International Journal of Cooperative Information Systems
© World Scientific Publishing Company

USING THE EVENT CALCULUS FOR TRACKING THE NORMATIVE STATE OF CONTRACTS

ANDREW D. H. FARRELL, MAREK J. SERGOT
Department of Computing, Imperial College, 180 Queen's Gate, London. SW7 2AB. United Kingdom.
{*andrew.farrell,m.sergot*}*@imperial.ac.uk*

MATHIAS SALLÉ, CLAUDIO BARTOLINI
Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, California. United States of America.
{*mathias.salle,claudio.bartolini*}*@hp.com*

Received (30th November 2004)
Revised (18th February 2005)

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.

Keywords: Contract State Tracking, Event Calculus

1. Introduction

An increasingly important aspect in the life-cycle of contracts is the automated monitoring of the *normative state of contracts*. 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 [23]. 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) [29]. 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, for deployment in a business context, we have constructed a Java implementation of the EC reasoning component and have 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 [15]) 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, in this paper, we have assumed a *closed policy* [13]. 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, nor is it always appropriate. Indeed, our framework is capable of supporting the co-existence of relations pertaining to any distinct normative concepts, such as permission and prohibition; however, the state tracking component itself is not capable of assessing whether the co-existence of such relations may be inconsistent, or counter-intuitive – such as the existence of a permission and prohibition to perform the same action (which would be inconsistent). Analysis of a contract representation is a separate exercise

(in our framework) for which there is a separate set of tools. 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.

- Obligation. Bentham (as presented in [15]) 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, that is, obligation to refrain from performing a prohibited action, may also be useful, as may be an obligation to realise a state-of-affairs. The concept of *directed obligation* (sometimes called *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 [4], 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 [27], 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 [4] 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. A presentation of the formal characterisation of these concepts is not necessary for the purposes of this paper.

  In legal theory, the importance of the following distinction has long been recognised. There are three quite different notions [4,25]:

(a) The power to create an institutional fact
(b) The permission to exercise that power
(c) The practical ability (physical capability, opportunity, know-how) to exercise

that power.

The following example, due to Makinson and quoted in [4], 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 [1,2] for examples of practical settings where power is separated from permission to exercise it.)

We should also mention the concept of *violation* of some obtaining normative relation, such as an obligation. In fact, with regard to the example mail service contract, presented in section 2, we say that a violation obtains when a service-level guarantee pertaining to the availability of the mail service ceases to be met by the service provider.

However, it should be noted that, in many cases of performing a conceptualisation of a contract, for representation in *ecXML*, where we have identified that a normative relation might be violated, we have not found it necessary to explicate a corresponding notion of violation. This is because, often, when a normative relation has been violated – such as an obtaining obligation instance reaching its deadline for fulfilment without being fulfilled, we do not need to record as much by asserting an explicit record of violation in the contract representation. Rather, carrying out a collection of (compensating) contract actions and informing pertinent parties of the violation occurrence typically suffices. There is considerable discussion in the deontic logic literature on producing adequate formalisations of violation and its consequences, usually in the context of so-called *contrary-to-duty* obligations (see, for example, [14]). Again, these technical details are not pertinent to the treatment adopted in our state tracking system.

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 [12]. In this paper, we use one such agreement for a mail service to ground our discussions. In order to exemplify the normative concepts

that have been introduced here, we now briefly present some excerpts from this agreement.

- Obligation: *Service Provider (SP) 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. We choose not to represent this as a form of directed obligation where we make explicit the counter-party to whom the obligation is owed. In this particular example, it seems natural to think that the obligation to pay the customer is owed to the customer, but this need not always be so. In some contexts, the obligation to make a payment to party x may be owed to another third party y. However this particular example, and others in the mail service example, are so simple that nothing is gained from making this distinction and the counter-party can be left unspecified without loss.
- Permission and (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 the Service Consumer (*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 other examples 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*, introduced as they are convenient for the needs of a specific example, 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. [4,18]) 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 $(\mathcal{C}/\mathcal{C}+)^{++}$ [28] 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 $(\mathcal{C}/\mathcal{C}+)^{++}$ as an alternative formalism for contract state tracking is a subject of current work.

