Contextualizing Commitment Protocols*

Amit K. Chopra*
North Carolina State University
Department of Computer Science
akchopra@ncsu.edu

Munindar P. Singh
North Carolina State University
Department of Computer Science
singh@ncsu.edu

ABSTRACT

Commitment protocols are modularized specifications of interactions understood in terms of commitments. Purchase is a classic example of a protocol. Although a typical protocol would capture the essence of the interactions desired, in practice, it should be adapted depending on the circumstances or *context* and the agents' preferences based on that context. For example, when applying purchase in different contexts, it may help to allow sending reminders for payments or returning goods to obtain a refund. We *contextualize* a protocol by adapting it via different transformations.

Our contributions are the following: (1) a protocol is transformed by composing its specification with a transformer specification; (2) contextualization is characterized operationally by relating the original and transformed protocols; and (3) contextualization is related to protocol compliance.

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Multiagent Systems

1. INTRODUCTION

This paper studies commitment protocols (for short, "protocols"), which are modularized interactions characterized in terms of the commitments among the interacting agents. Protocols are key in designing multiagent systems, such as for implementing business processes. Protocols would be composed and instantiated to realize such systems. Protocols geared for business applications are increasingly being standardized and published, although existing approaches do not offer a formal semantics.

Although protocols are modular and reusable, one size does not fit all. Consider a business example such as a purchase protocol. A basic purchase protocol would capture the essence of how agents should interact to effect a purchase. However, it might not be able to accommodate the many variations needed to handle all the different contexts in which it might be applied. The correct interactions might depend upon the *context* where it is applied: the social or trust relationships between the agents (would the seller insist on advance payment?), the goods being sold (do they need a prescription?), the geographic location (is it legal to sell Schedule C pharmaceuticals without a prescription?), legal considerations (should the seller verify the buyer's age?), and so on. Further variations of context might be inherited from the rules of an electronic marketplace (the Uniform Commercial Code in the US [8]) or a business culture, e.g., can goods be returned for a refund with no questions asked or can goods be returned if they are faulty, and so on. A merchant that knows how to participate in a basic purchase protocol would need to interact differently in each of the above cases.

Developing protocols from scratch for each of the possible contexts would be a difficult undertaking, tedious and error prone at best. This paper proposes an approach wherein protocols are generated by taking an existing protocol and transforming it. To this end, protocols and *transformers* are specified formally and declaratively. Ideally, each transformer is designed for a particular aspect of context. For example, one transformer might accommodate reminders; another might accommodate returns and refunds. In this way, the task of engineering protocols is greatly simplified.

The main contributions of this paper include the following: (1) specifying protocols and transformers formally and independently of each other, (2) formally characterizing transformations based on how they modify a protocol, and (3) determining an agent's compliance with a protocol that takes transformations into account.

This paper advances our program of research that treats interaction as a key abstraction for multiagent systems and treats protocols—as abstractions of interaction—as first-class citizens in contrast to object classes. Modeling protocols enables sophisticated management of interactions and of associated representations such as commitments. Architecturally, an emphasis on interactions is essential for open systems, because openness arises from making the end points dynamically changeable. A system being designed is guaranteed to be open when the interactions among its participants are reified, and the participants are described not as specific entities but in terms of the roles they play in those interactions.

Organization. The rest of the document is organized as follows. Section 2 presents a purchase protocol which serves as our running example. Section 3 develops a language for specifying protocols and a computational model for them. Section 4 presents contexts and transformers relevant to our running example. Section 5 semantically characterizes transformations and defines compliance. Section 6 discusses some key literature and directions.

2. RUNNING EXAMPLE

This paper uses purchase as a running example. This section

*This research was partially supported by the National Science Foundation under grant DST-0139037 and by DARPA contract F30603-00-C-0178.

*Student author.

AAMAS'06, May 8–12 2006, Hakodate, Hokkaido, Japan.

describes its "base" variant next. Figure 1 shows the base protocol in an informal graphical notation formalized later.

Commitments provide meaning to agent interaction. During the execution of a protocol, agents enter into, modify, and satisfy commitments. A commitment *C(x, y, p)* binds a debtor *x* to a creditor *y* for fulfilling the condition *p*. A conditional commitment *CC(x, y, p, q)* denotes that if condition *p* is brought about, then the commitment *C(x, y, q)* will hold. Following Singh [6], we use the following operations for the creation and manipulation of commitments: *create*, *discharge*, *cancel*, *assign*, *delegate*, and *release*. Commitment operations do not happen arbitrarily. They are typically governed by normative policies that characterize the set of circumstances under which the operations can happen.

