

Flexible Protocol Specification and Execution: Applying Event Calculus Planning using Commitments

Pinar Yolum
pyolum@eos.ncsu.edu

Munindar P. Singh
singh@ncsu.edu

Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534, USA

ABSTRACT

Protocols represent the allowed interactions among communicating agents. Protocols are essential in applications such as electronic commerce where it is necessary to constrain the behaviors of autonomous agents. Traditional approaches, which model protocols in terms of action sequences, limit the flexibility of the agents in executing the protocols. By contrast, we develop an approach for specifying protocols in which we capture the content of the actions through agents' commitments to one another. We formalize commitments in a variant of the event calculus. We provide operations and reasoning rules to capture the evolution of commitments through the agents' actions. Using these rules in addition to the basic event calculus axioms enables agents to reason about their actions explicitly to flexibly accommodate the exceptions and opportunities that arise at run time. This reasoning is implemented using an event calculus planner that helps us determine flexible execution paths that respect the protocol specifications.

Categories and Subject Descriptors

I.2.11 [Artificial Intelligence]: Distributed Artificial Intelligence;
D.2.2 [Software Engineering]: Design Tools and Techniques

General Terms

Design, Verification

Keywords

Agent communication languages and protocols; methodologies and tools; commitments

1. INTRODUCTION AND MOTIVATION

Multiagent protocols regulate the interactions between agents. Designing protocols that ensure meaningful conversations among agents is necessary but not sufficient to support the dynamic interactions among agents. Current formalisms used in modeling net-

work protocols, such as finite state machines and Petri Nets, specify protocols merely in terms of legal sequences of actions without regard to the meanings of those actions. Thus, applying these formalisms into multiagent settings result in protocols that are over-constrained. However, protocols should not only constrain the actions of the participants, but also recognize the open, dynamic nature of interactions by accommodating the key aspects of autonomy, heterogeneity, opportunities, and exceptions.

- *Autonomy*: Promoting the participants' autonomy is crucial for creating effective systems in open environments. Participants must be constrained in their interactions only to the extent necessary to carry out the given protocol, and no more.
- *Heterogeneity*: Participants can be of diverse designs and may adopt different strategies to carry out their interactions. It would help participants to accommodate heterogeneity in others. Current specifications don't allow this. For example, many e-commerce protocols assume that all participants are untrustworthy, and each step ensures that appropriately safe actions are taken by the various participants. This unnecessarily degrades performance where trust has been, or can be, established.
- *Opportunities*: Participants should be able to take advantage of opportunities to improve their choices or to simplify their interactions. Depending on the situation, certain steps in a protocol can be skipped. A participant may take advantage of domain knowledge and jump to a state in a protocol without explicitly visiting one or more intervening states, since visiting each state may require additional messages and cause delays.
- *Exceptions*: Participants must be able to modify their interactions to handle unexpected conditions. The exceptions of interest here are not programming or networking exceptions such as loss of messages and network delays, but are higher-level exceptions that result from the unexpected behavior of the participants. For example, a deadline may be renegotiated at a discount. This would obviously involve domain knowledge, but the protocol representation should allow it.

Our approach to specifying protocols is based on capturing the intrinsic meaning of actions. We model these intrinsic meanings through *social commitments*. Conceptually, social commitments capture the obligations from one party to another [1, 16]. We define operations to create and manipulate commitments. We view each action in the protocol as an operation on commitments. In

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AAMAS'02, July 15-19, 2002, Bologna, Italy.

Copyright 2002 ACM 1-58113-480-0/02/0007 ...\$5.00.

other words, by following the protocol, each agent creates and manipulates commitments, e.g., by fulfilling, canceling, and so on. In addition to providing a protocol specification that defines the actions as operations on commitments, we provide reasoning rules to operationalize the commitments.

We formalize these reasoning rules and the operations on commitments in a variant of event calculus which allows capturing of meanings of actions easily. Denecker *et al.* previously showed that the event calculus could be used to represent traditional network protocols accurately and succinctly [3]. We extend this result to multiagent protocols by incorporating commitments into the formalism.

Capturing the intrinsic meaning of the actions and explicitly representing them as part of the protocol brings in flexibility to the protocols, permitting the agents to reason about their and others' behavior during the execution of the protocol, and enabling them to modify their actions as best suits them.

The protocol specifications developed by our approach can be used in two ways:

- *Execution:* The protocol specification can be used at run time if the agents have the necessary resources to process logical formulae. Essentially, the agents can logically compute their transitions using the operations and the reasoning rules. After each action, the agent is aware of its pending commitments and any additional commitments it has to make to get to a final state. In this respect, following the protocol becomes generating paths that enable the agent to get to a desired final state from its current state. We give a detailed description of path generation in Section 6.
- *Compilation:* If the agents cannot reason logically, then they have to follow a formalism that does not require them to compute the transitions. One such formalism is the traditional finite state machines (FSMs). Even though FSMs are easy to execute, they are not easy to design in the first place. Without capturing the meaning of transitions, redundant transitions can be added or necessary transitions may be omitted. On the other hand, specifying a protocol using our approach is easy and this specification can be compiled into an FSM ensures that all necessary transitions are captured.

