USING THE EVENT CALCULUS FOR TRACKING THE NORMATIVE STATE OF CONTRACTS

ANDREW D. H. FARRELL, MAREK J. SERGOT

Department of Computing, Imperial College, London. SW7 2AZ. United Kingdom.
{andrew.farrell,m.sergot}@imperial.ac.uk

MATHIAS SALLÉ, CLAUDIO BARTOLINI

Hewlett-Packard Laboratories, Palo Alto, California, USA.
{mathias.salle, claudio.bartolini}@hp.com

In this work, we have been principally concerned with the representation of contracts so that their normative state may be tracked in an automated fashion over their deployment lifetime. The normative state of a contract, at a particular time, is the aggregation of instances of normative relations that hold between contract parties at that time, plus the current values of contract variables. The effects of contract events on the normative state of a contract are specified using an XML formalisation of the Event Calculus, called ecXML. We use an example mail service agreement from the domain of web services to ground the discussion of our work. We give a characterisation of the agreement according to the normative concepts of: obligation, power and permission, and show how the ecXML representation may be used to track the state of the agreement, according to a narrative of contract events. We also give a description of a state tracking architecture, and a contract deployment tool, both of which have been implemented in the course of our work.

1 INTRODUCTION

An increasingly important aspect in the life-cycle of contracts is the automated monitoring of the *normative state of contracts* while they are active. We define the normative state of a contract, at a particular time, to be the aggregation of instances of normative relations that hold between contract parties at that time, plus the values of contract variables at that time.

The term normative relation is used here as a generic term to refer to concepts such as obligation (of a buyer of goods to pay the agreed price to the seller by a certain date, say), various notions of permission, power (of a contracting party to change the terms of the contract, say), as well as other more complex normative concepts, such as right, directed obligation or relative duty, entitlement, and so on, as discussed below. A contract variable is a piece of numerical state whose value can change over the deployment lifetime of its containing contract. Its use will be normative in that it will have been agreed upon when the contract is formed.

In this paper, we describe a general approach to the tracking of state based on a version of the Event Calculus (EC), originally presented in [1]. Simply put, EC allows the expression of domain axioms which characterise how propositional properties of a domain (*fluents* in the AI terminology) change according to the occurrence of domain events. Various forms of reasoning can be undertaken using such a set of domain axioms, such as *planning* (a sequence of actions that will take the domain from an initial state to a goal state), *prediction* (where given an initial domain state, and a sequence of domain events, an *event narrative*, we seek a resulting domain state), and *postdiction* (where given a current domain state, and a set of a domain events, we seek an initial domain state) [2]. In this terminology, state tracking is a special case of prediction, except that we shall also want to have access to all intermediate states as well as the initial and final ones. The Event Calculus is presented in a logic programming framework, and is usually implemented using a logic programming language such as Prolog or using techniques from deductive databases. In this work, in order to facilitate integration with other components of a general contract state tracking architecture, we have constructed a Java implementation of the reasoning and used XML as a convenient file format. We call this version of the Event Calculus *ecXML*.

For the purpose of tracking the normative state of contracts, we use *ecXML* to express how arbitrary normative relations may change according to contract-related events. By arbitrary we mean that we are not restricted to any particular set of normative concepts. It is worth noting, however, that we have found the following concepts in particular to be useful in our work:

- *Permission*. Bentham (as presented in [3]) used the term *liberty* to mean: "*You have a right to perform whatever you are not under obligation to abstain from the performance of*". Bentham further distinguished two types of liberty: *naked liberty* to do action α – where others have the freedom to (attempt to) prevent α, and *vested liberty* – where others have an obligation not to prevent α. This notion of *vested liberty* is the sort of definition that we would attribute to (our use of) *permission*. Note also that, for convenience, we have assumed a *closed policy* [4]. That is, absence of a permission to perform an action is taken to be the same as presence of a prohibition to perform the action. This is not a necessary restriction, however.
- *Obligation*. Bentham (as presented in [3]) describes an obligation as being imposed by a legislator whenever a law of type *command* or *prohibition* is imposed:

  > "*…where the provision of the law is a command or a prohibition, [concerning the performance of an act], it creates an offence: if a command, it is the non-performance of the act that is the offence: if a prohibition, the performance…Moreover the law, in constituting an act an offence, is said to impose thereby an obligation on the persons in question not to perform it.*"

  In our work, we have found the concept of obligation to adhere to a command to be directly useful. Obligation to adhere to a prohibition, which is a sort of refrain, may also be useful, as may be an obligation to realise a state-of-affairs. The concept of *directed obligation* or *relative duty*, where one party is the bearer of an obligation owed to another counterparty, is also an important concept though it will not feature in the example contract to be discussed in this paper.

- *Institutionalised Power*. This concept, as proposed in [5], is a generalisation of the standard legal concept variously described in the literature as *legal power*, *legal capacity* or *legal competence*. It refers to the characteristic feature of all norm-governed organisations/institutions whereby designated agents are *empowered*, by the institution, to create or modify facts that have a conventional significance within that institution, usually by performing a special kind of act (such as when a priest performs a marriage or when a chair-person of a formal meeting declares the meeting closed). We find it helpful to employ a terminology, originally due to Searle, which makes a distinction between *brute* and *institutional* facts. Physical possession of an object is a brute fact, whereas ownership of the object is an institutional fact. Whether an agent owns something that it possesses depends on whether possession *counts as* ownership in the institution in question. Jones and Sergot [5] present a formalisation of institutionalised power, where within a particular institution, certain actions or states-of-affairs count as other kinds of actions or states-of-affairs. On this account, (institutionalised) power arises when an agent's performance of some *brute* action counts, within a given institution, as an action that creates an institutional fact.

In legal theory, the importance of the following distinction has long been recognised. There are three quite different notions [5,6]:

1. The power to create an institutional fact
2. The permission to exercise that power
3. The practical ability (physical capability, opportunity, know-how) to exercise that power.

The following example, due to Makinson and quoted in [5], may help to clarify the distinction. Consider the case of a priest of a certain religion who does not have permission, according to instructions issued by the ecclesiastical authorities, to marry two people, only one of whom is of that religion, unless they both promise to bring up the children in that religion. He may nevertheless have the power to marry the couple even in the absence of such a promise, in the sense that if he goes ahead and performs the ceremony, it still counts as a valid act of marriage under the rules of the same church even though the priest may be subject to reprimand or more severe penalty for having performed it. In this case the priest is *empowered* to marry a couple, but at the same time he may not be permitted to do so. It is commonplace, first to empower an agent to create a certain institutional fact, and then separately to impose restrictions on when that agent may exercise his power. In many practical examples, however, especially where a contract does not spell out in detail the precise means by which contracting parties exercise their powers (for example, to change the price or delivery time of purchased goods), it is often the case that power implies a (vested) permission to exercise it. We use the term *vested power* in these cases. (See [7,8] for examples of practical settings where power is separated from permission to exercise it.)