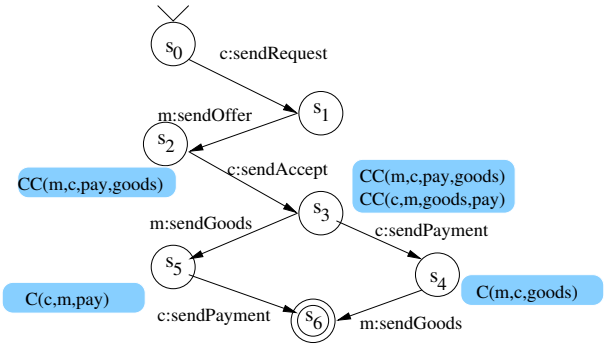

Figure 1: The base purchase protocol

The base purchase protocol has two roles: merchant and customer abbreviated m and c, respectively. For simplicity, we assume reliable, synchronous communication. Thus, sending a message is concurrent with its receipt. The protocol has the following steps. (1) The customer begins the protocol by sending a request for quotes to the merchant. (2) The merchant sends an offer in response. By sending an offer, the merchant creates the conditional commitment CC(m,c,pay,goods) meaning that if the customer pays, then the merchant will send the goods. (3) The customer accepts the offer, thereby creating another conditional commitment CC(c,m,goods,pay) meaning that if the merchant sends the goods, then the customer will pay. (4) If the customer sends a payment to the merchant, then CC(c,m,goods,pay) is discharged and CC(m,c,pay,goods) is reduced to C(m,c,goods) meaning that the merchant is now committed to sending the goods. Conversely, if the merchant sends goods to the customer, then CC(m,c, pay,goods) is discharged and CC(c,m,goods,pay) is reduced to C(c,m,pay) meaning that the customer is now committed to paying for the goods. (5) If the customer has paid in the previous step, then the merchant sends the goods, thereby discharging its commitment. Conversely, if the merchant has sent the goods in the previous step, then the customer sends the payment, thereby discharging its commitment. In either case, no commitments or conditional commitments hold in the resulting state, which is indicated as the final state of the protocol.

3. PROTOCOLS, FORMALLY

We specify protocols declaratively in $C+$ [2], an action description language based on the nonmonotonic causal logic, which captures *universal causation*—every fact that obtains must be caused.

We specify protocols in $C+$ because it supports *elaboration tolerance*. This means that a specification can be refined merely by adding to the existing specification; nothing need be removed from it even if the desired effect is that some inferences are disabled. Elaboration tolerance is a major motivation behind nonmonotonic reasoning. For protocols, elaboration tolerance means that certain interactions can be removed or modified simply by adding axioms to an existing specification. As Section 4 shows, the parts that are to be added correspond to protocol transformers.

3.1 The Action Description Language C+

A specification in $C+$ consists of a set of causal laws. The *signature* of a $C+$ specification is the set of constants that occur in it. ^{act} and ^{fl} represent the sets of actions and fluents, respectively. Each constant c is assigned a nonempty finite domain *Dom(c)* of symbols. An *interpretation* of is an assignment of values $c = v$ for each $c \in$ where $v \in$ *Dom(c)*. A parameterized constant represents a family of constants, the members of which are all its ground instances.

An action description in $C+$ describes a transition system, a graph with states as vertices and actions as edges. Here, a state s is an interpretation of ^{fl} and a transition is a triple $\langle s, e, s' \rangle$ where s and s' are states, and e is an interpretation of ^{act}.

Below, we explain informally the meanings of the kinds of laws we use. Here $a, b \ldots$ represent boolean action literals, and $f, g, \ldots$ represent boolean fluent literals (except in declarations, where they are treated as names); $A, B, \ldots$ and $F, G, \ldots$ represent formulas (combinations using $\wedge, \neg, \vee$) of actions and fluents, respectively.

SCHEMA 1. *a :: exogenousAction.*

Schema 1 declares an exogenous action a, which means that there is a default cause for a to happen or not happen. In other words, a simply happens or does not happen. We model *protocol actions* such as sendRequest and sendOffer as exogenous actions.

SCHEMA 2. *a :: action.*

Schema 2 declares an action that is not exogenous. Such actions are termed *auxiliary actions*. Auxiliary actions must be caused by protocol actions. Commitment operations are auxiliary actions.

Let P_A and A_A denote the sets of protocol and auxiliary actions, respectively. Then $P_A \cup A_A =$ ^{act}, the set of actions of an action description. All actions are atomic—they either happen or do not happen.

SCHEMA 3. *f :: inertialFluent.*

Schema 3 declares a fluent with the property that its value persists from one state to the next unless changed by some other law.

SCHEMA 4. *f :: sdFluent.*

Schema 4 declares f as a static fluent, that is, the value of f depends only on other fluents, not on actions.

SCHEMA 5. *nonexecutable a if b.*

Schema 5 ensures that a and b do not happen concurrently. We use axioms of this schema for every pair of distinct protocol actions, thereby ensuring that no two protocol actions are concurrent.

SCHEMA 6. *a causes b if F.*

Schema 6 says that in a state where F (the precondition) holds, if a happens, then b happens concurrently. If F is the formula $\top$, the law reduces to a causes b. In this case, in a state where a happens, b happens concurrently.

We restrict the usage of Schema 6 so that a and b are protocol and auxiliary actions, respectively. This makes sense if we think of protocol actions causing commitment actions. Also, it does not make sense for b to be a protocol action as we apply Schema 5 to restrict protocol actions to be nonconcurrent.

SCHEMA 7. *a causes F if G.*

Schema 7 says that in a state where G holds, if a happens, then F holds in the next state. If G is the formula $\top$, the law reduces to a causes F. In this case, in a state where a happens, F holds in the next state.

SCHEMA 8. *nonexecutable a if F.*

Schema 8 specifies that in a state where F holds, a cannot happen. Axioms of this schema help impose an ordering on the protocol actions.

SCHEMA 9. *a causes a.*

Schema 9 specifies that the action a happens by default if a is a positive literal. If a is negative literal, then a does not happen by default.

SCHEMA 10. *caused f if f.*

Schema 10 says that the fluent f holds by default if f is a positive literal. If f is a negative literal, then f does not hold by default.

SCHEMA 11. *caused f if F.*

Schema 11 says that f holds in a state where F holds.