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 [9]).

- *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*[a]. 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. 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. Note that in *ecXML* instances of normative relations are represented as fluents. Moreover, 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

[a]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).

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

represents an instance of a normative relation of type `o1`. It specifies two parameters which are set on instantiation of the relation, namely: the customer's charge (being the current value of the contract variable – multi-valued fluent – `vDailyCharge`) and the current billing month (being the value of the `Month` parameter – as specified by the contained <`para`> element – of the event causing the instantiation of the normative relation).

In the ECSTA architecture, all fluents are relative to a specific contract. Furthermore, for convenience in this paper, we assume that the party and counter-party of a normative relation is known when a contract is authored. For these reasons, it is unnecessary to refer to contracting parties (service provider and customer) and other fixed features of the contract in parameters of fluents pertaining to normative relations. These details may be stored in some database, where the name of the normative relation may be used to reference them. This is merely for convenience. In other cases, where it may be necessary to refer explicitly to the respective parties in a normative relation, the *ecXML* representation can simply carry these as additional fluent parameters.

In *ecXML*, we characterise an *event* as a pair: (`E,Q`), as described later. 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,Q`) initiates a period of time for which a fluent `F` has a particular value `V` (or just (`E,Q`) initiates `F=V` for short) and/or terminates a period of time for which fluent `F` has value `V` (or `E` 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`> (floating-point greater or equals), <`leq`> (floating-point less or equals), <`gt`> (greater), <`lt`> (less), <`deq`> (floating-point 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 floating-point) 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`> (floating-point value), <`dpara`> (floating-point event parameter), <`dtpara`> (floating-point 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.

Although not required by the Event Calculus in general, in *ecXML* a boolean fluent is initially `false` by default unless it has an `initially` statement making it `true` initially. A multi-valued fluent (whose domain is the set of reals) has the value `0.0` by default unless it has an `initially` statement assigning it some value initially.

- A finite set of <`happens`> statements of the form:

```
<happens>
   <event id="E" qual="Q" timestamp="T" ...>
      event parameters
   </event>
</happens>
```

meaning that the contract event (`E`,`Q`) happened at time `T`.

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 Event Calculus 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. 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, which are hard-coded into a contract reasoner for *ecXML*, and which do not exist as part of the representation of a contract, are as follows:

- `holds(F,T) if initiated(F,T1,T) 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 a time later than `T1` and before, or at, `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,T) if T>=T1>0
  and
  <happens>
     <event id="E" qual="Q" timestamp="T1" ...> event parameters </event>
  </happens>
  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` (as determined by the event narrative) **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 T>=T2>T1
  and
  <happens>
     <event id="E" qual="Q" timestamp="T2" ...> event parameters </event>
  </happens>
  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` (as determined by the event narrative) **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,T)` are similar. Note, as already stated, there will be no <`terminates`> statements for multi-valued fluents, within an *ecXML* representation, as termination of a multi-valued fluent occurs just when it is initiated. Included in the EC axioms for multi-valued fluents, used within an *ecXML* reasoner, is the following definition:

  ```
  terminated(F,V,T1,T) if initiated(F,V1,T2,T) and T2>T1
  ```

  meaning that a multi-valued fluent has been terminated after `T1` but before or at `T`, if it has been initiated at or later than `T2` (later than `T1`), but before or at `T`.

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` does not hold 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.

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 facilitates the notion of contract templates, which are customised for particular scenarios. *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 agreement, 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. 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>
```

As can be seen, 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 (specifying 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*. In the example above, the <`para`> statements say that the customer's charge is $25.00 and the billing month is October.

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.

4. *ecXML* Representation of Example Contract

Now 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 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, as well as the effects on the contract whenever 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, which we will call `slg1`, 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`.

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

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

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>
```

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.

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.

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. ECSTA also supports contracts written in *ctXML*, which is a language for contracts that is less verbose than *ecXML*. We now present some details regarding capabilities of the architecture.

