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

Andrew D.H. Farrell, David Trastour, Athena Christodoulou
HP Laboratories Bristol
HPL-2004-20
February 16th , 2004*

E-mail: {andrew.farrell, david.trastour, athena.christodoulou}@hp.com

utility computing, service-level agreement, contract, performance monitoring, contract state tracking, event calculus

Utility Computing (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. A crucial stage in the life-cycle of contracts (such as SLAs) is their automated performance monitoring at run-time. In this work, we define an ontology to capture aspects of SLAs that are pertinent to performance monitoring, and generalise these aspects so that the ontology may be applicable to other contract domains. The ontology is formalised as an XML-based language, called CTXML (contract tracking XML). The semantics for CTXML are presented in terms of a computational model based on the Event Calculus.

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

Andrew D H Farrell, David Trastour, Athena Christodoulou
HP Labs
Filton Road, Stoke Gifford, Bristol, BS34 8QZ, United Kingdom
{andrew.farrell,david.trastour,athena.christodoulou}@hp.com

Abstract

Utility Computing (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. A crucial stage in the life-cycle of contracts (such as SLAs) is their automated performance monitoring at run-time.

In this work, we define an ontology to capture aspects of SLAs that are pertinent to performance monitoring, and generalise these aspects so that the ontology may be applicable to other contract domains. The ontology is formalised as an XML-based language, called CTXML (contract tracking XML). The semantics for CTXML are presented in terms of a computational model based on the Event Calculus.

1. Introduction

Utility Computing (UC) [11] 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 predicates within Service-Level Agreements (SLAs). SLAs are essential for formalising the objectives of a UC service, and to manage expectations [13].

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 [6]. In our view, performance monitoring is (at least) concerned with 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 [16]. 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 contracts, thus far, can be largely characterised in one of two ways: (i) As general-purpose reasoning frameworks that (mainly) have not been applied in actual, deployed systems – some of which are described in section 7; or (ii) In the case of SLAs, as being fairly limited in capability [5]. 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. Firstly, (in section 2), the conceptualisation of contracts that has been used in this work is presented; followed (in section 3) by an example contract (namely, an SLA for a UC scenario), used to ground our discussions. Then, (in sections 4 and 5), a description of the contract tracking ontology, developed in this work, and its semantics are given. The paper proceeds to describe implementation and related work (in sections 6 and 7), and concludes (in section 8).

2. Contracts conceptualised

It is a useful abstraction to consider that contracts (such as SLAs) are comprised of norms. A norm may be defined as: "*a principle of right action binding upon the members of a group and serving to guide, control, or regulate proper and acceptable behaviour*" [1].

In our work, we consider norms to be templates, which can be instantiated to yield (normative) relations that hold between contract parties. An example might be the norm: ‘a service consumer is obliged to pay for service provision’. When instantiated, it yields a relation that now holds between a service consumer and provider – that is, that the consumer is obliged to pay for service provision. In time, the norm may be instantiated again, creating a further relation. In fact, it may be that the first relation still persists (i.e., the consumer is yet to fulfil their original obligation to pay), meaning that there is now more than one relation pertaining to the same norm.

In this presentation, it will be assumed that at most one relation pertaining to a norm may exist at any time. This is for convenience; the general case is treated in [7]. For simplicity, the existence of a relation pertaining to a norm will be described as the norm being active, and the lack of an extant relation will be described as the norm being inactive.

Crucially, it is considered here that: (i) a contract expresses norms between contract parties, whereby the actual state of the contract at any time is determined by which norms are active; (ii) norms within a contract will define the effects on the contract state of events that are presented to the contract (contract events).

3 Example Contract

In this paper, we use the following Mail Service UC SLA in order to ground our discussions.

- 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.
- 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 $u<=s$
- Whenever $u>s+e$, where e is a level of tolerance in GBytes, SC will not be able to receive emails.
- In the case that the mail service is unavailable, SP is obliged to restore it within $t_{recover}$ minutes. SP will pay $\$p_{recover}$ for every $t_{recover}$ minutes that it is unavailable. SP is obliged to pay any penalties to SC within a month of their accruement.
- 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.