3.2 Protocol Specification

Listing 1 specifies commitments and commitment operations in $C+$. Only partial listings are shown due to space limitations. Complete listings for all specifications in this paper are available at http://research.csc.ncsu.edu/mas/code/. Listing 1 is independent of any particular protocol, and is incorporated into each protocol. Commitments and conditional commitments are modeled as inertial fluents (Lines 10–11). Commitment operations are modeled as simple (not exogenous) actions (Lines 12–13). Causal laws of Schema 7 are used to model commitment operations (Lines 18–19). By default, all commitment actions are disabled (Line 27). These laws are required because of universal causation—they explain situations in which the commitment actions do not happen.

```
1  % The listing referred to as 'com-ops' from other listings
2  :- sorts
3   role;
4   condition.
5  :- variables
6   x, y, z          ::role;
7   p,q              ::condition.

9  :- constants
10  ccommitment(role,role,condition,condition),
11  commitment(role,role,condition) :: inertialFluent;
12  create(role,role,condition) :: action;
13  discharge(role,role,condition) ::action;
14  ...

16 % '<>', '++', '&', '-' represent logical
17 % inequality, or, and, and negation, respectively.
18  create(x,y,p) causes commitment(x,y,p) where x<>y.
19  discharge(x,y,p) causes -commitment(x,y,p) where x<>y.
20 % Creation of conditional commitment
21  ccreate(x,y,p,q) causes ccommitment(x,y,p,q)
22                          where x<>y & p<>q.
23  ...

25 % By default, all commitment actions are disabled;
26 % they must be caused by protocol actions.
27  -create(x,y,p) causes -create(x,y,p).
28  ...
```

Listing 1: The basic commitment operations captured in $C+$

Listing 2 specifies the purchase protocol of Figure 1. Note that it includes Listing 1. Protocol actions are declared on Lines 11–12 as exogenous actions. They correspond to the labels on the edges in Figure 1. Fluents that capture the state of the protocol are declared in Lines 15–16. Line 21–22 capture the modeling of the sendRequest action. Lines 25–28 show a part of the modeling of the sendGoods action. Lines 26–28 capture the discharging of the merchant's commitment to send goods. Other protocol actions are appropriately captured, but are not shown here. Towards the end of the listing, the initial and final states of the protocol are defined. The initial state is one where none of the fluents (except initial) hold. The final state is one where a successful exchange of goods and payment has happened. Fluents initial and final are declared to be statically determined (Line 18) as their value depends only on other fluents. Conceptually, initial and final are metafluents: they are part of the ontology for specifying protocols. The nonexecutable axioms serve to impose an ordering on actions. For instance, a sendRequest can happen only in the initial state (Line 22).

```
1  % Include basic commitment operations from Listing 1
2  :-include 'com-ops'.

4  :- objects
5    merchant, customer ::role;
6    goodsc, payc, acceptc ::condition.

8  :- constants

10 % Protocol actions: parameter specifies the performer
11  sendRequest(role), sendOffer(role), sendAccept(role),
12  sendGoods(role), sendPayment(role) ::exogenousAction;

14 % Fluents to characterize the state of the protocol
15  request(role,role), offer(role,role), accept(role,role),
16  goods(role,role), pay(role,role) ::inertialFluent;

18  initial, final :: sdFluent.

20 % Request
21  sendRequest(customer) causes request(customer,merchant).
22  nonexecutable sendRequest(customer) if -initial.

24 % Partial modeling of goods
25  sendGoods(merchant) causes goods(merchant,customer).
26  sendGoods(merchant) causes cdischarge(customer,merchant,
27          goodsc,payc) if ccommitment(customer,merchant,
28          goodsc,payc).
29  ...

31 % Similar axioms for Offer, Accept, and Payment
32  ...

34 % By default, in initial state
35  caused initial if initial.

37 % Not in initial state if any fluent holds
38  caused -initial if request(x,y).
39  caused -initial if offer(x,y).

41 % Similar axioms relating initial and other fluents
42  ...

44 % By default, not in final state
45  caused -final if -final.

47 % In final state, if pay and goods
48  caused final if pay(customer,merchant) &
49                  goods(merchant,customer).
```

Listing 2: The base purchase protocol in $C+$

The sending of a message is captured by the occurrence of the corresponding protocol action. Receiving a message is assumed to be synchronous with its sending. The fluents describe the state of the protocol. Commitment actions give meaning to protocol actions in terms of how protocol actions affect commitments. Therefore, it makes sense to model a commitment action as happening concurrently with the protocol action that causes it.

3.3 Transition Systems

As noted earlier, an action description in $C+$ formally describes a transition system. The transition system of a protocol is based on the transition system of its $C+$ action description with the removal of redundant transitions and unreachable states.

DEFINITION 1. *The set of transitions of a protocol P specified as an action description D is $T_P = \{\langle s, e, s'\rangle \mid \langle s, e, s'\rangle \in T_D$ and $s \neq s'\}$, where T_D denotes the set of transitions of D.*

Thus the transition system of a protocol is different from its action description. In the transition system semantics of $C+$, a nonaction (i.e., the nonoccurrence of an action $a \in {}^{act}$, denoted by the interpretation $a =$ false), is also considered an action. T_D contains transitions where e is interpreted such that every $a \in {}^{act} =$ false. Effectively, no action happens in such transitions. T_D also contains transitions where e is interpreted such that some $a \in {}^{act} =$ true, but the state does not change. (Such transitions could be produced, for example, by axioms of Schema 7 when the action a happens in a state where G does not hold.) Both these types of transitions appear as self-loops around states and are computationally redundant. Definition 1 removes such transitions.

DEFINITION 2. *The set of states of a protocol P specified as an action description D is $V_P = \{s \mid s \in V_D$ and $\exists\langle s', e, s''\rangle \in T_P$ such that $s = s'$ or $s = s''\}$, where V_D denotes the set of states described by D, and T_P denotes the set of transitions of P.*

Definition 2 removes isolated vertices (states). Such vertices have no incoming or outgoing edges.

