Processes = Protocols + Policies *

A Methodology for Business Process Development

Nirmit Desai† Ashok U. Mallya Amit K. Chopra Munindar P. Singh

{nvdesai, aumallya, akchopra, singh}@ncsu.edu
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7535, USA

ABSTRACT

Business process modeling and enactment are notoriously complex, especially in open settings where the participants are autonomous, requirements must be continually finessed, and exceptions frequently arise because of real-world or organizational problems. Traditional approaches, which attempt to capture processes as monolithic flows, have proved inadequate in addressing these challenges.

This paper describes a novel approach centered around the concepts of protocols and policies. A (business) *protocol* is a modular, public specification of an interaction among different roles that achieves a desired goal. A *policy* is a typically private description of a participant's business logic that controls how it participates in a protocol. We conceptualize a business *process* be conceptualized as a cohesive set of protocols, to be enacted by agents playing the specified roles in the protocols. This approach presupposes a semantics for protocols that supports reasoning about them and, especially, about enacting them in a flexible, context-sensitive manner. We develop such a semantics based on *commitments*, which capture the essence of the relationships among roles without unnecessarily constraining their behaviors. This paper develops OWL-P, a language for specifying protocols, and associated tools. OWL-P specifications of protocols compile into skeletons for each role. Each skeleton corresponds to a set of rules with place-holders for policies. Developing an agent involves composing the skeletons for its intended roles and supplying the necessary policies to flesh them out.

The key benefits of this approach are (1) a separation of concerns between protocols and policies in contrast to traditional monolithic approaches; (2) reusability of protocol specifications based on design abstractions such as specialization and aggregation; and (3) flexibility of enactment of processes in a manner that respects local policies while adapting continually.

Categories and Subject Descriptors

[**Service Computing & Applications**]: e-Business; [**Software engineering techniques for service-based development**]: Service design principles; [**Core service activities and technologies**]: Service composition; [**Service & AI Computing**]: Multi-agent based service models

*This research was sponsored by NSF grant DST-0139037 and a DARPA project.

†The first three authors are full-time students.

Keywords

Business Process, Software Design, Multiagent Systems, Rule-based Systems

1. INTRODUCTION

Unlike traditional business processes, processes in open, Web-based settings typically involve complex interactions autonomous, heterogeneous *business partners*. Conventionally, business processes are modeled as centralized flows, which are horizontally complete, specifying exact steps for each participant. Because of the exceptions and opportunities that arise in open environments, business relationships cannot be preconfigured to full detail. Thus, flow-based models are difficult to develop and maintain in the face of evolving requirements. Further, conventional models do not facilitate flexible actions by the participants.

This paper proposes an innovative approach for business process modeling and enactment, which is based on a combination of protocols and policies. The key idea is to capture meaningful interactions reusably as *protocols*. Protocols can involve multiple roles and address specific purposes such as ordering, payment, shipping, and so on. Protocols are given a semantics in terms of how their steps affect commitments among various roles: this semantics supports modeling abstractions such as refinement and aggregation as well as enactment abstractions such as delegation, assignment, and scoping into necessary contexts. In this manner, protocols go beyond the rigid two-party, request-response interactions of today's approaches, such as RosettaNet's Partner Interface Processes (PIPs) [14]. In order to maximize participants' autonomy and to be reusable, protocols should emphasize the essence of the interactions and omit local details. Such details are supplied by each participant's *policies*. For example, when a protocol allows a participant to choose from multiple actions, the participant's policy decides which one to perform. Policies help decide the contents of the messages sent, and the processing of the messages received.

In other words, protocols are public, whereas policies are private. The decentralized modeling and enactment of business processes is the key feature of our approach. This paper seeks to develop the main techniques needed to make this promising approach practical. Our contributions include a language and ontology for protocols called OWL-P, which is coded in the Web Ontology Language (OWL) [12]. OWL-P describes concepts such as roles, the messages exchanged between the roles, and declarative rules that describe the effects of sending messages in terms of commitments. OWL-P compiles into Jess rules [6] into which local policies can be readily inserted in a principled manner.

From the software engineering point of view, the clear separation of protocols and policies offers certain advantages. Protocols can be reused across business processes. Protocols may not only be reused directly, they are also amenable to abstractions such as refinement and aggregation [10].

1.1 Running Example

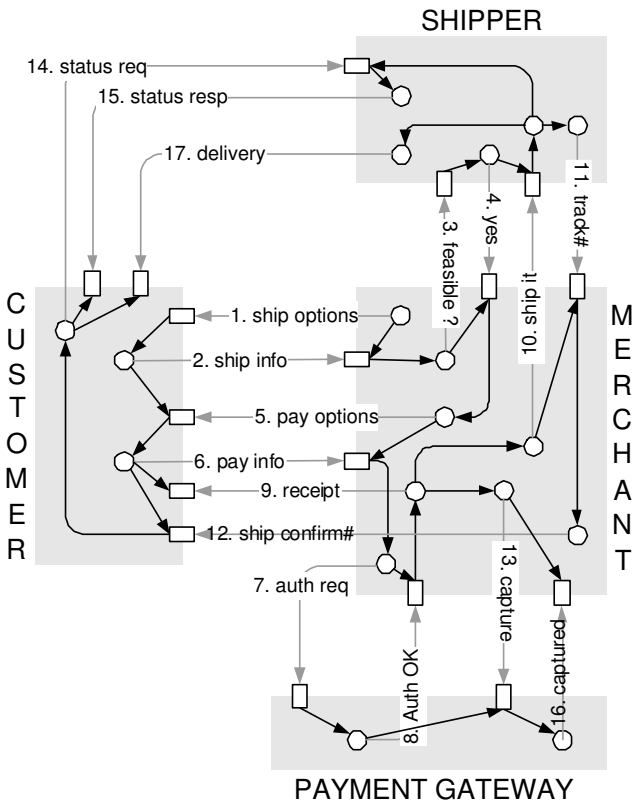

Figure 1: A purchasing process

As a running example, let's consider a business process involving a small number of parties. Figure 1 depicts a purchase process where items to be purchased have already been selected and the price has been agreed upon. This process involves a Customer who wants to buy items, a Merchant who sells items, a Shipper who is a logistics provider, and a Payment Gateway who authorizes payments. The payment-related interactions are imported from the Secure Electronic Transactions (SET) standard [15]. In Figure 1, each participant is shown via a separate shaded region, the graph made of dark edges denotes the *flow* of the given participant. Circular nodes represent the participant's internal business logic or policies, e.g., to decide the parameters of an out-bound message. Rectangular nodes represent external interfaces through which a participant receives messages. Thus, an ordering of dark arrows, circles, and rectangles represents the local process of the participant. When there are multiple out-edges from a node, all of them are taken concurrently. The messages are labeled with numbers to indicate a possible order in which they might occur. For example, messages 9–17 could occur in any order.