We have based the development of our approach and the *ecXML* language on the representation of a representative sample of agreements, of the order of tens of agreements, from the domain of web services and other service-oriented domains, such as Utility Computing [9]. In this paper, we use one such agreement for a mail service as an example contract to illustrate our work. Some examples from the mail service agreement, presented in section 2, are as follows.

- Obligation: *Service Provider will pay $p for every whole t minutes that it (the service) is unavailable.* We will say that when this norm is activated by the occurrence of a contract event it creates an instance of a normative relation of type obligation. It seems (to us) to be unnatural to conceive of this relation as a directed obligation owed by the service provider to the customer, though this is a possible reading.
- Permission & (vested) power: *SP may terminate the mailbox service without notice* (in certain circumstances). The action of *SP* terminating the mailbox service is considered to be an institutional action. To effect this action, *SP* may actually carry out some other action, such as removing *SC's* service record from a database collection recording active mailboxes. This action would be a *brute* action in the sense described previously, which counts, in the institution created by the contract, as an institutional action which terminates the service. One possibility for representation of this contract excerpt would be to refer explicitly to the brute action, and specify the conditions under which there is permission to carry it out. However, as the agreement makes no mention of the actual brute action that effects termination, there is little point in drawing out permission as a concept to apply separately. In such circumstances we say simply that the provider has a vested power to terminate the mailbox service, and thus also a vested permission to carry out the associated implicit brute action that will effect termination. In circumstances where brute actions are spelled out in the contract, we represent explicitly the *counts as* relation between brute and institutional actions, and represent the conditions under which performance of the brute action is permitted.

It is important to note that for the purposes of state tracking, terms of the *ecXML* language that represent normative concepts are treated as *unanalyzed primitives* whose intended reading is left implicit. We do *not* incorporate in the *ecXML* representation of a contract any explicit theory of normative concepts and their inter-relationships. In other work (see e.g. [5,10,11] we have considered the formal representation of various classes of normative relations, but this is not applied in the contract state tracking representation described in this paper. Thus for example, there is no explicit connection in the *ecXML* representation between terms representing an obligation to perform a certain action and terms representing permission to do so. This is in contrast to the use of a formalism such as $(C+)^{++}$ [12,13] which we have under development which is specifically designed for the representation of norms and institutional concepts and which has an explicit formal semantics for permission and *counts as* relationships between actions. Experiments with the use of $(C+)^{++}$ as an alternative formalism for contract state tracking is a subject of current work.

For the purposes of contract state tracking, we introduce terms representing various classes of normative concepts as they are convenient for the needs of a specific example. We find there is little added value, here, in making explicit the informal reading we ascribe to these terms. Of course this leaves the question of whether the *ecXML* representation of a contract is consistent in some appropriate sense. We see this as a separate question however. We are developing a separate tool which will test the consistency (suitably defined) of an *ecXML* representation of a contract (for instance, that the same action is not both obligatory and prohibited at the same time). For the purposes of state tracking, we assume that the contract representation has been tested for consistency, and we focus only on tracking the normative relations that are created as contract events occur.

The remainder of this paper is structured as follows. In section 2, we present the example contract and provide a brief analysis. In section 3, we give an informal overview of our use of *ecXML* in representing contracts for state tracking. In section 4, we provide a presentation of the example contract represented in *ecXML*. In sections 5 and 6, we present the Event Calculus State Tracking Architecture – our implementation for state tracking – and the Contract Visualiser – a tool for visualising the deployment status of contracts. In section 7, we present related work, and in section 8 conclude the paper.

2 EXAMPLE CONTRACT

In this paper, we use the following mail service agreement in order to ground our discussions. (The mail service is provided as a web service [14]).

- The Service Provider (*SP*) will provide a mail service to the Service Customer (*SC*), which includes a mailbox with a quota of *s* GBytes. *SC* will be charged a fixed monthly fee of $s*c_0$ 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.
- Whenever $u>s+e$, where *e* is a level of tolerance in GBytes, *SP* may prevent *SC* from receiving 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.

In order to represent a contract for the purpose of 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 instantiate an instance of a normative relation, namely an obligation bearing on *SC* to pay *SP* for service within a month.

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, which occurs when the specified time period expires before *SC* fulfils its obligation (to pay for service) on time. We shall say that an effect of this event is to instantiate an instance of another normative relation, namely (vested) power of *SP* to terminate the mailbox service.

3 *ecXML* -- AN XML FORMALISATION OF THE EVENT CALCULUS

From the perspective of what needs to be represented for contract state tracking, we need some way of representing the *effects of events* on contract state. For this, we use the XML based formalisation of the Event Calculus, *ecXML.* We have developed a Java implementation of a reasoner – the Event Calculus State Tracking Architecture (ECSTA) – for contracts written in *ecXML*.

In the following, we assume that such a reasoner is informed of the occurrence of *external events* that are named in the contract, such as *a user's mailbox utilisation going over its set quota.* The reasoner will also generate its own internal timer events, such as the monthly billing event, previously described. In both cases, such events are termed simply: *contract events.*

In the Event Calculus, and *ecXML*, a state is characterised by the values of *fluents*, which are properties whose values change according to the occurrence of (contract) events[1]. Fluents can be multi-valued, which means that in any given state they have a value from some designated set of possible values, or they can be boolean, which means that they have the possible values *true* or *false*. It is convenient to treat the special case of boolean fluents separately. In *ecXML*, the values of all multi-valued (non-boolean) fluents are real (floating point) numbers.

The following shows a boolean fluent `slg1_ok` in the XML notation:

```
<fluent id="slg1_ok"/>
```

The significance of the `slg1_ok` fluent in the example contract is explained later.

In *ecXML*, contract variables (as well as counting variables – see later) are represented as *multi-valued fluents.* The following example specifies that the current value of a contract variable `vDailyCharge` is 0.

```
    <mvfluent id="vDailyCharge">
          <num val="0"/>
    </mvfluent>
```

In general, a fluent (boolean or multi-valued) may have structure and additional parameters. For example, in the representation of the mail service agreement to be discussed in later sections, the (boolean) fluent

```
<fluent id=″o1″>
   <apara name=″Charge″><value id="vDailyCharge"/></apara>
   <apara name=″Month″>October</apara>
</fluent>
```

represents an instance of a normative relation of type `o1` in which the customer's charge for the billing month of October is the current value of the contract variable (multi-valued fluent) `vDailyCharge`. (In the ECSTA architecture, all fluents are relative to a specific contract, and for this reason it is unnecessary to refer to contracting parties (service provider and customer) and other fixed features of the contract in parameters of fluents.)