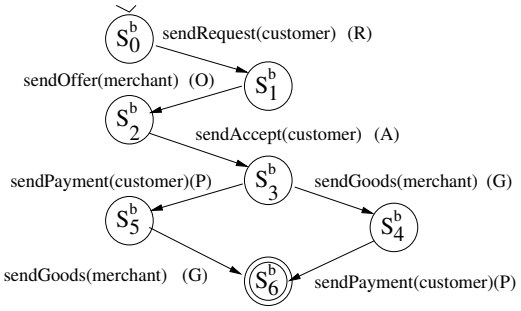

Figure 2: Transition system corresponding to Listing 2

Formally, then, a protocol P is described by $\langle T_P, V_P, V_I, V_F\rangle$, where $V_I \subseteq V_P$ such that $\forall v \in V_I : v \models$ initial, and $V_F \subseteq V_P$ such that $\forall v \in V_F : v \models$ final. Figure 2 shows the transition system corresponding to base purchase (Listing 2). Let's call this protocol P_B (B for base). Table 1 shows the interpretation of the states.

Lemma 1 shows that exactly one protocol action occurs in any transition. We will use this result later.

LEMMA 1. *In the transition system of a protocol P with protocol actions $\{p_1, \ldots, p_n\}$ and auxiliary actions $\{a_1, \ldots, a_m\}$, for every transition $\langle s, e, s'\rangle \in T_P$, exactly one $p_i =$ true $(1 \leq i \leq n)$, in interpretation e.*

Proof. The proof follows from the assumptions specified above about protocol actions and auxiliary actions. By Schema 5, no two protocol actions are concurrent. Therefore, at most one $p_i =$ true $(1 \leq i \leq n)$, in the interpretation e. If some $p_i =$ true, we are done. Else, Definition 1 removes nonaction loops, there must be auxiliary $a_j =$ true $(1 \leq j \leq m)$. However, by Schema 6, auxiliary actions are caused only by protocol actions. Thus some $p_i =$ true in e.

State	Fluents
s_0	*initial*
s_1	*request(customer, merchant)*
s_2	*request(customer, merchant), offer(merchant, customer), CC(customer, merchant, goodsc, payc)*
s_3	*request(customer, merchant), offer(merchant, customer), accept(customer, merchant), CC(customer, merchant, goodsc, payc), CC(merchant, customer, payc, goodsc)*
s_4	*request(customer, merchant), offer(merchant, customer), accept(customer, merchant), goods(merchant, customer), C(c, m, payc)*
s_5	*request(customer, merchant), offer(merchant, customer), accept(customer, merchant), pay(customer, merchant), C(m, c, goodsc)*
s_6	*request(customer, merchant), offer(merchant, customer), accept(customer, merchant), goods(merchant, customer), pay(customer, merchant), final*

Table 1: Interpretation of states in Figure 2: only fluents that obtain are shown.

4. CONTEXTS AND TRANSFORMERS

Section 3 showed how to specify protocols in $C+$, and developed a computational model for them. This section exploits the elaboration tolerance of $C+$ to support the independent specification of protocols and transformers. A protocol transformer is a $C+$ specification that encodes a generic way to handle an aspect of context. To apply a transformer to a protocol, we simply append it to the protocol specification.

A transformer specification may use *external* constants, i.e., fluents and actions that are not declared in it. External constants must be declared in a protocol to which the transformer is applied. Such constants represent part of the interface between the protocol and the transformer. Below we model a few transformers for the base purchase protocol (P_B).

4.1 The Return-Refund Transformer

EXAMPLE 1. *Sending returns and refunds: In some settings, a customer may return goods bought and obtain a refund (if a payment was made). Likewise, a merchant may refund the customer's payment unilaterally if it is unable to deliver the promised goods.* ∎

```
1  :- constants

3    sendReturn(role), sendRefund(role)::exogenousAction;
4    returned(role,role), refunded(role,role)::inertialFluent.

6  % Capturing the effects of the return action

8    sendReturn(customer) causes returned(customer,merchant)
9          & -goods(merchant,customer) if
10         goods(merchant,customer).
11   sendReturn(customer) causes create(merchant,customer,
12         goodsc) if pay(customer,merchant) &
13         goods(merchant,customer).
14   sendReturn(customer) causes cancel(customer,merchant,
15         payc) if commitment(customer,merchant,payc)
16         & goods(merchant,customer).
17   sendReturn(customer) causes ccreate(merchant,customer,
18         payc, goodsc) if commitment(customer,merchant,
19         payc) & goods(merchant,customer).
20   sendReturn(customer) causes ccreate(customer,merchant,
21         goodsc, payc) if commitment(customer,merchant,
22         payc) & goods(merchant,customer).

24 % Sending goods causes them to be not returned.
```

```
26   sendGoods(merchant) causes -returned(customer,merchant).

28 % Axioms for refunds
29   ...
```

Listing 3: Return-Refund transformer in C+

Listing 3 shows a transformer that adds refunds and returns to base purchase. The actions and fluents are declared in Lines 3–4. Lines 8–22 capture the semantics of return. sendReturn(customer) is successful only if goods have already been sent, and it undoes the effect of the sending of goods by the merchant (Lines 8–10). The remaining causal laws capture the effects of performing a return depending on whether payment has already been sent (Lines 11–13) or not (Lines 14–22). In addition, sending goods undoes the effect of returning them (Line 26). This enables cyclic behavior of sending and returning goods. Refunds (not shown in the listing) are captured similarly.

