Specifying and Resolving Preferences Among Agent Interaction Patterns[*]

Ashok U. Mallya [†]
Veraz Networks
926 Rock Avenue, Suite 20,
San Jose, CA 95131, USA
amallya@veraznet.com

Munindar P. Singh
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206, USA
singh@ncsu.edu

ABSTRACT

A strength of commitment protocols is that they enable agents to act flexibly, thereby enabling them to accommodate varying local policies and respond to exceptions. A consequent weakness is that commitment protocols thus fail to distinguish between possible executions that are normal and those that may be allowed but are not ideal. This paper develops an approach for specifying *preferences* among executions that are allowed by a protocol. It captures sets of executions via an event constraint specification language and gives them a denotational characterization based on branching-time models. This paper develops algorithms for choosing the best execution path by considering the interplay between the preference specification of a protocol and local policies of agents interacting using the protocol, thereby giving the specifications a natural operational characterization. The value of the concepts developed is illustrated by its application to a recent practical framework for protocols called OWL-P. Further, the paper shows that the operational and denotational characterizations of preference specifications coincide.

Categories and Subject Descriptors

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

1. INTRODUCTION

Commitment protocols capture the essence of the desired interactions in high-level terms. Protocols regulate the externally observable, social behavior of agents, distinguishing what is allowed from what is not. Current approaches, however, do not make any finer distinctions about what is *normal* and what is not.

This paper is about the main consequences of taking a knowledge engineering stance toward commitment protocols.

[*]This research was supported by the NSF under grant DST-0139037.
[†]Ashok Mallya was a full-time doctoral student at NCSU when this paper was submitted; he has since graduated.

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'06 May 8–12 2006, Hakodate, Hokkaido, Japan.
Copyright 2006 ACM 1-59593-303-4/06/0005 ...$5.00.

- To accommodate the openness of the given system, protocols must apply in a wide range of contexts. That is, they must generally allow multiple execution paths. For example, a purchase protocol should allow the possibility that payment might be made before delivery of goods, in which case a refund should be made if delivery is unsuccessful.
- To accommodate agent autonomy, protocols must enable the participating agents to choose their actions and responses to the above kinds of conditions as they see fit. For example, a purchase protocol should allow the possibilities that negotiations may fail, that the payment may be made via a third party, that an agent awaiting missing goods may send a reminder for them.
- To be realistic, the protocols themselves must be based on a study of different usage scenarios, wherein not all possible executions are considered equal. For example, as a practical matter, we would all recognize that it is more normal for a purchase protocol to lead to an exchange of goods and money than for a refund to be issued or for the goods to be repossessed by the merchant because of lack of payment.

In other words, protocols should be flexible enough to accommodate exceptions, but should be described in such a manner that exceptions are distinguished from normal executions. Moreover, protocol descriptions should ideally enable the specification of a hierarchy of exceptions, some being more acute (i.e., less desirable) than others.

Protocol Preferences and Agent Policies. To capture the above motivations, we propose that protocol specifications be enhanced with modular, pluggable descriptions of the *preferences* among executions of the protocol. These preferences are the protocol designer's view of what are the most normal or most desired executions. In this sense, they have a normative force. Individual agents would have their own *policies* for how they participate in a given protocol. The policies should generally be in line with the protocol preferences. (Verifying compliance with protocol preferences, however, is nontrivial—we return to this point in Section 6.)

Protocols can be refined to yield protocols that serve the same goal but impose additional requirements. For example, payment by cash is a refinement of payment (in general). Agents contemplating interacting can negotiate about the refinement of the protocol that they will enact. For example, if a merchant accepts only cash, payment by check or charge card is ruled out and payment by cash is the only kind of payment that will work.

More interestingly, agents who are participating in a protocol may negotiate about the specific actions that each would take (from among those that are allowed by the given protocol). For example, given that two parties agree to participate as seller and buyer in a purchase protocol, they might then negotiate about whether the seller is willing to send the goods before the buyer makes a payment. Protocol preferences thus provide a basis for argumentation among the parties.

Approach. As explained in greater detail below, in our framework, protocols are denotationally characterized using sets of runs (i.e., computations), and are operationalized via translation into executable rulesets. With the above background, the approach of this paper proceeds as follows to enable augmenting protocol specifications with preferences.

- We augment an existing protocol specification language with an ability to specify preferences among different executions of protocols. For simplicity of specification, we use a simple language based on linear temporal logic to specify expressions over protocol events to identify executions that are of interest.
- We show how preferences based on such expressions can be operationalized into executable rulesets.
- We show how the above preferences are mapped into a graph whose points are sets of runs, and characterized via branching time models.
- We show that the model-theoretic and the operational characterizations coincide.

We provide the example of a typical interaction between a merchant and a customer through this paper to illustrate our approach.

Contributions. Our main contribution is in developing a new methodology for designing commitment protocols. Our methodology can be used for the following

Modeling exceptions. In traditional process models, exceptions are modeled in an ad hoc manner, when the designer demarcates blocks of the process and assigns exception handlers to those blocks. In our approach, exceptions can be defined separately from the specification of the process or the protocol. As a result, different exception conditions can be assigned to the same protocol based on the context of the protocol execution. This is similar in spirit to aspect-oriented software development.

Selecting protocols. When multiple protocols are available for an agent to realize a certain interaction, that agent can negotiate with the other parties involved in the protocol and, based on its preferences, find out if an execution of that protocol with those participants would be acceptable to it. For example, a customer agent that does not wish to enact a hotel booking protocol in which the hotel can cancel the room and award a refund can choose which hotel to interact with if its preferences are made clear before enactment and if the hotel and the customer try and negotiate the protocol to enact based on their preferences.