In the Event Calculus, the effects of events are expressed by specifying the fluents that they *initiate* and *terminate*. We say that an event of type *E* initiates a period of time for which a fluent *F* has a particular value *V* (or just *E* initiates *F*=*V* for short) and/or terminates a period of time for which fluent *F* has value *V* (or *E*

[1] In the following description, we talk about *ecXML* being used to represent contracts. It should be noted, however, that *ecXML* is a general-purpose language for describing and tracking how the state of an *arbitrary* domain changes (according to an event narrative).

terminates *F=V* for short). The representation of a contract in *ecXML* is a conjunction of:

- A finite set of `<initiates>` statements of the form:
  ```
  <initiates>
    <event id="E" qual="Q"/>
    <fluent id="F"> parameters </fluent>
    condition
  </initiates>
  ```
  meaning that the occurrence of a contract event of type *(E,Q)* initiates a period of time for which the boolean fluent *F* is true if *condition* holds, and of the form:
  ```
  <initiates>
    <event id="E" qual="Q"/>
    <mvfluent id="F"> math expr
         parameters
    </mvfluent>
    condition
  </initiates>
  ```
  meaning that the occurrence of contract event of type *(E,Q)* initiates a period of time for which the multi-valued fluent *F* has the value given by *math expr* if *condition* holds.
- A finite set of `<terminates>` statements of the form:
  ```
  <terminates>
    <event id="E" qual="Q"/>
    <fluent id="F"/>
    condition
  </terminates>
  ```
  meaning that the occurrence of a contract event of type *(E,Q)* initiates a period of time for which the boolean fluent *F* is false if *condition* holds.

Since multi-valued fluents can have only one value at any given time, it is not necessary to include `<terminates>` statements for multi-valued fluents. Or to put it another way, to say that *E* terminates a boolean fluent *F* is effectively to say that *E* initiates a period of time for which *F* is false. It is convenient to treat the special case of boolean fluents separately in this way.

Conditions in *ecXML* `<initiates>` and `<terminates>` statements may refer to the values of other fluents and to the occurrence of other events recorded in the event narrative. They are constructed using `<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 event parameter), `<btpara>` (boolean contract parameter), `<occurs>` (event occurrence), and `<holds>` (for value of a fluent at a given time).

The statements `<geq>, <leq>`, `<gt>`, `<lt>`, and `<deq>` take real-valued (or *double*) operands, which are provided as mathematical expressions. A mathematical expression in *ecXML* can be a simple numerical value, such as

```
<num val=″0″/>
```

or constructed using `<mul>`, `<add>`, `<sub>`, `<div>`, `<num>` (double value), `<dpara>` (double event parameter), `<dtpara>` (double contract parameter), and `<value>` (contract variable value). For example, the following expression adds the value of a contract variable `vDailyCharge` to the value of a contract parameter `sc0`. (Contract parameters and contract variables are discussed a little later.)

```
<add>
   <value id="vDailyCharge"/>
   <dtpar name=″sc0“/>
</add>
```

The representation an event narrative in *ecXML* is a conjunction of:

- A finite set of `<initially>` statements of the form:
  ```
  <initially>
    <fluent id=″F″> parameters </fluent>
  </initially>
  ```
  meaning that boolean fluent *F* holds in the initial state, and of the form:
  ```
  <initially>
    <mvfluent id=″F″> math expr
        parameters
    </mvfluent>
  </initially>
  ```
  meaning that multi-valued fluent *F* has the value given by *math expr* in the initial state.
- A finite set of `<happens>` statements of the form:
  ```
  <happens>
    <event id="E" qual=″Q″
       timestamp=“T″ instance_id="…">
        event parameters
    </event>
  </happens>
  ```
  meaning that the contract event *(E,Q)* happened at time *T*.

The XML representation of events themselves will be described presently. *ecXML* also provides a `<timer>` feature for generating timing events, where these may be one-off or recurrent. For example, we may wish to generate a timing event for an instance of an obligation, where the occurrence of the timing event would signify the deadline for fulfilment of the obligation instance. One can think of this as simply a mechanism for adding further `<happens>` statements into the event narrative.

The EC predicates `holds(`*`F,T`*`)` and `holds(`*`F,V,T`*`)`, representing, respectively, that boolean fluent *F* holds (is true) at time *T* and multi-valued fluent *F* has value *V* at time *T*, provide the means for querying the state of a contract at any time. The EC provides axioms that define the `holds` predicates in terms of the event narrative (`<initially>` and `<happens>` statements) and the `<initiates>` and `<terminates>` specifications. These definitions are as follows:

- `holds(`*`F,T`*`)` **if** `initiated(`*`F,T1`*`)` **and** T^3T1 **and**
  not `terminated(`*`F,T1,T`*`)`
  meaning that fluent *F* holds at time *T* **if** *F* is initiated at some time *T1* before or at time *T* and it is not terminated at any time between *T1* and *T*.
- `initiated(`*`F,0`*`)` **if**
  ```
      <initially>
        <fluent id="F"> parameters </fluent>
     </initially>
  ```
  meaning that fluent *F* is initiated at time *0* if *F* is asserted to hold in the initial state (as determined by *ecXML* `<initially>` statements for *F* in the event narrative).
- `initiated(`*`F,T1`*`)` **if** `happens(`*`E,T1`*`)` **and** *T1>0* **and**
  ```
   <initiates>
       <event id="E" qual="Q"/>
      <fluent id="F"> parameters </fluent>
      condition
    </initiates>
  ```
  meaning that fluent *F* is initiated at time *T1* greater than *0*, **if** an event *(E,Q)* happens at *T1* **and** *(E,Q)* initiates *F* (as determined by *ecXML* `<initiates>` statements for *F* in the contract) **and** *condition* holds at time *T1*.
- `terminated(`*`F,T1,T`*`)` **if** `happens(`*`E,T2`*`)` **and** $T^3T2>T1$ **and**
  ```
   <terminates>
     <event id="E" qual="Q"/>
     <fluent id="F"/>
     condition
    </terminates>
  ```
  meaning that boolean fluent *F* is terminated at time *T2* later than *T1* and before, or at, time *T* **if** an event *(E,Q)* happens at *T2* **and** *(E,Q)* terminates *F* (as determined by *ecXML* `<terminates>` statements for *F* in the contract) **and** *condition* holds at time *T2*.