This paper studies the framework for protocol specification and execution in detail. Compilation of a commitment-based protocol into an FSM is discussed elsewhere [18]. The rest of this paper is organized as follows: Section 2 introduces our running example with pointers to different scenarios that may arise. Section 3 describes the necessary background about event calculus. Section 4 describes the operations and reasoning rules on commitments. Section 5 explains the specification of protocols. Section 6 depicts how planning can be used to generate scenarios of a protocol, and Section 7 discusses our work with respect to the literature.

2. RUNNING EXAMPLE

As a running example, we study the NetBill protocol developed for buying and selling of goods on the Internet [13].

EXAMPLE 1. As shown in Figure 1, the protocol begins with a customer requesting a quote for some desired goods, followed by the merchant sending the quote. If the customer accepts the quote, then the merchant delivers the goods and waits for an electronic payment order (EPO). The goods delivered at this point are encrypted, that is, not usable. After receiving the EPO, the merchant forwards the EPO and the key to the intermediation server, which

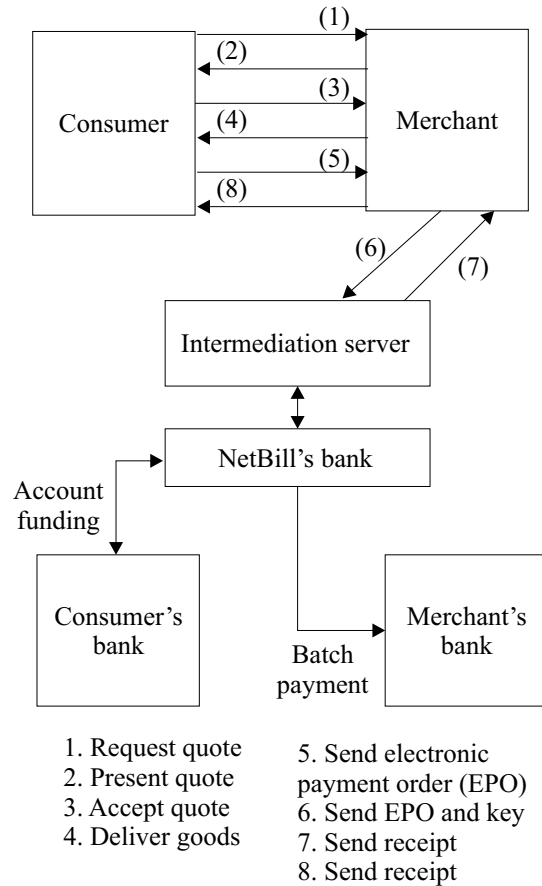


Figure 1: The NetBill payment protocol

then contacts the bank to take care of the funds transfer. When the funds transfer completes, the intermediation server sends a receipt back to the merchant. The receipt contains the decryption key for the sold goods. As the last step, the merchant forwards the receipt to the customer, who can then successfully decrypt and use the goods. For our present purposes, we leave out the banking procedures in the protocol, thus simplifying it to the point where if a merchant gets an EPO, he can take care of it successfully. ■

Traditional representations of protocols are inadequate in settings where autonomous agents must flexibly interact, e.g., to handle exceptions and exploit opportunities.

EXAMPLE 2. Consider the following scenarios that may arise in the NetBill protocol:

- Before the customer asks for a quote, the merchant wants to advertise his goods by sending a quote to the customer.
- The customer wants to send an 'accept' message without prior conversation on the price of the goods, due to emergency, insignificance of money, etc.
- The merchant wants to send the goods without a prior acceptance of the customer, similar to the trial versions in the software industry, where after a certain period of time the customer is required to pay to continue using the goods. This scenario is shown in Figure 2.

- After receiving the goods, the customer may send the EPO to the bank instead of the merchant. By delegating the payment to the bank, the customer makes the bank responsible for ensuring that the money gets to the merchant. ■

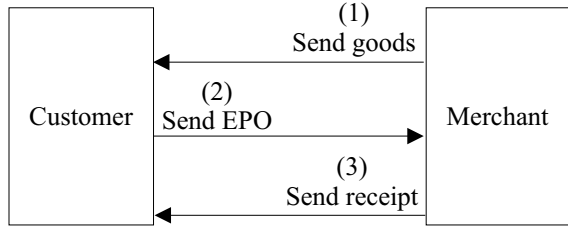


Figure 2: An alternative scenario

The scenarios depicted in Example 2 can not be handled by a protocol representation that specifies the legal sequences of actions but does not define the content of the actions or of the intervening states.

3. EVENT CALCULUS

The event calculus (EC), introduced by Kowalski and Sergot [7], is a formalism to reason about events. *Events* in EC initiate and terminate *fluents*, which are properties that are allowed to have different values at different time points. Their value is manipulated by the occurrence of events. A fluent starts to hold after an event that can initiate it occurs. Similarly, it ceases to hold when an event that can terminate it occurs.

The event calculus used in this paper is a subset of Shanahan's version [10]. It is based on many-sorted first-order predicate calculus, with the addition of eight predicates to reason about the events. We now introduce these predicates and the axioms with which to reason about the predicates.

In the following, a, b, \dots refer to events, f, g, \dots refer to fluents; and t, t_1, t_2, \dots refer to time points. The variables that are not explicitly quantified are assumed to be universally quantified. \leftarrow denotes implication and \wedge denotes conjunction. The time points are ordered by the $<$ relation, which is defined to be transitive and asymmetric.