4 Contract Tracking Ontology

Figure 1 presents the contract tracking ontology that has been devised in this work. The ontology has also been formalised as an XML-based contract language, called *CTXML* (*c*ontract *t*racking *XML*).

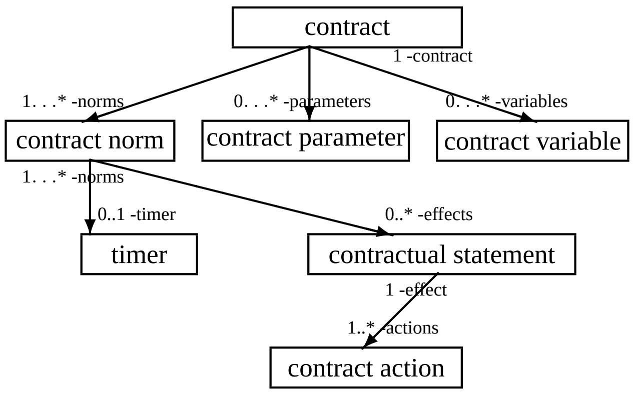

Figure 1. Contract Tracking Ontology

With reference to the figure, a contract is conceived as consisting of one or more contract norms, as well as zero or more contract parameters – which allow for the customisation of a contract for a particular instantiation context – and zero or more contract variables – which are used to maintain live, numerical contract state (their use is normative in that it is agreed by all parties when the contract is signed).

A contract norm may be considered as corresponding to one of many (Holfeldian-inspired) normative concepts, including (non-exhaustively): obligation, privilege, entitlement or power (see [7]). A contract norm will usually specify one or more of the following:

- One or more contractual statements, which define the effect of contract events (pertaining to the norm) on the contract. It is considered that a norm is *triggered* by a contract event that pertains to it.
- A timer for the norm, which is possibly recurrent.
- One or more parameters. That is, a contract norm may be parameterised, whereby a contract event that pertains to the norm may pass data to the contractual statements contained within the norm.

In our work, we have considered the following conceptualisations of contract norms to be useful for the representation of contracts:

- Contract management norms, of which we define two types: Periodic and Event
- Obligation norms
- Privilege norms

In turn, contract management norms (CMNs) represent the principal means of defining the effects of contract events on contract state. They contain a single contractual statement which is executed when the norm is triggered. Note that a CMN will either be conceptualised as an event CMN, or a periodic CMN. An event CMN is triggered by an external event. Contrastingly, a periodic CMN describes a (possibly recurring) timer which triggers the norm.

An obligation norm is concerned with an obligation that bears on a party to perform one or more (non-contractual) actions. It will typically contain a contractual statement that specifies the effects on the contract in case of violation of the obligation norm, and a contractual statement that specifies the effects on the contract in case of fulfilment of the norm. It is considered that such a norm is triggered by violation and fulfilment events. An obligation norm will also specify a timer for the actions associated with the obligation to be performed by the pertinent party. Like a CMN, an obligation norm may be parameterised.

A privilege norm is concerned with (non-contractual) actions that a party is permitted to perform. It is considered illegal behaviour for a party to carry out a (non-contractual) action for which it does not have the privilege. (As a consequence, there does not exist a need for explicit prohibition norms). Furthermore, a privilege norm is considered to be a vested privilege in that other parties undertake that they will not attempt to prevent the bearer of the privilege from exercising it.

Examples of these norms represented in *CTXML* for the Mail Service SLA (introduced in section 3) are now presented. Note that, in the sequel, contract events take the form: `(norm, qualification, parameters)`, where the first argument is the unique pertaining norm, the second argument is a qualification for the event – which names the contractual statement in the norm to be executed, and the third argument is a list of event parameters that are passed on to the contractual statement.