  The definitions for the multi-valued versions of predicates `holds(`*`F,V,T`*`)` and `initiated(`*`F,V,T1`*`)` are similar. Further, for multi-valued fluents, the value of the fluent is terminated whenever an

`<initiates>` event happens, that is, the definition included also the following:

terminated(*F,V,T1,T*) **if** happens(*E,T1*) **and** $T^{3}T1>0$ **and**

```
 <initiates>
    <event id="E" qual="Q"/>
    <mvfluent id="F"> math expr
      parameters
    </mvfluent>
    condition
  </initiates>
```

Consider the following example of the *holds* axiom being applied in the context of contract representation. Let us say that we have the following *ecXML* statements:

- The occurrence of a `bill_timer` timeout event *initiates* an instance of an obligation relation `o1`
```
    <initiates>
       <event id="bill_timer"/>
       <fluent id="o1"/>
    </initiates>
```

- The occurrence of a fulfilment event for `o1` *terminates* `o1`**.**
```
    <terminates>
      <event id="o1" qual="fulfilment"/>
      <fluent id="o1"/>
    </terminates>
```
  (For use of *ecXML* in the representation of contracts, there is a built-in feature which treats *fulfilment* events without the need to write `<terminates>` statements of this form directly.)

- A billing event *happens* 1 month into the contract
```
    <happens>
     <event id="bill_timer" timestamp=M1/>
    </happens>
```
  where `M1` is a string with a value of the UTC time corresponding to the start of the contract plus 1 month.

- A fulfilment event for `o1` *happens* 1.5 months into the contract
```
    <happens>
      <event id="o1" qual="fulfilment"
                             timestamp=M15/>
    </happens>
```
  where `M15` is a string with a value of the UTC time corresponding to the start of the contract plus 1.5 months.

According to the *holds* axioms given previously, `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. `o1` holds at time 1.25 months because the billing event that occurred at 1 month initiates `o1` and `o1` has not been terminated between 1 month and 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.

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. Also, *counting variables*, similarly to contract variables, are used to maintain live numerical state. However, statements using them are added to a contract representation for housekeeping purposes: in contrast to contract variables, their use is not normative and their use is not agreed between contract parties.

It is also convenient to set up *state definitions* for a contract which allow us to monitor states of interest. For example, the following defines a state, for the mail service SLA, pertaining to the mail service being unavailable whenever the boolean fluent `slg1_ok` does not hold:

```
<statedefn id="sUnavailable">
   <statenorm id="slg1_ok" active="false"/>
</statedefn>
```

`<statenorm>` in turn is evaluated in terms of the `holds` predicates.

We conceptualise events in *ecXML* as pairs. The first argument of the pair is the identifier of a pertaining normative relation, an (internal) timer, or some external event. The second argument is an (optional) qualification of the event. In the case of a pertaining normative relation, for example, it may be that an instance of the normative relation has been fulfilled. Here, the event would be: `(norm-identifier, fulfilment)`. Whenever an event occurrence pertaining to a normative relation is asserted within *ecXML*, that is, within a `<happens>` statement, there is also associated with the event an instance identifier, which specifies the instance of the pertaining norm with which the event is concerned. Events in *ecXML* have the following XML schema:

```
<xsd:element name="event">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="para"
      minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:choice>
          <xsd:element ref="bool" minOccurs="0"/>
          <xsd:element ref="num" minOccurs="0"/>
        </xsd:choice>
        <xsd:attribute name="name"
           type="xsd:string" use="required"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="id"
     type="xsd:string" use="required"/>
  <xsd:attribute name="qual" type="xsd:string"/>
  <xsd:attribute name="timestamp" type="xsd:dateTime"/>
  <xsd:attribute name=”instance_id"
     type="xsd:nonNegativeInteger"/>
 </xsd:complexType>
</xsd:element>
```

As can be seen from the schema, an *ecXML* event has an *id* attribute which is required, along with an optional *qualification* attribute. Together these constitute the conceptual pair that characterises an *ecXML* event: `(id, qualification)`. An *ecXML* `<event>` statement may also contain an optional *timestamp* attribute (pertaining to the UTC time of the event), and optional *instance_id* attribute (which if present in an event gives the unique instance identifier of the instance of the norm pertaining to the event). If an `<event>` statement is used within a `<happens>` statement, it may specify event parameters, which are elaborated as `<para>` statements within *ecXML*. An example of an `<event>` statement pertaining to the fulfilment of `o1` is:

```
<event id=”o1” qual=”fulfilment”
    timestamp=”UTC time” instance_id="…">
   <para name=”Charge”><num val="25.00"/></para>
   <para name=”Month”>October</para>
</event>
```

For *ecXML* statements pertaining to the initiation and termination of normative relations, the following should be noted. Every time the conditions of an `<initiates>` statement are satisfied, a new instance of the given normative relation is created. Whereas, every time a `<terminates>` statement holds, all instances of the given normative relation are considered to be destroyed. In practice, `<terminates>` statements are used with normative relations for which it only makes sense to have at most one instance outstanding at any one time.

As already noted, we have not sought in this work to give a fixed name and definition to particular types of normative relation, such as obligation, or permission. Instead, we allow complete freedom over the naming of normative relations. Consequently, a normative relation is only characterised by how it is initiated and terminated, and has no explicit definition of its informal semantics in our representation of contracts.

In *ecXML*, instances of normative relations and values of contract variables are represented as fluents. It is worth emphasising that, by using the presented *holds* axioms, we can ask queries such as: *return all of the extant instances of a given normative relation (while optionally limiting the query to a particular parameterisation of the relation), indexed according to instance identifier.*

4 ecXML REPRESENTATION OF EXAMPLE CONTRACT

In the sequel, we provide an explanation of how the example mail service contract is represented in *ecXML.* The following *ecXML* statements cater for the contract excerpt: *Whenever u>s, where u is the mailbox utilisation in GBytes, SP will charge SC c_1 for each GByte over s, calculated daily*. It is assumed, in accommodating this excerpt, that an external event `daily_charge_event` is entered into the event narrative daily providing the contract reasoner with the daily charge that the customer has accrued, where this charge will be zero if the value of *u* has not gone above *s* for that day. The daily charge is accumulated in the contract variable `vDailyCharge`.

The first statement simply initialises the contract variable `vDailyCharge` to zero at contract initiation:

```
<initially>
   <mvfluent id="vDailyCharge">
      <num val="0"/>
   </mvfluent>
</initially>
```

The next statement says that when a `daily_charge_event` occurs, add the value of the event’s `Charge` parameter, corresponding to the charge for the day, to the contract variable `vDailyCharge`.

```
<initiates>
   <event id="daily_charge_event”/>
   <mvfluent id="vDailyCharge">
      <add>
         <value id="vDailyCharge"/>
         <dpara name="Charge"/>
      </add>
   </mvfluent>
</initiates>
```

Here `<dpara>` accesses the value of the `Charge` parameter in the event's XML description.

The following *ecXML* statements accommodate the representation of the contract excerpt: *All billing of SC occurs monthly*. They set up the timer `bill_timer` to generate monthly timeout events to initiate instances of an obligation `o1`, which bears on *SC* to pay for service provision.

The first *ecXML* statement simply says that the timer is initially active. That is, fluent `bill_timer`, is initially set.

```
<initially>
   <fluent id="bill_timer"/>
</initially>
```

The second statement says that `bill_timer` is recurrent with a period of one month.

