case of an exception and ini-

tiate the conditional more specific obligation till the end of the exceptional situation. After this point the exception norm is terminated and we re-initiate the initial "default" obligation. Note that we can also represent norms which hold initially via the *initially* axiom in order to simulate "non-temporal" norms. A third way is to represent conflicts as **defeasible deontic rules** with defined priorities (*overrides*) between conflicting norms, i.e. we weaken the notion of implication in such a way that the counterintuitive sentences are no longer derived (see. defeasible logic).

Defeasible Logic: We adapt two basic concepts in ContractLog to solve conflicting rules (e.g. conflicting positive and negative information) and to represent rule precedences: Nute's defeasible logic (DfL) [8] and Grosof´s Generalized Courteous Logic Programs (GCLP) . There are four kinds of knowledge in DfL: *strict rules, defeasible rules, defeaters and priority relations*. We base our implementation on a meta-program [9] to translate defeasible theories into logic programs and extended it to support priority relations *r1>r2 : overrides(r1,r2)* and conflict relations in order to define conflicting rules not just between positive and negative literals, but also between arbitrary conflicting literals. Example:

Rule1 "discount": All gold customers get 10 percent discount."
Rule2 "nodiscount": Customers who have not paid get no discount."
ContractLog DfL: … overrides(discount, nodiscount) … // rule 1 overrides rule 2

GCLP is based on concepts from DfL. It additionally implements a so called *Mutex* to handle arbitrary conflicting literals. We use DfL to handle conflicting and incomplete knowledge and GLCP for prioritisation of rules. A detailed formulation of our implementation can be found in [4].

Description Logics: Inspired by recent approaches to combine description logics and logic programming [10] we have implemented support for RDF/RDFS/OWL descriptions to be used in ContractLog rules. At the core of our approach is a mapping from RDF triples (constructed from RDF/XML files via a parser) to logical facts: RDF triple:*subject predicate object* → LP Fact: *predicate(subject, object)*, e.g.:

$a : C$, i.e., the individual *a* is an instance of the class *C*: *type(a,C)*

$< a, b >: P$, i.e., the individual *a* is related to the individual *b* via the property *P: property(P,a,b)*

On top of these facts we have implemented a rule-based inference layer and a class and instance mapping[4] to answer typical DL queries (RDFS and OWL Lite/DL inference) such as class-instance membership queries, class subsumption queries, class hierarchy queries etc. This enables reasoning over large scale DL ontologies and it provides access to ontological definitions for vocabulary primitives (e.g. properties, class variables and individual constants) to be used in LP rules. In addition to the existing Java type system, we allow domain independent logical objects in rules to be typed with external ontologies (taxonomical class hierarchies) represented in RDF, RDFS or OWL.

[4] to avoid backward-reasoning loops in the inference algorithms

4 RBSLA language

Real usage of a formal representation language which is usable by others than its inventors immediately makes rigorous demands on the syntax: declarative syntax, comprehension, readability and usability of the language by users, compact representation, exchangeability with other formats, means for serialization, tool support in writing and parsing rules etc. The **r**ule **b**ased **SLA** language (**RBSLA**) tries to address these issues. It stays close to the emerging XML-based Rule Markup language (RuleML) in order to address interoperability with other rule languages and tool support. Therefore, it adapts and extends RuleML to the needs of the SLA domain.

RuleML is a standardization initiative with the goal of creating an open, vendor neutral XML/RDF-based rule language. The initiative develops a modular specification and transformations via XSLT from and to other rule standards/systems. RuleML arranges rule types in a hierarchical structure comprising reaction rules, transformation rules, derivation rules, integrity constraints, facts and queries. 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. However, the latest version 0.88 is still mostly limited to deduction 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, complex event processing and state changes as well as normative concepts and violations to norms are missing; in order to pick up a few examples. As such improvements must be made. RBSLA therefore adds the following aspects to RuleML:

- *Typed Logic and Procedural Attachments*
- *External Data Integration*
- *Event Condition Action Rules with Sensing, Monitoring and Effecting*
- *(Situated) Update Primitives*
- *Complex Event Processing and State Changes (Fluents)*
- *Deontic Norms and Norm Violations and Exceptions*
- *Defeasible Rules and Rule Priorities*
- *Built-Ins, Aggregate and Compare Operators, Lists*
- *If-Then-Else-Syntax and SLA Domain Specific Syntax*

It is very important to note, that serialization of RBSLA in XML using RuleML does not require any new constructs, i.e., it can be done by using existing RuleML features. However, for the reason of brevity and readability RBSLA introduces apposite abbreviations for the prime constructs needed in SLA representation. A XSLT transformation might be applied normalizing the syntax into the usual RuleML syntax. We first present the abbreviated, compact RBSLA syntax and we will give an example of the normalization afterwards. Lack of space prevents us from giving an extensive description of the XML Schema representation of the RBSLA model. More details can be found in [11] and the latest version can be downloaded from our website [5]. Here, we blind out some details such as optional “oids” etc. and sometimes use a more compact DTD notation to present the concrete syntax.

Typed Logic: Logical terms (variables (Var), individuals (Ind) and complex terms (Cterm)) are either un-typed or typed. RBSLA supports the following type systems and data types: ***java:type , rdf:type , rdfs:type , owl:type , xsi:type , sql:type***.

Example:
```
<Var java:type="java.lang.Integer">1234</Var>
<Ind xsi:type="xs:nonNegativeInteger">12</Ind>
<Var rdfs: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, i.e. they are unified and evaluated against each other. We therefore extend the unification process so that the following rules 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 superclass 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 superclass, 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

Procedural Attachments: $O_m[p^1..p^n] \rightarrow [r^1..r^n]$. Here, O denotes an object or a class which is also an object, m denotes a method invocation, $p^1..p^n$ the parameters and $r^1..r^n$ the list of result objects which might also be a Boolean true or false value. The serialization in RuleML extends complex terms: Cterm(Ctor | Attachment). The first element of an ***<Attachment>*** is either a link on a qualified Java class, a variable bound to a Java object/class instance or a nested complex term which itself constructs/returns an object. The second element specifies the method call: Attachment((Ind|Var|Cterm), Ind). The parameters for the method invocation are the subsequent elements under the Cterm element, which are defined after an attachment. RBSLA supports Java (via reflection), e.g.: **java.lang.Integer$_{parseInt}$[1234]→Integer(1234)**

```
<Cterm java:type="java.lang.Integer">                    // (types are optional)
        <Attachment>
                <Ind java:type="java.lang.Class">java.lang.Integer</Ind>
                <Ind java:type="java.lang.reflect.Method">parseInt</Ind>
        </Attachment>
        <Ind java:type="java.lang.String">1234</Ind>
</Cterm>
```

The result of a method invocation finally replaces the complex term and is used in the further derivation process. Results can be bound to variables via the *Equal* element:

```
<Equal>
        <Var> Varible </Var>
        <Cterm> ... Attachment ... </Cterm>
</Equal>
```

External Data Integration: RBSLA supports a smooth integration of facts managed by external databases in particular SQL databases or XML/RDF files. It therefore provides different built-in predicates such as **<Location>** (database location with username and password), **<Select>** (SQL select query), **<XML>** (construct a DOM tree from a XML file) etc. to access and query the database in order to reuse the result as facts in a knowledge system.

Event Condition Action Rules: An ECA rule **<Eca>** is a rule with four terms: **<(Time?,Event?,Condition?,Action)>**. Each term defines a reference **<Ref>** on a derivation rule (or a fact) which implements the respective functionality of the ECA term. This offers maximum flexibility. Logical rule chaining between derivation rules facilitates the implementation of complex functionalities which can be referenced and reused in several ECA rules and procedural attachments might be applied for active sensing, monitoring and triggering actions. A reference is semantically interpreted as a RuleML query (conclusion less derivation rule). Therefore, the only semantics we add is that of the ECA paradigm, which executes the ECA rule in a forward directional manner, i.e., it proceeds with the next rule term iff the query on the currently referenced derivation rule succeeds.

Example:
```
<Eca>
        <Time><Ref><Ind>everyMinute</Ind></Ref></Time>
        <Event><Ref><Ind>serviceUnavailable</Ind></Ref></Event>
        <Condition><Ref><Ind>notMaintenance</Ind></Ref></Condition>
        <Action><Ref><Ind>sendNotification</Ind></Ref></Action>
 </Eca>