1.1.1 Shortcomings of Traditional Approaches

The above process can be captured via a traditional flow-based approach such as BPEL [2]. Such a representation would be functionally correct, but still be inadequate from the perspective of service-oriented computing (SOC). The following are the main limitations:

Lack of Semantics. Traditional approaches expose low-level interfaces, e.g., via WSDL [21], but associate no semantics with the participants' actions. This lack precludes flexible enactment (as needed to handle exceptions) as well as reliable compliance checking. For this reason, we cannot determine if a deviation from a specific sequence of steps is significant.

Lack of Reusable Components. The local processes of the partners are not reusable even though the patterns of interaction among the participants might be. Local processes are monolithic in nature, and formed by *ad hoc* intertwining of internal business logic and external interactions. Since business logic is proprietary, local processes of one partner are not usable by another. If a new customer were to participate in this SOC environment, its local process would have to be developed from scratch.

1.1.2 Contributions, Scope, and Significance

This paper develops an approach for designing processes that recognizes the fundamental interactive nature of open environments where the autonomy of the participants must be preserved. Protocols can be readily reused and combined with different policies. Commitments provide the basis for a semantics of the actions of the participants, thereby enabling the detection of violations and making the resultant protocols flexible. The significance of this work derives from the importance of processes in modern business practice. Over 100 limited business protocols have been defined [14];this approach will enable the development and usage of an ever-increasing set of protocols for critical business functions. We demonstrate the practicality of our approach by embedding it in an ontology and language for specifying protocols. Not only is this approach conducive to reuse, refinement, and aggregation, it has also been implemented in a prototype tool.

Organization

Section 2 introduces some key concepts and terminology. Section 3 describes our protocol specification language and its semantics. Section 4 discusses composite protocols and their construction and shows by an exception handling scenario, how protocol composition can be leveraged for refining protocols. Section 5 shows how augmenting policies with protocols can be used to develop processes. It also presents our prototype framework. Section 6 evaluates our approach and shows how the problems identified in Section 1.1.1 can be handled in our approach. It also compares our work with current research efforts in the area and charts out directions for future work.

2. CONCEPTS AND TERMINOLOGY

Figure 2 shows our conceptual model for a treatment of business processes based on protocols and policies. Boxed rectangles are abstract entities (interfaces), which must be combined with business policies to yield concrete entities that can be fielded in a running system (rounded rectangles). Abstract entities should be published, shared, and reused among the process developers. They correspond to service specifications in SOC terminology. We specify a business protocol using rules termed *protocol logic* that specify the interactions of the participating *roles*. Roles are abstract, and are adopted by agents to enable concrete computations.

Whereas the protocol logic specifies the protocol from the global perspective, a *role skeleton* specifies the protocol from the perspec-

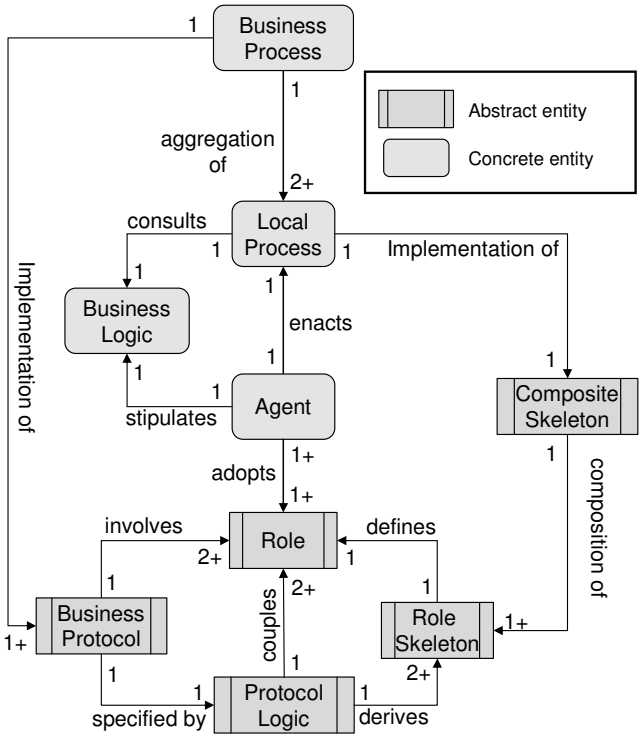

Figure 2: Conceptual model for business processes based on protocols and policies

tive of one of the participant roles. Thus, each role skeleton defines the behavior of the respective role in the given protocol.

An *agent* represents a real-world business partner with its local business logic. An agent may participate in multiple business protocols by playing a role in each of them. For example, a bookstore may play the role of a seller while interacting with customers and the role of a buyer while interacting with publishers. When an agent needs to participate in multiple protocols, a *composite skeleton* can be constructed by combining respective role skeletons according to some composition constraints. For example, in a supply chain process, a supplier would be a merchant when interacting with a retailer in a trading protocol and would be an item-sender in a shipping protocol for sending goods to the retailer. A composite skeleton for such a supplier could be composed by combining role skeletons for a trading merchant and a shipping item-sender. The resultant composite skeleton could be published and then reused for developing local processes of other suppliers.

An agent's private policies or *business logic* are described via rules. The *local process* of an agent is an executable realization of a composite skeleton obtained by integration of the protocol logic of the composite skeleton and the business logic of the agent. A *business process* is the aggregation of the local processes of all the agents participating in it. Conversely, a business process is an implementation of the constituent business protocols.

2.1 Protocols and Commitments

To enable protocols to be enacted flexibly and yet in a manner where the compliance of agents with specific protocols can be determined requires that we provide protocols a semantics that is rigorous yet not rigid. The concept of *commitments* has been proposed to capture a variety of contractual relationships, while allowing manipulations such as delegation and assignment, which are essential for open systems [17]. For example, a customer's agreement to pay the price for the item after it is delivered is a commitment that the customer has towards the merchant. Violations of commitments can be detected; in some important circumstances, violators can be penalized. Such enforceability of contracts is necessary in practical settings where the participants are autonomous and heterogeneous [19]. Since commitments might be unfamiliar to some readers, we introduce them first.