Contract Templates

An *ecXML* representation of a contract is loaded into ECSTA as a *contract template*. A contract template can be instantiated multiple times, with different *template parameters*, which allow the customisation of a contract template for a particular contract instance.

The functions pertaining to contract templates supported by ECSTA are:

- Discover Registered Contract Templates – For discovering the contract templates that are currently available for instantiation by the particular ECSTA deployment
- Register Contract Template – For registering a contract template, for later instantiation
- Deactivate Contract Template/s – For deactivating contract template/s, meaning that it/they can not be used for instantiation until reactivated
- Reactivate Contract Template/s – For reactivating deactivated contract template/s
- Destroy Contract Template/s – To remove contract template/s permanently from the ECSTA database, that is to destroy it/them

Contract Parameters

A contract (template) parameter allows for the customisation of a contract template for a particular contract instance. For example, the penalty that a service provider may pay to a "gold-standard" customer may be different from that paid to a "silver-standard" customer. That apart, the contract may be identical for both types of customer, and so the same contract template may be used with different values set for the given penalty contract parameter for contract instances pertaining to gold customers to those pertaining to silver customers.

The functions pertaining to contract parameters supported by ECSTA are:

- Discover Contract Parameters – Used to discover contract parameters of a contract template
- Change Contract Parameters – Used to change contract parameters of extant contracts belonging to the same contract template

Contract Instances

An *ecXML* contract loaded as a contract template may be instantiated to create a contract instance. The functions pertaining to contract instances supported by ECSTA are:

- Discover Instantiated Contracts – To discover instantiated contracts in the ECSTA database
- Instantiate Contract Template – To instantiate a contract template
- Reactivate Contract – To deactivate contract/s within the ECSTA database
- Deactivate Contract – To reactivate deactivated contract/s within the ECSTA database
- Destroy Contract – To permanently remove a contract from the ECSTA database, that is to destroy it
- Retrieve Contract – To retrieve contracts from the ECSTA database

Contract Clauses and User Rules

A contract will contain clauses and user rules, the aggregation of which constitute the contract. Clauses are normative in that their use has been agreed (a priori to contract instantiation) by all parties with an interest in the contract. The use of user rules in a contract is non-normative in the sense that there is no requirement for agreement on their use between parties, and indeed their use may not be transparent to all parties. User rules are typically employed by individual parties to realise additional inferences on contract state within their own ECSTA deployments.

The functions pertaining to contract clauses and user rules supported by ECSTA are:

- Add Contract Clauses and User Rules – To add contract clauses and user rules to contract/s and template/s
- Overwrite Timestamps in Contract Clauses and User Rules – To overwrite the timestamps of contract clauses and user rules in extant contract/s. Clauses and rules have associated temporal characterisations of when they are applicable – this function “overwrites” such temporal characterisations
- Register for Clause and User Rule Triggering Notification Events – To register for notification of the triggering of a particular clause or user rule on the given contracts / templates
- Deactivate Clause and User Rule Triggering Notification Events – To deactivate notification of the triggering of a particular clause or user rule on the given contracts / templates

- Reactivate Clause and User Rule Triggering Notification Events – to reactivate deactivated notification of the triggering of a particular clause or user rule on the given contracts / templates

Contexts

ECSTA also supports *what-if* experimentation, which we call *simulation*, where a speculative event narrative for a contract can be asserted and querying on the resultant contract can be carried out. For this purpose, we introduce the notion of contexts. There exists a *real context*, with which all contract instances are associated. The real-time contract monitoring operates over this context. All other contexts are *simulation contexts*. For the purpose of performing simulation, a collection of contract instances, belonging to the real context or some extant simulation context, may be cloned to create a new simulation context. This cloning process creates duplicate contract instances. Contract events are asserted to a particular context, either the real context, or a simulation context. These events propagate down to the pertinent contract instances associated with the particular context. Querying is done at the level of a contract instance, associated with the real context, or a simulation context.

