Using the Event Calculus for the Performance Monitoring of Service-Level Agreements for Utility Computing

Andrew D H Farrell, Marek J Sergot,
Department of Computing,
Imperial College London,
United Kingdom.
{adf02, mjs}@doc.ic.ac.uk

Mathias Salle, Claudio Bartolini,
HP Labs,
Palo Alto, California,
United States of America.
{mathias.salle, claudio.bartolini}@hp.com

Abstract

The automated performance monitoring of contracts, in terms of tracking contract state, is an important issue investigated in this work. We define contract state to be the sum of the normative relations that hold between contract parties. In order to facilitate state tracking, we define an XML formalisation of the Event Calculus, ecXML. This language is used to describe how a contract's state evolves, according to events that are described in the contract.

The work is grounded in the domain of Utility Computing (UC). UC is concerned with the provisioning of computational resources (compute-power, storage, network bandwidth), on a per-need basis, to corporate businesses. Service-level Agreements (SLAs) - contracts between a provider and a customer - are a sine qua non in the deployment of UC.

1. Introduction

Utility Computing (UC) [1] offers an opportunity to corporate businesses to maximise the efficiency and efficacy of their IT service provision (both in-house and to customers). It will allow them to out-source large areas of their IT service provision to UC-data centres, which will agree to provide computational resources, packaged as services to them.

The levels of service that are agreed between a UC service-provider and customer are mandated by Quality-of-Service (QoS) guarantees, written as Service-Level Guarantees (SLGs) within Service-Level Agreements (SLAs) [2]. An example SLG might be:

- Service Availability should be greater or equal to 99%, weekdays 9a.m.–5p.m.
- Service Availability should be greater or equal to 95%, at all other times
- Availability metric is measured over each calendar month; penalty for SLG violation: *refund to customer monthly fee*

SLAs are essential for formalising the objectives of a UC service, and to manage expectations [3].

In this work, we have been interested in monitoring the state of contracts while they are active. We define *contract state* as the *sum of the normative relations that hold between contract parties.* An example of a normative relation might be: *Service Provider is* ***permitted*** *to terminate the SLA.*

The work that has been realised here has been concerned with one particular aspect of the life cycle of a contract (such as an SLA), namely, automated run-time performance monitoring [4]. In our view, performance monitoring is concerned with (at least) two functional aspects: (i) Tracking the effect of events (pertinent to a contract) on contract state – the contractual (or, normative) relations that hold between contract parties – and informing interested parties of past, present and (possible) future contract states; and, (ii) Assessing the current state of the contract, in terms of its utility (that is, worth), and other metrics related to business intelligence [5]. The work presented in this paper is primarily concerned with the first of these, which is known as automated contract (state) tracking to distinguish it.

Notably, approaches to automated tracking of contract state, thus far, can be largely characterised in one of two ways [6]: (i) As general-purpose reasoning frameworks that (mainly) have not been applied in actual, deployed systems; or (ii) In the case of SLAs, as being fairly limited in capability. The work presented here is considered to be distinguished from such approaches in that: (i) It has been developed in the

context of a *real-world* deployment scenario (namely, SLAs for UC), while being generalised so to be applicable to other domains; and (ii) It represents an advance (over many approaches) in what can be realised regarding performance monitoring for contracts.

This paper is structured as follows. In section 2, we present an example contract (namely, an SLA for a UC scenario), used to ground our discussions. In section 3, a brief analysis of this contract is given. In section 4, an informal introduction to the Event Calculus is presented. In section 5, a brief overview of the representation of the example contract in the Event Calculus is given. In sections 6 and 7, we discuss the implementations of a reasoner for contracts written in the Event Calculus, and of a deployment tool for SLAs. In section 8, we present related work and conclude the paper in section 9.

2. Example contract

In this paper, we use the following Mail Service UC SLA in order to ground our discussions. Portions in bold are referred to in the course of this paper.