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

As explained earlier, `<timer>` statements may be seen as a mechanism for adding extra `<happens>` assertions to the event narrative.

The following *ecXML* statements represent the contract excerpt: *SC will be charged a fixed monthly fee of $s*c_0$ for the service*. The first statement specifies that a timeout event pertaining to `bill_timer` initiates an instance of an obligation relation of type `o1`. The relation `o1` has a single parameter `Charge` which, in the given *ecXML*, is assigned the value obtained by summing the current (accumulated) daily charge, given by the contract variable `vDailyCharge`, with the value (currently) assigned to the contract parameter `sc0`.

```
<initiates>
   <event id="bill_timer"/>
   <fluent id="o1">
      <apara name="Charge">
         <add>
            <value id="vDailyCharge"/>
            <dtpar name="sc0"/>
         </add>
      </apara>
   </fluent>
</initiates>
```

The next statement specifies that the same `bill_timer` event also has the effect of setting the contract variable `vDailyCharge` to zero:

```
<initiates>
   <event id="bill_timer” />
   <mvfluent id="vDailyCharge">
      <num val="0"/>
   </mvfluent>
</initiates>
```

The following timer for obligation relation `o1` accommodates the contract clause: *SC is given a month thereafter to pay*. Note the single iteration of the `<run>` statement.

```
<timer id="o1">
   <run iters="1">
      <dur="P1M"/>
   </run>
</timer>
```

The next *ecXML* statement pertains to the contract excerpt: *If SC fails to pay within the given time, SP may terminate the mailbox service without notice.* It says that a timeout event for an instance of an obligation relation `o1` has the effect of initiating an instance of the (vested) power relation `r1`, which corresponds to: *SP may terminate the mailbox service without notice.*

```
<initiates>
   <event id="o1” qual=”timeout"/>
   <fluent id="r1"/>
</initiates>
```

The following *ecXML* statements accommodate 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.* This excerpt is part of a Service-Level Guarantee (SLG) pertaining to the provision of the mail service. An SLG captures a level of service that must be maintained by a service provider, according to a service-level predicate, such as: *95% availability, Mondays-Fridays, 9a.m.-5p.m., and 99% availability at all other times.* An SLG will also capture what actions may, or should be, taken by one or more contract parties whenever the service level captured by the SLG is violated or restored. The contract excerpt, in this case, states that the service provider will be liable to pay the service customer money whenever the SLG is violated. We assume that some external agent tells us when the SLG has been violated, that is that the mail service is unavailable, and when it has been restored.

Following is an *ecXML* statement that simply stipulates that a fluent that tracks the status of `slg1`, called `slg1_ok`, is initially true. This fluent is useful for the purposes of the state tracking definitions that are given later in this section.

```
<initially>
   <fluent id="slg1_ok"/>
</initially>
```

Next, a violation event for `slg1` terminates `slg1_ok` signifying that the SLG is now being violated:

```
<terminates>
   <event id="slg1” qual=”violation"/>
   <fluent id="slg1_ok"/>
</terminates>
```

The same violation event also triggers a timer `service_unavail_timer`. The purpose of this timer is to trigger the collection of *$p* every *t* minutes, where *t* is the value of the contract parameter `t_unavailable`. It also initiates an instance of the obligation relation `o3`, which has no specified timeout, and which bears on the service provider to restore service as soon as possible.

```
<initiates>
   <event id="slg1” qual=”violation"/>
   <fluent id="service_unavail_timer"/>
</initiates>

<timer id="service_unavail_timer">
   <run>
      <durtpar name="t_unavailable"/>
   </run>
</timer>

<initiates>
   <event id="slg1” qual=”violation"/>
   <fluent id="o3"/>
</initiates>
```

The next *ecXML* statement accumulates the *$p* penalty that the provider is charged every *t* minutes for unavailability of the mail service. It does so according to `service_unavail_timer` events, which occur according to the timer set up previously. It uses the contract variable `vPenalty` to store the accumulated charge.

```
<initiates>
   <event id="service_unavail_timer”/>
   <mvfluent id="vPenalty">
```

```
            <add>
               <value id="vPenalty"/>
               <dtpar name="p_penalty"/>
            </add>
         </mvfluent>
      </initiates>
```

On restoration of `slg1`, the SLG status fluent `slg1_ok` is initiated:

```
      <initiates>
         <event id="slg1" qual="restoration"/>
         <fluent id="slg1_ok"/>
      </initiates>
```

Also, the timer that causes the provider to be penalised *$p* every *t* minutes is terminated:

```
      <terminates>
         <event id="slg1" qual="restoration"/>
         <fluent id="service_unavail_timer"/>
      </terminates>
```

Normative relation `o3` pertaining to the obligation bearing on the service provider to restore service is also terminated:

```
      <terminates>
         <event id="slg1" qual="restoration"/>
         <fluent id="o3"/>
      </terminates>
```

The next two *ecXML* statements initiate and terminate instances of the (vested) power fluent `r2`**,** corresponding to whether the service provider is empowered to refuse to allow the customer to receive emails. These accommodate the contract clause: *Whenever u>s+e, where e is a level of tolerance in GBytes, SC will not be able to receive emails.* Whenever a service customer's utilisation goes above the limit specified in this clause, a `quota_violation_event` contract event will be received. Its parameter `Over` will be set to true. The first statement says that when this happens `r2` should be initiated. Whenever a service customer's utilisation ceases to be above the limit specified in this clause, a `quota_violation_event` contract event will again be received. This time, however, its parameter `Over` will be set to false. The second statement says that when this happens the relevant instances of relation `r2` should be terminated.

```
      <initiates>
         <event id="quota_violation_event"/>
```

```
   <fluent id="r2"/>
   <beq>
      <bpara name="Over"/>
      <bool val="true"/>
   </beq>
</initiates>

<terminates>
   <event id="quota_violation_event"/>
   <fluent id="r2"/>
   <not>
      <beq>
         <bpara name="Over"/>
         <bool val="true"/>
      </beq>
   </not>
</terminates>
```

There are a number of *ecXML* statements that are concerned with the payment of penalties to the service customer by the service provider on a monthly basis. They are very similar to the *ecXML* statements concerned with the billing of the service customer for service provision shown above. There are also a number of *ecXML* statements concerned with maintaining *housekeeping information* required by the service provider. These statements manipulate the values of counting variables that pertain to monies earned, and penalties paid out, as well as penalties that the service customer has failed to pay in time for the contract instance. These *ecXML* statements are all dealt with straightforwardly in similar fashion to those shown above and are not presented here for reasons of brevity.

Finally, some state definitions may be specified. These allow a contract party to track states of interest.

