

# SLA Representation, Management and Enforcement

Adrian Paschke, Martin Bichler

*Internet-based Information Systems, Technical University Munich*  
[paschke, bichler]@in.tum.de

## Abstract

As the IT industry is becoming more and more interested in service oriented business models and upcoming technologies like Web Services or Grid Computing, the need for automated contract management increases accordingly. In this paper we evolve a formal representation for contractual agreements together with their monitoring and enforcement by standard components of logic programming. We incorporate different logical formalisms like Horn Logic, Event Calculus, Deontic Logic and ECA rules into one logical framework to execute complex contract rules on an individual or a group level together with their normative relationships (permissions, obligations and prohibitions). This logical framework supports a high level architecture for the automation of electronic contracts capable of representing complex business rules and business policies, detecting contract violations, authorisation control, conflict detection, service billing, reporting and other contract enforcement processes.

## 1. Rule-based Service Level Agreements

For enterprises which outsource IT-services it is important to receive the services in the agreed quality and quantity, in particular, if the core business depends directly on the services. Therefore, clear agreements regarding main service obligations and warranties as well as service quality and sanction regulations are required, if these standards can not be met. For this purpose, Service Level Agreements (SLAs) between the service provider and the service consumer are contracted. Nowadays, contractual agreements and their implicit rules are typically defined in natural language, and contract monitoring is done manually to a large degree. Recent system management tools (e.g. IBM Tivoli Service Level Advisor, Remedy Service Level Agreements etc.) try to execute contract rules in procedural application logic (implemented in Java, C++ etc.). However, this approach is restricted to the implemented application features on IT infrastructure level and therefore not very flexible to efficiently represent, manage and enforce diverse contract

rules in a dynamic environment. We propose representing contracts by different logical formalisms, which are combined in one logical framework – the **ContractLog**. We make use of a standard component of logic programming (a rule engine) to monitor and enforce the contracts. The essential advantages of our approach are:

1. Formal representation of contracts in reusable, dynamically extensible and coexisting contract modules on different business/operational and individual/group level.
2. Contract rules are separated from the service management application and allow easier maintenance and management.
3. Rules can be automatically linked (a.k.a. rule chaining) and executed by standard components of the logic programming in order to automate complex business policies and individual contractual agreements.
4. (Proactive) detection of contract violations and automatic reaction by event processing rules (ECA) and further logical formalisms for reasoning about time, events and their effects on the contract state.
5. (Proactive) verification of rule sets and detection of rule conflicts as well as (automatic) conflict resolution.
6. Contract enforcement based on contractual norms relating to permissions, prohibitions, obligations and event-triggered actions.
7. Integration of semantically defined domain models, such as contract ontologies or role/system models etc.

The main contribution of this paper, is the development of adequate knowledge representation concepts to describe SLAs with complex contract rules and a prototypical implementation which supports a high level architecture – a rule-based Service Level Management System (**RBSLM**) for the automation of electronic contracts capable of representing complex business rules and business policies, detection of contract violations, authorisation control, conflict detection, service billing, reporting and other contract enforcement processes.

The subsequent paper is structured as follows: In section 2 we introduce the logical formalisms on which we base our ContractLog framework for contract representation. In section 3 we work out the main concepts of our approach and their implementation. In section 4 we will introduce the logic programming tool - the rule engine Mandarax -which we used for executing our logical framework. Finally in section 5 and 6 we will discuss related work and will provide a conclusion.

## 2. Logic Formalisms

Before we evolve our logical ContractLog framework in section 3 we first will introduce the main concepts of the core logical formalisms which we use:

### Horn Logic and Derivation Rules

In contrast to simple logics, like Propositional Logic or Boolean logic, which are not expressive enough to represent most contract rules, First Order Logic (FOL) is to an large degree epistemologically adequate (~ expressive enough), because of rule chaining, quantifiers etc. However, this expressiveness is paid with low computational tractability. Therefore, horn logic - a computational tractable subpart of FOL – is used in logic programming to express derivation rules (a.k.a. horn clauses). Derivation rules derive knowledge from existing (facts) or other derived knowledge (other rules) usually by a backward-reasoning (query-driven) inference process (based on unification and resolution). [1] A example is a discount rule, such as “A customer gets a discount of  $p_{gold}$  if he is a gold customer”, which can be written as a derivation rule  $discount(Customer, p_{gold}) \leftarrow gold(Customer)$ . Rule chaining allows formulating a complex discount policy by several rules, e.g.:

$$\begin{aligned} discount(Customer, p_{gold}/p_{silver}/p_{bronze}) &\leftarrow gold/silver/bronze(Customer) \\ gold/silver/bronze(Customer) &\leftarrow \\ &spending(Customer, >5000\$/3000\$/500\$, last\ year) \end{aligned}$$