- The Service Provider (*SP*) will provide a mail service to the Service Consumer (*SC*), which includes a mailbox with a quota of *s* GBytes. ***SC* will be charged a fixed monthly fee of $s \times c_0$ for the service**.
- **In the case that the mail service is unavailable, *SP* will pay *$p* for every whole *t* minutes that it is unavailable.** *SP* is obliged to pay any penalties to *SC* within a month of their accruement.
- **Whenever $u>s$, where *u* is the mailbox utilisation in GBytes, *SP* will charge *SC* c_1 for each GByte over *s*, calculated daily, until $u \leq s$**
- Whenever $u>s+e$, where *e* is a level of tolerance in GBytes, *SC* will not be able to receive emails.
- **All billing of *SC* occurs monthly, and *SC* is given a month thereafter to pay. If *SC* fails to pay within the given time, *SP* may terminate the mailbox service without notice.**

3. Brief Analysis of Contract

For the purposes of representing a contract in order to facilitate state tracking, we are concerned with identifying events described in the contract that can have an effect on contract state. Once identified, we need to express, in our representation, the effects on contract state of these events

For example, the contract excerpt: "*All billing of SC occurs monthly*" indicates a *monthly billing event*. One effect of such an event is that *SC* receives an invoice for service. But this is not an effect on contract state, *per se*. We shall say that another effect of this event – this time, on the contract state – is to activate a normative relation, namely an obligation bearing on *SC* to pay *SP* for service, where *"SC is given a month thereafter to pay"* – that is, *SC* has a month to fulfil its obligation to pay.

Another example is: *"If SC fails to pay within the given time, SP may terminate the mailbox service without notice"*. This statement talks about another event; that of *SC* failing to fulfil their obligation (to pay for service) on time. We shall say that an effect of this event is to activate another normative relation, namely a (vested) permission for *SP* to terminate the mailbox service

4. Using the Event Calculus

From the perspective of what needs to be represented for contract state tracking, then, we need some way of representing the **effects of events** on contract state. For this, we use the Event Calculus (EC) [7]. In this work, we have defined an XML formalisation of EC, called *ecXML*. We use XML as a convenient file format for our implementation of a reasoner – the Event Calculus State Tracking Architecture (ECSTA) – for contracts written in *ecXML*.

In the sequel, we will not describe the XML formalisation, rather we will introduce EC informally. We say that a contract (as represented for state tracking) in the Event Calculus is a conjunction of:

- A finite set of **initially** statements, e.g.:
  o1 **initially holds**[1] (if some condition holds)
- A finite set of **initiates** statements, e.g.:
  the occurrence of a billing event **initiates** *o1* (if some condition holds)
- A finite set of **terminates** statements, e.g.:
  the occurrence of a fulfilment event for *o1* **terminates** *o1* (if some condition holds)
- A finite set of **occurs** statements, e.g.:
  a billing event **occurs** 1 month into the contract
- Plus some other statements

Then, axiomatic to every contract in EC is the **holds** statement, which says: Normative relation *r* holds at time *T* if and only if:

- There is a statement which says that it holds initially *or* it has been initiated before or at *T* AND
- It has not been terminated, after its initiation, before or at *T*

For example:

[1] Where *o1* may, for example, be an obligation on *SC* to pay *SP* for service.

- The occurrence of a billing event **initiates** *o1*
- The occurrence of a fulfilment event for *o1* **terminates** *o1*
- A billing event occurs 1 month into the contract
- A fulfilment event for *o1* occurs 1.5 months into the contract

According to the *holds* statement, *o1* does not hold at 0.5 month because it does not hold initially and it has not been initiated before or at 0.5 month. However, *o1* does hold at time 1.25 months because the billing event occurring at 1 month initiates *o1* and it has not been terminated since 1 month but before or at 1.25 months. Finally, *o1* holds at time 2 months because notwithstanding its initiation at 1 month, it is terminated by the occurrence of the fulfilment event for *o1* at 1.5 months.

It is considered that a normative relation may be parameterised. Different parameterisations using the same relation name count as different normative relations. As such, there is always a single proposition that pertains to a normative relation. A parameterisation for *o1* might be: amount that *SC* owes, along with the billing month. An example normative relation pertaining to *o1* would then be: *o1(Charge, Oct2004)*. Note that if *o1* is not parameterised, then *o1* itself would count as the normative relation.

Specifying parameters may be useful to external components; in the example, a billing component may use the information provided in sending *SC* an invoice. Parameters may be passed back to the contract representation within *occurs* statements.