- A periodic CMN, `pcmn3`, defining its (recurrent) timer as being specified by the `pcnm3timer` timer clause; and specifying its (single) contractual statement to be: `pcmn3timeout` which is executed whenever `(pcmn3, timeout, [])` contract events occur. These events are generated internally according to `pcnm3timer`.

  ```
  <contractnorm id="pcmn3" timer="pcnm3timer">
    <csref name="timeout" id="pcmn3timeout"/>
  </contractnorm>
  ```

  This norm in part facilitates: "SP will pay $\$p_{recover}$ for every $t_{recover}$ minutes that it is unavailable" in the example SLA.
- An event CMN, `ecmn1`, specifying a single contractual statement: `ecmn1trigger` which is executed whenever `(ecmn1, trigger, [Charge])` contract events occur; and denoting that it is parameterised with a single parameter: `Charge`.

  ```
  <contractnorm id="ecmn1">
    <csref name="trigger" id="ecmn1trigger"/>
    <para name="Charge"/>
  </contractnorm>
  ```

  This norm in part facilitates: "SP will charge SC $\$c_1$ for each GByte over *s*, calculated daily, until $u{<=}s$" in the example SLA.
- An obligation norm, `o2`, defining its (one-off) timer as being specified by the `o2timer` timer clause; contractual statements for non-fulfilment (violation) and fulfilment of the obligation within the time specified by `o2timer` as being specified by the `o2violation` and `o2fulfilment` contractual statements, respectively – executed in response to `(o2, violation, [Charge])` and `(o2, fulfilment, [Charge])` contract events; and denoting that it is parameterised with a single parameter: `Charge`.

  ```
  <contractnorm id="o2" timer="o2timer">
    <csref name="violation" id="o2violation"/>
    <csref name="fulfilment" id="o2fulfilment"/>
    <para name ="Charge"/>
  </contractnorm>
  ```

  This norm in part facilitates: "SP is obliged to pay any penalties to SC within a month of their accruement" in the example SLA.
- A privilege norm, `p1`.

  ```
  <contractnorm id="p1"/>
  ```

  This norm in part facilitates: "If SC fails to pay within the given time, SP may terminate the mailbox service without notice" in the example SLA.

A timer clause is used to specify (a recurrent, or one-off) timer for periodic CMNs and obligation norms. Such a clause consists of one or more `run` clauses, which each specify a certain number of iterations of a particular timer duration. If the number of iterations is not explicitly specified (as in the example below), the run is considered to be *indefinitely recurring* according to the specified timer duration. An example of such a clause is now given, from the *CTXML* representation of the Mail Service SLA, for the timer used for contract norms: `pcmn1` and `pcmn2`. Here, the clause simply says that the timer will be indefinitely recurring with a period of 1 month.

```
<timer id="pcmn1pcmn2timer">
     <run><dur val="P1M"/></run>
</timer>
```

This clause in part facilitates: "SP will pay $\$p_{recover}$ for every $t_{recover}$ minutes that it is unavailable" in the example SLA.

A contractual statement clause comprises a list of contract actions, which are actions to be performed on the contract, in response to contract events. A contract action may be one of the following clauses (where the first three are considered to be atomic contract actions):

- **activate** – activates contract norm: `norm`. In *CTXML*: `<activate id="norm">` *activation parameters* `</activate>`
- **deactivate** – deactivates contract norm: `norm`. In *CTXML*: `<deactivate id="norm"/>`
- **assign** – assigns a numerical value, given by *expr*, to contract variable `cvar`. In *CTXML*: `<assign id="cvar">` *expr* `</assign>`
- **ifcond** – specifies a conditional contract action. In *CTXML*: `<ifcond then="..." else="..."/>`

An example of a contractual statement, with associated contract actions, represented in *CTXML* for the Mail Service SLA is now presented.

```
<contractualstmt id="pcmn1timeout">
   <ifcond then="ifcond1then">
     <gt><value id="vPenalty"/><num val="0"/></gt>
   </ifcond>
</contractualstmt>
```

```
<contractualstmt id="ifcond1then">
   <activate id="o2">
     <apara name="Charge"><value id="vPenalty"/></apara>
   </activate>
   <assign id="vPenalty"><num val="0"/></assign>
</contractualstmt>
```