DEFINITION 1. *A commitment* $\mathsf{C}(x, y, p)$ *denotes that the agent* x *is responsible 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 can be expressed in a suitable formal language.

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 where, q is called the precondition of the commitment. For example, the conditional commitment $\mathsf{CC}(c, b, \mathit{goods}(g), \mathit{pay}(p))$ means that the customer c is committed to pay the bookstore b an amount p if the bookstore delivers the book g to the customer. When the bookstore delivers the goods, i.e., when the $\mathit{goods}(g)$ proposition holds, the conditional commitment $\mathsf{CC}(c, b, \mathit{goods}(g), \mathit{pay}(p))$ is automatically converted into the base-level commitment $\mathsf{C}(c, b, \mathit{pay}(p))$.

2.2 Commitment Operations

Commitments are created, satisfied, and transformed in certain ways. The following are the conventional operations defined on commitments [17].

1. CREATE(x, c) establishes the commitment c in the system. This can only be performed by c's debtor x.
2. CANCEL(x, c) cancels the commitment c. This can only be performed by c's debtor x. Generally, cancellation is compensated by making another commitment.
3. RELEASE(y, c) releases c's debtor x from commitment c without c being fulfilled. This only can be performed by the creditor y.
4. ASSIGN(y, z, c) replaces y with z as c's creditor.
5. DELEGATE(x, z, c) replaces x with z as the c's debtor.
6. DISCHARGE(x, c) c's debtor x fulfills the commitment.

A commitment is said to be *active* if it has been created, but not yet discharged. The transformations of commitments are given as:

$\frac{C(x,y,p) \wedge p}{discharge(x,C(x,y,p))}$, $\frac{CC(x,y,p,q) \wedge p}{create(x,C(x,y,q)) \wedge discharge(x,CC(x,y,p,q))}$, and $\frac{CC(x,y,p,q) \wedge q}{discharge(x,CC(x,y,p,q))}$.

3. PROTOCOL SPECIFICATION

A business protocol is a specification of the allowed interactions among two or more participant roles. The specification focuses on the interactions and their semantics. What does it mean to send a certain message to a business partner? What is expected of the participants wishing to comply to a business protocol? How are the protocols specified? These are the questions we address in this section.

3.1 OWL-P: OWL for Protocols

OWL-P is an ontology based on OWL for specifying protocols; it functions as a schema or language for protocols. The main computational aspects of protocols are specified using rules. We employ the Semantic Web Rule Language (SWRL) [8] for defining rules. SWRL allows us to specify implication rules over entities defined as OWL-P instances. The availability of tools such as Protégé [13] is a motivation for grounding OWL-P on OWL.

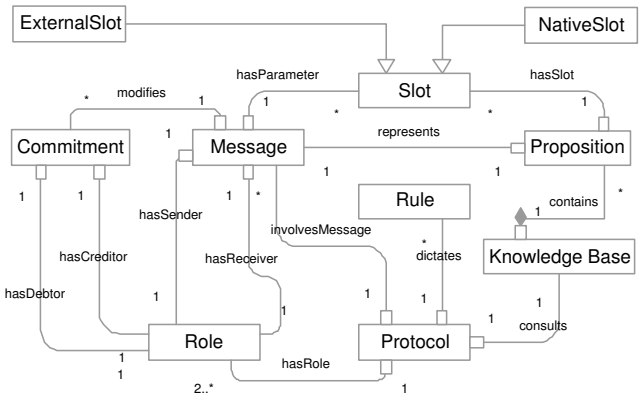

Figure 3: Basic OWL-P ontology

The important OWL-P elements and their properties are shown in Figure 3. An entity with a little rectangle represents the domain of the corresponding property. Many of the properties are self-explanatory and reflect the conceptual model introduced in Section 2.

A *Message* has a sender and a receiver, and can have several *Slots* as its parameters. We provide operational semantics for messages in terms of their effects on the commitments among the roles. Messages can be thought of as operators over commitments. Slots are analogous to data variables. A slot is said to be *defined* when it is assigned a value and it said to be *used* when its value is assigned to another slot. A slot in a protocol may be assigned a value produced by another protocol and hence be represented as an *External Slot*. An external slot is untyped until it is given the type of the external value to which it is bound. By contrast, a *Native Slot* is typed and produced by the protocol.

A *Protocol* dictates several rules and consults a *Knowledge Base*. The rules are SWRL implication rule instances. A knowledge base consists of a set of *Propositions* which can have several slots. Propositions are analogous to slotted facts in the conventional rule-based systems. A proposition in a knowledge base may correspond to a message that has been sent or received, since every message being sent or received is recorded as a proposition in the knowledge base. Propositions are also used to represent active commitments or other domain specific propositions.

Figure 4 shows a protocol for ordering goods (along with others, to which we refer later). For readability, a leading and trailing * is placed around external slot names, as in *amount* and *itemID*. The customer requests a quote for an item, to which the merchant responds by providing a quote. Here, a commitment is created providing semantics for the message. The commitment means that the merchant guarantees receipt of the item if the customer pays the quoted price. The customer can either accept the quote or reject it(not shown). Again, the semantics of acceptance is given by the creation of another commitment from the customer to pay the quoted price if it receives the requested item. Figure 5 shows the rules for the Order protocol in the "antecedents ⇒ consequents" notation.

contains(KB, startProp) ⇒ send(C, reqForQuote(?itemID))

contains(KB, reqForQuoteProp(?itemID)) ⇒
send(M, quote(?itemID, ?itemPrice)) ∧
createCommitment(M, CC(M, C, pay(?itemPrice), goods(?itemID)))

contains(KB, quoteProp(?itemID, ?itemPrice)) ⇒
send(C, acceptQuote(?itemID, ?itemPrice)) ∧
createCommitment(C, CC(C, M, goods(?itemID), pay(?itemPrice)))

Figure 5: Rules for the Order protocol

reqForQuote, quote, and acceptQuote are OWL-P message instances (individuals in OWL terminology). Corresponding proposition instances are reqForQuoteProp, quoteProp, and acceptQuoteProp. pay and goods are other propositions while itemID and itemPrice are native slots. Note that a name ?x represents the slot ?x across the entire rule-base. Readers may notice that the itemID variable in the first rule is not assigned any value by the antecedents. It means that the rule is abstract and not executable, and as we will see in later sections, it can be augmented with business logic that produces such values. OWL-P implicitly dictates that the rules having undefined native slots must be augmented with the business logic that produces such values. How do these rules define the protocol? The next section describes the operational semantics of the protocol rules. The OWL-P ontology and protocol instance examples in their RDF/XML serialization, and corresponding Protégé projects are available at http://www4.ncsu.edu/∼nvdesai/owl/.