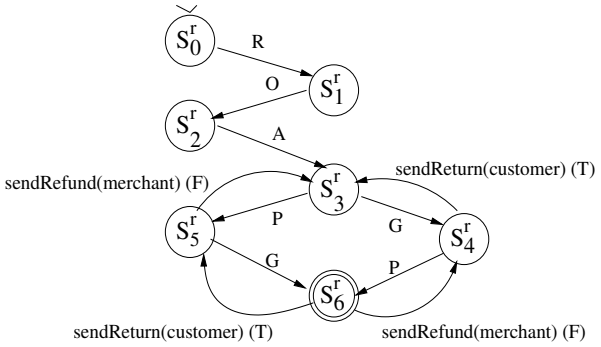

Figure 3: Transition system produced by applying the Return-Refund transformer

To apply this transformer to P_B (base purchase), we append it to the specification of P_B. We refer to the resulting protocol as P_R (for returns and refunds). Figure 3 shows the transition system of P_R. R, O, A, G, P mean the same as in Figure 2.

4.2 The Reminder Transformer

EXAMPLE 2. *Sending reminders: In some context where agents are notorious for not fulfilling their commitments on time, it would be useful for the creditor of a commitment to remind the debtor about the pending commitment. A customer can remind the merchant to send the goods and conversely, a merchant can remind the customer to send a payment.* ∎

```
1  :- constants

3    sendReminder(role) :: exogenousAction;

5  % Every protocol action must cause a transition into
6  % a new state. The num of the reminder serves to
7  % distinguish between two reminders.

9    reminder(role,role,num) :: inertialFluent;

11 % reminderNum keeps track of the number of reminders.
12 % To save space, we removed the lines that say
13 % maxReminders=6

15   reminderNum :: inertialFluent(0..maxReminders).

17 % Reminder: increment reminderNum after every reminder.
```

```
19 sendReminder(y)  causes reminder(x,y,i) & reminderNum=i+1
20         if commitment(x,y,p) & i=reminderNum where
21         i < maxReminders.
```

Listing 4: Reminder transformer in C+

Listing 4 shows a transformer that adds reminders to base purchase. Commitments are the only external constants in this transformer. Both the customer and the merchant can send reminders (limited to six maximum) depending on which commitment holds. The declaration of the fluent reminder(role, role, num) on Line 9 includes a parameter num. Recall that Definition 1 removes self-loops from the transition system. Therefore, protocol actions must be specified so that every time a protocol action happens, it causes a state change. Without the num parameter, reminders after the first reminder would not cause a state change, and would be removed by Definition 1.

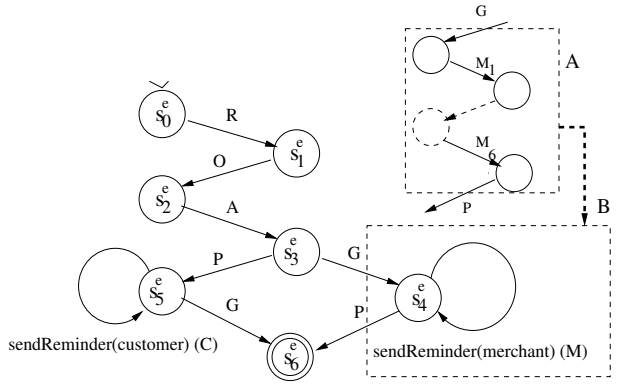

Figure 4: Transition system produced by applying the Reminder transformer

When we apply this transformer to P_B (by appending it to P_B), we obtain a protocol P_E (for reminders). Because the transition system obtained is large, Figure 4 shows the simplified transition system of P_E. Every reminder that the merchant sends transitions the protocol into a new state as shown in block A. We collapse block A into block B, thereby making it seem that reminders happen in a self-loop. We do the same for the customer's reminders. Note that such simplification is not possible in general. We make this simplification here, because we focus on commitments, and sending reminders does not change the commitments in any state.

4.3 The Pay-Before-Goods Transformer

This is an example of a transformer that *removes* some previously allowed interactions. Recall that base purchase allows sending goods and payment in any order.

EXAMPLE 3. *Pay first: In a context where customers are not trustworthy (but merchants are), merchants requires payment before delivery of goods.* ∎

```
1  % Goods sent only after payment.

3    nonexecutable sendGoods(merchant) if
4          -pay(customer,merchant).
```

Listing 5: Pay-Before-Goods transformer in C+

Listing 5 removes the path where the merchant sends goods before payment is received. When we append this transformer to P_B,

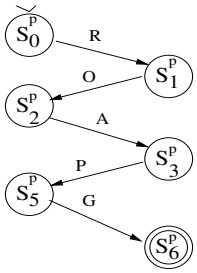

Figure 5: Transition system produced by applying the Pay-Before-Goods transformer

we obtain the protocol P_P (for pay first) as shown in Figure 5. Its transition system is similar to Figure 2, but with the goods-before-payment path removed.

4.4 The Discount Transformer

The previous transformers did not add any new commitments to base purchase; this does.