Here, the `pcnm1timeout` contractual statement consists of a single contract action – an `ifcond`. The `ifcond` action specifies a contractual statement, `ifcond1then`, to be performed if the condition of the `ifcond` holds. (It is possible for `ifcond` actions to also specify a contractual statement to be performed if the condition does not hold). The condition of the `ifcond`, in the example, stipulates contract variable `vPenalty` be greater than `0`. The `ifcond1then` contractual statement consists of a couple of contract actions – an `activate` action (for activating parameterised obligation norm `o2` with activation parameter `Charge` assigned to the current value of contract variable `vPenalty`), and an `assign` action (for resetting the value of the contract variable).

Finally, a contract may specify a list of initialising operations (itself a contractual statement – constrained to contain just `activate` operations) which are carried out on the contract when it is instantiated. Note that all contract norms are inactive, by default. As such, any norm that is required to be initially active should have a corresponding `activate` operation specified in this list.

4.1 Specialisation to SLA context

It useful to explicate an additional concept, which has been utilised within this work, that is specific to the context of representing SLAs. The concept is a service-level norm (SLN), which is a variation of an event CMN. An SLN pertains to a level of service, which is agreed between parties when the containing SLA is negotiated. A 'service level predicate', corresponding to the SLN, defines the level of service that must be upheld throughout the lifetime of the SLA.

In terms of the contract tracking ontology, an SLN defines up to two contractual statements. One that specifies contract actions that are to be performed in case of violation of the (service level predicate pertaining to the) SLN, and another that specifies contract actions that are to be performed in case of restoration of the SLN. It is considered that such a norm is triggered by violation and restoration contract events.

An example of an SLN represented in the Mail Service SLA is now presented, where it is triggered by `(sln1, violation,_)` and `(sln1, restoration, _)` contract events.

```
<contractnorm id="sln1">
  <csref name="violation" id="sln1viol"/>
  <csref name="restoration" id="sln1rest"/>
</contractnorm>
```

This clause in part facilitates: "The Service Provider. . .a mail-storage facility of up to *s* GBytes" and "In the case of unavailability of the mail service. . ." in the example SLA.

5 Semantics

The semantics attributed to the contract tracking ontology are presented in terms of how the execution of contractual statements, in response to contract events, changes the state of the contract. This is achieved by describing the computational model for determining the state of norms, in the context of a narrative of contract events, according to the contractual statements contained within a contract. The computational model that is described here is inspired by the Event Calculus (EC) [12].

5.1 Event Calculus overview

There are many variations on the Event Calculus (EC). In the sequel, we define an XML formalisation of a simplified form of the version described in [17], called *ecXML*. In this formalisation, contract events, which take the form: `(norm,qualification,parameters)`, are written as: `<event id="(norm,qualification)">`*parameters*`</event>`.

A contract in *ecXML* is a conjunction of:

- A finite set of **initially** clauses of the form:

  ```
  <initially>
    <fluent id="F">...</fluent>
  </initially>
  ```

  meaning that (boolean) fluent *F* holds initially. (A fluent is a property of a domain which can be attributed a value, where the value of the fluent is able to change over time). Multi-valued fluents are assigned an initial value using similar clauses.

- A finite set of **happens** clauses of the form:

  ```
  <happens time="T">
    <event ...>...</event>
  </happens>
  ```

  meaning that the given event happened at time *T*

- A finite set of **initiates** clauses of the form:

  ```
  <initiates>
    <event ...>...</event>
    <fluent id="F">...</fluent>
     condition
  </initiates>
  ```

  meaning that the given event initiates fluent *F* if *condition* holds. Similar clauses can be written giving how multi-valued fluents are initiated.