Organization. Section 2 introduces commitments and commitment protocols. Section 3 describes the language used to specify sets of execution sequences and its semantics. This section also describes how preferences among sets of execution sequences are specified. Section 4 gives an algorithm for resolving preferences among execution sequence sets so that agents can choose the best course of action. Section 5 lists important patterns that arise due to the interplay between local-actions, nonlocal actions, local policies, and protocol preferences. Section 6 concludes the paper with a summary of our contributions and a survey of related literature.

2. BACKGROUND

We formalize protocols as transition systems similar in spirit to commitment machines [9], which are declarative specifications. For compatibility with this model, we use the linear temporal logic-based event ordering language proposed by Singh [7] to specify preferences between sets of executions in that transition system. Although Singh's language is given semantics based on linear models, the models are related in an incremental manner to other possibilities: thus it supports a branching interpretation. We use expressions over events in this language to specify sets of runs of a protocol over which preferences can be expressed. This section briefly introduces the semantics of commitments to lay the groundwork for preference specification and introduces the language.

2.1 Commitments

A commitment $\mathsf{C}(x, y, p)$ denotes that the agent x is obliged to the agent y for bringing about the condition p. Here x is called the *debtor*, y the *creditor*, and p the *condition* of the commitment. The condition is expressed in a suitable formal language.

Commitment Operations. Commitments are created, satisfied, and transformed in certain ways. The following operations are conventionally defined for commitments, where $c \equiv \mathsf{C}(x, y, p)$: $\textsc{create}(x, c)$ establishes the commitment c, and can only be performed by c's debtor *x*. $\textsc{cancel}(x, c)$ cancels the commitment c, and can only be performed by c's debtor *x*. Generally, cancellation is compensated by making another commitment. $\textsc{release}(y, c)$ releases c's debtor *x* from commitment c, and only can be performed by the creditor *y*. $\textsc{assign}(y, z, c)$ replaces *y* with *z* as c's creditor. $\textsc{delegate}(x, z, c)$ replaces *x* with *z* as the c's debtor. $\textsc{discharge}(x, c)$ c's debtor x fulfills the commitment.

A commitment is *active* if it has been created, but not yet been operated upon by a *discharge*, *delegate*, *assign*, *cancel*, or *release*. A commitment is *satisfied* when its condition becomes true. Commitments can also be *conditional*, denoted by $\mathsf{CC}(x, y, p, q)$, meaning that x is committed to y to bring about q if p holds. A conditional commitment such as $\mathsf{CC}(x, y, p, q)$ becomes an unconditional commitment $\mathsf{C}(x, y, q)$ when its condition p holds. A commitment is *breached* when it is not possible that the commitment will be satisfied. Realistic settings assign deadlines to commitments to detect their breach or satisfaction [5].

Commitment Protocols. Commitment protocols are interaction patterns expressed in terms of the commitments that arise between the participants of the interaction. Commitment protocols have been operationalized via commitment machines. Commitment machines generate computations or *runs*, which are sequences of *states* that a valid protocol execution goes through. Propositions represent facts about the universe of discourse of the protocol such as the actions that the protocol participants have taken, commitments that have been created and operated upon, and messages that have been sent. Each state is a snapshot of the evolving state of the system (as the given agents interact), and is labeled by *propositions* that hold true there. State changes are caused by *messages* that the participants send to each other.

This paper builds on OWL-P [1, 2], which is a practical framework and an associated language for specifying, combining, and enacting commitment protocols. OWL-P (OWL for Protocols and Processes), uses OWL (Web Ontology Language) to specify protocol and protocol composition constructs. An OWL-P specification identifies roles that participate in the protocol, the messages that are exchanged (with the meanings of the messages in terms of commitments that are created), and a set of rules that constrain the set of runs of the protocol by specifying ordering, data flow, and other constraints.

The flexibility of commitment protocols arises because they use commitments and other propositions to assign meanings to states and thus capture the essence of an interaction. Treating commitments explicitly enables further kinds of sophisticated reasoning, such as involving delegation and other kinds of manipulation of commitments. For example, a customer in a purchase protocol can delegate to a third party (such as a bank) the payment commitment that the customer has towards the merchant.

2.2 Running Example

We illustrate our approach with an example interaction in which a merchant sends some goods to a customer in exchange for a payment. The exact protocol for this exchange involves variations because the customer could pay before the merchant ships the goods and, if not satisfied, can return the item to the merchant for a refund of the payment.

3. SPECIFYING PREFERENCES

OWL-P specifies protocols declaratively and leverages the semantics of commitments and their operations to make protocols flexible. However, OWL-P does not distinguish between the normal and abnormal executions of a protocol. For example, executions in a purchase protocol that involve returns and refunds or missing shipments are treated on par with the normal executions involving no missing shipments or delayed payments. For this reason, we extend OWL-P to capture preferences among sets of executions. We first devise a language over events to concisely specify sets of protocol executions. Next, we specify preferences among such sets using this language.

3.1 The Constraint Specification Language

We use the event-based linear temporal logic $\mathcal{I}$ as introduced by Singh [7] for specifying sets of executions. We repeat here the syntax and semantics of $\mathcal{I}$ from [7]. I is the start symbol of the BNF for the language of $\mathcal{I}$. In this BNF, *slant* indicates nonterminals, $\longrightarrow$ and $|$ are meta-symbols of the BNF, /* and */ begin and end comments, respectively, and all other symbols are terminals.

L_1. $I \longrightarrow$ *dep* $|$ *dep* $\wedge$ I /*conjunction: interleaving*/

L_2. *dep* $\longrightarrow$ *seq* $|$ *seq* $\vee$ *dep* /* disjunction: choice*/

L_3. *seq* $\longrightarrow$ *bool* $|$ *event* $|$ *event* $\cdot$ *seq* /* before: ordering*/

L_4. *bool* $\longrightarrow 0$ $| \top$

Dependency. A dependency is an expression generated by I. It specifies constraints on the occurrence and ordering of events.