EXAMPLE 4. *As an incentive for the customer to pay and complete the transaction, the merchant promises to give a 10% discount on the customer's next purchase if the customer pays.* ▮

```
1  :- objects
2  % 10 percent discount on next purchase.

4    discountNextPurchase :: condition.

6  :- constants

8    sendDiscountOffer(role) ::exogenousAction,
9    discountOffer(role,role) ::inertialFluent.

11 % DiscountOffer

13   sendDiscountOffer(merchant) causes discountOffer(
14           merchant,customer).
15   sendDiscountOffer(merchant) causes ccreate(merchant,
16           customer,payc,discountNextPurchase).

18 % DiscountOffer only if not customer has not paid.

20   nonexecutable sendDiscountOffer(merchant) if
21           pay(customer,merchant) ++
22           -offer(merchant,customer).
```

Listing 6: Discount transformer in C+

Listing 6 shows the corresponding transformer. A new action sendDiscountOffer(merchant) is introduced which creates a conditional commitment that if the customer pays, then it receives a 10% discount on the next purchase. This action can happen only before pay, and only after offer.

Applying this transformer to P_B yields the protocol P_D (for discount). In the resulting transition system, the paths in P_B remain and new paths with sendDiscountOffer(merchant) are added. Figure 6 shows part of the resulting transition system with one such added path, between s_0 and s_8. Note that in s_8, a final state, the fluent C(merchant, customer, discountNextPurchase) holds.

5. SEMANTICS OF TRANSFORMATIONS

The base purchase protocol and its transformations described above offer differing functionality, yet are similar in that they are all purchase protocols. The notion of protocol subsumption [3] helps

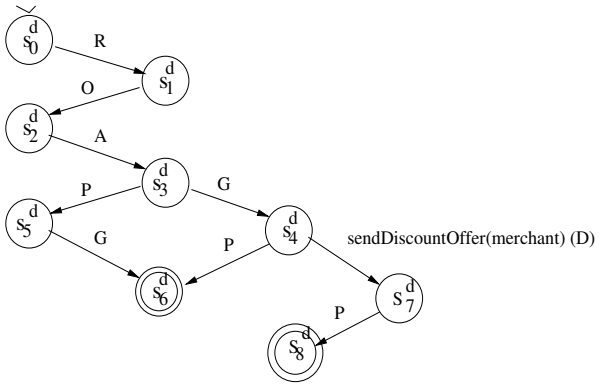

Figure 6: Transition system produced by applying the Discount transformer

characterize the relationship between a protocol and its transformations. The following reviews protocol subsumption, and applies it for understanding transformations.

5.1 Protocol Subsumption

Runs. A run represents a path in the transition system from a initial state to a final state.

DEFINITION 3. *A run of a protocol is a series of transitions* $\langle s_0, e_0, s_1\rangle$, $\langle s_1, e_1, s_2\rangle$, ...,$\langle s_{n-1}, e_{n-1}, s_n\rangle$ *such that* $s_0 \in S$, *the set of initial states, and* $s_n \in F$, *the set of final states.*

$\langle s_0, e_0, s_1, e_1, \ldots, e_{n-1}, s_n\rangle$ is an abbreviated representation of a run. Using Lemma 1, we further abbreviate a run by indicating only protocol actions instead of complete action interpretations, e.g., $\langle s_0, p_0, s_1, p_1, \ldots, p_{n-1}, s_n\rangle$ where $p_i \in P_A$, the set of protocol actions. For example, the set of runs corresponding to Figure 2 is $\{\langle s_0, R, s_1, O, s_2, A, s_3, G, s_4, P, s_6\rangle, \langle s_0, R, s_1, O, s_2, A, s_3, P, s_5, G, s_6\rangle\}$.

State Similarity. We introduce *state similarity* to compare states occurring in runs of different protocols. A state-similarity function f maps a state to a set of states, i.e., $f : \mathbf{S} \rightarrow 2^{\mathbf{S}}$. From f, we induce a binary relation $\approx_f \subseteq \mathbf{S} \times \mathbf{S}$, where $\approx_f = \{(s, f(s)) : s \in \mathbf{S}\}$. We require f to be such that $\approx_f$ is an equivalence relation. For example, commitments could be used to compare states. Two states are *commitment similar* if the same set of commitments hold in them. State-similarity forms the basis of protocol comparison.

Run Subsumption. Let $\prec_\tau$ be a temporal ordering relation on states in a run τ. $s \prec_\tau s'$ means that s occurs before s' in τ.

DEFINITION 4. *A run* τ_j *subsumes* τ_i *under a state-similarity function* f, *denoted by* $\tau_j \gg_f \tau_i$ *if for every state* s_i *that occurs in* τ_i, *there exists a state* s_j *that occurs in* τ_j *such that* $s_j \approx_f s_i$, *and for all* s'_i *that occur in* τ_i, *if* $s_i \prec_{\tau_i} s'_i$ *then there exists* s'_j *that occurs in* τ_j *such that* $s_j \prec_{\tau_j} s'_j$ *and* $s'_j \approx_f s'_i$.

For the sake of subsumption analysis, we omit the actions of a run, thus abbreviating a run $\langle s_0, p_0, s_1, p_1, \ldots, p_{n-1}, s_n\rangle$ to just $\langle s_0, s_1, \ldots, s_n\rangle$. Run subsumption is reflexive, transitive, and antisymmetric up to state similarity [3]. Longer runs subsume shorter runs, provided they have similar states in the same temporal order. For example, the run $\langle s^r_0, s^r_1, s^r_2, s^r_3, s^r_4, s^r_3, s^r_4, s^r_6\rangle$ in purchase with returns and refunds (Figure 3) subsumes under commitment similarity, the run $\langle s^b_0, s^b_1, s^b_2, s^b_3, s^b_4, s^b_6\rangle$ in base purchase (Figure 2, states with the same subscript being commitment similar). However, the run $\langle s^d_0, s^d_1, s^d_2, s^d_3, s^d_4, s^d_7, s^d_8\rangle$ in purchase with discount

(Figure 6) does not subsume under commitment similarity, any run in base purchase as state s_6^b does not have a corresponding commitment-similar state in purchase with discount.