3.2 Operational Semantics

Protocols are specified from the global perspective with an assumption of an abstract global knowledge base. In later sections, we will show how the abstraction of a global knowledge base maps to the perspectives of the participants having their local knowledge bases. Rules are assumed to be forward-chained. OWL-P defines several property predicates with operational semantics.

Table 1 lists the semantics for such property predicates of OWL-P. A proposition cannot be retracted from a knowledge base. A discharge commitment operation is automatic as stated in the Section 2.2 and hence not mentioned here. In the forthcoming examples, we may omit the OWL-P properties, e.g., contains, send, createCommitment when the meaning is clear.

4. COMPOSITE PROTOCOLS

The previous section describes how to specify individual protocols. However, real-world business process would typically involve multiple protocols and multiple roles. Now we show how protocols can be composed.

Conceptually, each component protocol achieves a business goal. Thus, several such protocols composed together would achieve the goals of the larger business process. Another motivation is to be able to refine or amend protocols with additional rules. Observe that the set of additional rules is a protocol itself and hence the problem boils down to the problem of combining the original protocol with the amendments. Also, the ability to compose protocols would allow significant reuse of published protocols. How can we construct such composite protocols? How do they facilitate reusability? How do they allow refinements of protocols? This section answers the above questions.

4.1 Construction of Composite Protocols

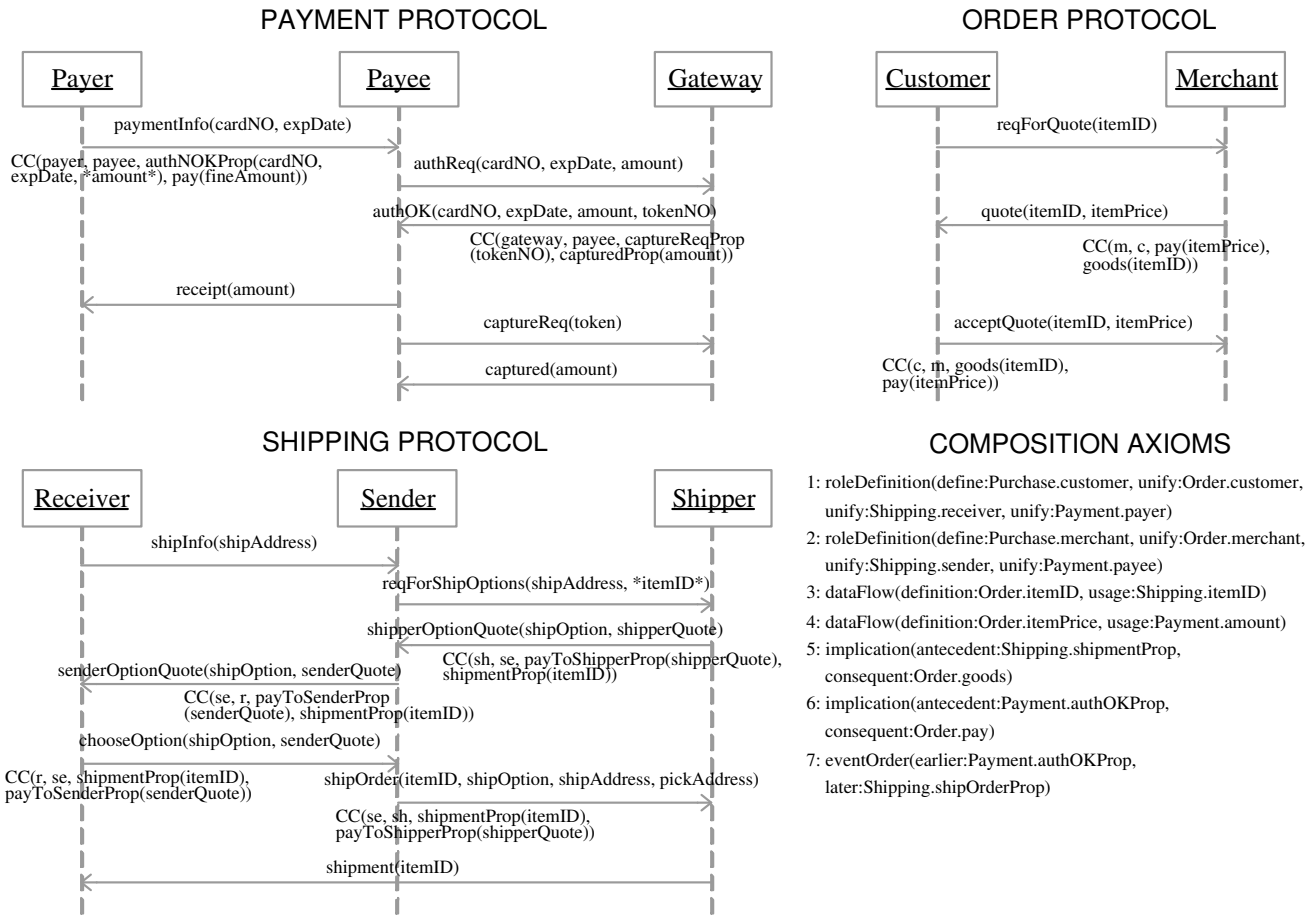

Figure 4: Example: Order, Shipping, and Payment protocols and their composition

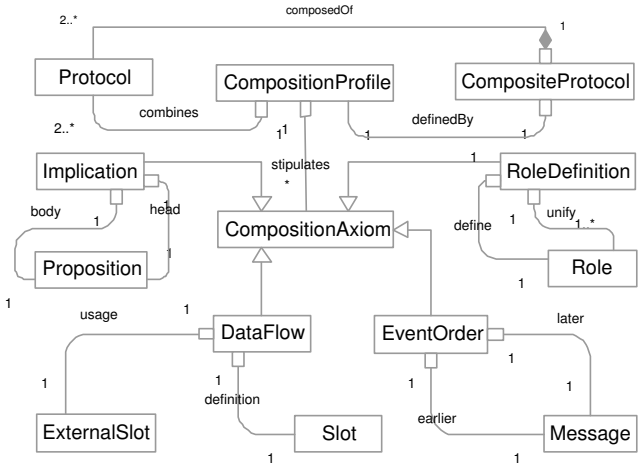

Figure 6: OWL-P composition classes and properties

Figure 6 describes the OWL-P classes and properties that deal with the problem of protocol composition. A *Composite Protocol* is an aggregation of component protocols and is defined by a *Composition Profile*. A composition profile describes the combination of two or more protocols by stipulating several *Composition Axioms*. Composition axioms define relationships among the protocols being combined. The operational semantics of an axiom specifies the way in which the relationships affect the composite protocol. Figure 4 depicts an Order protocol, a Shipping protocol, a Payment protocol, and a set of composition axiom instances stating the relationships among them.