1. $Initiates(a, f, t)$ means that fluent f holds after event a at time t .
2. $Terminates(a, f, t)$ means that fluent f does not hold after event a at time t .
3. $Initially_P(f)$ means that fluent f holds from time 0.
4. $Initially_N(f)$ means that fluent f does not hold from time 0.
5. $Happens(a, t_1, t_2)$ means that event a starts at time t_1 and ends at t_2 .
6. $HoldsAt(f, t)$ means that the fluent f holds at time t .
7. $Clipped(t_1, f, t_2)$ means that the fluent f is terminated between t_1 and t_2 .
8. $Declipped(t_1, f, t_2)$ means that the fluent f is initiated between t_1 and t_2 .

Based on the language of EC, the following axioms are defined [10]:

AXIOM 1. $HoldsAt(f, t) \leftarrow Initially_P(f) \wedge \neg Clipped(0, f, t)$ ■

All fluents that hold initially and are not terminated by any event from time 0 to time t continue to hold at time t .

AXIOM 2. $HoldsAt(f, t_3) \leftarrow Happens(a, t_1, t_2) \wedge Initiates(a, f, t_1) \wedge (t_2 < t_3) \wedge \neg Clipped(t_1, f, t_3)$ ■

Domain Description:

$Initiates(a, f, t_1)$

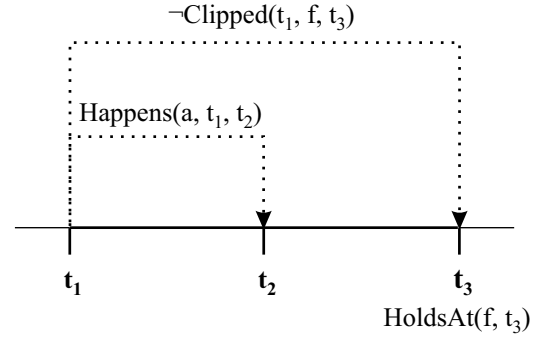


Figure 3: Axiom 2

As shown in Figure 3, after an event initiates a fluent, the fluent continues to hold if no other event that can terminate it occurs at a later time. These two axioms capture the fact that after a fluent begins to hold, an event that can terminate the fluent should occur in order to put an end to the holding of that fluent.

AXIOM 3. $Clipped(t_1, f, t_4) \leftrightarrow \exists a, t_2, t_3 [Happens(a, t_2, t_3) \wedge (t_1 < t_2) \wedge (t_3 < t_4) \wedge Terminates(a, f, t_2)]$ ■

Axiom 3 states that a fluent is said to be *clipped* if and only if an event occurs to terminate it.

Axioms 4 and 5 represent the duals of Axioms 1 and 2, respectively.

AXIOM 4. $\neg HoldsAt(f, t) \leftarrow Initially_N(f) \wedge \neg Declipped(0, f, t)$ ■

Axiom 4 states that a fluent does not continue to hold, if initially it did not hold, and there does not occur any event that initiates it.

Domain Description:

$Terminates(a, f, t_1)$

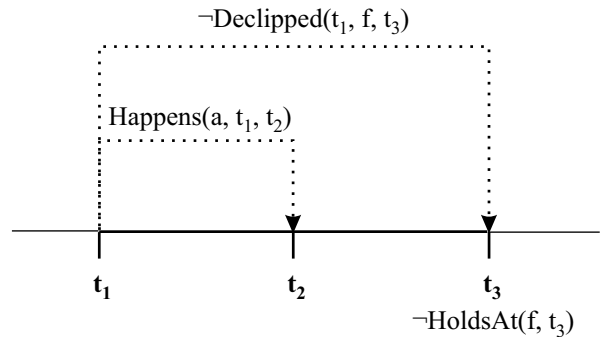


Figure 4: Axiom 5

AXIOM 5. $\neg \text{HoldsAt}(f, t_3) \leftarrow \text{Happens}(a, t_1, t_2) \wedge \text{Terminates}(a, f, t_1) \wedge (t_2 < t_3) \wedge \neg \text{Declipped}(t_1, f, t_3)$ ■

Following the same intuition, Axiom 5 states that if an event occurs and terminates a fluent, and no other event occurs to initiate it, then the fluent continues not to hold, as shown in Figure 4.

AXIOM 6. $\text{Declipped}(t_1, f, t_4) \leftrightarrow \exists a, t_2, t_3 [\text{Happens}(a, t_2, t_3) \wedge (t_1 < t_2) \wedge (t_3 < t_4) \wedge \text{Initiates}(a, f, t_2)]$ ■

Domain Description:
Initiates(a, f, t₂)

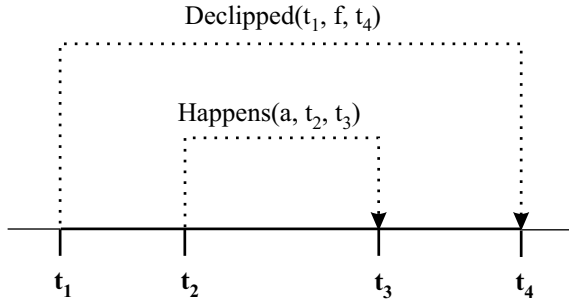


Figure 5: Axiom 6

A fluent is said to be *declipped* in a time period if and only if there exists an event that occurs and either initiates or releases the fluent in that time period. This axiom is illustrated in Figure 5.

AXIOM 7. $\text{Happens}(a, t_1, t_2) \rightarrow t_1 \leq t_2$ ■
Axiom 7 ensures that no event takes a negative amount of time.