Protocol Subsumption. Informally, a more *general* protocol allows more variations in its runs than a less general one. To help formalize this notion, we define a *span* of a protocol as a set of runs allowed by it. A transition system directly yields a span. We define the *closure* of a protocol as a span that is closed under run subsumption. That is, if a run is in the closure, then all the runs that subsume it (under some state-similarity function) are also in the closure. Closures are unique, and provide a firm basis for comparing protocols.

DEFINITION 5. *The transition-span of a protocol specification P, denoted by $[P]$, is the set of runs in its transition system.*

For example, $[P_B] = \{\langle s_0, s_1, s_2, s_3, s_4, s_6\rangle, \langle s_0, s_1, s_2, s_3, s_5, s_6\rangle\}$ is a transition-span of base purchase.

DEFINITION 6. *The closure of a protocol P under a state-similarity function f is given by $[\![P]\!]_f = \{\tau \mid \forall \tau' \in [P] : \tau \gg_f \tau'\}$.*

DEFINITION 7. *A protocol P subsumes a protocol P' under a state-similarity function f, denoted by $P[\![f]\!\rangle P'$, if $\forall \tau' \in [\![P']\!]_f$, $\exists \tau \in [\![P]\!]_f$, such that $\tau' \gg_f \tau$.*

Considering our example protocols, under commitment-similarity function f, (1) P_B (base purchase) subsumes P_R (with returns and refunds) and vice versa because $[\![P_B]\!]_f = [\![P_R]\!]_f$, (2) P_B subsumes P_E (with reminders) and vice versa because $[\![P_B]\!]_f = [\![P_E]\!]_f$, (3) P_B strictly subsumes P_P (pay-before-goods) because $[\![P_P]\!]_f \subset [\![P_B]\!]_f$, and (4) P_D (with discount) strictly subsumes P_B because $[\![P_B]\!]_f \subset [\![P_D]\!]_f$. (Comparing across Figures 2, 3, 4, 5, and 6, the states labeled with the same subscript are commitment similar.)

DEFINITION 8. *The set of induced runs of a protocol P under a state-similarity function f is $([P])_f = [\![P]\!]_f \setminus [P]$.*

In other words, the induced runs are not explicitly present in the transition system but are implicitly present in the specification.

5.2 Types of Transformations

A transformation can be viewed as a function that takes a protocol as input and produces a different protocol. In simple terms, a transformation either (1) adds, (2) removes, or (3) adds and removes runs from a protocol. The definitions below characterize transformations based on how they change the span of a protocol. The examples below use commitment similarity as the state-similarity function.

DEFINITION 9. *A transformation is closure expanding with respect to a state-similarity function f if the resulting transition-span has at least one additional run that does not belong to the closure of the original protocol.*

The transformation from P_B (Figure 2) to P_D (Figure 6) is closure expanding since it adds the run $\langle s_0^d, \ldots, s_8^d\rangle \notin [\![P_B]\!]_f$.

LEMMA 2. *A protocol resulting from a closure-expanding transformation with respect to a state-similarity function f strictly subsumes the original protocol under f.*

DEFINITION 10. *A transformation is closure preserving with respect to a state-similarity function f if the transition span of the resulting protocol is different from that of the original protocol, but their closures are identical (under f).*

No.	$[P]$		$[\![P]\!]_f$		Transformation
	Removes	Adds	Removes	Adds	
1	x	x	x	x	null
2	x	x	x	✓	Impossible
3	x	x	✓	x	Impossible
4	x	x	✓	✓	Impossible
5	x	✓	x	x	Closure preserving
6	x	✓	x	✓	Closure expanding
7	x	✓	✓	x	Impossible
8	x	✓	✓	✓	Impossible
9	✓	x	x	x	Closure preserving
10	✓	x	x	✓	Impossible
11	✓	x	✓	x	Closure contracting
12	✓	x	✓	✓	Impossible
13	✓	✓	x	x	Closure preserving
14	✓	✓	x	✓	Closure expanding
15	✓	✓	✓	x	Closure contracting
16	✓	✓	✓	✓	Not characterized

Table 2: Transformation possibilities

The transformations from P_B (Figure 2) to P_R (Figure 3) and to P_E (Figure 4) are both closure preserving since each transformation adds runs that were already present in $[\![P_B]\!]_f$. A transformation that removes runs from the transition span, but is still closure-preserving, is one that removes reminders from P_E.

LEMMA 3. *A protocol resulting from a closure-preserving transformation with respect to the state-similarity function f subsumes the original protocol under f, and vice versa.*

DEFINITION 11. *A transformation is closure contracting if the resulting transition-span is a proper subset of the transition-span of the original protocol.*

The transformation from P_B (Figure 2) to P_P (Figure 5) is closure contracting since it removes the run $\langle s_0^b, \ldots, s_4^b, s_6^b\rangle \in [P_B]$.

LEMMA 4. *A protocol subsumes a protocol resulting from a closure-contracting transformation under every f.*

Table 2 shows that the above-defined types of transformations represent an exhaustive list; there can be no other type of transformation. Each row is to be read as "a transformation that Removes (if ticked) and Adds (if ticked) from $[P]$, and at the same time Removes (if ticked) and Adds (if ticked) from $[\![P]\!]_f$ is of type Transformation". To understand the table, we present the following observations that follow from the definition of closure. (1) To change the closure, the transition span must be changed making the transformations in rows 2–4 impossible. (2) A transformation that only adds runs to the transition span cannot remove runs from the closure making transformations in rows 7–8 impossible. (3) Similarly, a transformation that only removes runs from the transition-span cannot add runs to the closure making rows 10 and 12 impossible. (4) The transformations of the type in row 16 cannot be generally characterized in terms of closures—depending on the specific transformation in consideration, the transformation may be expanding, preserving or contracting, or there may be no subset relation between the closures of the original and the resulting protocol—causing them to be labeled 'not characterized'. We lack the space to give detailed examples.

