Contract-related Agents

John Knottenbelt and Keith Clark

Dept of Computing, Imperial College London,
180 Queens Gate, London, SW7 2AZ, UK
{jak97,klc}@imperial.ac.uk

Abstract. We propose a simple event calculus representation of contracts and a reactive belief-desire-intention agent architecture to enable the monitoring and execution of contract terms and conditions. We use the event calculus to deduce current and past obligations, obligation fulfilment and violation. By associating meta-information with the contracts, the agent is able to select which of its contracts with other agents are relevant to solving its goals by outsourcing. The agent is able to handle an extendable set of contract types such as standing contracts, purchase contracts and service contracts without the need for a first-principles planner.

1 Introduction

Multi-agent systems is a growing research area and has already started to find application in industry in web services and the semantic web. There is also increased interest in agent coordination and choreography. Our approach sees contracts as a means of formally describing the relationships between agents in terms of obligations and permissions, as well as providing a coordination function.

By expressing the terms and conditions of a contract as a set of event-based rules – and so long as the participating agents agree on the history of events relevant to their contracts – an agent is able to obtain a completely unambiguous and indisputable view of the state of the contract at any given point in time.

We claim that the AgentSpeak(L)[11] architecture, with relatively few extensions, enables an agent to behave in a reactive manner (as is the case with service agents, where they react to obligations imposed on them) or a proactive manner where it makes use of agreed or newly proposed client contracts in order to impose obligations on other agents to do things for it. It may do this both to satisfy its own goals, or to discharged obligations it has arising from other contracts.

Starting off with a description of how contracts may be represented in the event calculus, we give an example of a short-term contract to conduct a purchase and a long-term standing contract to set-up short-term purchase contracts. In section 3 we discuss how the agents may communicate with each other and in section 4 we show how these communications can be used to effect the contract state (such as established facts, obligation fulfilment and violation). Section 5

briefly presents the AgentSpeak(L) architecture and how we have used and extended it to incorporate reasoning about contracts. We give the plan libraries for the customer and vendor agents which are able to monitor a general class of purchase and standing contracts, of which the contracts presented in section 2 are instances. Finally we review related work and concluding remarks.

2 Contract Representation

The core of the contract representation language is the event calculus [10], where communications are events and the contract rules specify how the events initiate and terminate obligation fluents. We make an implicit assumption that an agent is *permitted* to perform any communication that, taking into account the history of the use of the contract up to this point, will initiate an obligation on another party to the contract. This assumption has been sufficient for the examples studied so far, however, we do intend to investigate explicit representation of permissions in future work. In this paper we are using the Prolog variable syntax convention where variables begin with an uppercase letter.

We are using the full event-calculus[12] without the `releases` predicate since the examples we have considered so far do not require the use of non-inertial fluents. We also dispense with the `initiallyP` and `initiallyN` predicates by providing a contract start event, and writing `initiates(start, F, T)` and `terminates(start, F, T)` respectively. Figure 1 summarises the axioms of the event calculus.

```
holdsAt(F,T) ↔ happens(E,T1) ∧ T1<T ∧
  initiates(E,F,T1) ∧ not clipped(T1,F,T).
notHoldsAt(F,T) ↔ happens(F, T1) ∧ T1<T ∧
  terminates(E, F, T1) ∧ not declipped(T1,F,T).
clipped(T0,F,T1) ↔ ∃T[T0≤T ∧ T<T1 ∧ terminates(E,F,T)].
declipped(T0,F,T1) ↔ ∃T[T0≤T ∧ T<T1 ∧ initiates(E,F,T)].
```

Fig. 1. Event Calculus Summary

The body of a contract is represented by a binary relation, `contractClause`, between the label of the contract and the clauses belonging to the contract. Variables can be shared between the contract label and the clauses – those appearing in the label are conceptually parameters to the contract. If we were representing the contracts directly in Prolog, there would be one `contractClause` definition for each rule of the contract. For example, Figure 2 shows the first rule of a short term contract about a purchase transaction.

```
contractClause(
  customerVendorContract_purchase(customer[1]:C, vendor:V |
     vendorBank:VB, customerBank:CB, deliveryService:DS, item:I, price:P),
  initiates(start, oblig(V, achieve(value(invoice-no, _)), T+100), T).
```

Fig. 2. Contract Clause as Prolog

[1] To aid readability we employ the syntax `field:Value` to indicate a named field or parameter.

The label of the contract is `customerVendorContract_purchase(...)`. The parameters to the left of the `|` are the principals of the contract, usually the offeror and the offeree in a normal two party contract. The rule reads that at the start of the contract, the vendor (`V`) is obliged to announce the invoice number (`invoice-no`) relating to the purchase within 100 time units. `start` is the event marking the start of the contract's lifetime. `oblig` is a 3-place fluent relation between the bearer, the goal to be performed and its deadline. `achieve` indicates a state of affairs is to be achieved. `value` is a binary fluent relating contract variables to their values.