DEFINITION 1. We introduce a two argument *Happens* fluent to reason about events that start and end at the same time point. For simplicity, we will use this version of the *Happens* fluent hereafter.
 $\text{Happens}(a, t) \equiv_{def} \text{Happens}(a, t, t)$

4. COMMITMENTS FORMALIZED

Social commitments are commitments made from one agent to another agent to bring about a certain property. Commitments result from communicative actions. That is, agents create commitments and manipulate them through the protocol they follow. We represent commitments as properties in the event calculus, and develop a scheme where we model the creation and manipulation of commitments as a result of performing actions. Further, by allowing preconditions to be associated with the initiation and termination of properties, different commitments can be associated with communicative acts to model the communications among agents more concretely.

DEFINITION 2. A *base-level commitment* $C(x, y, p)$ relates a debtor x , a creditor y , and a condition p [11].

When a commitment of this form is created, x becomes responsible to y for satisfying p . The condition p does not involve other fluents or commitments.

Conditional commitments are useful when a party wants to commit only if a certain condition holds or only if the other party is also willing to make a commitment.

DEFINITION 3. A *conditional commitment* $CC(x, y, p, q)$ denotes that if the condition p is brought out, x will be committed to bring about condition q .

EXAMPLE 3. The base-level commitment $C(\text{merchant}, \text{customer}, \text{deliveredGoods})$ shows that the merchant is committing to the customer that goods will be delivered, whereas the conditional commitment, $CC(\text{merchant}, \text{customer}, \text{paidMoney}, \text{deliveredGoods})$ specifies that the merchant will commit to send the goods only if the customer commits to paying the money. ■

Below, we present a formal account of the operations that can be performed to create and manipulate commitments [15, 11] and show how these operations can be formalized in the event calculus. In the following discussion, x, y, z denote agents, c, c' denote commitments, and $e(x)$ denotes an event that is performed by x .

1. *Create*($e(x), c$) establishes the commitment c . The *create* operation can only be performed by the debtor of the commitment x . When x performs the event e , the commitment c is initiated.

Create($e(x), C(x, y, p)$):
{ $\text{Happens}(e(x), t) \wedge \text{Initiates}(e(x), C(x, y, p), t)$ }

2. *Discharge*($e(x), c$) resolves the commitment c . Again, the *discharge* operation can only be performed by the debtor of the commitment to mean that the commitment has successfully been carried out. When x performs the event e , the commitment c is terminated.

Discharge($e(x), C(x, y, p)$):
{ $\text{Happens}(e(x), t) \wedge \text{Initiates}(e(x), p, t)$ }

3. *Cancel*($e(x), c$) cancels the commitment c . Usually, the cancellation of a commitment is followed by the creation of another commitment to compensate for the former one. When x performs the event e , the commitment c is terminated.

Cancel($e(x), C(x, y, p)$):
{ $\text{Happens}(e(x), t) \wedge \text{Terminates}(e(x), C(x, y, p), t) \wedge \text{Initiates}(e(x), C(x, y, p'), t)$ }

4. *Release*($e(y), c$) releases the debtor from the commitment c . It can be performed by the creditor to mean that the debtor is no longer obliged to carry out his commitment.

Release($e(y), C(x, y, p)$): { $\text{Happens}(e(y), t) \wedge \text{Terminates}(e(y), C(x, y, p), t)$ }

5. *Assign*($e(y), z, c$) assigns a new agent as the creditor of the commitment. More specifically, the commitment c is eliminated, and a new commitment c' is created for which z is appointed as the new creditor.

Assign($e(y), z, C(x, y, p)$): { $\text{Happens}(e(y), t) \wedge \text{Terminates}(e(y), C(x, y, p), t) \wedge \text{Initiates}(e(y), C(x, z, p), t)$ }

6. *Delegate*($e(x), z, c$) replaces the debtor of the commitment with another agent z so that z becomes responsible to carry out the commitment. Similar to the previous operation, the commitment c is eliminated, and a new commitment c' is created in which z is the debtor.

Delegate($e(x), z, C(x, y, p)$): { $\text{Happens}(e(x), t) \wedge \text{Terminates}(e(x), C(x, y, p), t) \wedge \text{Initiates}(e(x), C(z, y, p), t)$ }

The creation and manipulation of commitments are handled with the above operations. In addition to these operations, we formalize reasoning rules on commitments that capture the operational semantics of our approach. Some of these operations require additional domain knowledge to reason about. For example canceling a

commitment might be constrained differently based on the domain. The reasoning rules we provide here only pertain to the create and discharge operations and the conditional commitments. To ensure completeness, we assume unique name axioms for events and fluents.

Postulate 1 states that a commitment is no longer in force if the condition committed to holds. For the condition p to hold, an event must occur to initiate it. In Postulate 1, when the event e occurs at time t , it initiates the property p , and therefore the commitment $C(x, y, p)$ can be terminated.

POSTULATE 1. $Terminates(e(x), C(x, y, p), t) \leftarrow HoldsAt(C(x, y, p), t) \wedge Happens(e(x), t) \wedge Initiates(e(x), p, t)$ ■

The following two postulates capture how a conditional commitment is resolved based on the temporal ordering of the commitments it refers to.