Event Literal Set. $\Gamma \neq \{\}$ is the set of event literals as generated by the nonterminal *event.* Each event e literal has a *complement* $\overline{e}$. Intuitively, initially, neither an event nor its complement holds; ultimately, one of them must hold. Γ_D is the set of literals mentioned in a dependency D and their complements. For example, $\Gamma_e = \{e, \overline{e}\}$. For a set of dependencies $\mathbb{D}$, we define $\Gamma_{\mathbb{D}}$ as $\Gamma_{\mathbb{D}} = \bigcup_{D \in \mathbb{D}} \Gamma_D$.

In our example, we denote the payment made by the customer by p, the sending of goods by the merchant by s, the return of the goods by ret, and a refund by ref. Using these event literals, we can specify the constraint "refund can be done only after payment has been made" as a dependency $\overline{ref} \vee p \cdot ref$.

Since state labels in OWL-P capture the history of events in the system, the event-based language (and semantics) that we use here can be applied to the state-based execution semantics of OWL-P.

The formal semantics of $\mathcal{I}$ is based on runs, i.e., sequences of events. Legal runs satisfy the following requirements:

1. Event instances and their complements are mutually exclusive.
2. An event instance occurs at most once in a computation.

Universe of Runs. $\mathbb{U}_{\mathcal{I}}$ is the universe of runs; it contains all legal runs involving event instances from Γ.

3.2 Constraint Language Semantics

For a run $\tau \in \mathbb{U}_{\mathcal{I}}$ and $I \in \mathcal{I}$, $\tau \models I$ means that I is satisfied over the run τ . This notion can be formalized as follows. Here, τ_i refers to the ith item in τ and $\tau_{[i,j]}$ refers to the subrun of τ consisting of its elements from index i to index j, both inclusive. $|\tau|$ is the last index of τ and may be ω for an infinite run. We use the following conventions in the specification of semantics below: $e, f, \overline{e}, \overline{f}$, etc. are literals; D, E, etc. are dependencies; i, j, k, etc. are temporal indices; and τ, etc. are runs. The semantics of $\mathcal{I}$ is

M_1. $\tau \models e$ iff $(\exists i : \tau_i = e)$

M_2. $\tau \models I_1 \vee I_2$ iff $\tau \models I_1$ or $\tau \models I_2$

M_3. $\tau \models I_1 \wedge I_2$ iff $\tau \models I_1$ and $\tau \models I_2$

M_4. $\tau \models I_1 \cdot I_2$ iff $(\exists i : \tau_{[0,i]} \models I_1$ and $\tau_{[i+1,|\tau|]} I_2)$

Denotation. The denotation $[\![D]\!]$ of a dependency D is the set of runs that satisfy D, i.e., $[\![D]\!] = \{\tau : \tau \models D\}$.

For example, the denotation of the dependency $\overline{ref} \vee p \cdot ref$ is the set of runs in which there is no refund ($\overline{ref}$) or, if there is a refund, a payment precedes it ($p \cdot ref$).

3.3 Residuation

The residual of a dependency D by an event e is denoted by D/e and corresponds to the largest set of runs satisfying the given dependency. Formally, $\nu \in [\![D/e]\!]$ iff $(\forall v : v \in [\![e]\!] \Rightarrow (v\nu \in \boldsymbol{U}_{\mathcal{I}} \Rightarrow v\nu \in [\![D]\!]))$

The residual represents what remains in the dependency after a certain event has occurred for the dependency to be satisfied on any run. The following are residuation rules as given by Singh:

R_1. $0/e \doteq 0$

R_2. $\top/e \doteq \top$

R_3. $(E_1 \wedge E_2)/e \doteq ((E_1/e) \wedge (E_2/e))$

R_4. $(E_1 \vee E_2)/e \doteq ((E_1/e) \vee (E_2/e))$

R_5. $(e.E)/e \doteq E$ if $e \notin \Gamma_E$

R_6. $D/e \doteq D$ if $e \notin \Gamma_D$

R_7. $(e' \cdot E)/e \doteq 0$ if $e \in \Gamma_E$ where e' is any event literal

R_8. $(\overline{e} \cdot E)/e \doteq 0$

Consider the dependency $p \cdot$ *ret* $\cdot$ *ref*, which encodes "payment followed by a return followed by a refund." The residual of this dependency with p is *ret*$\cdot$*ref*, which is what is left to be done to satisfy the dependency after p occurs. The residual of this dependency with s, however, is the dependency itself, since the dependency does not specify whether s occurs.

3.4 Preferences

The specification language $\mathcal{I}$ supports dependencies to succinctly specify sets of runs. To induce a preference structure over such sets, i.e., to specify if one set is preferred over another, we introduce the *preference relation*.

Preference Relation. Let $\mathbb{R} \subseteq 2^{\mathbb{U}_\mathcal{I}} \times 2^{\mathbb{U}_\mathcal{I}}$ be the preference relation between sets of runs. $\mathbb{R}$ is irreflexive, transitive, and antisymmetric. $(S_i, S_j) \in \mathbb{R}$ means that any run in S_i is preferred over any run in S_j. Consequently, $\mathbb{R}$ is downward closed in the sense that if $(S_i, S_j) \in \mathbb{R}$, then for all subsets S'_i and S'_j of S_i and S_j, respectively, $(S'_i, S'_j) \in \mathbb{R}$. For convenience, we apply $\mathbb{R}$ to dependencies, meaning that their denotations are appropriately related. $(D_i, D_j) \in \mathbb{R}$ means that any run in the denotation of D_i is preferred over any run in the denotation of D_j, i.e., $\forall \tau_i, \tau_j : \tau_i \in [\![D_i]\!]$ and $\tau_j \in [\![D_j]\!], \tau_i$ is preferred over τ_j.

A commitment protocol is associated with a set of dependencies. For a protocol P specified in terms of a set of dependencies D_P, the denotation of the protocol is the set of runs that the protocol allows, and is given by $[\![P]\!] = \bigcup\limits_{D \in D_P} [\![D]\!]$. We encode our merchant-customer example by the following dependencies

P_1. Refund only if payment has been done: $\overline{ref} \vee p \cdot ref$.