<Implies>                                   // referenced time derivation rule
        <Atom><Rel>... respective time function ... </Rel></Atom>     //body
        <Atom><Rel>everyMinute</Rel></Atom                            //head
</Implies>
<Implies>                                   // referenced event derivation rule
    [...]
```

Situated Update Primitives: Update primitives change the state of a knowledge system. RBSLA supports primitives to *add* (**<Assert>**) and *delete* (**<Retract>**; **<RetractAll>**) facts and rules. Typically, these primitives are applied in ECA rules (a la transaction logic).

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 (a la event calculus). Therefore, it defines the following elements:

- **Fluent:** <!Element Fluent(Ind|Var|Cterm)>
- **Parameters:** <!Element Parameter(Ind|Var|Cterm)>
- **Persistent Events:** <!Element Happens((Event|Action),Time)>
- **Believed/Planned Events:** <!Element Planned((Event|Action),Time)>
- **Effects of events:**
    - <!Element Initially(Fluent)>
    - <!Element Initiates((Event|Action), Fluent, Time)>
    - <!Element Terminates((Event|Action), Fluent, Time)>
- **Queries:**
    - <!Element HoldsAt(Fluent, Time)>
    - <!Element ValueAt(Parameter, Time, (Ind|Var|Cterm))>

Fluents or parameters might be used in addition to individuals, variables or complex terms in rules: (Ind|Var|Cterm|Plex|Fluent|Parameter). By default fluents do not hold, but can be defined as initially true.

Deontic Norms and Norm Violations: Deontic Norms such as obligations, permissions or prohibitions are defined as personalized, time-varying fluents: norm(S,O,A)

```
<Oblige><Ind>Subject</><Ind>Object</><Action><Ind>Action</></></Oblige>
<Permit><Ind>Subject</><Ind>Object</><Action><Ind>Action</></></Permit>
<Forbid><Ind>Subject</><Ind>Object</><Action><Ind>Action</></></Forbid>
<Waived><Ind>Subject</><Ind>Object</><Action><Ind>Action</></></Waived>
```

RBSLA defines a special violation event which happens if a norm is violated, e.g. an obligation which is not fulfilled in time: **<Violation>**<Ind>*Violation*</>**</Violation>**

Defeasible Rules and Rule Priorities: Beside strict rules which a represented as normal derivation rules "*head* → *body*" (<Implies>) RBSLA supports defeasible rules "*body* => *head*" and therefore introduces the new rule element **<Defeasible>**. Although, incompatible and conflicting literals between rules in general can be expressed as specializations of RuleML integrity constraints, RBSLA introduces a new **<Mutex>** element (for mutually exclusive, derived von GCLP), e.g.:

```
<Mutex>
        <Atom>   <Rel>discount</Rel> <Var>X</Var>       </Atom>
        <Atom>   <Rel>discount</Rel> <Var>Y</Var>       </Atom>
        <Atom><Cond>
                <Neg><Equal> <Var>X</Var> <Var>Y</Var> </Equal></Neg>
        </Cond></Atom>
</Mutex>
```

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

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

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>**: Given an input variable, test whether it is instantiated or not
- **<Type>**: Given an input variable, return the type name.

Numbers:
- **<Add|Sub|Mult|Div|Mod|Max|Min|Abs>**: Compute two numeric values and return the result.

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

Date and Time:
- **<Date> <Ind>Year</><Ind>Month</><Ind>Day</> </Date>**
- **<Time><Ind>Hour</>, <Ind>Min</>, <Ind>Sec</> </Time>**
- **<DateTime> ((<Date>,<Time>) | (<Ind> Epoch Value in millis </Ind>))</DateTime>**
- **<TimeSpan>, <Intervall>**
- **<Compare>, <Less>, <Equal>, <More>, <Add>, <Sub>** etc.

Lists:

- **<Plex><Var1/><Var2/>…<VarN/></Plex>**: List of the form [Var1 .. VarN]
- **<Plex><Var>Head</Var><repo><Var>Rest</Var></repo></Plex>:** List of the form [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. The result is the concatenated 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 and return the result.

Comparison:

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

If-Then-Else-Syntax and SLA Domain Specific Syntax: In order to make programming in RBSLA and specification of SLAs more efficient and easier, RBSLA provides an additional If-Then-Else Syntax for rules:

<If> <Atom> ... Body ... </Atom> **</If>**
<Then> <Atom>Head</Atom> **</Then>**
<Else> <Atom>Head of corresponding Naf rule </Atom> **</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. RBSLA provides direct serialization of reusable SLA metrics **<Metric>**. SLA metrics are used to measure the performance characteristics. They are either retrieved directly from the managed resources such as servers, middleware or instrumented applications or are created by aggregating such *direct metrics* into higher level *composite metrics.* Typical examples of direct metrics are the MIB variables of the IETF SMI such as *number of invocations, system uptime, outage period* etc. which are collected via measurement directives such as management interfaces, protocol messages, URIs etc. Composite metrics use a specific function averaging one or more metrics over a specific amount of time, e.g. *average availability*, or breaking them down according to certain criteria, e.g. *minimum throughput, maximum response time,* etc. Direct metrics contain measurement directives implemented as attachments:

```
<Metric type="rbsla#DirectMetric">
    <Rel><Ind>Metric Name</Ind></Rel>
    <Cterm><Attachment> [...] </Attachment> </Cterm>
</Metric>
```

Composite metrics use the built-in RBSLA aggregate operators over list structures which temporarily store the measurement results of direct metrics or they shift these expensive computations to highly optimized query languages such as SQL. Metrics might be (re)used in rules, e.g. in the event declaration of an ECA rule:

```
<Event><Ref><Ind>serviceUnavailable</Ind></Ref></Event>
<Implies>                                   // referenced rule
    <Less><Metric>…AverageAvailability …</Metric><Ind >0.98</Ind></Less>
    <Atom><Rel>serviceUnavailable</Rel></Atom>
</Implies>
```

As mentioned before the presented RBSLA syntax can be normalized to the usual RuleML syntax via XSLT, for example:

```
<Oblige>
        <Ind>Service Provider</Ind>
        <Ind>Service Consumer</Ind>
        <Action><Ind>payPenalty</Ind></Atom>
</Oblige>
```

Maps to:

```
<Cterm rdfs:type="rbsla#Obligation">                    (type are optional)
        <Ctor>oblige</Ctor>
        <Ind rdfs:type="rbsla#Provider">Service Provider</Ind>
        <Ind rdfs:type="rbsla#Consumer">Service Consumer</Ind>
        <Ind rds:type="rbsla#Action>payPenalty</Ind>
</Cterm>
```

RBSLA comprises an additional ontology (based on RDFS) which describes the SLA domain vocabulary and can be used to semantically annotate and type used objects. This restricts search space to clauses where the type restrictions are fulfilled and makes the derivation process more efficient.

We have implemented a transformation process based on Java which maps the RBSLA rules into the basic logical ContractLog rules (Prova/Prolog syntax). During the transformation process we apply automated "refactorings" on the rules in order to improve the execution efficiency: loops, order of rules / order of prerequisites in rules might matter or performance critical issues like extensive use of negation as failure or cuts might be applied. For example we do some narrowing on multiple rules which share the same set of prerequisites in order to reduce redundancies ($A_1,..,A_N \rightarrow B$ and $A_1,..,A_N \rightarrow C$ becomes $A_1,.., A_N \rightarrow A$; $A \rightarrow B$; $A \rightarrow C$). Other examples are removing disjunctions in the prerequisites (replacing $A \vee B \rightarrow C$ by two new rules $A \rightarrow C$ and $B \rightarrow C$), remove conjunctions from rule heads (transform $B \rightarrow (H \wedge H')$ via Lloyd-Topor transformation into two rules $B \rightarrow H$ and $B \rightarrow H'$), remove function symbols from rule heads or reducing typing information in rule chains etc.

To sum up the additional RBSLA layer solves the demands stated at the beginning of this section, in particular compatibility with other languages via transformations and a declarative, compact and readable rule representation. Additionally it provides means for refactoring and validation of rules during transformation into ContractLog.

5 RBSLM tool

The RBSLM tool splits into the **Contract Manager** (CM) and the **Service Dashboard** (SD). The CM is used to manage, write, maintain and update SLA rules. The SD visualizes the monitoring and enforcement process in the contract life cycle and supports further SLM processes. We first describe the CM.

The CM provides a basic editor for ContractLog rules and a repository approach which represents the structure and meta-data of a knowledge base. It supports two roles: the *business practitioner* and the *rule experts*. The rule experts have a background in logic and are therefore responsible for the basic design of contract rules. They fill the repository with needed and reusable domain specific measurement, monitoring and computing functions, interface implementations to existing databases or system tools, references on existing business objects (e.g. EJBs) as well as rule templates and other domain specific concepts (e.g. contract vocabularies). Addition-

ally, they specify test cases together with certain test data to be used for verification and validation of contract rules in order to ensure the correct usage and a high-quality of SLA rule sets [1]. The business practitioners are involved in the daily business. They make use of the predefined templates to write and maintain the contract rules. They do not need to know any implementation details of used functions, objects or interfaces nor do they need to have a complete overview of all the rules in the contract system, meaning they do not know what the effect is on existing rules when a rule is added or changed. They just use the GUI to adapt the templates and build rules by clicking together the needed functionality. The test cases safeguard the authoring of rules and allow validation of complete rule sets and contracts to detect anomalies like inconsistencies, incompleteness or redundancies referring to the intended goals.

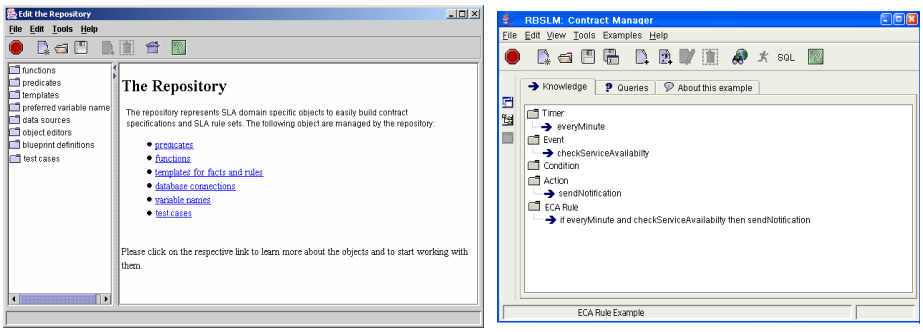

Fig. 3. The ContractManager GUI

Whilst the Contract Manager is used to manage the SLAs and their rules, the Service Dashboard visualizes status information and metrics during the monitoring and execution process. We provide different and adaptable visualisation views in order to satisfy the needs of different user roles e.g. for the customer advisor to face customer complains and problems, for the accountancy to forecast fees and recourse receivables and for the administrator to give detailed information to fix arising problems. Similar to the process of deriving quality metrics from base data to verifying contractual rules, the logic engine can also be used to get meaningful quantities for each of these parties. Predefined parsers and chart formations enable the user to present the information in an adequate way. New user views can be easily plugged into the framework dynamically through the class loader. The underlying Model-View-Controller notifies these views if data are changed in the knowledge base (observer pattern) and triggers the visualization process.

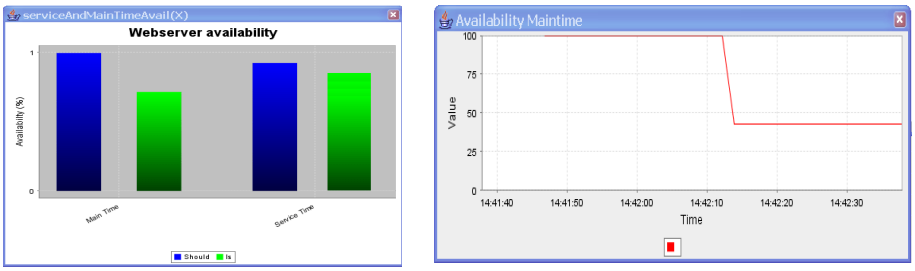

Fig. 4. Service Dashboard with different visualization views

6 Use Case

We now want to illustrate the SLA representation, monitoring and enforcement process. We therefore use the following example agreement:

*The service availability will be measured every t_{check} by a ping on the service. If the service is unavailable, the SP is **obliged** to restore it within $t_{deadline}$. If SP fails to restore the service in $t_{deadline}$ (violation), SC is **permitted** to cancel the contract.*"

RBSLA representation