POSTULATE 2. $Initiates(e(y), C(x, y, q), t) \wedge Terminates(e(y), CC(x, y, p, q), t) \leftarrow HoldsAt(CC(x, y, p, q), t) \wedge Happens(e(y), t) \wedge Initiates(e(y), p, t)$ ■

When the conditional commitment $CC(x, y, p, q)$ holds, if p becomes true, then the original commitment ceases to exist but a new base-level commitment is created, since the debtor x is now committed to bring about q . More specifically, In Postulate 2, when the event e occurs, it initiates p , which results in the termination of the original commitment, and the initiation of $C(x, y, q)$. This is similar to Colombetti's treatment of conditional commitments [2], but here we capture how conditional commitments are resolved into base-level commitments rather than how they can be violated.

POSTULATE 3. $Terminates(e(x), CC(x, y, p, q), t) \leftarrow HoldsAt(CC(x, y, p, q), t) \wedge Happens(e(x), t) \wedge Initiates(e(x), q, t)$ ■

Again, when the conditional commitment $CC(x, y, p, q)$ holds, if an event e that can initiate q occurs, q starts to hold and the original commitment is terminated. Since the creditor y has not committed to anything, no additional commitments are created.

Next, we present our approach for specifying protocols.

5. SPECIFYING PROTOCOLS

To represent a protocol, we need to represent the flow of execution within the protocol. In EC, two predicates are used to specify how the execution can evolve: *Initiates* and *Terminates*. In addition to defining which fluents they *initiate* or *terminate*, the required preconditions for activating these predicates can be specified. Therefore, the possible transitions in a protocol can be specified in terms of a set of *Initiates* and *Terminates* clauses.

DEFINITION 4. *A protocol specification is a set of Initiates and Terminates clauses that define which properties pertaining to the protocol are initiated and terminated by each action.*

Next we define how a protocol run is structured, which we represent by a set of actions that take place at specific timepoints.

DEFINITION 5. *A protocol run is a set of Happens clauses along with an ordering of the timepoints referred to in the predicates.*

Notice that a protocol specified as in Definition 4 does not indicate any starting states, final states or transitions among executions states. An agent can start a protocol by performing any of the actions whose preconditions match the current state of the execution. By appropriately increasing or decreasing the preconditions

of the actions, a protocol can be abbreviated or enhanced to allow a broader range of interactions. Although we do not represent the final states of a protocol explicitly, we can examine a protocol run to determine if any agent has backed out of its commitment.

DEFINITION 6. *A protocol run is complete if there are no pending base-level commitments; i.e., all the base-level commitments that have been created are resolved. Formally,*

$$\neg HoldsAt(C(x, y, p), t)$$

If a protocol run is not complete, that is, if there is an open base-level commitment after the execution of the protocol, we know that a participant has not fulfilled its commitment. This signals a violation of the protocol. Although we don't investigate the issue of protocol compliance by agents in this work, incomplete protocol runs provide evidence to identify non-compliant agents.

To continue our treatment of the NetBill protocol, we first define the fluents used in that protocol and then provide the protocol specification.

EXAMPLE 4. *The messages of Figure 1 can be given a content based on the following definitions. Since each action can be performed by only one party, we do not represent the performers explicitly.*

- **Roles:**

- *MR represents the merchant.*
- *CT represents the customer.*

- **Domain-specific fluents:**

- *request(i): a fluent meaning that the customer has requested a quote for item i.*
- *goods(i): a fluent meaning that the merchant has delivered the goods i.*
- *pay(m): a fluent meaning that the customer has paid the agreed upon amount m.*
- *receipt(i): a fluent meaning that the merchant has delivered the receipt for item i.*

- **Commitments:**

- *accept(i, m): an abbreviation for $CC(CT, MR, goods(i), pay(m))$ meaning that the customer is willing to pay if he receives the goods.*
- *promiseGoods(i, m): an abbreviation for $CC(MR, CT, accept(i, m), goods(i))$ meaning that the merchant is willing to send the goods if the customer promises to pay the agreed amount.*
- *promiseReceipt(i, m): an abbreviation for $CC(MR, CT, pay(m), receipt(i))$ meaning that the merchant is willing to send the receipt if the customer pays the agreed-upon amount.*
- *offer(i, m): an abbreviation for $(promiseGoods(i, m) \wedge promiseReceipt(i, m))$*

- **Definition of Initiates in the NetBill Protocol:**

- *Initiates(sendRequest(i), request(i), t)*
- *Initiates(sendQuote(i, m), promiseGoods(i, m), t)*
- *Initiates(sendQuote(i, m), promiseReceipt(i, m), t)*
- *Initiates(sendAccept(i, m), accept(i, m), t)*

- $\text{Initiates}(\text{sendGoods}(i, m), \text{goods}(i), t)$
- $\text{Initiates}(\text{sendGoods}(i, m), \text{promiseReceipt}(i, m), t)$
- $\text{Initiates}(\text{sendEPO}(i, m), \text{pay}(m), t) \leftarrow$
 $\text{HoldsAt}(\text{goods}(i), t)$
- $\text{Initiates}(\text{sendReceipt}(i, m), \text{receipt}(i), t) \leftarrow$
 $\text{HoldsAt}(\text{pay}(m), t)$

• **Definition of Terminates in the NetBill Protocol:**

- $\text{Terminates}(\text{sendQuote}(i, m), \text{request}(i), t) \leftarrow$
 $\text{HoldsAt}(\text{request}(i), t)$ ■

The following example describes an example protocol run, and depicts how the commitments are created and then resolved.