For ease of presentation, we adopt a more compact syntax, where the label is written once at the beginning of the contract, and the rules are written inside the following brace delimited block. Macro definitions for frequently used terms are written in small caps and marked with $\equiv$ and should be textually substituted by the reader as they occur.

2.1 Short-term Contracts

Figure 3 shows the full text for a purchase contract, parameterised by the item being purchased, the price, the vendor, the customer, the vendor's and customer's bank, and a delivery service.

```
customerVendorContract_purchase(
     customer:C, vendor:V |
     vendorBank:VB, customerBank:CB, deliveryService:DS, item:I, price:P) {
  Paid(R) ≡ paid(payer:C, payee:V, price:P, reference:R).
  Delivered(R) ≡ delivered(item:I, destination:C, invoice-no:R).
  InvoiceOblig(DL) ≡ oblig(V, achieve(value(invoice-no, _), DL).
  PayOblig(R, DL) ≡ oblig(C, achieve(Paid(R)), DL).
  DeliverOblig(R, DL) ≡ oblig(V, achieve(Delivered(R)), DL).

  initiates(start, InvoiceOblig(T+100), T).
  initiates(E, PayOblig(R, T+100), T) ← initiates(E, value(invoice-no, R), T).
  initiates(E, DeliverOblig(R, T+300), T) ← initiates(E, value(invoice-no, R), T).
  initiates(E, owns(owner:C, item:I), T) ←
    holdsAt(value(invoice-no, R), T) ∧ initiates(E, fulfilled(PayOblig(R, _))).

  terminates(E, owns(owner:V, item:I), T) ←
    holdsAt(value(invoice-no, R), T) ∧ initiates(E, fulfilled(PayOblig(R, _))).

  authoritative(V, value(invoice-no, _)),
  authoritative(VB, Paid(_)).
  authoritative(CB, Paid(_)).
  authoritative(DS, Delivered(_)).
}
```

Fig. 3. Simple Purchase Contract

The first `initiates` rule, as described above, obliges the vendor to determine an invoice number and to signal this as an event notification for the contract. The vendor does this by sending an inform message to the customer (see sections 3 and 4.1). When the invoice number has been notified, the second and third

`initiates` rules oblige the customer to concurrently pay within 100 time units and the vendor to deliver within 300 time units and to signal completion as contract related events.

The last `initiates` and the only `terminates` rule indicate that the customer will become the new owner of the item when the payment obligation has been fulfilled.

The `authoritative` clauses indicate which agents are authoritative for which fluents. The concept is related to controllable propositions[7] — the difference being that in our work the controlled proposition is limited in scope to a specific contract, rather than to the entire agent society. In the example, only the vendor has the authority to notify an invoice number. A notification by the customer would not be considered a contract event.

This mechanism is also useful to identify trusted-third parties: the banks are authoritative for the paid fluents (that is any payment from customer to vendor) and the delivery agent is authoritative for the proof of delivery fluent. An agent that is not authoritative for a fluent may attempt to communicate it, but the communication would have no effect in this contract.

2.2 Long-term Contracts

It is useful to agree a contract about what contracts may be agreed in the future. In Figure 4 we give an example standing contract specifying prices for rolls of wire mesh, fixing screws and sheets of tin roofing. The vendor agent is constrained by the standing contract to accept any purchase proposals matching the agreed criteria. The purchase proposal is a reference, by means of the contract label, to the simple purchase contract presented above.

```
customerVendorContract(customer:C, vendor:V |
   vendorBank:VB, customerBank:CB, deliveryService:DS) {

  initiates(E, oblig(V, do(X, replyTo(X, E)), T+100), T) ←
    proposeEvent(E, C, V, _).
  initiates(E, oblig(V, do(X, acceptEvent(X, E), T+100), T) ←
    proposeEvent(E, C, V,
       customerVendorContract_purchase(customer:C, vendor:V |
         vendorBank:VB, customerBank:CB,
         deliveryService:DS, item:I, price:P)) ∧
     agreedPrice(item:I, price:P).

   agreedPrice(item:wiremesh(width:10, height:10, gauge:10), price:10.00).
   agreedPrice(item:fixingscrews(gauge:5, amount:1000), price 6.99).
   agreedPrice(item:tinroofing(width:6, height:9), price 4.00).
}
```

Fig. 4. Standing Contract

The first `initiates` rule specifies that the vendor must reply to proposals (of any kind) from the customer within 100 time units. The important syntax here is the `do(...)` notation which indicates that the agent must bring about an event satisfying a particular constraint. In this case the event must satisfy the

`replyTo` constraint, meaning that it must be a valid reply to the accept event (i.e. an accept or a reject). The following section defines the `replyTo`, `proposeEvent` and `acceptEvent` predicates, which are common to all agents. Similarly the second `initiates` rule specifies that the vendor must accept proposals meeting the `agreedPrice` criterion.