A *Role Definition* axiom states which of the roles in the component protocols are played by the same agent, and defines the name of the unified (coalesced) role in the composite protocol. In the example, the first axiom states that the roles of a customer in Order, a payer in Payment, and a receiver in Shipping protocol are played by an agent who will play the role of a customer in the Purchase protocol. Similarly, the second axiom defines the merchant role in the Purchase protocol. The ununified roles must be played by different agents in the composite protocol. The instances of the roles being unified are discarded from the composite OWL-P and an instance is added for the newly defined role. When the cardinality of the *unify* property is one, the role being unified is just renamed as the defined role in the composite protocol.

A *Data Flow* axiom states a data-flow dependency among the protocols. A component protocol might be using a slot defined by another component protocol, possibly with a different name. A native slot cannot use a value defined by another protocol as it is always defined in the protocol in which it is used and a slot can be defined only once. Hence, the range of the *usage* property must be an external slot. In the example, the fourth axiom states that the slot

Predicate	Domain	Range	Context	Meaning
contains	KnowledgeBase	Proposition	Body	Proposition $\in$ KnowledgeBase ?
assert	Proposition	KnowledgeBase	Head	KnowledgeBase $\leftarrow$ KnowledgeBase $\cup$ Proposition
send	Role	Message	Head	Asynchronous send to the receiver
				assert(KnowledgeBase, MessageProp)
receive	Role	Message	Head	Asynchronous receive from the sender
				assert(KnowledgeBase, MessageProp)
createCommitment	Role	Commitment	Head	assert(Knowledgebase, CommitmentProp)

Table 1: Operational semantics of protocol rules

amount in the Payment protocol gets its value from the slot itemPrice in the Order protocol. Such a dependency exerts an ordering among the rule defining the slot and all the rules using it. None of the the rules using the slot can fire before the slot is assigned a value by the defining rule. Thus, the operational semantics of the axiom is given as:

```
def   : reqForQuotePrope(?itemID) ⇒ quote(?itemID,?itemPrice) ∧
        CC(M,C,pay(?itemPrice),goods(?itemID))
use0  : startProp ⇒ paymentInfo(?cardNO,?expDate) ∧
        CC(Pr,Pe,authNOKProp(···),pay(?fineAmount))
usei  : ···
usen  : captureReqProp(?token) ⇒ captured(?amount)
axiom : dataFlow(definition:Order.itemPrice, usage:Payment.amount)
```

```
use0  : startProp ∧ quoteProp(?itemID,?itemPrice) ⇒
        paymentInfo(?cardNO,?expDate) ∧
        CC(Pr,Pe,authNOKProp(···),pay(?fineAmount))
usei  : ···
usen  : captureReqProp(?token) ∧ quoteProp(?itemID,?itemPrice) ⇒
        captured(?itemPrice)
```

The order in which the rules using the slot fire, cannot be determined. Also, the dependencies among the rules using the slot may change, thus changing the order of their firing. Hence, making all rules that use a slot dependent on the rule that defines the slot is safer. Also, the usage slot takes the name of the defining slot and the definition of external slot is discarded from composite OWL-P.

An *Implication* axiom states that an assertion of proposition X in a component protocol implies an assertion of proposition Y in another component protocol. For example, the sixth axiom states that an assertion of authOKProp in the Payment protocol means an assertion of pay in the Order protocol. This can be easily achieved by adding an implication rule to the composite rulebase. Hence, the operational semantics of the axiom is:

```
axiom : implication(body:Payment.authOKProp, head:Order.pay)
```

```
rule  : authOKProp(?cardNO,?expDate,?amount,?tokenNO) ⇒
        pay(?itemPrice)
```

Unlike the DataFlow axiom, an *EventOrder* axiom explicitly specifies an ordering among the messages of the component protocols. For example, the seventh axiom states that an authOK message from the payment gateway must be received before a shipOrder message is sent to the shipper. This can be achieved by making the rule for the later event depend on the rule for the earlier event. Hence, the operational semantics of the axiom is:

```
earlier : authReqProp(?cardNO,···) ⇒
          authOK(?cardNO,?expDate,?amount,?tokenNO) ∧
          CC(G,Pe,···)
later   : chooseOptionProp(?shipOption,?senderQuote) ⇒
          shipOrder(?itemID,?shipOption,?shipAddress,?pickAddress) ∧
          CC(Se,Sh,···)
axiom   : eventOrder(earlier:Payment.authOK, later:Shipping.shipOrder)
```

```
later   : chooseOptionProp(?shipOption,?senderQuote) ∧
          authOKProp(?cardNO,?expDate,?amount,?tokenNO)⇒
          shipOrder(?itemID,?shipOption,?shipAddress,?pickAddress) ∧
          CC(Se,Sh,···)
```

Composition axioms have to be specified by a designer. There might be several ways of composing the component protocols yielding different composite protocols. As a special case, if the component protocols are completely independent of each other, no axioms need be specified and their OWL-P specifications can be simply aggregated yielding the OWL-P specification of the composite protocol. If deemed necessary, more subclasses of composition axiom can be defined as long as their properties and operational semantics are defined. A composite protocol exposes its compositionProfile and possesses all the properties of the component protocols. Hence, a composite protocol itself can be a component protocol in some other composition profile instance.

How can we determine whether additional component protocols are needed? To answer this question, we define *closed* and *open* protocols.

