ContractLog: An Approach to Rule Based Monitoring and Execution of Service Level Agreements

Adrian Paschke[1], Martin Bichler[1], and Jens Dietrich[2]

[1] Internet-based Information Systems, Technische Universität München
{Paschke,Bichler}@in.tum.de
[2] Information Sciences & Technology, Massey University
J.B.Dietrich@massey.ac.nz

Abstract. In this paper we evolve a rule based approach to SLA representation and management which allows separating the contractual business logic from the application logic and enables automated execution and monitoring of SLA specifications. We make use of a set of knowledge representation (KR) concepts and combine adequate logical formalisms in one expressive formal framework called **ContractLog.**

1 Introduction

Service Level Management (SLM) and Service Level Agreements (SLAs) are of growing commercial interest with a deep impact on the strategic and organisational processes as intensified interest in accepted management standards like ITIL[1] or the new BS15000[2] shows. Additionally, IT virtualisation and upcoming flexible IT infrastructures like e.g. new middleware products, storage area networks or grids services pave the way for new service oriented business models (e.g. "on-demand", "pay-per-use", "utility computing") with flexible and more individual contracts. [1] This needs new levels of flexibility and automation in SLA management not available with the current technology and tools [2, 3]. This paper proposes a rule based representation of SLAs using sophisticated, logic-based knowledge representation (KR) concepts as an alternative to natural language defined contracts or pure procedural implementations in programming languages such as Java or C++. We combine selected adequate logical formalisms in one expressive framework called **ContractLog** with which to describe formal rule based contract specifications which can be automatically monitored and executed**.** The essential advantages of ContractLog 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.

[1] IT Infrastructure Library (ITIL): www.itil.co.uk
[2] BS15000 IT Service Management Standard: www.bs15000.org.uk

A. Adi et al. (Eds.): RuleML 2005, LNCS 3791, pp. 209–217, 2005.

2. Rules can be automatically linked (rule chaining) and executed by rule engines in order to enforce complex business policies and individual and graduated contractual agreements.
3. Test-driven validation and verification methods can be applied to determine the correctness and completeness of contract specifications against user requirements [4] and large rule sets can be automatically checked for consistency. Additionally, explanatory reasoning chains provide means for debugging and explanation. [5]
4. Contract norms like rights and obligations can be enforced and contract violations can be (proactively) detected and treated via automated monitoring processes and hypothetical reasoning. [5]

The rest of the paper is structured as follows. In section 2 we define the requirements for a logic based rule language capable of representing, monitoring and enforcing service contracts. In section 3 we present an overview of our solution to meet these requirements – the ContractLog framework. In section 4 we describe our implementation effort based on the open source rule engine Mandarax and compare our rule based approach to common procedural implementations. Further information on our implementation and more details on the applied logical formalisms can be found in [3, 5, 6] and on our project web site [7]. Finally, in section 6 we conclude this paper with a short summary and an outlook on the higher level **R**ule **B**ased **SLA** language: **RBSLA** [7].

2 Requirements

We analyzed several real world SLAs from different service providers in different branches and several commercial tools like e.g. Tivoli SLA, in order to identify the problems and to derive the requirements on an adequate, automated SLA representation language. The two main problems which we found are:

1. In many companies SLAs are informally described in natural language. This leads to simplified SLA rules and many manual processes in management and monitoring of SLAs.
2. Existing SLM tools with their hard coded application logic and common reference models like ITIL are too little automated, flexible and adaptable.

As a consequence SLA management needs new ways of knowledge representation for contractual agreements and new technical solutions for contract monitoring and enforcement. Beside basic information about the roles of the parties, the contract life time, the agreed services, etc. SLAs contain (business) rules on rights and obligations, prices and costs, quality of service (QoS) and service levels, penalties for contract violations, termination conditions etc. Automated monitoring and execution of such rules requires formalization of the rule syntax. Logic-based rule languages and dedicated rule engines can be used to solve this task [2, 3]. However, SLAs have a number of requirements regarding an adequate knowledge representation. Figure 1 shows the main requirements. For a detailed description of these and further requirements not listed here see [6].

3 The ContractLog Framework

Table 1 summarizes the main concepts used in ContractLog, our solution to the requirements identified in section 2 (see fig. 1).