3 Communication

In our system, events are the act of sending messages. The messages are interpreted by the agents according to a shared ontology, the domain independent part of which is described here and in section 4. Only recorded events that are relevant to a contract may progress the contract state. Real world events, such as "the car leaving the drive way", may be put in the context of a contract by an inform event (or reported event, see below) to that effect.

We require that the events are observed by all contract principals because the state of the contract depends on the history of events relevant to it. In a two party contract, this requirement is trivially satisfied when one principal sends a message to another.

Although the exact format of a message and its transport details will vary from application to application and agent society to agent society, a well-formed message should include a time-stamp, a unique message identifier, a field identifying the message to which it is a reply (if any), message sender, message receiver, a message content, context and the interaction protocol or conversation identifiers. Messages relevant to a contract should include a context field corresponding to the contract label. This information can be inferred if the received message is in reply to an earlier message that was properly context-tagged. Figure 5 gives an example representation of an accept event for the long term contract between the customer and vendor (see Figure 4).

We list some predicates that can be used in the contract language and agent code either as tests on received messages, or as constraints on messages about to be sent (this commonly occurs when an agent is obliged to `do` a communication subject to some specified constraints). The predicates are implemented in terms of constraints on the message attributes above.

contractEvent(E, C) event `E` is in the context of contract `C`.

proposeEvent(PE, X, Y, P) `PE` is a propose event from `X` to `Y` for a proposal `P`. Proposals take the form of contract labels. Propose events must specify the propose protocol, which restricts the valid replies to accept or reject events.

acceptEvent(A, PE) `A` is an acceptance event in reply to a proposal event, `PE`. Accept events must specify the propose protocol. According to the protocol rules (see `replyTo` below), there can only be one reply to a proposal, so if there are any further accept or reject events they should be ignored. The receiving agent should check the validity of the proposal event.

rejectEvent(R, PE) `R` is a reject event in reply to a proposal event, `PE`. Like acceptEvents, rejectEvents should specify the propose protocol.

informEvent(IE, X, Y, F) `IE` is an inform event from `X` to `Y` that `F` is true. `F` is normally a fluent. Only agents that are authoritative about the fluent (see subsection 4.1) may establish it in the context of the contract.

replyTo(R, E) `R` is a reply to event `E`. If `E` specifies a protocol, `replyTo` constrains `R` to be a valid response in the protocol. `R` and `E` must share the same context and must agree on the protocol attribute. `R`'s `in-reply-to` attribute must equal `E`'s message identifier.

reportEvent(RE, E) indicates that RE is a report of an actual event E. This is most useful when E is an inform event from another contract that something has been achieved. Only events which actually occur may be reported – this constraint might be enforced by the requiring event senders to digitally sign their events.

requestEvent(E, A, F) `E` is a request event for agent `A` to bring about that F is true. A successful response is an inform event that `F` is now true.

```
accept(
  time:20050121144600, identifier:cv123, in-reply-to:cv122,
  sender:sales@wiremeshRus, receiver:jak97@imperial.ac.uk,
  content:propose(
    time:20050121130100,
    identifier:cv122,
    sender:jak97@imperial.ac.uk,receiver:sales@wiremeshRus,
    content:customerVendorContract(
      customer:jak97@imperial.ac.uk, vendor:sales@wiremeshRus,
      vendorBank:finance@bank1, customerBank:finance@bank2,
      deliveryService:deliver@pforce),
    protocol:propose),
  protocol:propose).
```

Fig. 5. Possible representation of an accept event

4 Contract Evaluation

An agent may have many contracts active at the same time. It is important to be able to consider the contracts independently of each other (for example to determine a contract's state), and also their combined effect (for example when outsourcing goals). For this reason we define a meta-interpreter predicate, `selon`[2], which evaluates queries relative to a specified contract. Subsection 4.3 describes how the individual contract effects are combined into the agent's belief store.

Figure 6 shows the core of the meta-interpreter. The first parameter is the contract label, and the second is the formula to be evaluated. The agent can now query what obligations are current with respect to a contract by asking `selon(C, holdsAt(oblig(A, G, DL), Now))` where `Now` is a time point representing the current time and `C` is the label of an active contract.

The symbols `not`, $\wedge$ and $\leftarrow$ are overloaded. Where they occur in a functional context (in the second argument to `selon`), they should be read as functional

[2] From the French, *selon*, meaning "according to"

```
selon(C, happens(E, T)) ← happens(E, T) ∧ contractEvent(E, C).
selon(C, happens(start, T)) ← happens(E, T) ∧
  initiates(E, activeContract(C), T).
selon(C, R) ← contractClause(C, R ← S) ∧ selon(C, S).
selon(C, P ∧ Q) ← selon(C, P) ∧ selon(C, Q).
selon(C, not P) ← not selon(C, P).
selon(C, holdsAt(F, T)) ← selon(C, happens(E, T1)) ∧ T1<T ∧
  selon(C, initiates(E, F, T1)) ∧ selon(C, not clipped(T, F, T1)).
selon(C, clipped(T0, F, T1)) ← selon(C, happens(E, T)) ∧ T0≤T ∧ T<T1 ∧
  selon(C, terminates(E, F, T)).
```

Fig. 6. Contract meta-interpreter

terms; where they occur in a logical context (as part of the definition of `selon`) they should be read as logical connectives. Further meta-interpreter rules are presented below defining the concepts of authoritative agents, reported events and obligation fulfilment and violation.