The functions pertaining to contexts supported by ECSTA are:

- Create Simulation Context – To create a simulation context, adding specified contracts to the new context
- Destroy Simulation Context – To delete a simulation context

Contract Events

Contract events are either input events or output events. Input events are events that are presented to ECSTA for it to process their effects on contract state. Output events are events that ECSTA generates based on contract clauses and user rules. Note output events are fed back as input events for further inference. There is an additional distinction made between contract output events, which are generated according to contract clauses, and user output events, which are generated according to user rules.

The functions pertaining to contract events supported by ECSTA are:

- Assert Input Contract Events – To assert contract contracts to a context.
- Register for Notification of Output Contract Events – To register an interest in being informed of particular contract events on given contracts / templates
- Deactivate Notification of Output Contract Events – To deactivate an interest in being informed of particular contract events on given contracts / templates
- Reactivate Notification of Output Contract Events – To reactivate an interest in being informed of particular contract events on given contracts / templates

Contract Querying

Contract querying can happen over the real and simulation contexts. The functions pertaining to contract querying supported by ECSTA are:

- Query Contract State – To query the state of a contract at a particular time
- Query Contract State History – To query the state history of a contract up until a particular time
- Query Contract Norm or Contract Variable State – To query the state, that is value, of a particular contract norm or contract variable, at a particular time
- Query Contract Norm or Contract Variable State History – To query the state history, that is value, of a particular contract norm or contract variable, up till a particular time

Shared Variables

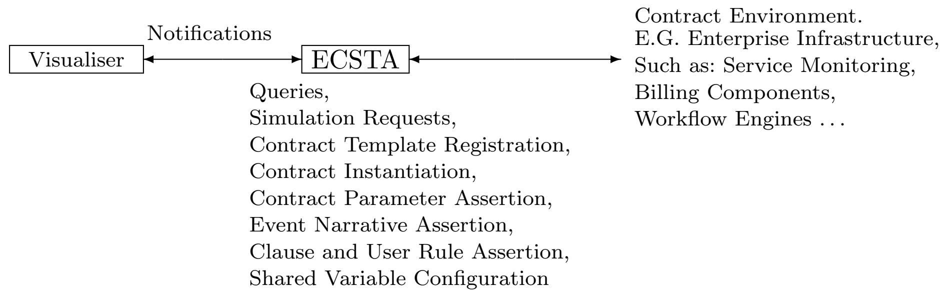
Shared variables are used for maintaining inter-contract state. The utility of shared variables is highlighted in the following example. Say in the context of web service provision, an *incident manager*, responsible for handling the effects of infrastructure failures or incidents on the fulfilment of 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 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* functions (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 function.*

The functions pertaining to shared variables supported by ECSTA are:

- Allocate Shared Variable – To allocate a shared variable for inter-contract reasoning
- Destroy Shared Variable – To destroy a shared variable for inter-contract reasoning
- Register Shared Variable Association – To register an association between a shared variable and a variable name used in individual contracts / templates
- Deactivate Shared Variable Association – To deactivate an association between a shared variable and a variable name used in an individual contract / templates
- Reactivate Shared Variable Association – To reactivate a deactivated association between a shared variable and a variable name used in an individual contract / templates

Fig. 1. Relationship between ECSTA and Contract Visualiser



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 ECSTA functions presented in section 5. The relationship between ECSTA and Contract Visualiser is depicted in figure 1.

In the following narrative, we present the evolution of a scenario, pertaining to the mail service agreement used in this paper, as it would be captured within Visualiser. As screenshots may be hard to read, we present the scenario in the form of tables which capture the same information as would be presented by Visualiser. For illustration, an actual screenshot for the final stage of the scenario is shown in figure 2.

In stage 1 of the scenario – see table 1 – we see that the state of the (mail service) contract instance is "OK" to begin with.

Table 1. Scenario Unfolds: Stage 1

Occurrence	Date/Time
`STATE: OK`	`Fri 3 Sep 2004 22-15-03`

In stage 2 – see table 2 – we see that a "Service Violation" event occurs causing the state of the contract instance to change to "Service Violation" and an obligation to be initiated bearing on the provider to restore the service.