A state is considered to be 'normal' whenever no instances of relations `r1` and `r2` hold, but where `slg1_ok` does.

```
<statedefn id="sNormal">
   <statenorm id="r1" active="false"/>
   <statenorm id="r2" active="false"/>
   <statenorm id="slg1_ok" active="true"/>
</statedefn>
```

An 'unavailable' state is considered to be one where `slg1_ok` does not hold.

```
<statedefn id="sUnavailable">
   <statenorm id="slg1_ok" active="false"/>
</statedefn>
```

A state in which the service customer cannot receive mail is defined as one where an instance of `r2` holds.

```
<statedefn id="sRefuseReceiveMail">
   <statenorm id="r2" active="true"/>
</statedefn>
```

State definitions for other states of interest are given in similar fashion. The interested reader may find a complete *ecXML* representation of the example at [15].

5 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 contract 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. That is, 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. Shared variables are used for maintaining *inter-contract* state, such as the number of times an SLG has been violated across all mail service agreements
- Register/Deactivate/Reactivate Shared Variable Association. This use-case simply pertains to the association of individual contracts to shared variables
- 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 in the context of web service provision, an *incident manager*, responsible for handling the effects of fabric incidents on the fulfilment of service agreements, would like to be notified when the number of violated obligations across a number of agreements goes above *x*.

As this requires reasoning across multiple agreements, 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 service agreement using the *Register Shared Variable Association* and *Add Contract 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 contract: *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.

6 CONTRACT VISUALISER

As well as the ECSTA reasoner, a tool called *Contract Visualiser* has been implemented which allows for the deployment management of contracts. It provides a user-interface to contract deployment tasks, and supports all of the use-cases given in section 5. The relationship between ECSTA and Contract Visualiser is captured in figure 1.

In figures 2 through to 9 a scenario is shown unfolding, as captured by Contract Visualiser. The scenario pertains to the mail service agreement used in this paper. (Note that the example shots of Contract Visualiser pertain to its use in the context of web service deployment. As such, the term *SLA* is used in place of contract, where SLA stands for *Service-Level Agreement*).

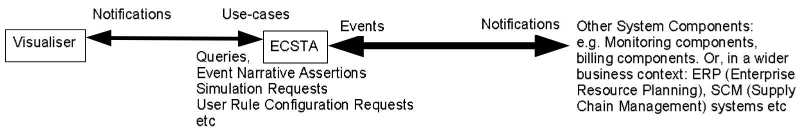
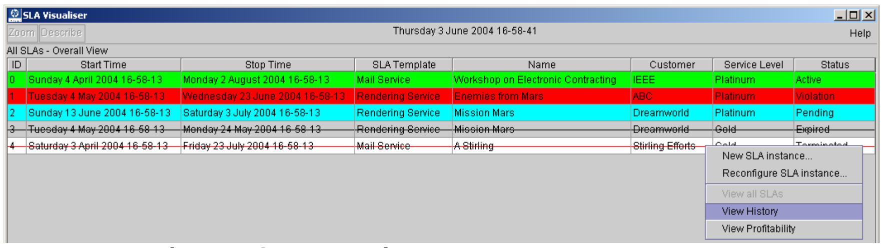

Notifications
Use-cases
Visualiser
ECSTA
Events
Notifications
Queries,
Event Narrative Assertions
Simulation Requests
User Rule Configuration Requests
etc
Other System Components:
e.g. Monitoring components,
billing components. Or, in a wider
business context: ERP (Enterprise
Resource Planning), SCM (Supply
Chain Management) systems etc

Figure 1: Relationship between ECSTA and Contract Visualiser



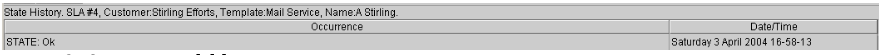
Thursday 3 June 2004 16-58-41

All SLAs - Overall View

ID	Start Time	Stop Time	SLA Template	Name	Customer	Service Level	Status
0	Sunday 4 April 2004 16-58-13	Monday 2 August 2004 16-58-13	Mail Service	Workshop on Electronic Contracting	IEEE	Platinum	Active
1	Tuesday 4 May 2004 16-58-13	Wednesday 23 June 2004 16-58-13	Rendering Service	Enemies from Mars	ABC	Platinum	Violation
2	Sunday 13 June 2004 16-58-13	Saturday 3 July 2004 16-58-13	Rendering Service	Mission Mars	Dreamworld	Platinum	Pending
~~3~~	~~Tuesday 4 May 2004 16-58-13~~	~~Monday 24 May 2004 16-58-13~~	~~Rendering Service~~	~~Mission Mars~~	~~Dreamworld~~	~~Gold~~	~~Expired~~
~~4~~	~~Saturday 3 April 2004 16-58-13~~	~~Friday 23 July 2004 16-58-13~~	~~Mail Service~~	~~A Stirling~~	~~Stirling Efforts~~	~~Gold~~	~~Terminated~~

Figure 2: Top-Level View in Contract 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.

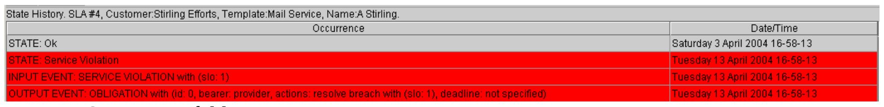


State History. SLA #4, Customer:Stirling Efforts, Template:Mail Service, Name:A Stirling.

Occurrence	Date/Time
STATE: Ok	Saturday 3 April 2004 16-58-13

Figure 3: Scenario Unfolds 1

In figure 3, we see that the state of SLA 4 is "Ok" to begin with.

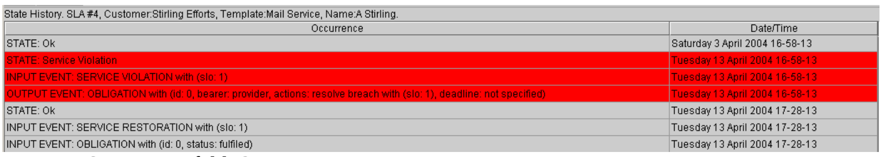


State History. SLA #4, Customer:Stirling Efforts, Template:Mail Service, Name:A Stirling.

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

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.

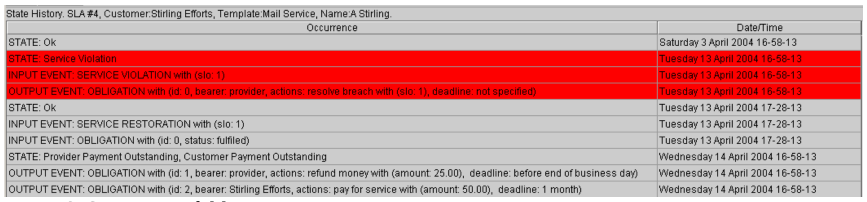


State History. SLA #4, Customer:Stirling Efforts, Template:Mail Service, Name:A Stirling.

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

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.