- A finite set of **terminates** clauses of the form:

  ```
  <terminates>
    <event ...>...</event>
    <fluent id="F">...</fluent>
     condition
  </terminates>
  ```

  meaning that the given event terminates fluent *F* if *condition* holds. Similar clauses can be written giving how multi-valued fluents are terminated.

Additionally the following axioms (for which a full XML formalisation is neither necessary nor appropriate) are also defined for *ecXML*:

- `holds(`*F,T*`)` **if** `initiated(`*F,T1,T*`)` **and not** `terminated(`*F,T1,T*`)` meaning that fluent *F* holds at time *T* **if** fluent *F* is initiated at time *T1* before, or at, time *T* **and** it is not terminated at a time later than *T1* but before, or at, time *T*. A similar axiom exists for multi-valued fluents. *Note that* it is the *holds* axioms which provide the means for querying the state of a contract at any time, and thus which realise the primary purpose of applying an EC-based semantics.
- `initiated(`*F,0,_*`)` **if**

```
<initially>
  <fluent id="F">...</fluent>
</initially>
```

  meaning that fluent *F* is initiated at time `0` **if** fluent *F* holds initially (as determined by any extant *ecXML* `<initially>` clause for *F* in the contract). A similar axiom exists for multi-valued fluents.
- `initiated(`*F,T1,T*`)` **if** `happens(`*E,T1*`)` **and** $T \geq T1 > 0$ **and**

```
<initiates>
  <event ...>...</event>
  <fluent id="F">...</fluent>
  ...
</initiates>
```

  meaning that fluent *F* is initiated at time *T1* before, or at, time *T*, and greater than `0`, **if** an event *E* happens at *T1* **and** *E* initiates *F* (as determined by any extant *ecXML* `<initiates>` clauses for *F* in the contract). A similar axiom exists for multi-valued fluents.
- `terminated(`*F,T1,T*`)` **if** `happens(`*E,T2*`)` **and** $T \geq T2 > T1$ **and**

```
<terminates>
  <event ...>...</event>
  <fluent id="F">...</fluent>
  ...
</terminates>
```

  meaning that fluent *F* is terminated at time *T2* later than *T1* and before, or at, time *T* **if** an event *E* happens at *T2* **and** *E* terminates *F* (as determined by any extant *ecXML* `<terminates>` clauses for *F* in the contract). A similar axiom exists for multi-valued fluents.

5.2 Event Calculus based semantics

As stated, the Event Calculus (EC) is used to provide a computational model for *CTXML* contractual statements. This is achieved by defining a mapping between contractual statements and expressions in EC. Note that, a contractual statement will have a distinct mapping for each contract norm to which it pertains.

Recall from section 4 that a contractual statement consists of the following types of contract actions: `activate`, `deactivate`, `assign`, and `ifcond`. The mapping for the first three contract actions – the atomic actions – is now presented.

- `<activate id="norm">`*activation_parameters*`</activate>` is mapped to:

```
<initiates>
  <event id="(pnorm,qualification)"/>
  <fluent id="norm"> activation_parameters</fluent>
</initiates>
```

  where `(pnorm,qualification)` is the event name that triggers the contractual statement with id: `qualification` within contract norm: `pnorm` and `norm` is the norm activated with the given *activation_parameters*.
- `<deactivate id="norm"/>` is mapped to:

```
<terminates>
  <event id="(pnorm,qualification)"/>
  <fluent id="norm"/>
</terminates>
```

  where `norm` is the norm deactivated.
- `<assign id="cvar">`*expr*`</assign>` is mapped to:

```
<initiates>
  <event id="(pnorm,qualification)"/>
  <mvfluent id="cvar"> expr</mvfluent>
</initiates>
```

  where `cvar` is the contract variable assigned to *expr*.