Conditions in our formalisation of the Event Calculus are used in *initially*, *initiates* and *terminates* statements. They may comprise *not, and, or, beq (boolean equals), geq (double greater or equals), leq (double less or equals), gt (greater), lt (less), deq (double equals), bool (boolean value), bpara (boolean parameter), btpara (boolean contract parameter), occurs (event occurrence),* and *holds (normative relation holds)* statements.

The statements: *geq*, *leq*, *gt*, *lt*, and *deq* take numerical (that is, double) operands, which may be provided by the following statements: *mul, add, sub, div, num (double value), dpara (double parameter), dtpara (double contract parameter),* and *value (contract variable value*).

Note that *contract variables* are used to maintain live, numerical state – their use is normative in that it is agreed by all parties when a contract is signed. A *contract parameter* is assigned a value at the instantiation of a contract, and facilitate the notion of contract templates, which are customised for particular scenarios.

Timers may be specified in our formalisation, where these may be one-off or recurrent. They are used to generate *occurs* statements which are then processed by a reasoner for the instantiated contract. An example of their use is for generating violation events for obligations, which typically have a time constraint for their fulfilment.

Finally, using our formalisation, it is possible to set up *state definitions* for a contract, which allow us to monitor states of interest. For example, we may say that a *"Payment Outstanding"* state exists *if* normative relation *o1* holds. Thus, it is the *holds* statement that allows us to query the state of a contract. That is, to ask if the contract is in a *"Payment Outstanding"* state, we ask whether *o1* holds.

5. Event Calculus Representation of Example Contract

Firstly, we give (unique) names to the identified normative relations and events in the contract. For the example contract, we shall use these names:

- Normative relations: *Obligation (on SC to pay for service)* is *o1*, and *Privilege (for SP to terminate mailbox service) is p1*
- Events: *Billing event* is *bill_timer*, and *Failure to fulfil obligation o1* is *o1_violated*

For each identified event, we need to state its effects in the Event Calculus on contract state. For example, we want to say that *bill_timer* initiates obligation *o1*. This would usually be written in *ecXML*, but is given here in English for clarity:

- The occurrence of event *bill_timer* **initiates** *o1* with a single **initiation parameter** *Charge* set to the value obtained by summing the current values of **contract variable** *vDailyCharge* and **contract parameter** *sc0*

This statement in part accounts for the following snippets of the example contract:

- *"SC will be charged a fixed monthly fee of s X c0 for the service"*. This is reflected in the use of *sc0* in the contract statement.
- *"Whenever u>s, where u is the mailbox utilisation in GBytes, SP will charge SC c1 for each GByte over s, calculated daily, until u• s"*. This is reflected in the use of *vDailyCharge* in the contract statement.

- *"All billing of SC occurs monthly, and SC is given a month thereafter to pay"*; in that, it initiates the obligation *o1* on *SC* to pay.

Another example is the contract excerpt: *"Failure on the part of SC to fulfil obligation o1 initiates privilege p1"*. This is written as:

- The occurrence of event *o1_violated* **initiates** *p1*

Finally, the contract excerpt: *"In the case that the mail service is unavailable, SP will pay $p for every whole t minutes that it is unavailable"* is, in fact, part of a Service-Level Guarantee (SLG), namely, the SLG pertaining to the provision of the mail service. Specifically, it describes what course of action is normative in the case that the SLG is violated by *SP*.

We assume that some *monitoring agent* tells us when the SLG has been violated, that is that the mail service is unavailable. This agent will generate an event, *SLG1_violated* say, to this effect; and will generate an event, *SLG_restored* say, when the mail service has been restored.

We shall say that there exists elsewhere in our representation of the example contract, a statement setting up a timer, *SLG1_timer*, with a period of *t* minutes. Also relation *o2* is defined as an obligation that bears on *SP* to restore the service.

This contract excerpt would then be written as follows:

- The occurrence of event *SLG1_violated* **initiates** *SLG1_timer*
- The occurrence of event *SLG1_violated* **initiates** *o2*
- The occurrence of event *SLG1_restored* **terminates** *SLG1_timer*
- The occurrence of event *SLG1_restored* **terminates** *o2*
- The occurrence of event *SLG1_timer* **initiates** contract variable *vPenalty* to the value obtained by summing the current value of *vPenalty* and *$p*

6. Event Calculus State Tracking Architecture (ECSTA)

A reasoner for contracts written in *ecXML*, called the Event Calculus State Tracking Architecture (ECSTA) has been implemented in Java, supporting: instantiation of contracts written in *ecXML*, assertion of event narratives including speculative narratives which can be unrolled, and querying of SLA state.