4.1 Authoritative Agents

In the simple purchase contract, the vendor was authoritative for the invoice number. We capture this authority with an extension to the meta-interpreter:

```
selon(C, initiates(E, F, T)) ←
  selon(C, authoritative(X, F)) ∧ informOrReportedEvent(E, X, F).

selon(C, terminates(E, G, T)) ← selon(C, incompatible(F, G)) ∧
  selon(C, authoritative(X, F)) ∧ informOrReportedEvent(E, X, F).

informOrReportedEvent(E, X, F) ← informEvent(E, X, _, F) ∨
  (reportEvent(E, I) ∧ informEvent(I, X, _, F)).
```

In the simple purchase contract, the delivery agent is authoritative for the delivery fluent. The delivery agent is not a principal of the contract, however, so in order for any delivery notification to have effect, it must be reported by one of the principals (in this case the customer or the vendor). Direct or indirect reporting of an inform event from the authoritative agent is deemed to be a valid contract event by virtue of the last rule.

We have borrowed the `incompatible` predicate, which states which fluents must be terminated in response to one being initiated, from the original event calculus[10].

4.2 Obligation Fulfilment and Violation

We adopt a similar semantics to Dignum et al.[6] with respect to deadlines. An obligation is fulfilled if the deadline has not yet expired. If the obligation was to achieve a state of affairs represented by a fluent fulfillment has to have been notified by an event that initiates the contract fluent. Where it was a more direct obligation to bring about an event characterized by a constraint, that the event has occurred is checked by showing that the constraint is now satisfied.

An obligation is violated if the deadline has elapsed and it has not been fulfilled. For simplicity's sake, we omit rules allowing a violation to be repaired (by meeting its sanction). We need three meta-interpreter rules to capture this: two for `achieve` and `do` fulfilment and one for violation.

```
selon(C, initiates(E,
  fulfilled(oblig(X, achieve(F), DL)), T)) ←
      selon(C, holdsAt(oblig(X, achieve(F), DL), T)) ∧
      T<DL ∧ selon(C, initiates(E, F, T)).
selon(C, initiates(E,
  fulfilled(oblig(X, do(E, Constraint), DL)), T)) ←
    selon(C, holdsAt(oblig(X, do(E, Constraint), DL), T))
    ∧ T<DL ∧ selon(C, Constraint).
selon(C, violated(oblig(X, G, DL), T)) ←
  selon(C, holdsAt(oblig(X, G, DL), T)) ∧ DL<T ∧
  not selon(C, holdsAt(fulfilled(oblig(B, G, DL)), T)).
```

4.3 Imported Fluents

Event calculus is used not only within the contract language definition, but also by the agent at the top-level to manage its beliefs. We need some rules to model that certain contracts have effects on the agent society outside of the contract itself. An example of this is the simple purchase contract which concludes with the transfer of ownership of the item from the vendor to the seller: reasoning solely with respect to the purchase contract will not allow the agent to realise that it does not own the item after selling it in the future. Since we need to track the ownership changes over the course of several contracts, we pool the ownership fluent into the agent's own belief store:

```
initiates(E, F, T) ← importedFluent(F) ∧
  holdsAt(activeContract(C), T) ∧ selon(C, initiates(E, F, T)).

terminates(E, F, T) ←  importedFluent(F) ∧
  holdsAt(activeContract(C), T) ∧ selon(C, terminates(E, F, T)).

importedFluent(owns(_, _)).
importedFluent(activeContract(_)).
```

The `importedFluent` predicate selects which contract fluents should be imported into the agent's belief store. `activeContract` is a fluent predicate indicating which contracts are active. Marking it as an imported fluent allows contracts to spawn sub-contracts.

5 Agent Architecture

We now describe an agent architecture in the style of AgentSpeak(L) to enable agents to respond to events related to all their active contracts in a timely fashion. We give a brief introduction to a simplified version of AgentSpeak(L), and then propose a plan library for the customer agent that will allow it to make use of the standing and purchase contracts. AgentSpeak(L) is chosen as a basis because it has a well understood operational semantics and there are available implementations such as [16] and [4].

5.1 AgentSpeak(L)

An AgentSpeak(L) agent architecture can be viewed as multi-threaded event-triggered interruptible logic programming system [11].

There are two kinds of events, belief updates and new goal events. Belief updates are represented as `+b` or `-b` depending on whether the particular belief, `b`, is now true or false. Belief events model the changes in the environment as perceived by the agent. New goal events are represented as `+!g`, where `g` is the goal to achieve.