P_2. Return only if goods have been shipped: $\overline{ret} \vee s \cdot ret$.

P_3. If payment has been made and goods have been returned, then refund: $\overline{p} \vee \overline{ret} \vee ref$.

Preference Graph. A preference graph specifies preferences among a set of dependencies. Each dependency labels one node of the graph, and a preference relation specifies preferences among these dependencies, and consequently among the nodes they label. A preference graph $L = \langle D_x, R \rangle$ specifies preferences among the elements of the set of dependencies D_x using the partial order induced by R over D_x. For the above L, we define its event literal set to be the set of all events that are mentioned in the dependencies in D_x, and their negations. $\Gamma_L = \bigcup\limits_{D \in D_x} \Gamma_D$.

Figure 1 shows a schematic representation of an example preference graph. This graph encodes the following:

D_1. The most preferred runs are those in which payment and shipping occur, and a return or a refund do not.

D_2. Less preferred than runs in D_1 are runs in which goods are returned and payment is refunded.

D_3. Also less preferred than runs in D_1 are runs in which the payment is refunded and the goods are returned.

D_4. Less preferred than all of the above are runs in which goods are returned, but no refund occurs.

D_5. Also preferred less than D_1, D_2, and D_3 are runs in which the goods are shipped, a refund is made, but the the goods are not returned.

Preference Node Denotation. The denotation of a preference node N_D labeled by D, with respect to a protocol P and a graph $L = \langle D_P, R \rangle$, where $D \in D_P$, is the set of runs allowed by the protocol that are also allowed by D, but not by nodes in L that are preferred over N_D. The motivation for the above is that each run can occur in the denotation of at most one node in the graph. Thus if a given run is allowed by two sets of runs, one preferred over the other, the run from the less preferred set would not "get credit" for this run. On each path from the top to the bottom of the graph, a given run can occur at most once. Formally, $[\![N_D]\!] = [\![P]\!] \cap ([\![D]\!] - \bigcup\limits_{(D_i, D) \in R} [\![D_i]\!])$.

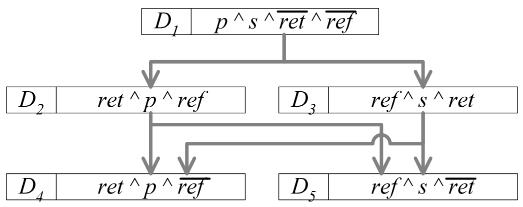
D_1 $p \wedge s \wedge \overline{ret} \wedge \overline{ref}$
D_2 $ret \wedge p \wedge ref$
D_3 $ref \wedge s \wedge ret$
D_4 $ret \wedge p \wedge \overline{ref}$
D_5 $ref \wedge s \wedge \overline{ret}$

Figure 1: A preference graph for our example, $L = \langle \{D_1, D_2, D_3, D_4, D_5\}, \{(D_1, D_2), (D_1, D_3), (D_2, D_4), (D_3, D_4), (D_2, D_5), (D_3, D_5)\} \rangle$.

Consider the preference node formed from D_4. Since the protocol contains no runs in which the payment has been made, goods have been returned, and the refund does not occur, $[\![D_4]\!] = \{\}$. However, the protocol allows runs in which the payment is refunded, the goods have been shipped, but are not returned. The preference node formed from D_5 captures such runs as the least preferred runs in the protocol.

4. RESOLVING PREFERENCES

How can an agent operationalize a preference graph during protocol enactment?

4.1 Evolution of Graphs

A preference graph evolves as events occur, causing the state of the protocol to change. The evolution changes the denotation of each node of the graph because events residuate dependencies that label graph nodes. Below, we describe, and give an algorithm for, the evolution of a (preference) graph. An *evolution tree* captures the evolution of a graph. Below, $\tau + e$ concatenates event e to run τ.

Evolution Tree. An evolution tree $\mathbb{T}_L$ of a graph L represents all possible states that L can evolve through as a result of events in Γ_L. An evolution tree is a tuple, $\mathbb{T}_L = \langle \mathbb{V}, V_{\tau_0}, \mathbb{E}, L \rangle$, where $\mathbb{V}$ is a set of *evolution* nodes, V_{τ_0} is a distinguished root node, $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ is a set of *evolution edges*, and L is a preference graph.

Evolution Node. An evolution node captures a state of a preference graph. An evolution node corresponds to a *path* in the evolution tree. This path is a legal run and is the sequence of the events that occurred for the evolution node to be generated, starting from the root of the evolution tree. Evolution nodes are represented as $V_\tau = \langle L, \tau \rangle$, where L is a graph (as evolved from the original preference graph) and τ is the path to V_τ from the root node of the evolution tree. The root node of an evolution tree T_L is $V_{\tau_0} = \langle L, \tau_0 \rangle$, where τ_0 is the zero-length run.

Evolution Edge. An edge labeled e that originates at V_τ ends at $V_{\tau+e}$. Thus edges are written as $\langle V_\tau, e, V_{\tau+e} \rangle$.

Given a graph L, we construct the evolution tree $\mathbb{T}_L$ for it by first creating the root evolution node $V_{\tau_0} = \langle L, \tau_0 \rangle$. From a node V_τ, we create an edge labeled e for each event e in Γ_L that does not occur in τ. For each edge e, we create a new evolution node $V_{\tau+e} = \langle L', \tau + e \rangle$. For this new node, we set $L' = \langle D', R' \rangle$, where D' is the set of nonzero dependencies obtained by residuating each dependency in L by e, and R' relates two dependencies in D' if the dependencies corresponding to them in D were related by R. That is, we preserve the preferences among the residual dependencies in the evolution nodes. Algorithm 1 generates an evolution tree of a specified finite depth.