### Deontic Logic

Deontic Logic (DL) studies the logic of normative concepts, such as obligation, permission and prohibition. The first attempt to build a formal theory was made by Mally 1926, but most of the contemporary interest in DL has been stimulated by Wright’s paper “Deontic Logic” [2], who proposed a first axiomatization for deontic logic:  $O\emptyset$ (obliged),  $P\emptyset$ (permitted),  $F\emptyset$ (forbidden),  $W\emptyset$ (waived). Various further formulations of DL have been proposed. Follesdal and Hilpinen [3] presented an approach based on propositional logic, what they call the standard system of deontic logic, which includes a core set of deontic axioms. 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*. Another limitation is the inability to express *personalized statements*. In real world applications deontic norms refer to an explicit concept of an agent. These limitations make it difficult to satisfy the needs of practical contract management. Therefore, we will extend SDL with a role-based model and further logic formalisms which overcome the described limitations (cf. section 3.3).

### Classical Logic Event Calculus

Kowalski and Sergot’s Event Calculus (EC) [4] is a formalism for reasoning about events and their effects in

a logic programming framework. It 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:

$happens(E, T)$	<i>event E happens at time point T</i>
$initiates(E, F, T)$	<i>event E initiates fluent F for all time &gt; T</i>
$terminates(E, F, T)$	<i>event E terminates fluent F for all time &gt; T</i>
$holdsAt(F, T)$	<i>fluent F holds at time point T</i>

We restrict our attention to classical logic formulations of the Event Calculus, where theories are written using Horn Clauses [5]. However we will make some extensions to the core set of axioms (cf. section 3.2 and 3.3).

## 3. Contract Representation: Combining Horn Logic, ECA-Rules, Event Calculus and Deontic Logic

Derivation rules are suitable to represent complex business rules and differentiated business policies, e.g. because of the possibility of rule-chaining and automatic deduction of query-driven inferences based on business data. First of all we represent contract rules by derivation rules. [cf. 1] However, we need further logical formalisms to express and reason about events, actions and their effects on the contractual state and contract norms. In the next two sections (3.1 and 3.2) we evolve an extended logical framework which incorporates further suitable logical formalisms. In section 3.3 we extend our logical framework with concepts for enforcing role-based contract norms, authorisations and conflict detection. Finally in section 3.4 we introduce abduction as a mean for reaching proactivity. Because of the limited space we can not present all details of our logical framework. However, more information and more extensive examples can be found in [6] and on our project site [7]

### 3.1. Events and Actions – ECA Rules

Many rules in SLAs contain event-triggered action statements, such as: “If **event** (E) happens and **conditions** ( $C_1..C_2$ ) hold then trigger **action** (A)”. This type of rule is called an Event-Condition-Action (ECA) rule. In SLAs additional measurement directives specify how often an event is measured in order to reduce monitoring costs for event detection. To represent this we extend the ECA rule with a timer part:

“Whenever **timer** elapsed and **event** (E) happened and **conditions** ( $C_1..C_2$ ) hold then trigger **action** (A).”

This general ECA rule has an active forward-reasoning push-style, which means that the action is triggered when

timer -, event - and condition clauses hold. This can be very effective for simple event rules, and some popular rule engines like Jess, Clips or ILog use forward-reasoning, mostly based on the RETE algorithm. However, as we argued elsewhere [cf. 1] using query-driven backward-reasoning in the context of contract management and SLA monitoring has some advantages. In particular, the integration of arbitrary data sources such as e.g. customer or system databases can be handled much more effectively and with less memory consumption. Therefore, we use backward-reasoning and simulate the active behaviour in an active ECA meta-interpreter simulation based on the following derivation rules:

**ECA Rule:**

$eca(R) \leftarrow timer(N, T) \wedge event(E) \wedge con(C) \wedge action(A)$

**ECA Sub-Rules:**

$timer(N, T) \leftarrow time(T) \wedge \langle time\ function \rangle$  holds at a specific time  
 $event(E) \leftarrow \langle check\ event(s) \rangle$  event rule  
 $cond(C) \leftarrow \langle conditions \rangle$  condition rule  
 $action(A) \leftarrow \langle trigger\ action(s) \rangle$  action rule