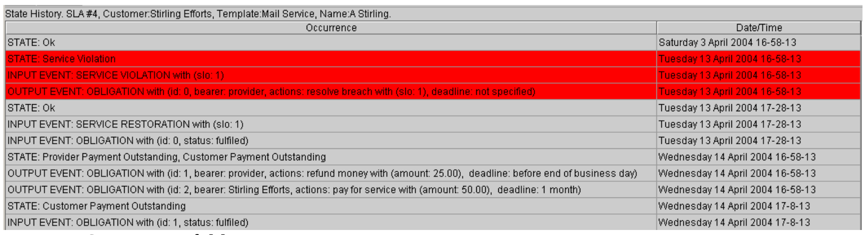


State History. SLA #4, Customer:Stirling Efforts, Template:Mail Service, Name:A Stirling.

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

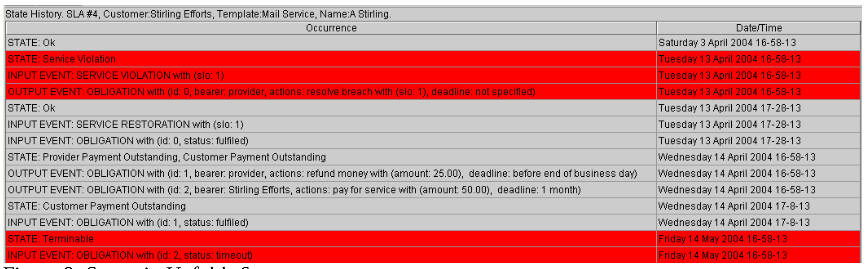
In figure 6, we see that two obligations are initiated (by timers that are specified in the SLA representation and maintained by the reasoner) stipulating that: the Service Provider must refund $25 to the Service Customer for poor service (before end of business day) and the Service Customer must pay $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

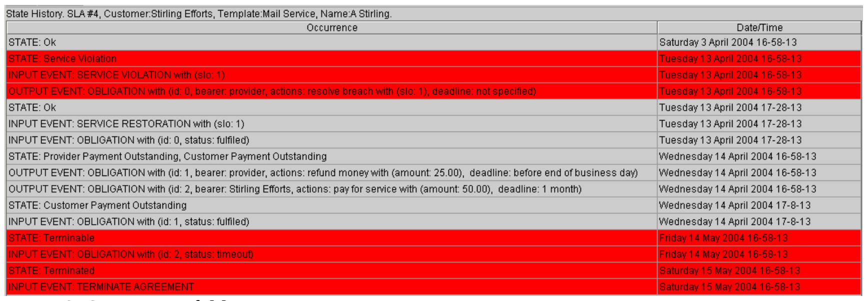
In figure 7, we see that an input event saying that the Service Provider has fulfilled its obligation to refund $25 to the service customer occurs causing: the state of the SLA moves from “Provider Payment Outstanding” + “Customer Payment Outstanding” to just “Customer Payment Outstanding”. The fulfilment 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 empowered 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 empowered to terminate the service, they do so: the SLA moves into a “Terminated” state.

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 are now presented. In [7,8], Artikis describes the representation in EC of ‘open’ multi-agent systems viewed as societies of computational agents, including variations on the Contract-Net and NetBill protocols [16,17], an argumentation protocol based on Brewka's reconstruction of Rescher's Theory of Formal Disputation (RTFD) [18], and resource allocation protocols among others. This work also explicitly employs the concepts of obligation, permission, and institutional power, and includes the specification of sanctions and penalties in the case of violations. The representation of these concepts as EC fluents is different from the methods employed in this paper, however. It is also worth noting that Artikis and colleagues have also employed other action languages from AI as an alternative to the use of EC, and specifically the action language C+ [19]. C+ provides a high-level notation for defining axioms specifying the effects of actions on domain fluents, and ways of characterising domain phenomena, such as the *common sense law of inertia*. It also has an explicit semantics in terms of labelled transition systems. Being able to describe contracts as transition systems is extremely useful for proving properties (using *model checking*) about the contracts. Also of note is an extended form of C+, called $(C+)^{++}$ [12,13], which is specifically defined for the representation of norms and institutional concepts. These extensions provide a treatment and formal semantics for institutionalised power, that is, *counts as*, relations between actions, and for the specification of permitted (or acceptable, or legal) states of a transition system and its permitted (or acceptable or legal) transitions and histories.

In [20], 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. The resulting formalism is used in conjunction with abductive reasoning techniques to perform *a priori* analysis of policy specifications. In [21], 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 monitoring their performance. In [22] Daskalopulu discusses the use of Petri-nets for contract state tracking, and assessing contract performance. Her approach is best suited for contracts which can naturally be expressed as protocols, or workflows. 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 [23] Milosevic and colleagues attempt to identify the scope for automated management of e-contracts; including: contract drafting, negotiation and monitoring. In [24], Abrahams and colleagues define the *EDEE* architecture (E-commerce application Development and Execution Environment). EDEE provides a mechanism for business process automation based on assessment and reasoning of interactions between intra-, inter-, and extra-organisational policy, and execution of business procedures informed by the combined legal effect of policy rules. Abrahams proposes *Event-Condition Obligation* rules for the writing of effect axioms for 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 [25] 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. In contrast to our approach, Grosof and colleagues are not concerned with maintaining live representations of contracts for state tracking purposes. A facility for tracking contract state is (ostensibly) lacking in their work. Rather, they seek to represent contracts for the purpose of communicating contract rules. In fact, Grosof's work would dovetail nicely with that of Leite and colleagues [26] who have suggested update semantics for *generalised logic programs*, where the meaning of a contract at any state would be given by the (*stable model* or *well-founded*) semantics of an individual logic program, which is derived from a combination of the logic programs pertaining to previous states and previously-occurring contract-related events, expressed by update rules.

8 CONCLUSIONS

In this paper we have proposed a formalisation of the Event Calculus in XML, called *ecXML*, and have shown informally its application to the representation of contracts to facilitate automated tracking of contract state. We have grounded our discussion using an agreement for a mail service (provided as a web service), one of a number of similar contracts and agreements we have represented in this approach.

Through using EC, we are able to represent a contract in terms of how its state evolves according to a narrative of (contract-related) events. Then, we are able to extract information pertaining to contract state, such as which norms are initiated, and what values contract variables have, for arbitrary times (in the past, or present). 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 state tracking is *externalised* as a separate component. 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 monitor state. That is to say, *ecXML* is a generic language for characterising how (both boolean – true or false – and numeric) properties of a domain change according to an event narrative, where the representation of contracts is just one application. Commensurately, the presented Event Calculus State Tracking Architecture may be used in many application domains.