5.3 Compliance

How can we use the above characterization of transformations? One use is in determining compliance. In open systems, checking whether an agent complies with stated protocols is nontrivial. A simple approach for verifying compliance is to check an agent's enactment of a protocol against the protocol's state machine: any deviation from the state machine would be a violation. But some deviations may be necessary for variations in context. A commitment-based approach can allow flexible enactments while enabling verification of compliance [9]. Practical settings, however, often require more structure than general-purpose commitment reasoning allows. In particular, agreement among the interacting parties (expressed in a protocol) may be needed to facilitate their interactions. For example, in a certain context, the parties involved may find sending reminders appropriate, but not allowing arbitrary returns. Where do we get this structure from?

The closure of a protocol specifies which runs it allows. This is important not just for deviations but also for accommodating agents who participate in multiple protocols in a potentially interleaved manner. By changing the similarity function f, we obtain different closures. Thus, the state-similarity function leads to a configurable definition of compliance. An appropriate similarity function can be defined for each domain of interest or institution. Below, enactment refers to an agent's interactions.

DEFINITION 12. *An agent is compliant with a protocol P in a domain f as long as any enactment belongs to $[\![P]\!]_f$.*

Given Definition 12, we relate transformations to compliance as follows. Consider a domain f in which agents use protocol P. If an agent replaces P with another protocol P' and acts in a manner compliant with P', then under what conditions can we say that it is also compliant with P? To be compliant with both P and P', an agent's enactment must belong to both $[\![P]\!]_f$ and $[\![P']\!]_f$.

LEMMA 5. *An agent compliant with P' in a domain f is compliant with protocol P if P' can be obtained from P using a closure-preserving transformation with respect to f.*

In a domain with commitment-similarity, an agent that is compliant with P_R (with returns and refunds) or P_E (with reminders) is compliant with P_B (base purchase), but an agent using P_P (pay first) or P_D (with discount) is not.

6. DISCUSSION

This paper has formalized protocols and transformers, demonstrated how they work, and shown some of their useful properties. When protocols and transformers are treated as software components [7], we can think of creating libraries of them, and using them as off-the-shelf components to create the desired multiagent systems. The key requirement behind components is that they are sufficiently rigorously defined that third-party developers can compose them in ways that the original designers might not have anticipated. This requirement is met for protocols by the present approach.

Some interesting challenges remain. How do we know that a transformer does not conflict with a protocol? For example, applying a Goods-Before-Pay transformer (the inverse of the example of Listing 5) on the Pay-Before-Goods protocol of Figure 5 will result in a protocol with no runs. Clearly, we can build a transition system for the transformed protocol and observe that it allows no runs. However, a characterization based on the specification language would be more perspicuous and helpful during design. In general, what kinds of correctness guarantees can we make regarding a transformer based on the schemas used in it?

Engineering protocols, like any other artifacts, presupposes the existence of effective tools. Because, in practice, protocols would be developed in multiple ways, more than one kind of tool is needed. This paper has concentrated on supporting protocol design from existing protocols. Others have addressed this problem via a composition of existing protocols [10] or as a refinement of part of an existing protocol [4]. In specification terms, a transformer is not different from a protocol. Thus the present approach has elements of composition in a rigorous manner and shows in what cases the resulting protocols are refinements of the original protocols. An alternative approach is to support protocol design from first principles, i.e., based on requirements analysis of the desired class of applications, an approach such as Dooley graphs [5] or Tropos [1] that yields specifications from early requirements would be natural. It would be interesting to see how the approaches can be combined so that we can develop a library of protocols and transformers from first principles, and enable the development of additional protocols as needed from the components in the library.

7. REFERENCES

[1] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Journal of Autonomous Agents and Multiagent Systems*, 8(3):203–236, May 2004.

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

[3] Ashok U. Mallya and Munindar P. Singh. An algebra for commitment protocols. *Journal of Autonomous Agents and Multiagent Systems special issue on Agent Communication*, 2006. To appear.

[4] Hamza Mazouzi, Amal El Fallah Seghrouchni, and Serge Haddad. Open protocol design for complex interactions in multi-agent systems. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 517–526, 2002.

[5] H. Van Dyke Parunak. Visualizing agent conversations: Using enhanced Dooley graphs for agent design and analysis. In *Proceedings of the 2nd International Conference on Multiagent Systems*, pages 275–282. AAAI Press, 1996.

[6] Munindar P. Singh. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7:97–113, 1999.

[7] Clemens Szyperski. Components and the way ahead. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 1, pages 1–20. Cambridge University Press, 2000.

[8] Uniform commercial code. http://www.law.cornell.edu/ucc/ucc.table.html.

[9] Mahadevan Venkatraman and Munindar P. Singh. Verifying compliance with commitment protocols: Enabling open Web-based multiagent systems. *Journal of Autonomous Agents and Multi-Agent Systems*, 2(3):217–236, September 1999.

[10] Benjamin Vitteau and Marc-Philippe Huget. Modularity in interaction protocols. In Frank Dignum, editor, *Advances in Agent Communication*, volume 2922 of *Lecture Notes in Computer Science*, pages 291–309, 2004.