**ECA Query:**

$eca(Rule)?$  (repeated by daemon)

A thread-based daemon process ensures the active behaviour of our ECA interpreter by frequently issuing the query “**eca(Rule)?**” (e.g. every second). That starts the verification of all ECA rules and the associated sub-rules. The rule engine will prove the leftmost rule clause in the rule body first and only proceed to the next clause if it succeeds. We have implemented an additional thread concept [cf. 6] which allows concurrency in proving the event and condition part of an ECA rule in an own thread, because this may take some time (e.g. think of accessing internet resources). A small example:

“If the service is unavailable then send a notification to the service provider. The service availability will be measured every minute by a service ping.”

$timer(minute, T) \leftarrow time(T) \wedge second(T) \bmod 60 = 0$  (~ every minute)  
 $event(unavailable) \leftarrow \neg ping(service)$   
 $action(notify) \leftarrow sendMessage(SP, "service\ unavailable")$   
 $eca(check) \leftarrow timer(minute, T) \wedge event(unavailable) \wedge action(notify)$   
Query:  $eca(Rule)?$  (Repeated by daemon)

Event detection and action triggering often requires procedural attachments [8] – procedure calls to a programming language, e.g. Java. The used rule engine supports this (see section 4). In the above example *ping()* and *sendMessage()* are procedural attachments.

**3.2. Integrating the Event Calculus**

A limitation of the described ECA concept is its inability to describe the effects of events and actions on the contract state as well as to derive the contract state

after an event happened. Therefore, we combine the Event Calculus with our general ECA rule concept and model the contract state and other changeable properties (e.g. deontic contract norms) as time-varying fluents:

$event(E, T) \leftarrow \langle check\ event(s) \rangle \wedge assert(happens(E, T))$   
 $action(A, T) \leftarrow \langle trigger\ action(s) \rangle \wedge assert(happens(A, T))$   
 $eca(R) \leftarrow timer(N, T) \wedge event(E, T) \wedge con(C) \wedge action(A, T)$   
 $initiates/terminates(E, F, T)$   
 $initiates/terminates(A, F, T)$

The special predicate *assert()* is used to insert knowledge about detected or triggered events and actions into the knowledgebase. We use the EC extension to describe event-based changes on deontic contract norms, such as obligation, permission or prohibition. A deontic norm consists of the normative concept (*norm*), 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 deontic norm as a fluent:  $norm(S, O, A)$  and write a corresponding rule as  $initiates/terminates(E, norm(S, O, A), T)$

Table 1 exemplarily shows the implementations of some typical SLA rules with our combined ECA/EC-rule concept in order to elucidate our approach and to illustrate its suitability for the representation of typical service contracts. More examples can be found in [6].

**Tab 1: Example rule implementation**

<p>The service provider SP will provide a storage service with a capacity of <i>c</i> GBytes to the service consumer SC. SC is <b>permitted</b> to use the service operations store, If the storage usage in GBytes <math>u &gt; c + t_{max}</math>, where <math>t_{max}</math> is the level of tolerance, the SC is <b>forbidden</b> to use the “store”-operation of the service until <math>u \leq c</math>.</p>
<p><math>initially(permit(SC, Service, store()))</math>  <math>event(exceeded, T) \leftarrow (getUsage(SC) &gt; (c + t_{max})) \wedge assert(happens(exceeded, T))</math>  <math>eca(exceeded) \leftarrow timer(everyday, T) \wedge event(exceeded, T)</math>  <math>initiates(exceeded, forbid(SC, Service, store()), T)</math>  <math>terminates(exceeded, permit(SC, Service, store()), T)</math></p>
<p>The service availability will be measured every <math>t_{check}</math> by a service ping. If the service is unavailable, the SP is <b>obliged</b> to restore it within <math>t_{deadline}</math>. If SP fails to restore the service in <math>t_{deadline}</math>, SC is <b>permitted</b> to cancel the contract.</p>
<p><math>event(available, T) \leftarrow \neg ping(Service) \wedge assert(happens(unavailable, T))</math>  <math>event(available, T) \leftarrow ping(Service) \wedge assert(happens(available, T))</math>  <math>eca(unavailable) \leftarrow timer(t_{check}, T) \wedge event(available) \wedge \dots</math>  <math>initiates(unavailable, oblige(SP, Service, start()), T)</math>  <math>terminates(available, oblige(SP, Service, start()), T)</math>  <math>trajectory(oblige(SP, Service, start()), T1, deadline, T2, (T2 - T1))</math>  <math>happens(elapsed, T) \leftarrow valueAt(deadline, T, t_{deadline})</math>  <math>initiates(elapsed, permit(SC, Contract, cancel()), T)</math></p>