```
<Eca>                                                    // ECA rule monitoring the service availability
      <Time><Ref><Ind> tcheck</Ind></Ref></Time>
      <Event><Ref><Ind>unavailable</Ind></Ref></Time>
      <Action><Ref><Ind>assertEvent</Ind></Ref></Action>
</Eca>

<Rule>                                                   // referenced time derivation rule
      <If>  <Intervall>… [monitoring schedule] … </Intervall>  </If>   // body
      <Then> <Atom><Rel> tcheck</Rel></Atom>  </Then>     // head
</Rule>

<Rule>                                                   // referenced event derivation rule
      <If>  <Metric>… [test availability] ... </Metric>   </If>   // body
      <Then> <Atom><Rel> unavailable</Rel></Atom>  </Then>     // head
</Rule>

                  […]

<Initiates>                                              // unavailable event initiates primary obligation
          <Event><Ind>unavailable</Ind></Event>
          <Oblige> … [obligation norm] … </Oblige>
</Initiates>
                  […]
<Rule>                                                   // if deadline elapsed then raise violation event
        <If><ValueAt><Parameter>deadline</Parameter><Time> tdeadline</Time><Ind>0</Ind></ValueAt></If>
        <Then><Assert><Violation><Ind>elapsed</Ind></Violation></Then>
</Rule>

<Initiates>                                              // violation event initiates reparation permission norm
          <Violation></Ind>elapsed</Ind></Violation>
            <Permit> … [cancel contract]  </Permit>
</Initiates>
                  […]
```

ContractLog representation

```
eca(everyTcheck ,  serviceUnavailable , assertUnavailable)          // ECA rule
Time:    everyTcheck (DT) ← < interval function for  tcheck,>      // referenced derivation rule
Event:   serviceUnavailable(DT) ← not ping(service)                // referenced derivation rules
Action:  assertUnavailable(DT) ← assert(happens(unavailable,T)     // referenced derivation rules