A full list of use-cases for ECSTA is as follows:

- Discover Registered Contract Templates, Register Contract Template, Deactivate/Reactivate/Destroy Contract Template
- Discover Instantiated Contracts, Instantiate/Reactivate/ Deactivate/Destroy Contract, Retrieve Contract
- Add Contract Clauses and User Rules, Overwrite Timestamps in Clauses and User Rules,
- Request/Change Contract Parameters
- Assert Input Contract Events
- Query Contract, i.e. query global state of contract, query particular fluent or contract variable (multi-valued fluent), query global state history of contract, query history of particular fluent or contract variable
- Register for/Deactivate/Reactivate Notification of Output Contract Events
- Register for/Deactivate/Reactivate Clause and User Rule Triggering Notification Events
- Allocate/Destroy Shared Variable
- Register/Deactivate/Reactivate Shared Variable Association
- Create/Destroy Simulation Context

One particularly useful functionality is for a user to register an interest in being notified of particular contract-related occurrences. This is supported through **user rules**.

Say an *incident manager*, responsible for handling the effects of UC-fabric incidents on the fulfilment of SLAs, would like to be notified when the number of violated obligations across a number of SLAs goes above *x*.

As this requires reasoning across multiple SLAs, we need to use the **Allocate Shared Variable** use-case to get the reasoner to allocate a **shared variable**. Say, the reasoner calls the shared variable *v1*. Then, we add the following *user rule* to each pertinent SLA using the **Register Shared Variable Association** and **Add SLA Clauses and User Rules** use-cases (written here in English, but would normally be *ecXML*): *Whenever a violation event for an obligation is received and is pertinent, increment v1*. Then, we add the following user rule, *u1*, to a single SLA: *For changes in the value of v1, where v1 goes above x, do nothing.* Importantly, rule *u1* is considered to be *triggered* whenever *v1* goes above *x*. Finally, we ask to be notified whenever rule *u1* is triggered, by using the **Register for Clause and Rule Triggering Notification Events** use-case.

7. SLA Visualiser

As well as the ECSTA reasoner, a tool called *SLA Visualiser* has been implemented which allows for the deployment management of SLAs. It provides a user-interface to SLA deployment tasks, and supports all of

the use-cases given in section 6. The relationship between ECSTA and SLA Visualiser is captured in figure 1.

In figures 2 through to 9 a scenario is shown unfolding, as captured by SLA Visualiser.

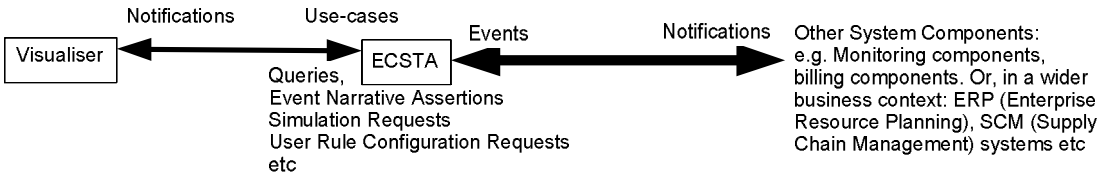

Figure 1: Relationship between ECSTA and SLA Visualiser

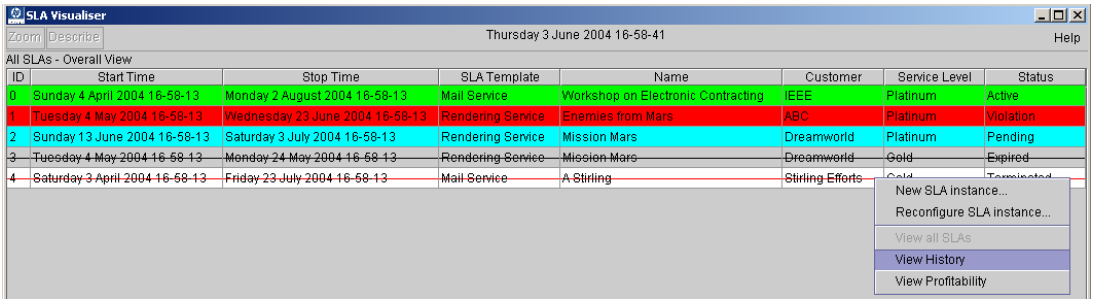

Figure 2: Top-Level View in SLA Visualiser

In figure 2, we select SLA 4 to look at its history. We see that it has been terminated, which would happen through the customer failing to pay for service.