`Ifcond` actions conceptually take the form: $\triangle \rightarrow \theta_{then} : \theta_{else}$. $\triangle$ is a boolean condition on the state of norms (inactive or active) in the contract and contract events. θ_{then} is a contractual statement that is executed should the condition hold when the `ifcond` is executed. θ_{else} is a contractual statement that is executed if the condition fails to hold. In mapping `ifcond` actions to EC, $\triangle$ becomes an additional condition placed on each contract action in θ_{then}; and **not** $\triangle$ becomes an additional condition placed on each contract action in θ_{else}. Generally speaking, there may be an arbitrary nesting to an `ifcond` action meaning that any atomic `activate`, `deactivate`, or `assign` actions specified within may be subject to a number of boolean conditions: $\Pi_1, \ldots, \Pi_n$, where for any boolean condition $\triangle_i$ within an `ifcond`, Π_i represents either $\triangle_i$ or **not** $\triangle_i$.

An `<activate id="norm">`*activation_parameters*`</activate>` contract action specified within an `ifcond` is mapped to:

```
<initiates>
  <event id="(pnorm,qualification)"/>
  <fluent id="norm"> activation_parameters</fluent>
  condition
</initiates>
```

Here an additional *condition* clause specifies that the contract action will only be applied if $\Pi_1, \ldots, \Pi_n$ all hold. Other atomic actions similarly have an additional *condition* clause when mapped.

Examples of such mappings for the Mail Service SLA are now presented.

- A violation event for `sln1` initiates (or activates) `pcmn3`, and terminates (or deactivates) `sln1_ok`.

```
<initiates>
  <event id="(sln1,violation)"/>
  <fluent id="pcmn3"/>
</initiates>
<terminates>
  <event id="(sln1,violation)"/>
  <fluent id="sln1_ok"/>
</terminates>
```

State name	Characterisation
`sNormal`	`-p1, -p2, +sln1_ok`
`sUnavailable`	`-sln1_ok`
`sRefuseRecMail`	`+p2`
`sSCToPay`	`+o3`
`sSPToPay`	`+o2`
`sTerminable`	`+p1`

Table 1. Definition of (equivalence classes for) states

- A timeout event for `pcnm1` initiates the assignment of (contract variable) `vPenalty` to `0` if the condition `vPenalty` greater than 0 holds.

```
<initiates>
  <event id="(pcnm1,timeout)"/>
  <mvfluent id="vPenalty">
    <num val="0"/>
  </mvfluent>
  <gt><value id="vPenalty"/><num val="0"/></gt>
</initiates>
```

The mapping of the (possibly extant) contractual statement containing initialising operations for the contract (which is constrained to contain only `activate` is simply (where `norm` is the norm activated):

`<activate id="norm">`*activation_parameters*`</activate>` is mapped to:

```
<initially>
  <fluent id="norm"> activation_parameters</fluent>
</initially>
```

Also, there is a mapping associated with the initialisation of contract variables in *CTXML*, thus (where `cvar` is the contract variable assigned):

`<contractvar id="cvar">`*expr*`</contractvar>` is mapped to:

```
<initially>
  <mvfluent id="cvar"> expr</mvfluent>
</initially>
```

Finally, there are other aspects of *CTXML*, such as timer clauses, that have mappings to *ecXML* that are not presented here for reasons of brevity.

6 Implementation

The Event Calculus-based computational model defined here has been implemented in two ways. Both implementations provide a query-interpreter for determining, at runtime, the state of contracts. Components outside the representation of the contract will post contract events via the query-interpreter, and be informed of (and be able to query the contract for) information relating to contract state.

- A comprehensive Prolog (Sicstus v3.10.1) implementation of a query-interpreter for querying contracts written in a Prolog version of *ecXML*

Event narrative (norm, qualification, parameters)	Activates	Deactivates	State History
`<contract initiation>`	`sln1_ok ...`		`sNormal`
`(sln1,violation,[])`	`pcmn3, o1`	`sln1_ok`	`sUnavailable`
`(sln1,restoration,[])`	`sln1_ok`	`pcmn3`	`sNormal`
`(ecmn2,trigger,[true])`	`p2`		`sRefuseRecMail`
`(ecmn2,trigger,[false])`		`p2`	`sNormal`
`(pcmn1,timeout,[])` `(pcmn2,timeout,[])`	`o2,o3`		`sNormal,` `sSCToPay` `sSPToPay`
`(o2,fulfilment,[$12.90])`		`o2`	`sNormal` `sSCToPay`
`(o3,timeout,[$20.50])`	`p1`	`o3`	`sTerminable`

Table 2. State History of SLA

- A comprehensive Java implementation of a query-interpreter for querying contracts written in either *ecXML* or the higher-level *CTXML*

Part of the API supported by these implementations is now presented.