DEFINITION 2. *A protocol is closed if none of the slots of the protocol are external slots, and all of the conditions of the commitments either created or transformed by the protocol can be discharged by propositions asserted in the protocol.*

DEFINITION 3. *A protocol is open if it is not closed.*

A designer's goal is to obtain a closed protocol by repeated applications of composition. Observe that in Figure 4, the Order protocol is open as its rules do not assert propositions pay and goods. The Payment protocol is open as its rules do not assert the proposition pay(fineAmount) and amount is an external slot. The Shipping protocol is open as its rules do not assert payToSenderProp and payToShipperProp, and itemID is an external slot. The composite Purchase protocol is also open as its rules do not assert the propositions pay(fineAmount), payToSenderProp, and payToShipperProp. A designer would choose protocols that assert these missing propositions and combine them with the Purchase protocol to obtain a closed composite protocol.

4.2 Refinement by Composition

Business protocols evolve continually as new requirements and new features routinely arise. Therefore, the ability to systemati-

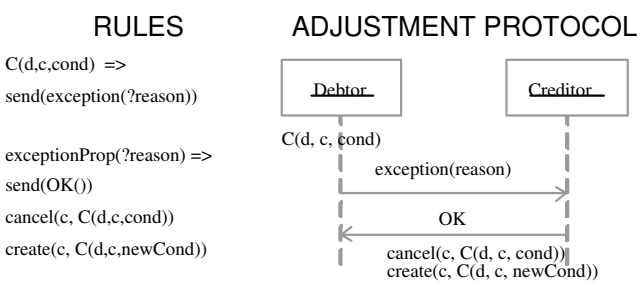

Figure 7: Handling exceptions by composition

cally refine protocols is valuable. In the composite Purchase protocol, consider a situation in which the customer has already paid the merchant for the goods and hence the commitment C(m, c, goods(itemID) is active. However, while trying to order the shipment, if a fire destroys the merchant's warehouse, the merchant will not be able to honor its commitment to ship the item. How can such exceptions be handled? The protocol could detect the violation due to an unfulfilled commitment, and the merchant could be held legally responsible. A more flexible solution would be to allow the merchant to refund money and release it from the commitment, provided the customer agrees to it. We can achieve this flexibility by combining the purchase protocol with the adjustment protocol shown in the Figure 7 using the following composition axioms:

```
1: roleDefinition(define:New.customer, unify:Purchase.customer,
   unify:Adjustment.creditor)
2: roleDefinition(define:New.merchant, unify:Purchase.merchant,
   unify:Adjustment.debtor)
3: Implication(body:Purchase.C(m,c,goods(itemID)),
   head:Cancel.C(d,c,cond))
4: Implication(body:Cancel.C(d,c,newCond),
   head:Purchase.C(d,c,refund))
```

The Adjustment protocol allows cancellation of the merchant's commitment if the customer deems it reasonable. The semantics of OK message specifies the creation of a new commitment for refunding the money. The proposition refund is an external slot in the New composite protocol, and must be composed with a protocol that can make a refund, to yield a closed protocol. The rule sending the exception message should be augmented with the merchant's business logic such that this rule is enabled only when the exception arises. Similarly, the rule for sending OK should be augmented with the customer's business logic such that it enables the rule only when it deems the reason valid.

Similar protocols for assigning, delegating, and releasing commitments can be defined. Adding new functionalities would involve composition of a set of rules for the new requirements with the original protocol.

5. PROCESSES

As described in Section 2, a process is an aggregation of the local processes of participating agents. However, an OWL-P specification of a protocol is a model of the interaction from a global perspective. To construct the local process of a participant, we need to derive the participant's view of the protocol, called its *role skeleton*. Section 5.1 describes the generation of role skeletons from an OWL-P specification.

5.1 Role Skeletons

Algorithm 1: deriveSkeleton(P, ρ): Generate the skeleton for a role

```
1  getPertinentRules(P, ρ) ;
2  foreach rule r in ρ.rules do
3      foreach proposition p in r.body do
4          if p is of the form CONTAINS(P.kb, atom) then
5              Replace atom with replace(atom, ρ, P) ;
6  Procedure getPertinentRules(P, ρ): Get all rules in
     which ρ sends or receives m
7  ρ.rules ← {} ;
8  foreach rule r in P.rules do
9      foreach proposition p in r.head do
10         if p is of the form SEND(ρ', ρ, m) then
11             Replace p with RECEIVE(ρ', ρ, m);
12             ρ.Rules ← ρ.rules ∪r;
13         if p is of the form SEND(ρ, ρ', m) then
14             ρ.Rules ← ρ.rules ∪r;
15 Procedure replace(P, ρ, atom): If atom isn't asserted in
     ρ.rules replace it by something that is
16 repl ← {};
17 if atom is asserted by a rule r_l in ρ.rules then
18     repl ← repl ∪ atom ;
19     return repl;
20 if atom is asserted by a rule r_g in P.rules then
21     foreach proposition p in r_g.body do
22         if p is of the form CONTAINS(P.kb, a) then
23             repl ← repl ∪ replace(a, ρ, P);
24     return repl
```

A role skeleton is one role's view of the protocol. Here, we present an algorithm for generating role skeletons from an OWL-P protocol specification.

OWL-P describes the protocol from a global perspective where the propositions are added to a global state and there are no distributed sites. The role skeletons describe the protocol from the perspective of the corresponding role. As in all distributed systems, the state of a protocol as seen by a role is changed only when a message is sent or received by that role. This observation forms the basis of our role skeleton-generation algorithm presented in Algorithm 1.

The algorithm invokes the procedure getPertinentRules(P, ρ) in line 1, which gathers all rules from the OWL-P specification for protocol P that have, the role ρ receiving or sending a message. The propositions asserted by this set of rules are the only propositions that ρ will *see*. We denote this set of rules by ρ.rules and the set of propositions by $Prop$. This procedure is defined from line 6 to line 14. Next, the algorithm invokes the procedure replace($P, \rho,$ atom), defined from line 15 to line 24. If a proposition atom $\in Prop$ triggers a rule $r \in \rho$.rules but atom is not asserted by any rules in ρ.rules, it means that atom was not seen by ρ. This procedure replaces atom with the last proposition that ρ did see, i.e., the proposition atom$'$ that was asserted in ρ.rules and leads to the atom being asserted in the OWL-P specification.

In the following, we show a rule in the Shipping protocol in Figure 4, and the same rule in the generated skeleton of the receiver. As the receiver would not be aware of the previous exchanges between the sender and the shipper, the antecedent of the rule for receiving senderOptionQuote should be adjusted as shown below.