Figure 3: Scenario Unfolds 1

In figure 3, we see that the state of SLA is “Ok” to begin with.

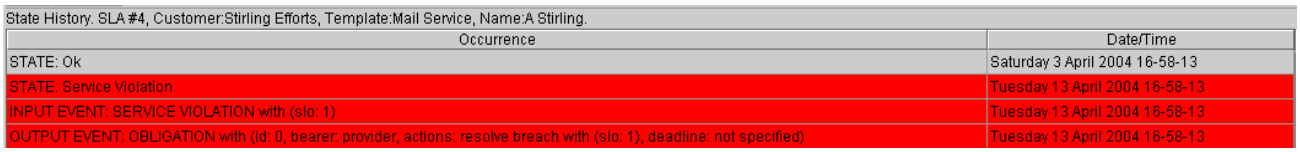

Figure 4: Scenario Unfolds 2

In figure 4, we see that a “Service Violation” event occurs causing: the state of the SLA to change to “Service Violation” and an obligation to be initiated bearing on the provider to restore the service.

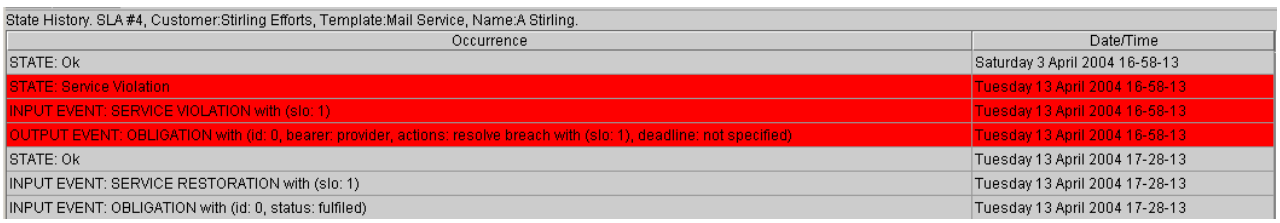

Figure 5: Scenario Unfolds 3

In figure 5, we see that a “Service Restoration” event occurs causing: the state of SLA to return to “Ok”. Also the obligation bearing on the provider to restore the service is fulfilled.

State History. SLA #4, Customer:Stirling Efforts, Template:Mail Service, Name:A Stirling.

Occurrence	Date/Time
STATE: Ok	Saturday 3 April 2004 16-58-13
STATE: Service Violation	Tuesday 13 April 2004 16-58-13
INPUT EVENT: SERVICE VIOLATION with (slo: 1)	Tuesday 13 April 2004 16-58-13
OUTPUT EVENT: OBLIGATION with (id: 0, bearer: provider, actions: resolve breach with (slo: 1), deadline: not specified)	Tuesday 13 April 2004 16-58-13
STATE: Ok	Tuesday 13 April 2004 17-28-13
INPUT EVENT: SERVICE RESTORATION with (slo: 1)	Tuesday 13 April 2004 17-28-13
INPUT EVENT: OBLIGATION with (id: 0, status: fulfiled)	Tuesday 13 April 2004 17-28-13
STATE: Provider Payment Outstanding, Customer Payment Outstanding	Wednesday 14 April 2004 16-58-13
OUTPUT EVENT: OBLIGATION with (id: 1, bearer: provider, actions: refund money with (amount: 25.00), deadline: before end of business day)	Wednesday 14 April 2004 16-58-13
OUTPUT EVENT: OBLIGATION with (id: 2, bearer: Stirling Efforts, actions: pay for service with (amount: 50.00), deadline: 1 month)	Wednesday 14 April 2004 16-58-13

Figure 6: Scenario Unfolds 4

In figure 6, we see that two obligations are initiated (by timers that are specified in the SLA representation and maintained by the reasoner) stipulating that: the Service Provider must refund $25 to the Service Customer for poor service (before end of business day) and the Service Customer must pay $50 for service to the Service Provider (within 1 month). This causes the SLA to move into state: “Provider Payment Outstanding” + “Customer Payment Outstanding”.