- `void get_output_events(Es,T)` – gets, `Es`, the output events that the contract generates at time `T`
- `void get_states(S)` – gets, `S`, the possible states of the contract; and `void get_state_history(H,T)` – gets, `H`, a history of states that the contract has been in, up to and including time `T`
- `boolean active_at(N,T)` – gives whether a norm, `N`, holds at a time `T`; and `double value_at(V,T)` – gives the value of a contract variable, `V`, at time `T`
- `void add_events(Es)` – used to add an event narrative, `Es`, specified in *ecXML*, to the contract; `void add_future_events(Es, T)` – used to add a future event narrative, `Es`, to the contract; and `void delete_future_asserted_events()` – used to remove all future events

Additionally, there is a means, provided for by the contract tracking ontology of defining equivalence classes for collections of contract states. It is the names of these equivalence classes that procedures such as `get_state_history/2` return for names of states. Notably, it is also possible to query possible future states by asserting the occurrence of future events (using `add_future_events/2`) and querying the resulting state. This is extremely useful in the SLA context, for calculating the utility (that is, worth) to the service provider of possible future service provisions [16], for instance.

6.1 Example Querying

In tables 1 and 2, a brief example of querying is shown using `get_state_history/2`. Firstly, in table 1, the equivalence classes for states that have been defined over the Mail Service SLA are shown. In the table, the definition of a particular state is characterised by which norms are active (prefixed with a '+') and which norms are inactive (prefixed with a '-').

Using this information, and looking at table 2, it is possible to see that the information returned by `get_state_history/2` is appropriate according to the event narrative and its consequences in terms of norms activated and deactivated.

7 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 the following. In [3], Artikis describes the effective representation in EC of: a variation on the Contract-Net protocol, an argumentation protocol based on Brewka's reconstruction of Rescher's Theory of Formal Disputation (RTFD), and the NetBill protocol. In [4], Bandara and colleagues develop methods for performing analysis and refinement of policy specifications. To this end, they formalise an EC-based notation for representing both policy and system behaviour specifications. In [9], Firozabadi and colleagues develop an EC-based framework for issuing privileges to agents in a community, through 'declaration' and 'revocation' authority certificates. It makes a distinction between the time a certificate is issued, or revoked, and the time for which the associated privilege is created, or discharged, enabling certificates to have prospective and retrospective effects.

There has been a good deal of research concerning the representation of contracts for the purpose of reasoning over, and monitoring, them at run-time. In [6] Daskalopulu discusses the use of Petri-nets for contract monitoring, and assessing contract performance. Their approach is best suited for contracts which can naturally be expressed as protocols. In [15] Milosevic and colleagues attempt to identify the scope for automated management of e-contracts; including: contract drafting, negotiation and monitoring.

In [2], 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 [10], 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 extended with Situated Courteous Logic Programming. The approach of Grosof and colleagues, has some similarities to our work. For instance, our approach is capable of facilitating conflict resolution in the way described by Grosof [10]. Furthermore, 'procedural attachment' [10] is realised here in a simple, but effective, way.

It is worth noting that, in the work of Grosof and colleagues, a sharp distinction is made between "*the representational mechanism for communicating contract rules, and the actual rule execution mechanisms employed by participating agents. Our concern here is with the former...*" [10]. Here there is a significant difference their approach and that employed here. We are very much concerned with the actual execution mechanisms of agents in considering how they might maintain live representations of contracts. Such a concern demands the use of a computational model that can effectively track the state of a contract over time – a facility that is (ostensibly) lacking in the work of Grosof. In fact, Grosof's work would apparently dovetail nicely with that of Leite and colleagues [14] who have suggested update semantics for *generalised logic programs*. The evolution of contract state would be manifested as a progression of logic programs, where the state of a contract at any particular time would be given by the (stable model or well-founded) semantics of the individual logic program most recent to that time. Contract events would be processed as program updates, expressed using the update language specified by Leite. In the approach presented here, the Event Calculus is used in order to facilitate reasoning of contracts over time.