Table 2. Scenario Unfolds: Stage 2

Occurrence	Date/Time
`STATE: Service Violation`	`Mon 13 Sep 2004 22-15-03`
`INPUT EVENT: SERVICE VIOLATION with (id: slg1)`	`Mon 13 Sep 2004 22-15-03`
`OUTPUT EVENT: OBLIGATION with (id: o0, bearer: provider, actions: resolve breech with (id: slg1), deadline: not specified)`	`Mon 13 Sep 2004 22-15-03`

In stage 3 – see table 3 – we see that a "Service Restoration" event occurs causing the state of contract instance to return to "OK". Also the obligation bearing on the provider to restore the service is fulfilled.

Table 3. Scenario Unfolds: Stage 3

Occurrence	**Date/Time**
STATE: OK	Mon 13 Sep 2004 22-45-03
INPUT EVENT: SERVICE RESTORATION with (id: slg1)	Mon 13 Sep 2004 22-45-03
INPUT EVENT: OBLIGATION with (id: o0, status: fulfilled)	Mon 13 Sep 2004 22-45-03

In stage 4 – see table 4 – we see that two obligations are initiated (by timers that are specified in the contract instance 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 contract instance to move into state: "Provider Payment Outstanding" + "Customer Payment Outstanding".

Table 4. Scenario Unfolds: Stage 4

Occurrence	**Date/Time**
STATE: Provider Payment Outstanding, Customer Payment Outstanding	Tue 14 Sep 2004 22-15-03
OUTPUT EVENT: OBLIGATION with (id: o1, bearer: provider, actions: refund money with (amount: 25.00), deadline: end bus. day)	Tue 14 Sep 2004 22-15-03
OUTPUT EVENT: OBLIGATION with (id: o2, bearer: Stirling Efforts, actions: pay for service with (amount: 50.00), deadline: 1 month)	Tue 14 Sep 2004 22-15-03

In stage 5 – see table 5 – 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 contract instance 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.

Table 5. Scenario Unfolds: Stage 5

Occurrence	**Date/Time**
STATE: Customer Payment Outstanding	Tue 14 Sep 2004 22-25-13
INPUT EVENT: OBLIGATION with (id: o1, status: fulfilled)	Tue 14 Sep 2004 22-25-13

In stage 6 – see table 6 – we see that the 1 month timer for the obligation

Fig. 2. Final stage of mail service scenario.

Contract Visualiser

Zoom Describe Saturday 16 October 2004 18-19-32 Help

State History. Contract #4, Customer:Mike Consulting, Template:Mail Service, Name:For Mike Maxwell.

Occurrence	Date/Time
STATE: OK	Friday 3 September 2004 22-15-3
STATE: Service Violation	Monday 13 September 2004 22-15-3
INPUT EVENT: SERVICE VIOLATION with (id: slg1)	Monday 13 September 2004 22-15-3
OUTPUT EVENT: OBLIGATION with (id: o0, bearer: provider, actions: resolve breech with (id: slg1), deadline: not specified)	Monday 13 September 2004 22-15-3
STATE: OK	Monday 13 September 2004 22-45-3
INPUT EVENT: SERVICE RESTORATION with (id: slg1)	Monday 13 September 2004 22-45-3
INPUT EVENT: OBLIGATION with (id: o0, status: fulfilled)	Monday 13 September 2004 22-45-3
STATE: Provider Payment Outstanding, Customer Payment Outstanding	Tuesday 14 September 2004 22-15-3
OUTPUT EVENT: OBLIGATION with (id: o1, bearer: provider, actions: refund money with (amount: 25.00), deadline: end bus. day)	Tuesday 14 September 2004 22-15-3
OUTPUT EVENT: OBLIGATION with (id: o2, bearer: Mike Consulting, actions: pay for service with (amount: 50.00), deadline: 1 month)	Tuesday 14 September 2004 22-15-3
STATE: Customer Payment Outstanding	Tuesday 14 September 2004 22-25-3
INPUT EVENT: OBLIGATION with (id: o1, status: fulfilled)	Tuesday 14 September 2004 22-25-3
STATE: Terminable	Thursday 14 October 2004 22-15-3
INPUT EVENT: OBLIGATION with (id: o2, status: timeout)	Thursday 14 October 2004 22-15-3
STATE: Terminated	Friday 15 October 2004 22-15-3
INPUT EVENT: TERMINATE AGREEMENT	Friday 15 October 2004 22-15-3

bearing on the service customer to pay for service has expired: this moves the contract instance into a "Terminable" state – the Service Provider is empowered to terminate the contract instance.