In the second rule a deadline  $t_{deadline}$  for the obligation can be found. We make use of *trajectories* and *parameters* [9] to model this as a kind of delayed effect. We distinguish between the already introduced “normal” EC fluents, which hold over intervals of time with non-zero duration, and *parameters*, which hold instantaneously. Additionally, we introduce the predicate *trajectory(F, T1, P, T2, X)*, which states that if fluent *F* is initiated at Time *T1* and continues to hold until time *T2*

then the parameter P has a value of X at time T2. We translate this into Event Calculus terms by a special extra domain independent axiom  $valueAt(P, T2, X)$ :

$$valueAt(P, T2, X) \leftarrow initiates(E, F, T1) \wedge happens(E, T1) \wedge T2 > T1 \\ \wedge trajectory(F, T1, P, T2, X) \wedge \neg clipped(T1, F, T2)$$

Therewith, we can express that the SC is permitted to cancel the contract if the SP fails to restore it in the deadline. Additionally, we can use trajectories and parameters for counting service usage times and other time-related measurements and to represent typical metering and accounting rules.

### 3.3. Norm Enforcement and Conflict Detection

#### Obligation Fulfilment

In order to fulfil an obligation the obliged action must be executed. This can be done automatically by an ECA rule “ $eca() \leftarrow \dots holdsAt(obligation(S, O, A), T) \wedge action(A) \dots$ ” or manually by an external agent. As soon as the action is performed ( $happens(A, T)$ ), which can be detected by an ECA rule, the obligation norm is terminated ( $terminates(A, obligation(S, O, A), T)$ ).

#### Authorisation with Permissions and Prohibitions

With the norms *permit* and *forbid* we can implement an authorisation control. We distinguish positive authorisation which by default forbids all actions and a negative authorisation which by default allows all actions and introduce a new predicate  $request(S, O, A, T)$  which holds if the subject (S) has a permission to do action (A) on the object (O) at time (T).

##### Positive Authorisation:

$$request(S, O, A, T) \leftarrow holdsAt(permit(S, O, A), T)$$

##### Negative Authorisation:

$$request(S, O, A, T) \leftarrow \neg holdsAt(forbid(S, O, A), T)$$

We use the request predicate to verify the authorisation before an action is performed, e.g. in an ECA rule or externally by asking a query  $request(S, O, A, T)$ ?

#### Group-Level and Role-based Deontic Norms

In the example rules (cf. table 1) we have used the undefined terms service provider (SP) or service consumer (SC).

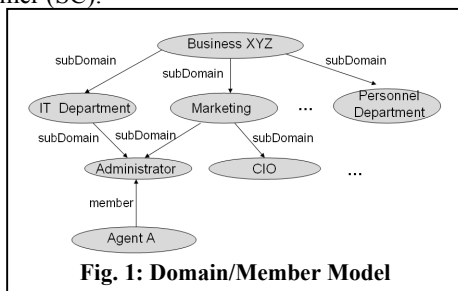


Fig. 1: Domain/Member Model

For practical contract management we need more detailed information. Therefore, we implement a general domain model as a taxonomy written in RDF. The model is structured in *(Sub)-Domains* and associated *Members* connected via the relations *subDomain* and *member*. A domain is a closed entity, e.g. a company, a department, a role etc. We use the RDF extension of the rule engine (cf. section 4) to process RDF statements as grounded facts of the kind:

$$subDomain(Domain, Sub-Domain) \\ member(Member, Domain)$$

Based on this we build the following axioms to reason over the domain model:

$$domain(Domain) \leftarrow subDomain(Domain, Sub-Domain) \\ member(Member) \leftarrow member(Member, Domain) \\ isMember(Member, Domain) \leftarrow member(Member, Domain) \\ isDerivedMember(Member, Domain) \leftarrow isMember(Member, Domain) \\ isDerivedMember(Member, Domain) \leftarrow isSubDomain(Domain, Sub-Domain) \wedge isDerivedMember(Member, Sub-Domain) \\ isSubDomain(Domain1, Domain2) \leftarrow subDomain(Domain1, Domain2) \\ \wedge Domain1 \neq Domain2 \wedge \neg isSubDomain(Domain2, Domain1)$$