1. Rule chaining
2. Default rules
3. Rule prioritization
4. Contract modularization
5. External data integration
6. Procedural attachments
7. OO type system and integration of business objects
8. Semantic (business) vocabularies and domain descriptions
9. Situated processing of events and actions
10. Temporal reasoning on events and their effects
11. Contract norms on an individual and group level
12. Verification, validation and conflict resolution
13. Declarative rule syntax and rule serialization

Fig. 1. Main requirements on a declarative rule language

Table 1. Main logic concepts of ContractLog

Logic	**Usage**
Derivation rules (horn logic with NaF)	Enables deductive reasoning on business rules.
Event-Condition-Action rules (ECA)	Active event detection and situative behaviour by event-triggered executable actions.
Event Calculus (temporal reasoning)	Temporal reasoning about dynamic systems, e.g. effects of events on the contract state.
Defeasible logic / GCLP (priorities)	Default rules and priority relations of rules. Facilitates conflict detection and resolution as well as revision/ updating and modularity of rules.
Deontic logic	Enables representing rights and obligations as deontic contract norms „permission, prohibition, obligation“ and norm violations (contrary-to-duty obligations) and exceptions (condit. defeasible obligations).
Description logic	Enables semantic domain descriptions (e.g. contract ontologies) to hold rules domain independent. Facilitate exchangeability and interpretation.
Procedural object-oriented logic / procedural attachments	Procedural attachments integrate object oriented programming into declarative rules. Merits the benefits of procedural logic (e.g. Java EJBs) and declarative logic programming (representing business logic).

In the following we want to describe the main formalism used in ContractLog. More information can be found in [1-3, 5, 6] and on our project site [1].

Derivation Rules with 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 level of flexibility in automatically combining rules to form complex business policies and graduated contract rules [3, 8]. On the other hand procedural logic as used in programming languages is highly optimized in solving computational problems - however, with the disadvantage that the complete control flow must be implemented. Procedural attachments and the

use of a typed logic[3] offer a clean way of integrating programming languages 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 includes accessing external databases using optimized query languages like SQL to temporarily populate the knowledge base with the needed facts for the inference processes at query time. After the query has been answered (using backward reasoning) these facts can be discarded from memory and therefore replication of data is not necessary any more, 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.

ECA Rules

A key feature of a SLA monitoring system is its ability to actively detect and react to events and many rules in SLAs are basically Event Condition Action (ECA) rules, e.g.: "*If the service becomes unavailable (Event) then send a notification message to the service administrator (Action)*". We implemented support for active ECA rules in our backward reasoning system based on an independent daemon process, which monitors all ECA rules by periodically querying the rule base using a thread pool for parallel execution of rules. We represent an ECA rule as a derivation rule: *eca(T,E,C,A)*. Each term *T* (time), *E* (event), *C* (condition) and *A* (action) references to a further derivation rule which implements the respective functionality of the term. The additional term *T* (time) is introduced to define the monitoring intervals/schedules in order to control monitoring costs for each rule. Example:

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

Rule chaining combining derivation rules can be used to build complex functionalities, which can be referenced from several ECA rules. More details on the ECA implementation can be found in [6].

Event Calculus

The Event Calculus (EC) [9] is a formalism for temporal reasoning about events and their effects on a knowledge system. 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. 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. countdowns, validity time of norms), continuous changes (e.g. time-based counter) and epistemic knowledge (planned events e.g. for hypothetical reasoning) [5, 6]:

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*

[3] ContractLog supports typed variables and constants based on the object-oriented type system of Java

initiates/terminates(E,F,T)	*event 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*	*occurred(E,T)*	*event E actually happened*
		derivedFluent(F)	*derived fluent F*

The EC and ECA can be combined and used vice versa, for example fluents (holdsAt) can be used in the condition parts of ECA rules or ECA rules can be used to persistently assert detected events to the EC knowledgebase and define e.g. ECA rules with post conditions (a.k.a. ECAP rules). The EC enables us to model the effects of events on changeable SLA properties (e.g. deontic contract norms describing rights and obligations) and to reason about the contract state at certain time points. In addition we can define complex state transition rules similar to workflows. This is very useful for deontic contract norms, e.g., for representing violations of norms (e.g. violation of fulfilling an obligation in a defined period).