EXAMPLE 5. *The protocol run shown in Figure 2 can be formalized by the following facts:*

- F1. $\text{Happens}(\text{sendGoods}(i, m), t_1)$
- F2. $\text{Happens}(\text{sendEPO}(m), t_2)$
- F3. $\text{Happens}(\text{sendReceipt}(i), t_3)$
- F4. $t_1 < t_2$
- F5. $t_2 < t_3$

Now we look at how the commitments among the participants evolve in the given protocol run. We also assume that initially no commitments or predicates hold.

1. $\text{HoldsAt}(\text{CC}(MR, CT, \text{pay}(m), \text{receipt}(i)), t_1) \wedge$
 $\text{HoldsAt}(\text{goods}(i), t_1)$

When the goods are sent at time t_1 , the fluent $\text{goods}(i)$ is initiated. Furthermore, following the protocol specification in Example 4, the commitment $\text{CC}(MR, CT, \text{pay}(m), \text{receipt}(i))$ is created. So now the goods have been delivered, and the merchant is willing to send the receipt if the customer pays.

2. $\text{HoldsAt}(\text{C}(MR, CT, \text{receipt}(i)), t_2) \wedge \text{HoldsAt}(\text{goods}(i), t_2) \wedge \text{HoldsAt}(\text{pay}(m), t_2)$

By sending the EPO at time t_2 , the customer initiates the fluent $\text{pay}(m)$. By Postulate 2, this ends the commitment $\text{CC}(MR, CT, \text{pay}(m), \text{receipt}(i))$ and creates the commitment $\text{C}(MR, CT, \text{receipt}(i))$. Since no event occurred to terminate $\text{goods}(i)$, it continues to hold.

3. $\text{HoldsAt}(\text{goods}(i), t_3) \wedge \text{HoldsAt}(\text{pay}(m), t_3) \wedge$
 $\text{HoldsAt}(\text{receipt}(i), t_3)$

At time t_3 the third fact is applicable, which initiates the fluent $\text{receipt}(i)$. Following Postulate 1, this terminates the commitment $\text{C}(MR, CT, \text{receipt}(i))$. Thus, we reach the state where the merchant has delivered the goods and the receipt, and the customer has paid. ■

6. EXECUTING PROTOCOLS

Planning is the construction of plans for automatic or semiautomatic execution by an agent. We consider plans that would lead from an initial state to a final state by applying the operators to transition among states. The form of logical reasoning that is commonly used in building event calculus planners is abduction. One such abductive event calculus planner is due to Shanahan [10]; we use this event calculus planner here to demonstrate how possible

paths can be generated between an initial state and a goal state given a protocol specification defined based on the preconditions and the effects of actions as in Definition 4.

We now walk through the steps to put together the protocol specification of the NetBill protocol and an initial state in the protocol to produce protocol runs that lead from the initial state to a sample final state. The reasoning rules explained before are manually compiled in the code. Due to lack of space, we only provide fragments of the program written in Prolog. The fluents used in them are the same as the ones used in Example 4, except here we add a transaction identifier to each fluent as the last argument. This identifier is used to ensure that the participants, by bringing about properties, resolve commitments with the corresponding transaction ids.

In the event calculus, the initial states are represented by conjunctive expressions consisting of Initially_P and Initially_N clauses to represent which fluents hold or do not hold at the beginning. In the planner, though, only one predicate, Initially , has been implemented. In order to indicate negative fluents, a new predicate neg has been introduced. Figure 6 gives an example of a clause to set up the initial state of the NetBill protocol.

```
axiom(initially(neg(goods(I, D))), []).
```

Figure 6: Example clause of an initial state

Operators are the actions in the domain. These actions are defined in the program through the executable clause. Figure 7 gives such an example. Recall that a protocol specification consists of

```
executable(sendrequest(I, D)).
```

Figure 7: Example clause of executable events

Initiates and Terminates clauses, specifying the preconditions and effects of the actions. Figure 8 gives an example axiom for the first part of the protocol description that is, the Initiates clauses. The first argument to the axioms is the Initiates clause, and the second argument is the set of preconditions needed for the Initiates clause to be applicable. The example axiom in Figure 8 corresponds to the second step of the Example 5. That is, if the merchant has not paid but promised the receipt, the action sendEPO will initiate the commitment, $c(\text{receipt}(I, D))$. Figure 9 gives an example of the

```
axiom(initiates(sendepo(M, D),
c(receipt(I, D)), T),
[holds_at(promisereceipt(I, M, D), T),
holds_at(neg(pay(M, D)), T)]).
```

Figure 8: Example of an Initiates clause

second part the protocol description, that is, the Terminates clauses. Again, the axioms have the same format as in Figure 8, where the first argument is the Terminates clause and the second argument is the set of preconditions. In other words, the axiom in Figure 9 states that if a party is committed to pay, sending an epo terminates its commitment of paying. Having the initial states, the goal states and the domain description, an event calculus planner can generate protocol runs that contain Happens clauses, and an ordering of time points of the actions that take place [10].