Figure 2 shows parts of the evolution tree generated using Algorithm 1 on the graph shown in Figure 1. In this figure, end denotes a special event signaling the end of the protocol. The complements of events that have not occurred are asserted when end occurs. In the figure, for example, since ret and ref have not occurred, their complements $\overline{ret}$ and $\overline{ref}$ are asserted when end occurs.

```
input  : A preference graph $L = \langle D_x, R \rangle$, and a limit
         MAX_DEPTH for the depth of the tree generated
output: An evolution tree $\mathbb{T}_L = \langle \mathbb{V}, V_{\tau_0}, \mathbb{E}, L \rangle$ of $L$
1  Create the root node $V_{\tau_0} = \langle L, \tau_0 \rangle$ ;
2  depth $\leftarrow 1$ ;
3  $V \leftarrow V_{\tau_0}$ ;
4  $\mathbb{T}_L.\mathbb{V} \leftarrow \{V\}$ ;
5  $\mathbb{T}_L.\mathbb{E} \leftarrow \{\}$ ;
6  siblings $\leftarrow \{V\}$;
7  while depth $<$ MAX_DEPTH $\&\&$ siblings $\neq \{\}$ do
8      foreach evolution node $V \in$ siblings do
9          siblings $\leftarrow$ siblings $-\{V\}$ ;
10         foreach event $e \in \Gamma_L$ such that $e \notin V.\tau$ do
11             $V_{V.\tau+e} \leftarrow$ createChildNode $(V, e)$ ;
12             $\mathbb{T}_L.\mathbb{V} \leftarrow \mathbb{T}_L.\mathbb{V} \cup \{V_{V.\tau+e}\}$ ;
13             $\mathbb{T}_L.\mathbb{E} \leftarrow \mathbb{T}_L.\mathbb{E} \cup E$ ;
14             if $V_{V.\tau+e}.L.D \not\models \{\}$ then
15                 siblings $\leftarrow$ siblings $\cup \{V_{V.\tau+e}\}$ ;
16     depth $\leftarrow$ depth $+1$ ;
17 return $\mathbb{T}_L$ ;
18 Procedure createChildNode $(V, e)$: Create a child
   evolution node for the node $V$ on event $e$
   input  : An evolution node $V = \langle L, \tau \rangle$, where
            $L = \langle D, R \rangle$, and an event $e$.
   output: An evolution node that represents the state of $L$ on
            event $e$.
19 $D' \leftarrow \{\}$ ;
20 $R' \leftarrow \{\}$ ;
21 foreach dependency $d \in V.L.D$ do
22     if $d/e \not\models 0$ then
23         $D' \leftarrow D' \cup (d/e)$ ;
24         foreach preference $(d_i, d_j) \in V.L.R$ do
25             if $d == d_i$ or $d == d_j$ then
26                 $R' \leftarrow R' \cup (d_i, d_j)$ ;
27 $L' = \langle D', R' \rangle$ ;
28 $V_{V.\tau+e} = \langle L, V.\tau + e \rangle$ ;
29 return $V_{V.\tau+e}$
```

Algorithm 1: GenerateEvolutionTree(L, MAX_DEPTH): Compute the evolution tree for a graph $L = \langle D_x, R \rangle$, where MAX_DEPTH is a bound on the depth of the tree to be generated.

4.2 Choosing Between Runs

Consider an agent X operating within the preference graph specified for a protocol in which it is participating. For such an agent to determine, the action to take (there is an event corresponding to every action instance), each evolution node is a decision point. X chooses the event would that take the protocol to the most preferred preference graph node. Since preference nodes are distributed among the evolution nodes, and X cannot change the state of the protocol at a finer granularity than that of evolution nodes, X determines which evolution node is the most preferred. To determine this, a *preference graph of evolution nodes* is constructed as follows: At any evolution node, to decide the next action to take, induce a graph among the child evolution nodes using the original preference structure, i.e., the preference structure specified in the root of the evolution tree. Once this is done, induce a graph on the evolution nodes based on the following criterion. Given two evolution nodes V_1 and V_2, create an edge from V_1 to V_2 (i.e., mark that V_1 is preferred over V_2) if and only if there is a nonzero dependency D_1 in V_1 which is preferred over all nonzero dependencies in V_2, and no nonzero dependency in V_2 is preferred over D_1. Formally,

$$(V_1, V_2) \in \mathbb{O} \quad \Leftrightarrow \quad \exists D_1 \text{ in } V_1 : \forall D_2 \text{ in } V_2, \\ (D_1, D_2) \in \mathbb{R} \text{ and } (D_2, D_1) \notin \mathbb{R}$$

where $\mathbb{O}$ is the preference relation over evolution nodes. $\mathbb{O}$ is irreflexive, antisymmetric, and transitive by the above definition. Algorithm 2 shows how a preference graph can be constructed between evolution nodes. Applying this algorithm to the children of the root evolution node V_{τ_0} in Figure 2, we obtain the evolution node graph shown in Figure 3. Figure 4 shows the same graph in detail. Since V_p contains the dependency D_1/p which is preferred over all dependencies in V_{ret} and no dependency in V_{ret} is preferred over D_1/p, we create an edge from V_p to V_{ret}. Similar reasoning for all four evolution nodes yields the graph of Figure 3.

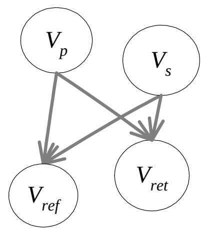

Figure 3: The preference graph for the evolution nodes V_p, V_s, V_{ret}, and V_{ref} shown in Figure 2, based on the preference graph L shown in Figure 1

Selecting the Best Path. Given the graph of preferences among the child (evolution) nodes, the best action to take (event to bring about) is the event that leads to a node in the evolution node graph over which no other node is preferred. However, the evolution node preference graph might have many such nodes. The agent can then choose any one of these. Section 5 discusses some criteria for path selection.