Table 6. Scenario Unfolds: Stage 6

Occurrence	Date/Time
`STATE: Terminable`	`Thu 14 Oct 2004 22-15-03`
`INPUT EVENT: OBLIGATION with (id: o2, status: timeout)`	`Thu 14 Oct 2004 22-15-03`

In stage 7 – see figure 2 and table 7 – we see that, in keeping with the Service Provider being empowered to terminate the service, they do so; the contract instance moves into a "Terminated" state.

Table 7. Scenario Unfolds: Stage 7

Occurrence	Date/Time
`STATE: Terminated`	`Fri 15 Oct 2004 22-15-03`
`INPUT EVENT: TERMINATE AGREEMENT`	`Fri 15 Oct 2004 22-15-03`

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 [1,2], 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 [19,24], an argumentation protocol based on Brewka's reconstruction of Rescher's Theory of Formal Disputation (RTFD) [10], 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 $\mathcal{C}/\mathcal{C}+$ [11]. $\mathcal{C}/\mathcal{C}+$ provides a high-level notation for defining laws 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 $\mathcal{C}/\mathcal{C}+$, called $(\mathcal{C}/\mathcal{C}+)^{++}$ [28], 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 [5], Bandara and colleagues develop methods for performing analysis and refinement of policy specifications, employing an EC-based representation of 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 [26], Sadighi and colleagues develop an EC-based framework for issuing privileges to agents in a community, through *declaration* and *revocation* authority certificates. A distinction is made 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 [20] Milosevic and colleagues attempt to identify the scope for automated management of e-contracts, including contract drafting, negotiation and monitoring. In [3] 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.

In [6], 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 [7] 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.

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 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 is 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 [8] we have discussed a higher-level syntax that we have defined, called *Contract Tracking 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.

We have assessed the adequacy of *ecXML* in expressing how the normative state of a contract evolves according to a narrative of contract events, by considering the representation of tens of service agreements from the domain of web services and other service-oriented domains, such as Utility Computing [12]. We have found it to be sufficient for 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 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 [21,22], value-passing CCS [16], π-calculus [17] and various logic-based formalisms such as $\mathcal{C}/\mathcal{C}+$ [11].

References

1. A.Artikis. *Executable Specification of Open Norm-Governed Computational Systems*. PhD thesis, Department of Electrical and Electronic Engineering, Imperial College, London, 2003.
2. A.Artikis, J.Pitt, and M.J.Sergot. Animated Specifications of Computational Societies. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS), Bologna*, pages 1053–1062, 2002.
3. A.Daskalopulu. Modelling Legal Contracts as Processes. In *11th International Confer-*