To query all direct members of a domain we use  $isMember(Member, Domain)$  and to get all derived members of a domain we use  $isDerivedMember(Member, Domain)$ . With these axioms and the domain model we can verify whether the requesting agent is a member of this domain:

$$request(S, O, A, T) \leftarrow holdsAt(permit(D, O, A), T) \wedge isDerivedMember(S, D)$$

#### Including Deontic Logic

We use  $O\emptyset$ ,  $P\emptyset$ ,  $F\emptyset$ ,  $W\emptyset$  modelled as personalized fluent and additionally include typical SDL axioms into our framework in terms of Event Calculus axioms, e.g.:  
 $O\emptyset \rightarrow P\emptyset : holdsAt(permit(S, O, A), T) \leftarrow holdsAt(oblige(S, O, A), T)$   
 $F\emptyset \rightarrow W\emptyset : holdsAt(waived(S, O, A), T) \leftarrow holdsAt(forbid(S, O, A), T)$   
 Since deontic norms such as *F* and *P* can be opposite to each other conflicts can arise. Therefore, it is important to provide a possibility to detect conflicts and to resolve them automatically if possible. Typical types of conflicts which can be found in literature are *deontic paradoxes* (e.g. exception conflicts or contrary-to-duty conflicts) *deontic conflicts* (a.k.a. deontic dilemmas or modality conflicts) and *application specific conflicts* like conflicts of duty, conflicts of interest, management conflicts, resource conflicts etc. [10]

#### Deontic Conflicts and Paradoxes

Deontic conflicts are triggered by contradicting norms. We distinguish *authorization conflicts* and *obligation conflicts*. Authorization conflicts arise if both norms forbid and permit hold at the same time for the same action, subject and target. Obligation conflicts arise if the norms forbid and oblige hold at the same time. We model both conflicts as the derived fluents *authConflict* and *obligConflict*. We distinguish *derived fluents* from

the already known primitive fluents ( $\sim$ normal EC fluents). Therefore, we introduce a new predicate *derivedFluent(F)* which describes the derived fluents and alter the basic holdsAt axioms to:

$$\text{holdsAt}(F, T) \leftarrow \dots \wedge \neg \text{derivedFluent}(F) \wedge \dots$$

$$\text{derivedFluent}(\text{authConflict}) \quad \text{derivedFluent}(\text{obligConflict})$$

Now, we can define authConflict and obligConflict as derived fluents and implement the conflict detection rules:

Authorization Conflict: Permit & Forbid

$$\text{holdsAt}(\text{authConflict}(S, O, A), T) \leftarrow \text{holdsAt}(\text{permit}(S, O, A), T)$$

$$\wedge \text{holdsAt}(\text{forbid}(S, O, A), T)$$

Obligation Conflict: Oblige & Forbid

$$\text{holdsAt}(\text{obligConflict}(S, O, A), T) \leftarrow \text{holdsAt}(\text{oblig}(S, O, A), T)$$

$$\wedge \text{holdsAt}(\text{forbid}(S, O, A), T)$$

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 ( $OA \wedge \neg A$ ) and a **contrary-of-duty (CTD) obligation** ( $OA \wedge \neg A \rightarrow OB$ ) 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** ( $OA$ ) 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 initiate 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.

### Application Specific Conflicts

One of the most common types of application specific conflicts cited in literature are conflicts of duty, which arise if the same agent is permitted to perform two actions that in the context of their domain are defined to be conflicting. Other application specific conflicts are conflicts of interest, resource conflicts or management conflicts. These conflicts arise in diverse situations and the corresponding conflict detection rules can take a number of different forms, e.g.:

$$\text{derivedFluent}(\text{applicConflict})$$

$$\text{holdsAt}(\text{applicConflict}(S1, O1, A1, S2, O2, A2), T) \leftarrow$$

$$\text{holdsAt}(\text{permit}(S1, O1, A1), T1) \wedge \text{holdsAt}(\text{permit}(S2, O2, A2), T2)$$

However, we have not found a general rule concept for all application specific conflicts, because of their specific nature - e.g. in some domains, concurrently or

sequential performed actions may cancel each others' effects and may combine to cause effects which none of the actions performed in isolation would achieve.