At the beginning of the agent cycle, the agent picks an event to handle from the set of unhandled events. The plan library is consulted to see if there are any plans that are triggered by the event. Each plan in the plan library has the syntax: `event:condition <- actions.`

For example, `+temperature(T) : T > 90 <- switch(heater, off).` is a plan from an environmental control agent. The plan is relevant to changes in temperature, and applicable when the temperature rises beyond 90 degrees. The action is to switch the heater off.

If the `event` in the head of the plan unifies with the selected event, the plan is said to be relevant. The `condition` is a formula in terms of the current beliefs of the agent and acts as a guard: the relevant plans whose condition formula evaluates to true are said to be applicable. Finally, one plan is selected from the applicable plans and an intention is created to monitor it.

The agent then picks an intention to execute, which involves executing the plan body (`actions`) one step at a time. A step may be either a physical action, an achieve goal (written `!goal`) or a test (written `?test`).

Goal achievement is handled by suspending the intention and adding a new goal event to the set of unhandled events. Future agent cycles will pick up the new goal event, and look in the plan library (as before) for an applicable plan to achieve it. The plan is then stacked on top of the intention that issued the achieve goal action, so that once the goal has been achieved, execution of that intention may continue.

Tests are queries to the agent belief store, and result in a set of variable assignments which are substituted into the remaining plan steps.

5.2 Extensions to AgentSpeak(L)

We extend the AgentSpeak(L) in the following ways:

- Plans may include belief update steps, of the form `+b` or `-b`. This effects the belief store of the agent in a similar way to Prolog's assert and retract.
- An agent may have an initial set of desires, which can be selected and posted as new goal events.
- In the example plan libraries below, we have also included some Prolog style horn clauses to ease readability. Since these definitions can be folded directly into the AgentSpeak(L) rules, they do not affect the operational semantics.
- "Fire and forget" goal execution, written `!!goal`. Instead of stacking the plan for the goal on top of the existing intention, create a separate intention for the achievement of the goal. This is useful when the agent requires simply to start off a process to achieve a goal, but not to wait for its achievement.

– The textually first applicable plan is selected if there is a choice and the agent commits to that plan.
– We implement the following physical actions:
  `notify` sends a message to all the principals of a given contract (see section 3), and logs it as a communication event.
  `waitReply` waits for a reply to a given notification message to be received, subject to a timeout.
  `waitContractEvent` waits for an event that is relevant to the specified contract. This is either an incoming communication event, or the lapsing of any of the current obligations' deadlines.
  `fail` abandons an executing plan and marks it as failed.

5.3 Meta-information about Contracts

Instead of writing a set of plans to address specific contracts, such as `customerVendorContract`, we can write plans that address a general class of standing and purchase contracts. We do this by abstracting common behaviour into agent-specific meta-information about the contracts. We define a binary relation `isa` which is true iff a particular contract belongs to a more general class of contracts. The schematic rules below we say that `customerVendorContract_purchase` as an instance of `purchaseContract` and `customerVendorContract` as an instance of `standingContract` that can be used to create new `customerVendorContract_purchase` contracts, so long as the item and price information match up with the agreed prices in the standing contract.

```
customerVendorContract_purchase(X) isa purchaseContract(X).
customerVendorContract(Y) isa standingContract(PC) :-
   PC isa purchaseContract(Y, item:I, price:P)
   selon(customerVendorContract(Y), agreedPrice(item:I, price:P)).
```

It is also useful to know when a contract is complete. This is dependent on the specific type of contract. For example, the standing contract above is open ended - it is never completed, whereas the purchase contract ends successfully with ownership of the item. `complete` is a binary predicate, first argument is the contract and the second argument is the time of evaluation.

```
complete(PC, T) :-
   PC isa purchaseContract(customer:C, _, item:I, _, _, _, _),
   selon(PC, holdsAt(owns(owner:C, item:I), T)).
```

5.4 Plan Library for Contract Execution

We now describe a plan library for executing arbitrary contracts. For each active contract, `C`, the agent must ensure that there is an intention to abide by it, by invoking a plan for the goal `monitor(C)`.

In the case of the vendor, we assume an initial desire to abide by their standing contract, which will result in the goal to `monitor` it. However, as this standing contract does impose any obligations on the customer agent, it is not necessary for that agent to actively monitor it. As we shall see, the customer may instead make use of the contract to achieve an ownership goal by creating an active purchase sub-contract that it will monitor.

```
+!monitor(C): now(Now) & complete(C, Now) <- true.
+!monitor(C): now(Now) & selon(C, holdsAt(oblig(Self, G, DL), Now)) &
               not(observed(C,oblig(Self, G, DL)))
<- +observed(C,oblig(Self, G, DL)) ; !G by DL in C ;
     !monitor(C).
+!monitor(C): now(Now) & selon(C, holdsAt(fulfilled(Oblig), Now)) &
               not(observed(C,fulfilled(Oblig))
<- +observed(C,fulfilled(Oblig)) ; !obligFulfilled(C, Oblig);
     !monitor(C).
+!monitor(C): now(Now) & selon(C, violated(Oblig, Now)) &
               not(C,observed(violated(Oblig)))
<- +observed(C,violated(Oblig)); !obligViolated(C, Oblig) ;
    !monitor(C).
+!monitor(C): true <- waitContractEvent(C) ; !monitor(C).
```

The conditions of the above plans query the state of the contract using the `selon` predicate. The `now` predicate gives the current time. When the contract is complete, as defined by the `complete` predicate in the contract meta-information, the first rule is applicable and the execution plan terminates.