State History. SLA #4, Customer:Stirling Efforts, Template:Mail Service, Name:A Stirling.

Occurrence	Date/Time
STATE: Ok	Saturday 3 April 2004 16-58-13
STATE: Service Violation	Tuesday 13 April 2004 16-58-13
INPUT EVENT: SERVICE VIOLATION with (slo: 1)	Tuesday 13 April 2004 16-58-13
OUTPUT EVENT: OBLIGATION with (id: 0, bearer: provider, actions: resolve breach with (slo: 1), deadline: not specified)	Tuesday 13 April 2004 16-58-13
STATE: Ok	Tuesday 13 April 2004 17-28-13
INPUT EVENT: SERVICE RESTORATION with (slo: 1)	Tuesday 13 April 2004 17-28-13
INPUT EVENT: OBLIGATION with (id: 0, status: fulfiled)	Tuesday 13 April 2004 17-28-13
STATE: Provider Payment Outstanding, Customer Payment Outstanding	Wednesday 14 April 2004 16-58-13
OUTPUT EVENT: OBLIGATION with (id: 1, bearer: provider, actions: refund money with (amount: 25.00), deadline: before end of business day)	Wednesday 14 April 2004 16-58-13
OUTPUT EVENT: OBLIGATION with (id: 2, bearer: Stirling Efforts, actions: pay for service with (amount: 50.00), deadline: 1 month)	Wednesday 14 April 2004 16-58-13
STATE: Customer Payment Outstanding	Wednesday 14 April 2004 17-8-13
INPUT EVENT: OBLIGATION with (id: 1, status: fulfiled)	Wednesday 14 April 2004 17-8-13

Figure 7: Scenario Unfolds 5

In figure 7, we see that an input event saying that the Service Provider has fulfilled its obligation to refund $25 to the service customer occurs causing: the state of the SLA moves from “Provider Payment Outstanding” + “Customer Payment Outstanding” to just “Customer Payment Outstanding”. The fulfilment of the obligation bearing on the Service Provider occurs just 10 minutes after it was initiated and within the business day as stipulated – the manifestation of the fulfilment may be that the billing system sent the customer a cheque, or organised a fund transfer.

State History. SLA #4, Customer:Stirling Efforts, Template:Mail Service, Name:A Stirling.

Occurrence	Date/Time
STATE: Ok	Saturday 3 April 2004 16-58-13
STATE: Service Violation	Tuesday 13 April 2004 16-58-13
INPUT EVENT: SERVICE VIOLATION with (slo: 1)	Tuesday 13 April 2004 16-58-13
OUTPUT EVENT: OBLIGATION with (id: 0, bearer: provider, actions: resolve breach with (slo: 1), deadline: not specified)	Tuesday 13 April 2004 16-58-13
STATE: Ok	Tuesday 13 April 2004 17-28-13
INPUT EVENT: SERVICE RESTORATION with (slo: 1)	Tuesday 13 April 2004 17-28-13
INPUT EVENT: OBLIGATION with (id: 0, status: fulfiled)	Tuesday 13 April 2004 17-28-13
STATE: Provider Payment Outstanding, Customer Payment Outstanding	Wednesday 14 April 2004 16-58-13
OUTPUT EVENT: OBLIGATION with (id: 1, bearer: provider, actions: refund money with (amount: 25.00), deadline: before end of business day)	Wednesday 14 April 2004 16-58-13
OUTPUT EVENT: OBLIGATION with (id: 2, bearer: Stirling Efforts, actions: pay for service with (amount: 50.00), deadline: 1 month)	Wednesday 14 April 2004 16-58-13
STATE: Customer Payment Outstanding	Wednesday 14 April 2004 17-8-13
INPUT EVENT: OBLIGATION with (id: 1, status: fulfiled)	Wednesday 14 April 2004 17-8-13
STATE: Terminable	Friday 14 May 2004 16-58-13
INPUT EVENT: OBLIGATION with (id: 2, status: timeout)	Friday 14 May 2004 16-58-13

Figure 8: Scenario Unfolds 6

In figure 8, we see that the 1 month timer for the obligation bearing on the service customer to pay for service has expired: this moves the SLA into a "Terminable" state – the Service Provider is permitted to terminate the SLA.