### Conflict Resolution

Using one of the fluents *authConflict*, *obligConflict* or *appConflict* conflicts can be detected. In some cases conflicts can be automatically avoided or resolved. This is called a mild dilemma. For instance, in case of an authorisation conflict we might terminate the permission: *terminates*( $E, \text{permit}(S, O, A), T$ )  $\leftarrow \text{holdsAt}(\text{authConflict}(S, O, A), T)$  or reject the request:

$$\text{request}(S, O, A, T) \leftarrow \neg \text{holdsAt}(\text{authConflict}(S, O, A), T)$$

$$\wedge \text{holdsAt}(\text{permit}(S, O, A), T)$$

In other cases like with obligation conflicts or with some application specific conflicts we can not resolve the conflict, which is therefore called a deadlock dilemma. In this case we need human interaction, e.g. a revocation of the obligation or an automated conflict handling e.g. with rule priorities. To define priorities for conflicting rules we make use of concepts from defeasible and default logics. We will describe our general approach to rule prioritization which is generally applicable for conflicting rules in a separate paper.

### 3.4. Proactivity – Integrating Abductive Logic

Up to now, we have used deductive logic. However, proactivity needs knowledge about the future. Therefore, we make use of abductive logic. Abduction is commonly defined as the problem of finding a set of hypotheses ( $\sim$  a plan) of a specified form that, when added to a given formal specification, allows a goal state to be inferred, without causing contradictions. For us this means to determine the sequences of events that need to occur such that a violation used as goal state happens. We extended our current logical framework using the Event Calculus-based abductive proof procedure presented in [11] which reduces the computational complexity to two timepoints – the time before the exception and the time after it. By using a violation contract state, e.g. one of the deontic conflict fluents defined in section 3.3, as goal state we are able to determine the critical event sequences (a plan in terms of happens predicates) that would result in the faulty contract state:

$$\text{rules}(\text{initiates/terminates}(\text{Event}_{1..n}, \text{Fluent}_{1..exception}, T))$$

$$\text{goal}(\text{holdsAt}(\text{Fluent}_{\text{exception}}, T))$$

$$\text{plan}_{1..n}(\text{happens}(\text{Event}_{1..n}, T))$$

$$\text{history}(\text{happens}(\text{Event}_{1..n-1}, T))$$

We compare the plans to the history of events and proactively conclude a future violation if the history coincides with a plan to a certain degree (e.g. at  $\text{Event}_{n-1}$ ). Additionally we can conclude that, if we do not find any event sequence (plan) for a dedicated conflict or violation that the contract specification is free of this

conflict. This means, we can additionally use abductive logic to verify the soundness and well-formedness of contract specifications.

## 4. Implementation

As runtime engine we use the rule engine Mandarax together with the Prova language extension. Mandarax is an open source java-based rule engine for backward reasoning derivation rules. Beside classical logical concepts it combines declarative logic programming (DP) with Object-oriented programming (OOP) and supports a type system based on Java, procedural attachments which wrap java methods and external clause sets e.g., to ground rules on data stored in external databases. This allows us combining the benefits of DP (good for logic) and OOP (good for representing objects and for using Java for complex computation tasks).

We made several extensions to Mandarax [cf. 7], such as the ECA meta-interpreter, the Event Calculus, the Deontic Logic implementation or the Mandarax RDF extension. Additionally, we developed a library of useful SLA related domain-specific functions which for example allow us computing the average availability of the service per month in rules such as: “*If average availability of the service measured over one month is below 99% then ...*”. Lack of space prevents us from giving more details. However, more information can be found on our project site [7].

We translated our example contract and further rules found in typical SLAs for measurement, billing, reporting, etc. into our ContractLog framework and found it to be sufficiently expressive and computationally tractable. The EC is able to infer the set of maximal validity intervals (MVIs) over which the fluents maximally hold. In the case of EC with absolute times and total ordering, as we implemented it for our contract domain, the worst case complexity of deriving all the MVIs for a given property has been proven to be  $O(n^3)$ , where  $n$  is the number of recorded events for a given property. [12] In case of a loose setting, e.g. because of over-long monitoring intervals in our ECA rules or uncertainty about the event order, we can make use of the solution of Cervesato and colleagues [13], which reduces complexity from  $O(2^n)$  to  $O(n^5)$ .