In order to interpret the code, we provide the description of a final state; Figure 10 shows such an example. The final state depicted in Figure 10 means that goods (software) have been sent with

```
axiom(terminates(sendepo(M, D),
  c(pay(M, D)),T),
  [holds_at(c(pay(M, D)), T)]).
```

Figure 9: Example of a *Terminates* clause

transaction identifier 51 and the customer is committed to send the money for this transaction. This yields the result shown in Figure 11, where each R is a possible protocol run starting from the initial state where no commitments or fluents hold, and ending in the final state depicted in Figure 10.

```
abdemo([holds_at(goods(software, 51),t),
  holds_at(c(pay(price, 51)),t)],R).
```

Figure 10: Description of a final state

As described before, each protocol run consists of *Happens* clauses and an ordering of time points. The last argument in each *Happens* clause denotes a timepoint at which the event happens. The ordering of these time points are then shown with the *before* clauses. The two protocol runs in Figure 11 correspond to the following scenarios: The first one starts with the merchant sending a quote for the software, followed by the customer sending an accept message. Recall that at this point both agents become committed to each other: customer to send the money, and merchant to send the software and receipt. As the last action, the merchant sends the goods. Therefore, at the end of the protocol run, the merchant has sent the goods and the customer is committed to send the money. The second protocol run starts with the customer sending an accept message, followed by the merchant sending the goods. After this message, by Postulate 2 the customer becomes committed to pay.

```
R=[ [happens
  (sendquote(software,H816,51),t191),
  happens
  (sendaccept(software,H816,51),t190),
  happens
  (sendgoods(software,H601,51),t189)],
  [before(t191,t), before(t191,t189),
  before(t191,t190), before(t190,t),
  before(t190,t189), before(t189,t)]] ;
R=[ [happens
  (sendaccept(software,H601,51),t193),
  happens
  (sendgoods(software,H601,51),t192)],
  [before(t193,t), before(t193,t192),
  before(t192,t)]] ;
```

Figure 11: Protocol runs computed by the planner

These protocol runs can be used by an agent at run time to decide if an action is appropriate at a particular state of the execution. This enables the agents to cope with the exceptions by reconstructing plans as necessary. Thus, agents can execute protocols flexibly by taking advantage of opportunities, and handling exceptions.

7. DISCUSSION

Traditionally, protocols have been specified using formalisms like finite state machines, or Petri Nets, that only capture the legal sequences of actions. The main advantage of these formalisms is that they are easy to implement and can be trivially followed by reactive agents. However, since the semantic content of the actions is not captured, the agents cannot reason about their actions, which

means that they cannot take advantage of opportunities that arise or handle unexpected situations at run time. To remedy this situation, we develop an approach for protocol specification that embodies the commitments of agents to one another. Specification of protocols in terms of commitments allows agents to reason about their actions, enabling them to take care of the unexpected situations that may arise at run time. Agents that follow these protocols can decide on the actions they want to take by generating protocol runs with a planner as we have demonstrated. In designing protocols, we can exploit the strengths of the event calculus to reason about actions and commitments. The event calculus provides an elegant way to represent the changes of the world through the actions in a protocol, and enables us to uniformly represent commitments, operations on them, and reasoning rules about them. Based on this formal grounding, multiagent protocols can be specified rigorously yet flexibly.

Event calculus has been theoretically studied, but has not been used for modeling commitments or commitment-based protocols. Denecker *et al.* [3] use event calculus for specifying process protocols. Their specification also captures the preconditions of actions as well as the execution of the protocol through actions. Since they are specifying process protocols only, they use domain propositions to denote the meanings of actions. In multiagent systems, in addition, protocols should respect agents' autonomy and enable them to interact flexibly to exploit opportunities and to handle exceptions. In order to achieve this, we use commitments to denote the meanings of the actions.

Commitments have been studied before [1, 5] but have not been used for protocol specification as we have done here. Permissions and prohibitions are also useful in protocol specifications. Permissions in our approach can be accommodated through preconditions of the actions such that an action would only initiate properties if the party performing the action had the necessary permissions. Prohibitions, on the other hand, forbid a party from bringing out a proposition. Prohibitions can be formulated through commitments. For example, if a client prohibits a merchant to send advertisements, the merchant becomes committed to not sending the advertisements. In many settings, we would expect prohibitions to be constrained, so that only authoritative roles can issue prohibitions.

Multiagent interactions should satisfy three desirable criteria: meaningful, verifiable, and declarative [12]. First we describe these criteria and show how our approach satisfies them. Next we review the recent literature, considering how these systems satisfy the same criteria.

- *Meaningful.* The messages should be represented with their content instead of being treated as mere tokens. Our approach is meaningful since we capture the meanings of the actions via commitments.
- *Verifiable.* The protocol semantics should allow detection of agents that are not complying with a given protocol without assuming that we can examine the internal reasoning (or the source code) of the agents. In our approach, we achieve verifiability through the commitments we capture. By keeping track of the commitments that are created and resolved in the system, we can infer which agents have not acted according to their commitments. These agents have not complied with the protocol.
- *Declarative.* A declarative, as opposed to a procedural, semantics should specify what actions should be brought out in the protocol, rather than how they are brought out. Our approach is declarative in that it specifies what properties each

action individually brings about rather than specifying procedurally how agents can get from a start state to a final state.

Lespérance *et al.* [8] develop a tool, ConGolog, for developing reactive control mechanisms that are capable of handling exceptions. Their system's execution is based on a declarative domain description, which is similar to our definition of a protocol specification in that it specifies the preconditions and effects of the actions in the domain. We share their intuition in reconstructing plans at run-time to handle exceptions. The main difference between Lespérance *et al.*'s approach and ours is that they define the meanings of actions in terms of only domain propositions, whereas we define them in terms of domain properties and commitments. In this respect, their representation is not verifiable. Again considering our example protocol, assume a protocol run where a customer agent performs a sendAccept action, after receiving a sendQuote action and no other actions take place. In our approach, we can easily detect that the merchant has to actually send the goods to the customer, and the customer has to send the EPO. In Lespérance *et al.*'s approach, all we know is that the merchant sent a quote to the customer and the customer accepted the quoted price. In order to decide if the protocol is in a good state requires examining the individual agent programs.