initiates(unavailable, oblige(SP, Service, start()),T)     // defines the primary obligation initiated by an certain event
 terminates(available, oblige(SP, Service, start()),T)  //  defines the event which normally terminates the obligation
trajectory(oblige(SP,Service,start()),T1,deadline,T2,(T2 - T1))  // defines the period in which the norm must be fulfilled
happens(elapsed,T) ← valueAt(deadline,T, tdeadline)        // defines the violation of the obligation norm
initiates(elapsed, permit(SC, Contract, cancel()),T)  // initiates the derived permission to cancel the contract
```

The RBSLA representation is translated via the RBSLA compiler (transformation) into the ContractLog representation. The Prova engine (based on Mandarax rule engine) together with the ContractLog extensions execute and monitor the contract.

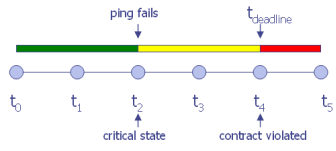

Fig. 5. Contract tracking

We assume that the service becomes unavailable at time t_2 and is restarted by the service provider at time t_5 which is after $t_{deadline}$. The ECA meta interpreter of the ContractLog framework monitors the ECA rule. Every t = t_{check} it pings the service via a procedural attachment and asserts the respective event to the

knowledgebase if the service is unavailable. This leads to the conclusions illustrated in fig. 5. The derived status information can be used to feed periodical reports, enforce rights and obligations or visualize monitoring results on quality aspects in the Service Dashboard.

7 Conclusion and Key Findings

In this paper we have described our declarative rule based approach to SLA representation and management. We have given an insight into the logical core, the ContractLog framework which underpins the declarative RBSLA language. Based on this we have implemented a prototypical rule based service level management tool (RBSLM). In contrast to conventional pure procedural programming approaches our declarative logic based approach simplifies maintenance, management and execution of SLA rules and allows easy combination and revision of rule sets to build sophisticated and graduated contract agreements, which are more suitable in a dynamic service oriented environment than the actually used, simplified rules. Although the framework described in this paper is still a proof of concept implementation, we have attempted to gather some data on its performance and usability. The important reasoning task in SLA monitoring and enforcement is query answering which is known to be only semi-decidable. However, the combination of highly optimized OO programming and database techniques as well as the use of adequate logical formalism implemented on the basis of horn logic and meta-programming techniques makes possible the high efficiency of the framework although it provides rich expressiveness. Usability analyses and qualitative comparisons with other representation approaches such as WSLA have revealed the higher flexibility and automation of our rule based approach.

References

1. Dietrich, J. and A. Paschke. *On the Test-Driven Development and Validation of Business Rules*. in *ISTA05, Massey, New Zealand*. 2005.
2. Paschke, A. and M. Bichler. *SLA Representation, Management and Enforcement - Combining Event Calculus, Deontic Logic, Horn Logic and Event Condition Action Rules*. in *EEE05*. 2005. Hong Kong, China.
3. Paschke, A., M. Bichler, and J. Dietrich. *ContractLog: An Approach to Rule Based Monitoring and Execution of Service Level Agreements*. in *RuleML 2005*. Galway, Ireland.
4. Paschke, A., *ContractLog - A Logic Framework for SLA Representation, Management and Enforcement*. 7/2004, IBIS, TUM, Technical Report.
5. Paschke, A., *RBSLA: Rule-based SLA, http://ibis.in.tum.de/staff/paschke/rbsla/index.htm*..
6. Paschke, A., *Rule Based SLA Management - A rule based approach on automated IT service management (in german language)*. 6/2004, IBIS, TUM, Working Paper.
7. Kowalski, R.A. and M.J. Sergot, *A logic-based calculus of events*. New Generation Computing, 1986. 4: p. 67-95.
8. Nute, D., *Defeasible Logic*, in *Handbook of Logic in Artificial Intelligence and Logic Programming Vol. 3*, D.M. Gabbay, C.J. Hogger, and J.A. Robinson, Editors. 1994, Oxford University Press.
9. Antoniou, G., et al. *A flexible framework for defeasible logics*. in *AAAI-2000*. 2000.
10. Grosof, B.N., et al. *Description Logic Programs: Combining Logic Programs with Description Logic*. in *WWW03*. 2003: ACM.
11. Paschke, A., *RBSLA: A Rule Based Service Level Agreements Language*. 8/2004, IBIS, TUM, Technical Report.