State History. SLA #4, Customer:Stirling Efforts, Template:Mail Service, Name:A Stirling.

Occurrence	Date/Time
STATE: Ok	Saturday 3 April 2004 16-58-13
STATE: Service Violation	Tuesday 13 April 2004 16-58-13
INPUT EVENT: SERVICE VIOLATION with (slo: 1)	Tuesday 13 April 2004 16-58-13
OUTPUT EVENT: OBLIGATION with (id: 0, bearer: provider, actions: resolve breach with (slo: 1), deadline: not specified)	Tuesday 13 April 2004 16-58-13
STATE: Ok	Tuesday 13 April 2004 17-28-13
INPUT EVENT: SERVICE RESTORATION with (slo: 1)	Tuesday 13 April 2004 17-28-13
INPUT EVENT: OBLIGATION with (id: 0, status: fulfiled)	Tuesday 13 April 2004 17-28-13
STATE: Provider Payment Outstanding, Customer Payment Outstanding	Wednesday 14 April 2004 16-58-13
OUTPUT EVENT: OBLIGATION with (id: 1, bearer: provider, actions: refund money with (amount: 25.00), deadline: before end of business day)	Wednesday 14 April 2004 16-58-13
OUTPUT EVENT: OBLIGATION with (id: 2, bearer: Stirling Efforts, actions: pay for service with (amount: 50.00), deadline: 1 month)	Wednesday 14 April 2004 16-58-13
STATE: Customer Payment Outstanding	Wednesday 14 April 2004 17-8-13
INPUT EVENT: OBLIGATION with (id: 1, status: fulfiled)	Wednesday 14 April 2004 17-8-13
STATE: Terminable	Friday 14 May 2004 16-58-13
INPUT EVENT: OBLIGATION with (id: 2, status: timeout)	Friday 14 May 2004 16-58-13
STATE: Terminated	Saturday 15 May 2004 16-58-13
INPUT EVENT: TERMINATE AGREEMENT	Saturday 15 May 2004 16-58-13

Figure 9: Scenario Unfolds 7

In figure 9, we see that, in keeping with the Service Provider being permitted to terminate the service, they do so: the SLA moves into a "Terminated" state.

8. Related Work

There have been many diverse research contributions that have utilised the Event Calculus (EC) for the purpose of reasoning over the effects of events on a logic theory. Those closest to the topics of this paper include [8-10].

There has been a good deal of research concerning the representation of contracts for performance monitoring. In [4] Daskalopulu discusses the use of Petri-nets for contract monitoring, and assessing contract performance. Her approach is best suited for contracts which can naturally be expressed as protocols. One particular desirability of using Petri-nets is that they naturally facilitate analysis. In the context of contract representation, an example would be to show that a contract will always terminate in a favourable state for one, or more, contract parties. It is possible, however, to carry out analysis of this nature using the formalism described here. Moreover, our representation has many advantages over Petri-nets (some of which are as a result of a rule-based approach).

In [11] Milosevic and colleagues attempt to identify the scope for automated management of e-contracts; including: contract drafting, negotiation and monitoring. In [12] Abrahams defines the EDEE architecture (E-commerce application Development and Execution Environment). Abrahams proposes *Event-Condition Obligation* rules for handling occurrences. *Prima facie* obligations are derived from the rules, where subsequent obligation choice decides which of these apply, and action choice decides which of those that apply will be fulfilled. In [13] Grosof and colleagues have sought to address the representation of business rules for e-commerce contracts. For this purpose, they have developed the SWEET (Semantic WEb Enabling Technology) toolkit, which enables communication of, and inference for, e-business rules written in RuleML. These approaches demonstrate many common themes with our approach.

9. Conclusions

In this work, we have proposed a formalisation of the Event Calculus in XML, called *ecXML*. We have informally shown its application to the representation of contracts to facilitate automated tracking of contract state for performance monitoring. We have grounded our discussion in the domain of SLAs for Utility Computing (UC), and have briefly presented how an example UC SLA could be represented.

Through using EC, we are able to extract information regarding which normative relations are initiated, and what values contract variables have, for arbitrary times (in the past, or present), according to a supplied event narrative. It is also possible to simulate the effects on contract state of a hypothetical event narrative, which we have found useful for carrying out prediction.