Deontic Logic

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* and *dependencies between norms*, e.g. violations of 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 extended the concepts of DL with a role-based model and integrated it in our Event Calculus implementation in order to model the effects of events on deontic norms and to represent dependencies between deontic norms. [5] 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 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)*. Additionally, we implemented typical DL inference axioms in ContractLog, e.g.: $O_{s,o}A \rightarrow P_{s,o}A$: *holdsAt(permit(S,O,A),T)* ← *holdsAt(oblige(S,O,A),T)* or $F_{s,o}A \rightarrow W_{s,o}A$: *holdsAt(waived(S,O,A),T)* ← *holdsAt(forbid(S,O,A),T)* etc. and additional rules to deal with deontic conflicts, violations of deontic norms and their contrary-to-duty paradoxes, e.g. Authorization Conflict: *holdsAt(authConflict(S,O,A),T)* ←*holdsAt(permit(S,O,A),T). holdsAt(forbid(S,O,A),T)*. The tight combination of the time based EC with role based deontic norms 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. In particular *derived fluents* and *delayed effects* (with *trajectories and parameters* [5]) offer the possibility to define exceptions and violations of contract norms and their consequential secondary norms e.g., conditional 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. [5, 6] Example:

"*If the service is unavailable, the SP is **obliged** to restore it within $t_{deadline}$. If the SP fails to restore the service in*

$t_{deadline}$, *(violation) the SC is* ***permitted*** *to cancel the contract (consequence).*"
Representation in ContractLog
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 obligation must be fulfilled*
happens(elapsed,T) ← *valueAt(deadline,T,* $t_{deadline}$*)* // *defines the violation event which happens when the deadline is reached*
initiates(elapsed, permit(SC, Contract, cancel()),T) // *initiates the derived permission to cancel the contract*

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 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 (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) [10] and Grosof´s Generalized Courteous Logic Programs (GCLP) [11]. There are four kinds of knowledge in DfL: *strict rules, defeasible rules, defeaters and priority relations*. We base our implementation on the meta-program found in [12] 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 [6].

Description Logics

Inspired by recent approaches to combine description logics and logic programming [13, 14] 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] [7] to answer typical DL queries (RDFS and OWL Lite/DL inference) such as class-instance membership queries, class subsumption queries, class hierarchy queries etc. For example:

RDFS inference examples:
$C \subseteq D$, i.e., class *C* is subclass of class *D: type(a, D)* ← *subClassOf(C,D), type(a,C)*
$Q \subseteq P$, i.e., *Q* is a subproperty of *P: property(Q,a,b)* ← *subPropertyOf(Q,P), property(P,a,b)*
$T \subseteq \forall P.C$,i.e., the range of property P is class C: *type(b,C)* <-- *range(P, C), property(P, a, b)*
$T \subseteq \forall P`.C$,i.e., the domain of property P is class C: *type(b,C)* <-- *range(P, C), property(P, a, b)* *etc.*

OWL inference examples:
$C \equiv D$, i.e., class C is equivalent to class D: *type(a,C)* <-- *equivalentClass(C, D), type(a,D)*
type(a,D) <-- *equivalentClass(C, D), type(a,C)*
$P^{+} \subseteq P$ i.e., property P is transitive: *property(P,a,c)* ← *type(P,"owl:TransitiveProperty"),property(P, a,b),property(P,b,c)* *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 Implementation and Discussion

We implemented the ContractLog framework based on the backward-reasoning rule engine Mandarax [8] and the Prova language extension [15], which provides a Prolog related syntax. Mandarax is an open source java-based rule engine for backward reasoning derivation rules. It provides a typed logic (typed rule terms) and procedural attachments which wrap java methods. This allows combining the benefits of LP and object-oriented programming and provides a high level of flexibility and automation. It offers the option to restrict the applicability of rules and to control the level of generality in queries and most importantly it makes possible the desired tight integration of Java code into logical rules, e.g. using monitoring functions from existing system management tools and delegating computation intensive tasks to Java (e.g. for computing average performance values and service levels), or triggering action functionalities from existing business object implementations like EJBs in ECA rules. Additionally, it supports clause sets to ground rules on data stored in external databases. This enables integrating facts from external databases (e.g. a data warehouse) via highly optimized query languages such as SQL into rule executions. Because we are

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

using well understood and sound logic formalism and implement them on the basis of horn logic our logic framework stays computational tractable and efficient although it provides rich expressiveness.