Smith *et al.* [14] develop protocols in which actions are given a content based on joint intentions. We agree with them on the necessity of declarative content. They model the content of actions with mental attributes whereas we use social constructs. In this respect, their approach is not verifiable.

Pitt and Mamdani [9] develop an agent communication language (ACL) framework in terms of protocols, and show how an agent replies to a communication by choosing one of the communications allowed by the given communication. They give content to messages based on social constructs, similar to the present approach.

d'Inverno *et al.* [4] develop interaction protocols for the multi-agent framework, Agentis. They model protocols as a composition of various services and tasks requested and offered among agents. d'Inverno *et al.*'s protocol model consist of four levels: registration, service, task and notification. In all levels of Agentis, the protocols are specified with FSMs. The FSM representations allow Agentis to specify the protocols formally yet easily. The protocols defined in Agentis are verifiable, since all transitions in Agentis FSMs are public (i.e., externally visible). However, their specification is not declarative since FSMs procedurally specify the sequences of actions that reach a goal state when executed in the described sequence. Thus FSMs specify how a certain goal state can be reached rather than specifying what each action brings out. In addition to not being declarative, the Agentis protocol semantics is not meaningful, since each message is treated as an arbitrary token without considering the meaning attached to it. As we have shown above, representations that are not meaningful are not adequate to accommodate flexible interactions among autonomous agents, since agents cannot exercise their autonomy by choosing among actions without knowing what each action means.

8. ACKNOWLEDGMENTS

A preliminary version of this paper appears as a poster in the Autonomous Agents 2001 Conference [17]. We thank Peter Wurman, Matt Stallman, Michael Winikoff, Mehdi Dastani, and the anonymous reviewers for useful comments.

This material is based upon work supported by the National Science Foundation under grant IIS-9624425 (Career Award). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the National Science Foundation.

9. REFERENCES

- [1] C. Castelfranchi. Commitments: From individual intentions to groups and organizations. In *Proceedings of the International Conference on Multiagent Systems*, pages 41–48, 1995.
- [2] M. Colombetti. A commitment-based approach to agent speech acts and conversations. In *Proceedings of the Workshop on Agent Languages and Conversation Policies*, 2000.
- [3] M. Denecker, K. V. Belleghem, G. Duchatelet, F. Piessens, and D. D. Schreye. A realistic experiment in knowledge representation in open event calculus : Protocol specification. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 170–184, 1996.
- [4] M. d'Inverno, D. Kinny, and M. Luck. Interaction protocols in Agentis. In *Proceedings of the 3rd Int. Conference on Multiagent Systems (ICMAS)*, pages 112–119. July 1998.
- [5] L. Gasser. Social conceptions of knowledge and action: DAI foundations and open systems semantics. In [6], pages 389–404. 1998. (Reprinted from *Artificial Intelligence*, 1991).
- [6] M. N. Huhns and M. P. Singh, editors. *Readings in Agents*. Morgan Kaufmann, San Francisco, 1998.
- [7] R. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.
- [8] Y. Lespérance, K. Tam, and M. Jenkin. Reactivity in a logic-based robot programming framework. In *Intelligent Agents VI: Agent Theories, Architectures, and Languages*, pages 173–187, 2000.
- [9] J. Pitt and A. Mamdani. A protocol-based semantics for an agent communication language. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 486–491, 1999.
- [10] M. Shanahan. An abductive event calculus planner. *Journal of Logic Programming*, 44:207–239, 2000.
- [11] M. P. Singh. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7:97–113, 1999.
- [12] M. P. Singh. A social semantics for agent communication languages. In *Proceedings of the 1999 IJCAI Workshop on Agent Communication Languages*. Springer-Verlag, 2000.
- [13] M. A. Sirbu. Credits and debits on the Internet. In [6], pages 299–305. 1998. (Reprinted from *IEEE Spectrum*, 1997).
- [14] I. A. Smith, P. R. Cohen, J. M. Bradshaw, M. Greaves, and H. Holmback. Designing conversation policies using joint intention theory. In *Proceedings of the 3rd Int. Conference on Multiagent Systems (ICMAS)*, pages 269–276. July 1998.
- [15] M. Venkatraman and M. P. Singh. Verifying compliance with commitment protocols: Enabling open web-based multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 2(3):217–236, Sept. 1999.
- [16] D. N. Walton and E. C. W. Krabbe. *Commitment in Dialogue: Basic Concepts of Interpersonal Reasoning*. State University of New York Press, Albany, 1995.
- [17] P. Yolum and M. P. Singh. Designing and executing protocols using the event calculus. In J. P. Müller, E. Andre, S. Sen, and C. Frasson, editors, *The Proceedings of the Fifth International Conference on Autonomous Agents*, 2001.
- [18] P. Yolum and M. P. Singh. Commitment machines. In *Proceedings of the 8th International Workshop on Agent Theories, Architectures, and Languages (ATAL-01)*. Springer-Verlag, 2002. In press.