ence and Workshop on Database and Expert Systems Applications, pages 1074–1079. IEEE C.S. Press, 2000.
4. A.J.I.Jones and M.J.Sergot. A Formal Characterisation of Institutionalised Power. *Journal of the IGPL*, 4(3):429–45, June 1996.
5. A.K.Bandara, E.C.Lupu, and A.Russo. Using Event Calculus to Formalise Policy Specification and Analysis. *4th IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2003), Lake Como, Italy*, 2003.
6. A.S.Abrahams. *Developing And Executing Electronic Commerce Applications with Occurrences*. PhD thesis, University of Cambridge, 2002.
7. 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, *1st ACM Conference on Electronic Commerce (EC-99), Denver, Colorado, USA*. ACM Press, New York, NY, USA, November 1999.
8. Andrew D.H.Farrell, Marek J.Sergot, Claudio Bartolini, Mathias Salle, David Trastour, and Athena Christodoulou. 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 2004), San Diego, CA, USA*, 6 July 2004.
9. G.Alonso, F.Casati, H.Kuno, and V.Machiraju. *Web Services. Concepts, Architectures and Applications (ISBN: 3-540-44008-9)*. Springer, 2004.
10. G.Brewka. Dynamic Argument Systems: A Formal Model of Argumentation Processes Based on Situation Calculus. *Journal of Logic and Computation*, 11(2):257–282, 2001.
11. Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic Causal Theories. *Artificial Intelligence*, 153:49–104, 2004.
12. http://www.hp.com, Hewlett-Packard. HP Utility Data Center - Technical White Paper. 2001.
13. Sushil Jajodia, Pierangela Samarati, V.S.Subrahmanian, and Eliza Bertino. A Unified Framework for Enforcing Multiple Access Control Policies. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 474–485. ACM Press, New York, NY, USA, 1997.
14. Andrew Jones and José Carmo. Deontic logic and Contrary-to-duties. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic (Second Edition): Volume 8*. Kluwer Academic Publishers, 2002.
15. Lars Lindahl. *Position And Change, A Study in Law and Logic. Synthese Library Volume 122*. D.Reidel Publishing Company, 1977.
16. Robin Milner. *Communication and Concurrency (ISBN: 0-13-115007-3)*. Prentice-Hall, 1989.
17. Robin Milner. *Communicating and Mobile Systems: The Pi-Calculus (ISBN:0-521-65869-1)*. Cambridge University Press, 1999.
18. M.J.Sergot. A Computational Theory of Normative Positions. *ACM Transactions on Computational Logic*, 2(4):581–622, October 2001.
19. M.Sirbu. Credits and Debits on the Internet. *IEEE Spectrum*, 34(2):23–29, 1997.
20. O.Marjanovic and Z.Milosevic. Towards Formal Modelling of e-Contracts. In *Proceedings of 5th International Enterprise Distributed Object Computing Conference (EDOC 2001), 4-7 September 2001, Seattle, WA, USA*, pages 59–68. IEEE Computer Society, 2001.
21. Wolfgang Reisig and Grzegorz Rozenberg. *Lectures on Petri Nets I: Basic Models (ISBN 3-540-65306-6)*. 1998.
22. Wolfgang Reisig and Grzegorz Rozenberg. *Lectures on Petri Nets II: Applications (ISBN: 3-540-65307-4)*. 1998.

23. R.Kowalski and M.Sergot. A Logic-Based Calculus of Events. *New Generation Computing*, 4:67–95, 1986.
24. R.Smith and R.Davis. Distributed Problem Solving: the Contract-net Approach. In *Proceedings of Conference of Canadian Society for Computational Studies of Intelligence*, pages 217–236, 1978.
25. F. B. Sadighi and M. J. Sergot. Power and Permission in Security Systems. In Bruce Christianson, Bruno Crispo, James A. Malcolm, and Michael Roe, editors, *Security Protocols. 7th International Workshop, Cambridge, April 1999*, LNCS 1796, pages 48–53. Springer, 2000.
26. F. B. Sadighi, M. J. Sergot, and O. Bandemann. Using Authority Certificates to Create Management Structures. In Bruce Christianson, Bruno Crispo, James A. Malcolm, and Michael Roe, editors, *Security Protocols. 9th International Workshop, Cambridge, April 2001*, LNCS 2467, pages 134–145. Springer, 2002.
27. John R. Searle. *Speech Acts*. Cambridge University Press, Cambridge, 1969.
28. Marek Sergot. The language $(\mathcal{C}+)^{++}$. In J. Pitt, editor, *The Open Agent Society*. Wiley, 2005. (In press). Extended version: Technical Report 2004/8. Department of Computing, Imperial College, London.
29. Murray Shanahan. The Event Calculus Explained. In M.J.Wooldridge and M.Veloso, editors, *Artificial Intelligence Today, Lecture Notes in Artificial Intelligence*, volume 1660, pages 409–430. Springer, 1999.