An inherent desirability of using EC is that the computation of tracking contract state – in the context of an event narrative – is *externalised* as a separate component, rather than buried within an

implementation for contract monitoring. This promotes better modularisation and makes for simplified code maintenance. Also, as a consequence, it means that the state tracking component may be re-used for a range of automated reasoning tasks for which it is appropriate to track state.

A comprehensive Java-based implementation of a generic EC reasoning component, called the Event Calculus State Tracking (ECSTA) architecture has been developed. In fact, *ecXML* can be seen as the *language of the machine*, and the implementation is capable of supporting any contract language that might be defined, so long as it has a tractable mapping to *ecXML*. All that is required to support a different language is the writing of a translator plug-in, which outputs *ecXML*. The ability to support multiple languages is an example of the re-use of the *ecXML* state tracking component.

ecXML has been evaluated against tens of SLAs, which are considered to be representative for UC. We have found it to be sufficient for facilitating contract tracking (as defined in this paper) for these SLAs. We have also designed our implementation to be capable of supporting a high number of contracts simultaneously and to support event narratives with a very large number of events. We have optimised the implementation for querying, and have found it to work extremely efficiently. In the future, it is our intention to evaluate the sufficiency of *ecXML* at facilitating contract tracking for other sorts of SLAs, and for contracts from other domains.

The work described herein represents a small part of a larger effort considering a unifying approach to the management and utilisation of contracts, policies and business rules at all levels of a business enterprise, including: management of IT infrastructure and hardware, management of business processes using business rules authored by business managers and analysts, and management of agreements between trading partners. For more information concerning this work, see [14].

10. References

[1] Hewlett-Packard (www.hp.com). "*HP Utility Data Centre - Technical White Paper*". October, 2001.

[2] Buco MJ, Chang RN, Luan LZ, Ward C, Wolf JL, Yu PS. "*Utility computing SLA management based upon business objectives*". In *IBM Systems Journal*, **43(1)**:159-78, 2004.

[3] J.J.Lee, R.Ben-Natan. "*Integrating Service Level Agreements: Optimising Your OSS for SLA Delivery*". Wiley, New York. 2002.

[4] A.Daskalopulu. "*Modelling Legal Contracts as Processes*". In Proceedings of *11th International Conference and Workshop on Database and Expert Systems Applications*, p. 1074-9. IEEE C. S. Press.

[5] M.Salle, C.Bartolini. "*Management by Contract*". In Proceedings of *IEEE/IFIP Network Operations and Management Symposium (NOMS 2004)*, 19-23 April 2004, Seoul, Korea.

[6] A.D.H.Farrell. "*Logic-based formalisms for the representation of Service Level Agreements for Utility Computing*". Master's thesis, Imperial College, London, U.K., 2003.

[7] R.Kowalski, M.Sergot. "*A Logic-Based Calculus of Events*". In *New Generation Computing*, **4**:67-95, 1986.

[8] A.Artikis. "*Executable Specification of Open Norm-Governed Computational Systems*". PhD thesis, Imperial College, London, U.K., 2003.

[9] A.K.Bandara, E.C.Lupu, A.Russo. "*Using Event Calculus to Formalise Policy Specification and Analysis*". In Proceedings of *4th IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2003)*, Lake Como, Italy, 2003.

[10] B.S.Firozabadi, M.Sergot, O.Bandmann. "*Using Authority Certificates to Create Management Structures*". In Proceedings of *Proceedings of Security Protocols, 9th International Workshop*, London, UK, April 2001.

[11] O.Marjanovic, Z.Milosevic. "*Towards Formal Modelling of e-Contracts*". In Proceedings of *Fifth IEEE International Enterprise Distributed Object Computing Conference*, Seattle, USA, September, 2001.

[12] A.S.Abrahams. "*Developing And Executing Electronic Commerce Applications with Occurrences*". PhD thesis, Cambridge University, 2002.

[13] B.N.Grosof, Y.Labrou, H.Y.Chan. "*A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML*". In M.P.Wellman, editor, Proceedings of *1st ACM Conf. on Electronic Commerce (EC-99)*, Denver, CO, USA, November 1999. ACM Press, New York, NY, USA.

[14] *http://www.doc.ic.ac.uk/~adf02/phd*.