P : shipperOptionQuoteProp(?shipOption,?shipperQuote) ⇒
senderOptionQuote(?shipOption,?senderQuote) ∧
CC(Se,Re,payToSenderProp(?senderQuote),shipmentProp(?itemID))

Re : **shipInfoProp(?shipAddress)** ⇒
receive(senderOptionQuote(?shipOption,?senderQuote)) ∧
CC(Se,Re,payToSenderProp(?senderQuote),shipmentProp(?itemID))

We observe that the composition axioms defined in Section 4 are also applicable for combining role skeletons to construct a composite skeleton.

5.2 Policies

Generation of a role skeleton may not be enough to obtain a local process of a participant. As we mentioned earlier, some of the rules of the protocols may be abstract, meaning that values of some of the native slots in the rule must be produced by the role's business logic. Hence, a role skeleton must be augmented with business logic to obtain a local process. How can we determine whether an augmented role skeleton is a local process? To answer this question, we first define *concrete* and *abstract* role skeletons, and a *local process*. A role skeleton is *concrete* if all of its native slots are defined. A role skeleton is *abstract* if it is not concrete.

DEFINITION 4. *A local process is a role skeleton that is concrete and derived from a closed protocol.*

contains(KB, startProp) ⇒ receive(C, reqForQuote(?itemID))

contains(KB,reqForQuoteProp(?itemID)) ∧ **quotePolicy(?itemPrice)** ⇒
send(M, quote(?itemID, ?itemPrice)) ∧
createCommitment(M, CC(M, C, pay(?itemPrice), goods(?itemID)))

contains(KB, quoteProp(?itemID, ?itemPrice)) ⇒
receive(C, acceptQuote(?itemID, ?itemPrice)) ∧
createCommitment(C, CC(C, M, goods(?itemID), pay(?itemPrice)))

contains(KB, reqForQuoteProp(?itemID)) ⇒
call(policyDecider, quotePolicy(?itemID))

Figure 8: Augmented role skeleton for the merchant role in the Order protocol

We propose that the business logic be specified in terms of the local policy rules of the agents. Figure 8 shows the skeleton of the merchant role in the Order protocol augmented with the policy rules of the merchant agent. The last rule is the policy rule which calls a business logic operation to decide how much to quote. The operation would assert the quotePolicy proposition and that would activate the second protocol rule. Observe that this pattern of augmenting policy rules is general and will be applied to the rules where the agent has to make a decision and respond. It would also assign a value to native slots that are not defined.

Figure 9 shows the interplay between the protocol rules and the policy rules of an agent. The business logic could involve processing such as looking up a legacy database or waiting for human input. When a message is received by the messaging interface, it is checked against the protocol rules to see whether it can be accepted, according to the protocol, at this state. For example, the merchant will not receive the acceptQuote message unless it knows that a quote was sent before as shown in the third rule in Figure 8.

5.3 Usage

Figure 10 summarizes our methodology with a scenario involving a customer interested in purchasing goods online. Software

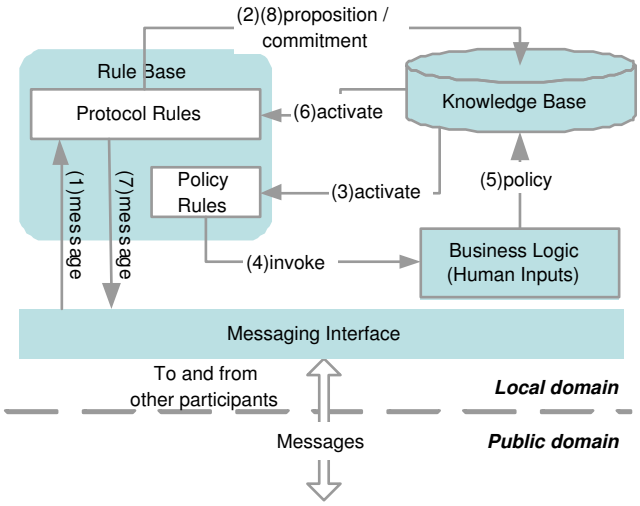

Figure 9: Agent architecture: protocol and policy interplay

designers design protocols and register them with protocol repositories. They may also construct composite protocols and reuse the existing component protocols from the repository. A merchant that wishes to sell goods online looks up the repository for a suitable Purchase protocol. It generates the skeleton for the merchant role, augments it with its local policies, and deploys the result as a service. The service profile for this service would contain an OWL-P description of the Purchase protocol. The service can be registered with the a UDDI registry. If a customer wishes to buy goods online, it searches the UDDI registry, finds the merchant, and acquires the OWL-P skeleton for the customer role from the merchant. The customer enacts its local process by augmenting the skeletons with its local policies and starts interacting with the merchant.

5.4 Prototype Framework

Figure 9 shows a prototype enactment framework for an agent. We employ JADE (Java Agent Development Framework) [9] as the messaging interface and Jess [6] as our rule system for both the protocol rules and the policy rules. Jess offers seamless interoperability with Java objects. Jess rules can be fired from Java and Java objects can be accessed from Jess. Also, Jess knowledge base can be accessed from Java. These features enable us to easily augment the protocol rules with the policy rules. The rules in OWL-P can be automatically translated into Jess rules and Java classes. The policy rules can either be directly specified in Jess and augmented with the protocol rules, or they can also be specified in SWRL and then translated into Jess rules.

6. DISCUSSION

Developing business processes for open systems poses significant challenges because of the complexity of the interactions and the autonomy of the partners. Traditional technologies such as workflow systems lack the flexibility and agility that modern business processes need. It is natural that Semantic Web technologies be applied in business process management to improve flexibility and exception handling. Our work fits into this emerging direction.

Interactions between services are more naturally organized as protocols than as procedural scripts, such as those based on WSDL [21] operations. Protocols specify the *what* of an interaction, and leave the *how* to the individual partners, thus supporting their autonomy. This yields flexibility in implementation, particularly in

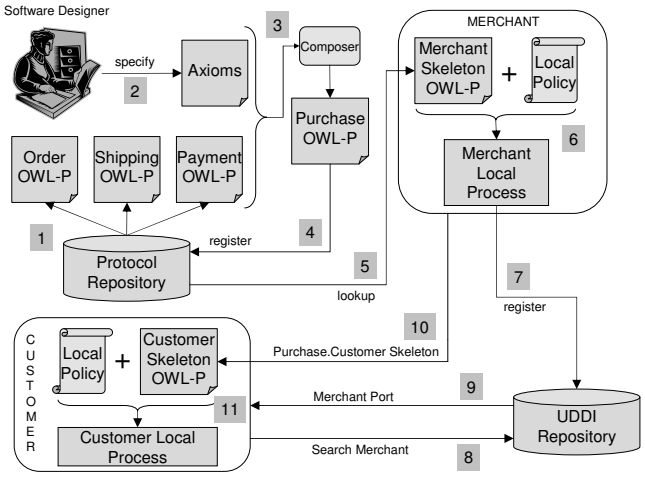

Figure 10: Usage scenario

dealing with exceptions. The following outlines our main contributions and contrasts them with the literature.