In contrast to procedural programming approaches where the control flow must be completely implemented, the logic based rule approach allows a more compact representation of SLAs. Additionally, dynamic reaction on external events with ECA rules and temporal conclusions about their effects on the contract state, e.g. on rights, obligations or violations are enabled by computational models like the Event Calculus. A static procedural code representing this type of temporal event based logic would have been much more cumbersome to implement and maintain. Other examples are *graduated rules* e.g. graduated penalty rules for missing certain availability levels, *dynamic rules*, e.g. to adapt to special situations or complex *dependencies between rules*. From these examples it is easy to see why the declarative style of logic programming can be superior to pure procedural languages in situations when flexibility and code economy are required to represent business logic which is likely to change over time.

5 Conclusion and Outlook

In this paper we have describe a rule based approach to SLA representation and management. We have summarized the requirements on an adequate representation language and evolved our ContractLog framework on the basis of horn rules and meta programming techniques to solve this needs. In contrast to conventional pure procedural programming approaches our 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 applied, simplified rules and the less adaptable procedural management tools. However, real usage of a representation language which is usable by others than its inventors immediately makes rigorous demands on the syntax: e.g. 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. and in particular a declarative syntax. We try to address these requirements with our superimposed declarative **R**ule **B**ased **SLA** language **RBSLA** [7], which is implemented on top of ContractLog. It adapts and extends RuleML [16] to the needs of the SLA domain. The main additional features we introduce are: (1) *definitions and terms* defining the meaning of the concepts used in SLAs by referencing on external contract vocabularies and semantic ontologies (RDFS/OWL); (2) *ECA rules* including monitoring schedules/intervals, active event monitoring/measurement functions and actively triggered, executable actions; (3) *deontic personalized contract norms with consequential violations and penalties* triggered by time based events; (4) *integration of external data and system/object functionalities* via procedural attachments, clause sets and typed constants and variables (Java or RDFS/OWL); (5) *modularization of contract structures and rule sets* including defeasible rules and priority relations; This includes a transformation implementation which maps the declarative RBSLA into

executable ContractLog rules (Prova/Prolog syntax) and additionally performs validation, optimization and refactoring of the declarative rule sets during this process.

References

1. Bichler, M., Diernhofer, N., Fay, F., König, C., MacWilliams, A., Paschke, A., Setzer, T., Völk, G., *Dynamic Value Webs for IT-Services: IT-Service Technologies and Management*. 2004, Siemens SBS / TUM, research study, 10/2004: Munich.
2. Paschke, A. and M. Bichler, *Rule-based Languages for the Representation of Electronic Contracts - A concept for using Knowledge-based Systems in the Development of flexible Internet-based Information Systems. (in german language)*. 04/2003, IBIS, TUM, Working Paper.
3. Paschke, A., *Rule Based SLA Management - A rule based approach on automated IT service management (in german language)*. 6/2004, IBIS, TUM, Working Paper.
4. Dietrich, J. and A. Paschke. *On the Test-Driven Development and Validation of Business Rules*. in *ISTA05*. 2005.
5. 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.
6. Paschke, A., *ContractLog - A Logic Framework for SLA Representation, Management and Enforcement.* 7/2004, IBIS, TUM.
7. Paschke, A., *RBSLA: Rule-based SLA, http://ibis.in.tum.de/staff/paschke/rbsla/index.htm*. 2005.
8. Dietrich, J. *A Rule-Based System for eCommerce Applications*. in *KES 2004*. 2004.
9. Kowalski, R.A. and M.J. Sergot, *A logic-based calculus of events.* New Generation Computing, 1986. 4: p. 67-95.
10. 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.
11. Grosof, B.N., *A Courteous Compiler From Generalized Courteous Logic Programs To Ordinary Logic Progams*. IBM, 1999.
12. Antoniou, G., et al. *A flexible framework for defeasible logics*. in *AAAI-2000*. 2000.
13. Levy, A. and M.-C. Rousset. *A Representation Language Combining Horn Rules and Description Logics*. in *ECAI96*. 1996.
14. Grosof, B.N., et al. *Description Logic Programs: Combining Logic Programs with Description Logic*. in *WWW03*. 2003: ACM.
15. Kozlenkov, A. and M. Schroeder, *Prova*. 2004, http://comas.soi.city.ac.uk/prova/.
16. Wagner, G., S. Tabet, and H. Boley. *MOF-RuleML: The abstract syntax of RuleML as a MOF model. OMG Meeting*. 2003.