The second rule states that if there is a new obligation on the agent, a goal of the form `G by DL in C` is posted. This goal event will be handled by other plans in the plan library (see customer and vendor agent's plan libraries below). When the plan to achieve the goal completes, the `monitor(C)` goal is reposted to carry on contract execution.

The third and forth rules monitor the contract for obligation fulfilment and violation. `obligFulfilled` and `obligViolated` goals are posted, which may be handled elsewhere in the agent's plan library to keep track of, for example, the reliability and reputation of the contract participants.

The last rule states that if the contract is not yet complete, the agent waits for a communication event or for the earliest outstanding obligation deadline to lapse before consulting the contract again. Although the condition of the last plan is always true, our plan selection function selects the textually first applicable plan.

5.5 Plan Library for Customer Agent

The role of the customer agent is to respond to desires to own an item. These desires are manifested by achievement goals, which gives rise to intentions to satisfy them. We show how the agent may make use of standing contracts (or other means of achieving ownership) in an example plan library.

```
+!owns(owner:Self, item:I) :
    now(Now) & not(holdsAt(owns(owner:Self, item:I), Now)) &
    holdsAt(activeContract(SC), Now) &
    SC isa standingContract(PC) &
    PC isa purchaseContract(customer:Self, vendor:V, item:I, price:P,
            vendorBank:VB, customerBank:SelfBank, deliveryService:DS) &
    reliable(V), reliable(DS) &
    fairPrice(item:I, price:P)
```

```
 <- ?proposeEvent(Proposal, Self, V, PC) ;
    notify(SC, Proposal) ;
    waitReply(Reply, Proposal, Now + 100) ;
    !enact(Reply, Proposal).

+!enact(timedOut, Proposal) : true <- +noResponseTo(Proposal).
+!enact(Reply, Proposal) : rejectEvent(Reply, Proposal) &
    rejectEvent(Proposal, Self, V, _) <- +rejected(Proposal).
+!enact(Reply, Proposal) : acceptEvent(Reply, Proposal) &
    proposeEvent(Proposal, Self, V, PC) <- !monitor(PC).
```

The plan above is applicable to the goal of achieving ownership of a particular item, if the agent does not already own it, and there is an agreed standing contract mandating an acceptable price for the item with a (believed) reliable vendor and delivery service. The plan body constructs a proposal event and sends it to the vendor in the context of the standing contract, waits for a reply and then acts on that reply. The vendor agent is obliged to respond with an accept event within 100 time units, and should they fulfill that obligation the resulting purchase contract will be monitored by the customer. If no response comes in time, or the proposal is rejected, a belief to that effect is stored effecting the customer's future reliability estimate of the vendor.

There is only one possible obligation on the customer arising from the purchase contract, and that is to pay for the item. We make the simplifying assumption that the customer has enough money in his account:

```
+!achieve(paid(payer:Self, payee:V, price:P, reference:R)) by DL in PC :
    now(Now) & holdsAt(activeContract(BC), Now) &
    BC isa bankContract(customer: C, bank:B)
<- ?requestEvent(Request, SelfBank, paid(payer:C, payee:V, price:P,
      reference:R)) ;
   notify(BC, Request) ; waitReply(Reply, Request, Now + 100);
   ?reportEvent(Report, Reply);
   notify(PC, Report).
```

After instructing the bank to transfer the money (in the context of the contract between the customer and their bank), the customer waits for an acknowledgement that this has been done and forwards it to the vendor in the context of the purchase contract. It is this reported event that causes the paid fluent to become established, and consequently for the customer to have fulfilled the payment obligation to the vendor (see section 4.1).

5.6 Plan Library for Vendor Agent

The following plan library enables the vendor to accept and reject proposals for purchase contracts. If the vendor is obliged to accept it, then by the generic contract execution plan library, a goal will be posted to of the form `do(X, acceptEvent(X, E)) by DL in SC` which is handled by the plan below.

```
+!do(X, acceptEvent(X, E)) by DL in SC: proposeEvent(E, C, Self, Proposal)
<- ?acceptEvent(X, E) ; notify(SC, X) ; !!monitor(Proposal).
```

The condition of this plan extracts the proposed contract from the content of the message, and the plan body constructs an accept event and sends it the customer in the context of the standing contract. A separate intention is then created to monitor the proposed contract. The plan for rejecting a proposal is similar, except no intention is created to monitor the proposed contract.