## 5. Related Work

There have been many diverse research contributions in the field of contract/SLA management. Various approaches to contract representation have been made, mostly either directly buried in procedural application logics (Java, C++ etc.) or based on a separate

representation language like XML which is interpreted and executed with associated procedural application components. Only some approaches use a logic-based declarative knowledge representation. Closest to our approach are the work of Farrell et al [14] who developed the Contract Tracking XML language (CTXML) with a computational model based on the Event Calculus and the work of Grosz and colleagues [8] who developed the Semantic Web Enabling Technology (SWEET) toolkit, which represents business rules written in RuleML and extended it with Situated Courteous Logic Programming (procedural attachments and priorities). Therefore, their approaches have some similarities to our work. However, the strength of our approach lies in the combination of adequate logical formalisms and further logical constructs to deal with numbers, time, domain models, lists etc. into one framework for contract representation and enactment.

## 6. Conclusion and Further Steps

In this paper we have developed a logical framework usable for representing, monitoring and enforcing service contracts like SLAs with a combination of adequate logical formalisms. The particular advantages of our logical approach in contrast to traditional procedural programming approaches is its high flexibility, its dynamic extendibility and its high potential for the automation of contract enforcement processes such as the detection of contract violations, authorisation control, conflict detection, service billing and reporting. We plan to make further experiences with different real world service contracts in order to prove the soundness and usability of our logical framework. Future work is needed to implement a high level contract and service management tool based on our logical framework capable of modelling, managing, monitoring and executing thousands of contracts. This will include enhancement such as:

- More thorough investigations of the logical correlations between rules across different service contracts.
- Complex event processing and event correlation in order to deal with “event storms”.
- Mapping of the formal specifications into our exchangeable and reusable language format based on XML (RuleML) called the Rule-Based SLA (RBSLA) language [cf. 7].

## References

1. Paschke, A., M. Bichler, *Rule-based Languages for the Representation of Electronic Contracts*. Technical Report, 04/2003, IBIS, TUM, Munich.
2. Wright, G.H., *Deontic Logic*. Mind, 1951. 60: p. 1-15.
3. Follesdal, D. and R. Hilpinen, eds. *Deontic Logic: An introduction*. Deontic Logic: Introductory and Systematic Readings, ed. R. Hilpinen. 1971, D.Reidel Publishing.

4. Kowalski, R.A. and M.J. Sergot, *A logic-based calculus of events*. New Generation Computing, 1986. **4**: p. 67-95.
5. Shanahan, M., *Solving the Frame Problem*. 1997, London: MIT.
6. Paschke, A., *ContractLog: A Logic Framework for SLA Representation, Management and Enforcement*. Technical Report, 07/2004, IBIS, TUM, Munich.
7. Paschke, A., <http://ibis.in.tum.de/staff/paschke/rbsla/index.htm>, *RBSLA: Rule-based SLA*, project site, 2004.
8. Grosz, B.N., Y. Labrou, and H.Y. Chan. *A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML*. EC-99. 1999. Denver UK: ACM Press.
9. Shanahan, M.P. *Representing Continuous Change in the Event Calculus*. ECAI 90. 1990. Stockholm, Sweden.
10. Lupu, E.C., M.S. Sloman, *Conflicts in Policy-Based Distributed Systems Management*. IEEE Transactions on Software Engineering - Special Issue on Inconsistency Management, 1999. **25**: p. 852-869.
11. Russo, A., et al. *An Abductive Approach for Analysing Event-Based Requirements Specifications*. in *18th In.l Conference on Logic Programming*. 2002. Copenhagen, Denmark.
12. Chittaro, L. and A. Montanari. *Efficient Handling of Context Dependency in the Cached Event Calculus*. in *TIME94*. 1994. Pensacola, FL, USA.
13. Cervesato, I., L. Chittaro, and A. Montanari. *What the Event Calculus actually does and how to do it efficiently*. GULP-PRODE'94. 1994. Peñíscola, Spain.
14. Farrell, A., et al. *Performance Monitoring of Service Level Agreements for Utility Computing using Event Calculus*. in *1st IEEE Int. Workshop on Electronic Contracting*. 2004. San Diego, CA.