8 Conclusions

In this work, we have proposed an ontology, formalised as an XML-based language, *CTXML*, to facilitate the automated tracking of contract state (that is, the active contract norms that hold between contract parties).

We have used the Event Calculus to provide a computational model for *CTXML*. An inherent desirability of such an approach is that it removes any rigid coupling that might otherwise be introduced between the ontology and its implementation. That is, with this approach, we implement generic state tracking, through EC, as a separate component to supporting the particular ontologies defined here. This means that for any ontology that would naturally lend itself to having an EC-based semantics (namely, any for which state tracking would be part of the desired reasoning using the ontology), the same state tracking component can be used. This approach thus promotes the simple support of multiple contract languages. In fact, for each distinct contract language that would be supported by our implementation, only a component that parses contracts written in such a language would need to be written. Such components would output an internalised version of *ecXML*, as do the parsing components that have been written for *ecXML* and *CTXML*.

A comprehensive Java-based implementation of a generic EC reasoning component, along with query-interpreters for *CTXML* and *ecXML* have been developed. The implementation and *CTXML* ontology have been evaluated against tens of SLAs, which are considered to be representative for UC. We have found the ontology 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 *CTXML* at facilitating contract tracking for other sorts of SLAs, and for contracts from other domains. We also intend to enhance the functionality supported by our implementation, in order to make it useful for supporting the calculation of contract utility [16]. Some of these enhancements will include:

- Ability to reason over possible future states of a contract, while allowing the contract to be updated according to incoming contract events.

- Ability to manage *user-variables* that a particular contract party may define for the purposes of their own monitoring. These will not be contract variables as their use will not have been agreed between contract parties; in fact, they are unlikely to be transparent to other parties. These will be defined along with rules for describing how the values of these variables are updated.

- Ability to manage dynamic addition and deletion of contract clauses.

- Enhancement of our support of ‘procedural attachment’ [10]. Currently, our implementation will output events pertaining to which norms have been made active, or inactive, at a particular time, as well as generating events pertaining to changes in the values of contract variables. An added desirability is the capability of specifying rules for generating output events, such as ‘generate an output event when user-variable *x* has a value greater than *y*’.

References

[1] Merriam-Webster On-line Dictionary (www.m-w.com/cgi-bin/dictionary).

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

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

[4] A. K. Bandara, E. C. Lupu, and 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, June 2003.

[5] R. Boreham and M. Morciniec. Contract Monitoring. *HP Labs Technical Report: HPL-2002-265*.

[6] A. Daskalopulu. Modelling Legal Contracts as Processes. *11th International Conference and Workshop on Database and Expert Systems Applications, IEEE C. S. Press*, pages 1074–1079, 2000.

[7] 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.

[8] A. D. H. Farrell, M. J. Sergot, D. Trastour, and A. Christodoulou. Performance Monitoring of Service-Level Agreements for Utility Computing using the Event Calculus and CTXML (Extended version). Available on request.

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

[10] B. N. Grosof, Y. Labrou, and 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.

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

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

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

[14] J. Leite, J. Alferes, and L. Pereira. Multi-dimensional Dynamic Logic Programming. In F. Sadri and K. Satoh, editors, *Proceedings of the CL-2000 Workshop on Computational Logic in Multi-Agent Systems (CLIMA’00)*, July 2000.

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

[16] M. Salle and C. Bartolini. Management by Contract. *HP Labs Technical Report: HPL-2003-186*.

[17] M. Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia, ISBN: 0262193841*. MIT Press, 1997.