Now we consider that the standing contract also obliges the agent to reply to a proposal even if it is not obliged to accept it. We define two auxiliary predicates `obligedToAccept` which is true iff the vendor is obliged to accept the proposal, and `shouldAccept` which is true iff the it is in the vendor's interest to accept the proposal. `shouldAccept` includes checks like there is available stock, taking into account already committed stock, and that the proposed price of the item is at least twice the cost price.

```
obligedToAccept(SC, E, T) :-
    selon(SC, holdsAt(oblig(Self, do(X, acceptEvent(X, E)), DL), T)).

shouldAccept(Proposal, T) :-
    Proposal isa purchaseContract(customer:C, vendor:Self | price:P,
              vendorBank:SelfBank, customerBank:CB, deliveryService:DS),
    costPrice(item:I, price:CostPrice),
    warehouse(item:I, availability:Warehouse),
    committed(item:I, level:Committed),
    Warehouse - Committed > 0,
    P >= CostPrice * 2.
```

The following two plans make use of the predicates to decide whether to accept or reject the proposal. The plan bodies are simply goals to accept or reject which will be handled by the plans as the start of this subsection.

```
+!do(X, replyTo(X, E) by DL in SC: proposeEvent(E, C, Self, Proposal) &
    now(Now) & shouldAccept(Proposal, Now) &
    not(obligedToAccept(SC, E, Now))
<- +!do(X, acceptEvent(X, E)) by DL in SC.

+!do(X, replyTo(X, E) by DL in SC: proposeEvent(E, C, Self, Proposal) &
    now(Now) & not(shouldAccept(Proposal, Now)) &
    not(obligedToAccept(SC, E, Now))
<- +!do(X, rejectEvent(X, E)) by DL in SC.
```

There are two obligations that may result on the vendor during execution of the purchase contract. The first is an obligation to announce an invoice number, and the second is to arrange for delivery of the item.

```
+!achieve(value(invoice-no, _)) by DL in PC : true
<- -invoiceNo(Last) ;
    ?New is Last + 1; +invoiceNo(New) ;
    ?informEvent(E, Self, value(invoice-no, New)) ; notify(PC, E).
```

To achieve a fresh value for the invoice number, the vendor increments a belief atom, `invoiceNo`, and then creates an inform event asserting its value to send to the customer in the context of the purchase contract, `PC`.

```
+!achieve(delivered(item:I, destination:C, invoice-no:R)) by DL in PC :
   PC isa purchaseContract(customer:C, vendor:Self | price:P,
            vendorBank:SelfBank, customerBank:CB, deliveryService:DS) &
   now(Now) & holdsAt(activeContract(DC), Now),
   DC isa deliveryContract(customer:V, deliveryService:DS)
<- ?requestEvent(Request,DS,delivered(item:I,destination:C,invoice-no:R)) ;
   notify(DC, Request) ; waitReply(Reply, Request, Now + 100) ;
   ?reportEvent(Report, Reply) ; notify(PC, Report).
```

The vendor agent has no built-in capability to achieve delivery, so it must make use of a third party. The above plan checks that its has an active contract, DC, with the delivery service `DS` mentioned in the purchase contract (which we assume it will have).

The vendor must create a request event to achieve the delivery and sent it to DS in the context of `DC`. If successful, the delivery service will reply with an inform that the item has been delivered, which is then forwarded by the vendor to the customer in the context of the purchase contract, thus fulfilling the vendor's obligation to deliver the item.

6 Related Work

Our architecture shares the concepts of plan library, beliefs, intentions with AgentSpeak(L). The addition of contracts not only reifies the concept of obligations, but also extends the built-in behaviour of the agent by allowing it to outsource goals that it cannot achieve itself.

In Agent0[13], agents are programmed by specifying a set of capabilities (commitment rules). Instead of building the commitment rules directly into the agent, our architecture allows these rules to be specified in the contract in the form of event calculus initiates and terminates rules.

Verharen's cooperative information agents [14] [15] are based on the language action perspective. The architecture specifies three main categories of activities: tasks (plans to achieve tasks organised with dependencies between the tasks), transactions (message sequences organised with temporal ordering constraints), and contracts, which are represented as deontic state machines of transaction transitions. Our system does not mandate such a conceptual break down, rather we envisage that higher level contract languages may be translatable into our simpler event calculus syntax.

Social integrity constraints[1] can be used to specify and verify agent interaction protocols. The constraints express expectations on events that ought to (or ought not to) occur given the occurrence of a previous triggering event. Each expectation has an associated priority such that those with higher priority are more ideal. The concept is related to obligations, but their focus is different: social integrity constraints focus essentially on constraining the event history, whereas obligations focus on constraining a particular agent behaviour. It would be an interesting to attempt the expression of the one in terms of the other.

Artikis et al[2] describe a system for animating and specifying computational societies. The system takes a global perspective, and so in order to make inferences about the state of the society all the events relevant to it must be known.

This is in contrast to our work, where conclusions are reached relative to each contract rather than the society as a whole.