A comprehensive Java-based implementation of a generic EC reasoning component, called the Event Calculus State Tracking (ECSTA) architecture, has been developed. In this context, we think of *ecXML* as the *language of the machine*: although it may appear to be somewhat verbose, this is not a critical issue because *ecXML* representations are meant to be automatically generated by some suitable authoring tool. For example, elsewhere [27] we have discussed a higher-level syntax that we have defined, called ***C**ontract **T**racking **XML*** (*CTXML*), for representing contracts for the purpose of state tracking. It provides a considerably more concise syntax than *ecXML*, together with a mapping to *ecXML*. The *ecXML* implementation, however, 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. The implementation, moreover, is designed to be capable of supporting a large 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 course of our work, we have empirically evaluated the adequacy of *ecXML* in expressing how the normative state of a contract evolves according to a narrative of contract events. We have done so by considering the representation of tens of service agreements from the domain of web services and other service-oriented domains, such as Utility Computing [9]. We have found it to be sufficient in representing such agreements. We plan to further evaluate *ecXML* by considering other sorts of agreements from these domains, as well as considering contracts from other domains.

The work described herein represents a small part of a larger effort related to the representation of workflow (where we consider contracts to be a type of workflow) from multiple perspectives: *control, data, organisational* and *normative*. We are seeking to realise a unified modelling approach across these perspectives so that we might facilitate the proving of workflow properties across them. One aspect of the organisational perspective concerns *organisational policies* which may be considered to constrain the enactment of workflows. It is an interesting research problem to consider how we might build flexibility into the specification of workflows (across these perspectives) so that the enactment of a workflow may be dynamically composed so to best satisfy organisational policies. Furthermore, if some policies have to be overridden in the enactment of a workflow, how do we choose the appropriate policies to override?

We are interested in evaluating the advantages and shortcomings of a number of formalisms for the modelling of workflows not only for proving workflow properties but also as enactment metaphors. Some of the formalisms considered in our work are: Petri-nets [28,29], value-passing CCS [30], π-calculus [31] and various logic-based formalisms such as C+ [19].

9 REFERENCES

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

[2] Murray Shanahan. "*The Event Calculus Explained"*. In M.J.Wooldridge, M.Veloso, editors, *Springer Lecture Notes in Artificial Intelligence, 1660*: pp. 409-30, 1999. Springer.

[3] Lars Lindahl. "*Position And Change, A Study in Law and Logic"*, D. Reidel Publishing Company, 1977.

[4] Sushil Jajodia, Pierangela Samarati, V.S.Subrahmanian, Eliza Bertino. "*A unified framework for enforcing multiple access control policies"*. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*: pp. 474-85, 1997. ACM Press New York, NY, USA.

[5] Andrew J.I.Jones, Marek Sergot. "*A Formal Characterisation of Institutionalised Power*". In *Journal of the IGPL*, **4(3)**: pp.429-45. June, 1996.

[6] Babak Sadighi Firozabadi, Marek J.Sergot. "*Power and Permission in Security Systems"*. In *Proceedings of the 7th International Workshop on Security Protocols, Lecture Notes In Computer Science*: pp. 48-59, 1999.

[7] A.Artikis. "*Executable Specification of Open Norm-Governed Computational Systems"*. Ph.D. thesis, Department of Electrical & Electronic Engineering, Imperial College, London, 2003.

[8] A.Artikis, J.Pitt, M.J.Sergot. "*Animated Specifications of Computational Societies"*. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS), Bologna*: pp. 1053-62, 2002.

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

[10] M.J.Sergot. "*Normative Positions"*. In P.McNamara, H.Prakken, editors, *Norms, Logics and Information Systems*: pp. 289-308, 1998. IOS Press, Amsterdam.

[11] M.J.Sergot. "*A Computational Theory of Normative Positions*". In *ACM Transactions on Computational Logic*, **2(4)**: pp.581-622. October 2001.

[12] M.Sergot, R.Craven. "*(C/C+)++: An action language for representing norms and institutions"*. In *Proceedings of Workshop on Automated Reasoning, Liverpool, UK*, 15-16 April 2003.

[13] M.Sergot. "*The language (C/C+)++"*. In J.Pitt, editor, *Deliverable D6(2) of ALFEBIITE EU-Project (IST-1999-10298)*: pp. 55-84, 2002.

[14] G.Alonso, F.Casati, H.Kuno, V.Machiraju. "*Web Services. Concepts, Architectures and Applications.",* Springer, 2004, ISBN: *3-540-44008-9*.

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

[16] M.Sirbu. "*Credits and Debits on the Internet*". In *IEEE Spectrum*, **34(2)**: pp.23-9. 1997.

[17] R.Smith, R.Davis. "*Distributed Problem Solving: the Contract-net Approach"*. In *Proceedings of Conference of Canadian Society for Computational Studies of Intelligence*: pp. 217-36, 1978.

[18] G.Brewka. "*Dynamic Argument Systems: A Formal Model of Argumentation Processes Based on Situation Calculus*". In *Journal of Logic and Computation*, **11(2)**: pp.257-82. 2001.

[19] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, Hudson Turner. "*Nonmonotonic causal theories*". In *Artificial Intelligence*, **153(1-2)**: pp.49-104. 2004.

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

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

[22] A.Daskalopulu. "*Modelling Legal Contracts as Processes".* In *11th International Conference and Workshop on Database and Expert Systems Applications*: pp. 1074-9. IEEE C. S. Press.

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

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

[25] 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, *1st ACM Conf. on Electronic Commerce (EC-99)*, Denver, CO, USA, November 1999. ACM Press, New York, NY, USA.

[26] J.A.Leite, J.J.Alferes, L.M.Pereira. "*Multi-dimensional Dynamic Logic Programming".* In F.Sadri, K.Satoh, editors, *Proceedings of the CL-2000 Workshop on Computational Logic in Multi-Agent Systems (CLIMA'00), London, England*, July 2000.

[27] Andrew D.H.Farrell, Marek J.Sergot, Claudio Bartolini, Mathias Salle, Athena Christodoulou, David Trastour. "*Using the Event Calculus for the Performance Monitoring of Service-Level Agreements for Utility Computing".* In *Proceedings of First IEEE International Workshop on Electronic Contracting (WEC 04), San Diego, CA, USA*, 6 July 2004.

[28] Wolfgang Reisig, Grzegorz Rozenberg. "*Lectures on Petri Nets I: Basic Models*", Springer, 1998, ISBN: *3-540-65306-6*.

[29] Wolfgang Reisig, Grzegorz Rozenberg. "*Lectures on Petri Nets II: Applications*", Springer, 1998, ISBN: *3-540-65307-4*.

[30] Robin Milner. "*Communication and Concurrency*", Prentice Hall, 1989, ISBN: *0-13-115007-3*.

[31] Robin Milner. "*Communicating and Mobile Systems: The Pi-Calculus*", Cambridge University Press, 1999, ISBN: *0-521-65869-1*.