4.3 Types of Events

So far, we have discussed a single agent's perspective. In agent interaction, however, events are brought about by different agents during the enactment of a protocol. Choosing the best path, if one is available, is therefore not up to a single agent. We therefore recognize two types of events, *observed* and *controlled*.

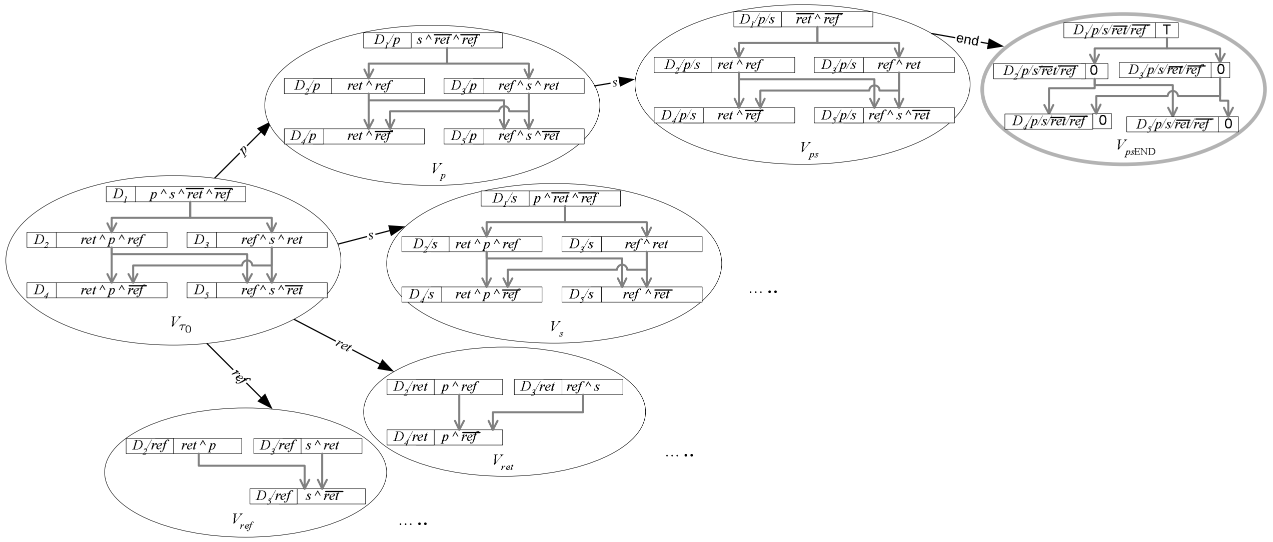

Figure 2: A partial evolution tree for the preference graph shown in Figure 1. Note that the denotations of preference nodes are not taken into consideration in this figure, so that we can illustrate the generation of the evolution nodes clearly. If denotations were taken into account, $[\![D_4]\!] = \{\}$, and this figure would have fewer dependencies. END is a special event signaling the end of the run.

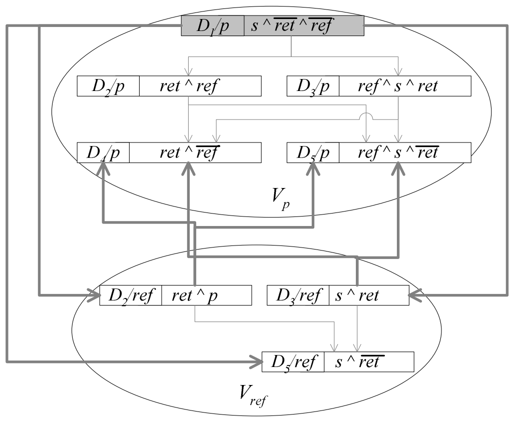

Figure 4: A detailed view of a part of the evolution node preference graph shown in Figure 3. Grey arrows indicate the preference relation. V_p is preferred over V_{ref} because of D_1/p, which is shown in a grey rectangle.

Controlled Events. For a given agent, an event that it can bring about is called a controlled event. We introduce a distinguished controlled event, Noop, which means "do nothing" (in this protocol). An agent can perform a Noop to wait.

Observed Events For a given agent X, an event which X can observe, but not bring about, is called an observed event. We distinguish two unique observed events, start and end, which are observed by agents participating in a protocol when the protocol begins and ends, respectively.

Whether an event is observed or controlled is decided when the events are described for the system being modeled. In our example, payment (p) and return (ret) are controlled events for the customer, and observed events for the merchant, whereas sending goods (s) and refund (ref) are controlled events for the merchant and observed events for the customer.

5. PATTERNS FOR PATH SELECTION

Given that an agent will make a choice at every node of the evolution tree, and that events can be controlled or observed, we identify typical patterns of decision-making that agents can adopt in common situations. Before describing these patterns, we explain rule-based protocols of the type found in the OWL-P framework.

5.1 Rule-Based Protocols

We envision the application of our algorithms in a rule-based protocol enactment framework, such as OWL-P. OWL-P enables agents to specify their *local policies* independent of the protocol specification. The general structure of rules in such protocols is the Event-Condition-Action (ECA) structure,

$$\begin{array}{c}\text{On } e\\ \text{if } \mathit{localPolicy}(e, x)\\ \text{then do } a(x)\end{array}$$

Here, e is an event and $a(\cdot)$ is an action (with a corresponding event e_a), which depends on the local policy for its parameters.

As an example, consider a purchase protocol which requires that the receiver of a $\mathit{rfq}(c, m, \mathit{itemID})$ message (which is a request for a price quote for an item)–the merchant m–respond the the sender–the customer c–with a $\mathit{quote}(m, c, \mathit{itemID}, \mathit{price})$ message (which is a price quote). The rule for this requirement will be

$$\begin{array}{c}\text{On } \mathit{rfq}(c, m, \mathit{itemID})\\ \text{if } \mathit{localPolicy}(\mathit{rfq}(c, m, \mathit{itemID}), \mathit{price})\\ \text{then do } \mathit{quote}(m, c, \mathit{itemID}, \mathit{price})\end{array}$$

Where multiple choices of action are afforded by the protocol on the same triggering event, there will be one rule for each action, with the same event and condition. In such a case, the local policy decides which rule to enable. In case the policy enables multiple rules, the preference specification of the protocol is used to decide which action to take. Based on these principles, we describe patterns of decision-making that agents can use.