Kollingbaum and Norman describe a system of supervised interaction[8] where agents are supervised by a third party called an authority. The authority registers contracts between the agents, witnesses the communications between the agents and enforces the norms specified in the contracts. Our system does not require this infrastructure, although it does admit a logging agent should the particular situation demand it. Furthermore the agents themselves are responsible for enforcing the contractual norms.

It is important that the contract is carefully constructed so that prohibitions do not completely prevent the fulfillment of obligations. It is feasible to statically check contracts for these kinds of potential conflicts[3]. The NoA architecture[9] solves conflicts by prioritising permissions over prohibitions — obligation consistency is determined by considering the action effects of plans to handle the obligation. If all plans include actions that are prohibited or interfere with other obligations, the obligation is found to be inconsistent and is not adopted.

Conflicts between obligations and desires may also emerge, and if so conflict resolution will be important. The BOID architecture[5] describes a method of resolving these conflicts. Beliefs, obligations, intentions and desires are represented as separate components with feedback loops between them. Each component builds extensions (closure under logical consequence) of their propositional theories, and conflicts between the components are resolved by prioritising the components one over the other.

7 Conclusion

Contracts are a powerful and high level approach to programming agent behaviour. Furthermore, specifying the contractual relationships between agents separately to the agents' capabilities is not only good software engineering, because concerns are separated, but also facilitates analysis and verification since the contracts are represented in a formal language, the event calculus. Event calculus is especially suitable for contract language representation because the semantics are unambiguous and, given a reliable log of events, the conclusions derived cannot be disputed.

Finally, our agent architecture provides a simple, powerful, extensible means to implement *passive* (by monitoring fulfilment and violation of obligations), *reactive* (by reacting to new obligations), *proactive* (by taking advantage of contracts to oblige other agents) and *opportunistic* (by accepting proposals that are in the agent's interest, but not necessarily obliged to accept) behaviours.

References

1. M. Alberti, D. Daolio, P. Torroni, M. Gavanelli, E. Lamma, and P. Mello. Specification and verification of agent interaction protocols in a logic-based system. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 72–78, New York, NY, USA, 2004. ACM Press.

2. A. Artikis, J. Pitt, and M. Sergot. Animated specifications of computational societies. In C. Castelfranchi and L. Johnson, editors, *Proceedings of Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1053–1062. ACM Press, 2002.
3. A. K. Bandara, E. C. Lupu, and A. Russo. Using event calculus to formalise policy specification and analysis. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, page 26. IEEE Computer Society, 2003.
4. R. H. Bordini and J. F. Hübner. Jason - A Java-based agentSpeak intepreter used with saci for multi-agent distribution over the net. `http://jason.sourceforge.net`, 2005.
5. J. Broersen, M. Dastani, J. Hulstijn, Z. Huang, and L. van der Torre. The BOID architecture: conflicts between beliefs, obligations, intentions and desires. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 9–16. ACM Press, 2001.
6. F. Dignum, J. Broersen, V. Dignum, and J.-J. Meyer. Meeting the deadline: Why, when and how. In *Proceedings of the 3rd Conference on Formal Aspects of Agent-Based Systems (FAABS III)*, 5 2004.
7. V. Dignum, J.-J. Meyer, F. Dignum, and H. Weigand. Formal specification of interaction in agent societies. In *Formal Approaches to Agent-Based Systems*, number 2699 in LNAI. Springer, 2002.
8. M. J. Kollingbaum and T. J. Norman. Supervised interaction: creating a web of trust for contracting agents in electronic environments. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 272–279. ACM Press, 2002.
9. M. J. Kollingbaum and T. J. Norman. Norm consistency in practical reasoning agents. In *Proceedings of PROMAS Workshop on Programming Multiagent Systems, AAMAS 2003*, 2003.
10. R. A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(4):319–340, 1986.
11. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. Velde and J. W. Perram, editors, *Agents Breaking Away (LNAI 1038)*, pages 42–55. Springer-Verlag, 1996.
12. M. Shanahan. The event calculus explained. In M.J.Wooldridge and M.Veloso, editors, *Artificial Intelligence Today*, volume 1600 of *LNAI*, pages 409–430. Springer Verlag, 1999.
13. Y. Shoham. Agent0: A simple agent language an its interpreter. In *Proc. of AAAI-91*, pages 704–709, Anaheim, CA, 1991.
14. E. Verharen and F. Dignum. Cooperative Information Agents and communication. In P. Kandzia and M. Klusch, editors, *Cooperative Information Agents, First International Workshop*, number 1202 in LNAI, pages 195–209, 1997.
15. E. Verharen, F. Dignum, and S. Bos. Implementation of a cooperative agent architecture based on the language-action perspective. In M. Singh, editor, *Intelligent Agents IV*, volume 1365 of *LNAI*, pages 31–44. 1998.
16. M. Winikoff. An agentspeak meta-interpreter and its applications. In *Proceedings of the Third international Workshop on Programming Multi-Agent Systems*, 2005.