5.2 Clear Choice

If an agent has to choose between paths in an evolution tree, and exactly one most-preferred node exists in the evolution node preference graph, the agent takes the path to that node. This situation is shown in Figure 5, and is termed *Clear Choice* .

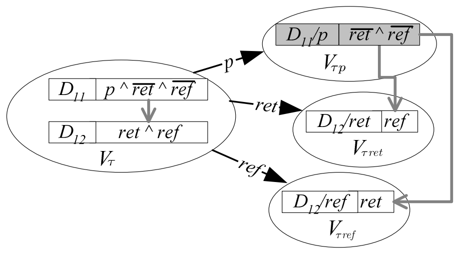

Figure 5: Clear choice: An agent can unambiguously decide which path to choose, since the evolution node preference graph has exactly one node $V_{\tau p}$, over which no other node is preferred.

Using Noop. Sometimes, even when there is a clear choice, an agent might not take any action because of its local policy. Consider the situation shown in Figure 6. This is a variant of our example in which the customer can send an rfq, which is a request for a price quote, to the merchant, to which the merchant can reply with q, which is a quote. At V_τ, the customer c can send either an *rfq* or a payment. According to the preference graph (dictated by the protocol), *rfq* is preferred over payment. However, the customer might not wish to send an *rfq*. Further, instead of performing the only other alternative, payment (p), the customer might want to wait, hoping for a quote (q), or a shipment (s) from the merchant. Since the protocol does not require that c act immediately (most realistic protocols have a notion of timeouts and deadlines), c can perform a Noop, which is do nothing, waiting till its policy allows it to perform an *rfq* or till the protocol forces c to perform p (which will also be a policy decision, if the policy has been designed to not violate the protocol).

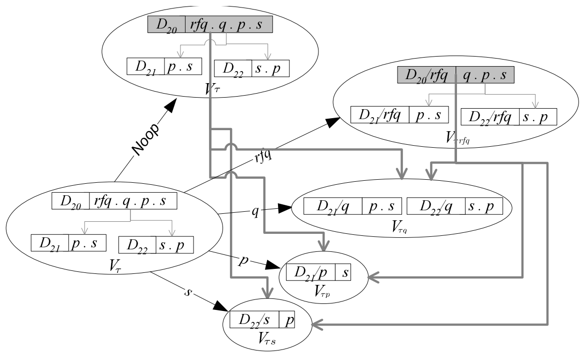

Figure 6: Using a Noop when local policy does not allow the best choice and the protocol does not impose immediate deadlines on the other choice(s). rfq is the best choice for the customer, followed by p, but the the customer chooses to wait using a Noop. This creates two most-preferred nodes, V_τ and V_{rfq}.

5.3 Unclear Choice

The choice of action to take (or event to generate) will be unique if there is exactly one evolution node over which no other node is preferred. However, this is not always the case. Although there will never be a cycle among the preference relations between the evolution nodes (straightforward proof by construction of the evolution node preference graph), it is possible that there are multiple most-preferred nodes, as in Figure 3, where both V_p and V_s are most-preferred, i.e., the customer can either pay or wait for the merchant to send the goods. Since the choice here is between a controlled and an observed event, we call this situation *Unclear Nonlocal Choice*. In such a case, the agent may choose to wait for other agents participating in the protocol to take some action.

There can also be cases in which two or more controlled events are most-preferred (*Unclear Local Choice*). An agent will rely on its local policy to make a choice when this happens. In the Noop example, if there were no preference between D_{20} and D_{21}, the customer would have to decide whether to send an *rfq* or to send a payment. In OWL-P, such local policy-based decisions drive the protocol by enabling agents choose one run from among the many runs that a protocol allows.

6. DISCUSSION

Protocols help us capture agent interactions perspicuously. The very strength of protocols, namely, their flexibility, can pose a challenge. Executions allowed so as to make a protocol flexible under exceptional conditions can begin to be selected over superior executions even when the exceptional conditions do not obtain. This paper proposes that protocols be specified generically, and preference conditions among different sets of runs be stated separately, depending on the context of usage of the protocol. Preferences serve as a rough-and-ready means to capture design goals wherein the normal executions are preferred, yet abnormal executions arising from exceptions or unexpected actions by some of the agents are allowed.

This paper identified a denotational semantics for preference conditions understood as a graph. Next, it showed how such a graph evolves as events and actions take place. It proposed a simple method by which an agent can decide upon its next action, describing an anomalous condition and giving an algorithm for identifying it.

Preferences are often understood solely in decision-theoretic terms. Decision theory is clearly important for modeling preferences and coming up with strategies for acting according to them. However, a denotational understanding of preferences can support the proper application of decision theory.

Related Literature. Yolum and Singh [10] developed one of the first operationalizations of commitments. They specified protocols by listing legal states in terms of the commitments and domain propositions that hold at that state, and using an event calculus planner to generate the set of runs that were allowed. Winikoff *et al.* [9] advance this line of research. Fornara and Colombetti have also proposed a commitment-based interaction protocol framework [3]. However, none of the above approaches specify or operationalize a notion of preferences among the various execution sequences allowed by a protocol. Our work, therefore, is a significant step in this direction. Grosof *et al.* [4] implement rule based agent interaction systems where rules are prioritized. Grosof and colleagues propose *Courteous Logic Program*s, or CLPs. In a CLP, when there is ambiguity regarding which rule to fire, i.e., a conflict arising because multiple rules can be fired at a particular state of the world, the priorities assigned to the rules are used to resolve the conflict. Our work is similar to CLPs in this respect, but different in that we propose a scheme in which preferences among *runs* are specified, independent of a protocol specification. Further, we also present a methodology for translating these preferences into rules that can be embedded into the (rule-based) protocol specification.

Our work is based on the concept of social interaction among agents, which gives importance only to the observable behavior of agents. We describe how preferences among runs can exist. However, we do not study how agents can reason about the benefits of using one set of protocol runs over another. Pasquier and Chaib-Draa [6] introduce the cognitive dissonance theory into multiagent communication by incorporating the theory and dialogue game protocols into agent interactions. Their theory explores ways in which agents can decide when to start dialogues with other agents and what kind of dialogues to initiate, among other things. This line of research is complementary to and would strengthen the interaction framework we have presented here. Preferences among the available runs of a protocol have also been studied from the game theoretic point of view by Otterloo *et al.* [8]. They describe a logic that can be used for reasoning about a strategy to adopt in a game when the preferences of other agents in the game are known. The work differs from ours because of the use of games instead of commitment protocols. Also, preferences of agents are assumed to be known by other agents, which does not always apply in real-world applications such as the business interaction we have outlined in this paper. We plan to incorporate such reasoning among agents into our framework.

7. REFERENCES

[1] N. Desai, A. U. Mallya, A. K. Chopra, and M. P. Singh. OWL-P: A methodology for business process modeling and enactment. In *AAMAS AOIS Workshop*, July 2005.

[2] N. Desai, A. U. Mallya, A. K. Chopra, and M. P. Singh. Interaction protocols as design abstractions for business processes. *IEEE Trans. Software Engg.*, 31(12):1015–1027, 2006.

[3] N. Fornara and M. Colombetti. Defining interaction protocols using a commitment-based agent communication language. In *Proc. 2nd AAMAS*, pp. 520–527. ACM Press, July 2003.

[4] B. N. Grosof and T. C. Poon. SweetDeal: Representing agent contracts with exceptions using XML rules, ontologies, and process descriptions. In *Proc. WWW 2003*, pp. 340–349.

[5] A. U. Mallya, P. Yolum, and M. P. Singh. Resolving commitments among autonomous agents. In F. Dignum, editor, *Advances in Agent Communication*, vol. 2922 of *LNAI*, pp. 166–182, Berlin, 2003. Springer-Verlag.

[6] P. Pasquier and B. Chaib-Draa. The cognitive coherence approach for agent communication pragmatics. In *Proc. 2nd AAMAS*, pp. 544–551. ACM Press, July 2003.

[7] M. P. Singh. Distributed enactment of multiagent workflows: Temporal logic for web service composition. In *Proc. 2nd AAMAS*, pp. 907–914. ACM Press, July 2003.

[8] S. van Otterloo, W. van der Hoek, and M. Wooldridge. Preferences in game logics. In *Proc. 3rd AAMAS*, pp. 152–159. ACM Press, July 2004.

[9] M. Winikoff, W. Liu, and J. Harland. Enhancing commitment machines. In *Proc. AAMAS DALT Workshop*, 2004.

[10] P. Yolum and M. P. Singh. Flexible protocol specification and execution: Applying event calculus planning using commitments. In *Proc. 1st AAMAS*, pp. 527–534. ACM Press, July 2002.

```
input  : An evolution node $V_\tau$, and a root preference graph
         $L_r = \langle D_r, R_r \rangle$
output : A graph $L_e = \langle V_e, R_e \rangle$ representing the
         preferences among the child nodes of $V_\tau$.

$V_e \leftarrow$ All child nodes of $V_\tau$;
$R_e \leftarrow \{\}$ ;
foreach child node $V_{\tau+x}$ of $V_\tau$ do
    foreach child node $V_{\tau+y}$ of $V_\tau$ do
        foreach dependency $d_x$ in $V_{\tau+x}$ do
            isPrefOverAll ← true ;
            foreach dependency $d_y$ in $V_{\tau+y}$ do
                if (getRootDep ($d_x$, $V_{\tau+x}$, $D_r$),
                getRootDep ($d_y$, $V_{\tau+y}$, $D_r$)) $\notin R_r$ or
                (getRootDep ($d_y$, $V_{\tau+y}$, $D_r$),
                getRootDep ($d_x$, $V_{\tau+y}$, $D_r$)) $\in R_r$
                then
                    isPrefOverAll ← false ;
            if isPrefOverAll then
                $R_e \leftarrow R_e \cup (V_{\tau+x}, V_{\tau+y})$;
                break ;
return $\langle V_e, R_e \rangle$ ;
Procedure getRootDep ($d, V_{\tau'}, D_0$): Find which
dependency among $D_0$ node was residuated to $d$ in node
$V_{\tau'}$
input  : A dependency $d$, its evolution node $V_{\tau'}$,and a set
         of dependencies $D_0$.
output : A most-preferred dependency in $D_0$ that was
         residuated to $d$ along path $\tau'$; null if no such
         dependency exists.
foreach $d_0 \in D_0$ do
    $d_t \leftarrow d_0$ ;
    for $i \leftarrow 0$ to $|\tau'| - 1$ do
        $d_t \leftarrow d_t / \tau'_{[i]}$ ;
    if $d_t == d$ then
        $D_{root} \leftarrow D_{root} \cup \{d_t\}$;
    foreach $d_r \in D_{root}$ do
        isPrefOverAll ← true ;
        foreach $d'_r \in D_{root}$ do
            if $(d_r, d'_r) \notin \mathbb{R}$ then
                isPrefOverAll ← false ;
                break ;
            if isPrefOverAll then
                return $d_r$ ;
return null ;
```

Algorithm 2: GenerateEvolutionGraph(V_τ, L_r): Compute the preference graph among child nodes of an evolution node V_τ given the preference graph L_r of the root evolution node.