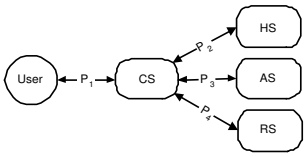

Figure 11: Trip Panning

Composition. Composition can take place at two levels. At one level, a designer can compose published business protocols to create a new service. For example, in designing an auction service, a designer may compose the specifications of a bidding protocol with a payment protocol such as SET (Secure Electronic Transaction) [15]. Thus protocol specifications can be reused saving the designer considerable effort. At another level, existing deployed services can be composed to obtain a new service composition. The trip planning scenario of Figure 11 shows service composition. The user wants to book a flight to some city, reserve a hotel room there, and also rent a car. To do so, the user interacts with the composed service (*CS*) which in turn interacts with the airline service (*AS*), the hotel service *HS*, and the car rental service *RS*, thus simplifying the user's interface.

Our approach supports both protocol composition and service composition. Protocols may not only be composed, they may also be systematically refined as shown in Section 4.1. Service composition is achieved by way of protocol composition; by composing the protocol specifications of the partner services.

BPEL [2] is a process language designed to specify the static composition of Web services. However, it mixes protocols and policies, which makes specifications unsuitable for reuse. The Semantic Web Services Initiative has produced OWL-S [4], which includes a process model for Web services. The model is constructed using procedural constructs such as sequence, choice, and so on. An important feature of OWL-S is the use of semantic annotations to facilitate dynamic composition. A composed service is produced at runtime based on the user's constraints. While dynamic service composition has some advantages, it assumes a perfect markup of the services being composed. Dynamic composition in OWL-S involves ontological matching between inputs and outputs. Such a matching might be difficult to obtain automatically given the heterogeneity of the web. For this reason, we do not emphasize dynamic service composition. Our goal is to provide a human designer with tools to facilitate service composition. Unlike BPEL, which specifies the internal orchestration of services, WSCI [20] specifies the conversational behavior of a service using control flow constructs. However, these specifications lack a semantics, which makes them difficult to compose and reuse.

Several other approaches aim to solve the service composition problem by emphasizing formal specifications to achieve verifiability. Solanki *et al.*[18] employ interval temporal logic to specify and verify ongoing behavior of a composed service. Their use of "assumption" and "commitment" (different meaning than here) assertions allows better compositionality. Gerede *et al.*[3] treat services as activity-based finite automata to study the decidability of composability and existence of a lookahead delegator given a set of existing services. However, these approaches consider neither the autonomy of the partners, nor the flexibility of composition.

Software Engineering. The preceding sections presented a methodology for designing business processes. We start with protocol specifications, extract role skeletons from them, and augment role skeletons with business policies to come up with a business process. The protocols and policies are separate rule-bases. However, there is a clean, uniform interface between the two; they are tied together through the use of policy variables in the protocol rules. This greatly enhances the modularity of the software. New policies can be plugged in with only local changes.

In addition, our methodology advocates and enables reuse of protocols as building blocks of business processes. Protocols can not only be composed, they can also be systematically refined to yield more robust protocols. Mallya and Singh [10] treat these concepts formally. The MIT Process Handbook [11], in a similar vein, catalogues different kinds of business processes in a hierarchy. For example, *sell* is a generic business process. It can be qualified by *sell what*, *sell to who*, and so on. Our notion of a protocol hierarchy bears similarity with the Handbook. Our effort is focused on providing formal semantics to the hierarchy. RosettaNet [14] bears similarity to our approach in that it centers around publishing protocols and designing the business processes around them. However, it is currently limited to two-party request-response interactions called Partner Interface Processes (PIPs) and more importantly, PIPs lacks a formal semantics.

Exception-handling is critical to building robust business processes. Commitments play an important role in exception-handling. The violation of a commitment represents an exception. Giving protocols a commitment-based semantics helps a designer anticipate such exceptions and handle them in a context-sensitive manner. This usually leads to refinements in the protocol, yielding more robust protocols.

Baïna *et al.* [1] advocate a model-driven Web service development approach to ensure compliance between a service's implementation and its external protocol specifications. This is a valuable approach, which can be extended for protocol implementations as well.

Enactment Flexibility. Our work with commitments draws from work in multiagent systems. A commitment exists in some particular context [16]. The definition of commitment given in Section 2.1 represents a simplification wherein the context is omitted. A context represents a social institution that governs some aspects of handling of the commitment, particularly those related to excep-

tion handling. For example, commitments in a Purchase protocol are created in the context of an institution (fictional here) such as the Federal Trading Commission. The commission stipulates that it is necessary for a merchant to refund a customer in case of a failure to deliver goods. It also states the cases when a merchant can delegate the delivery of goods to another merchant or when a party can cancel its commitments. Though these *norms* are public, they are more in the nature of policies than protocols. These norms have the potential to change the interaction specified by the protocol. However, encoding them into the Purchase protocol makes the protocol unwieldy and unsuitable for reuse in other contexts. Plus, agents can be imagined as entering and leaving contexts at runtime, and thereby adopting and dropping the associated norms, and changing the interaction dynamically.

When an agent moves into a particular context, we say it is *scoped* into the context. More importantly for us, its protocol is scoped into the context. Scoping enables a protocol and context to be specified independently of each other, yet it dynamically changes the interaction, making the interaction more robust. Although agents in a business process will generally be static, that is, they will not change their contexts dynamically, scoping has relevance to business processes. A designer can use a tool that scopes a protocol into a context which captures the characteristics of the environment into which the service will be deployed, and produce a new protocol with better exception-handling characteristics. We are currently working on the formal development of such techniques.

The representation of business contracts is an interesting area of research. Grosof and Poon [7] represent agent contracts in OWL and RuleML. They develop an ontology for processes and contracts. Davulcu *et al.* [5] develop a logic for specifying contracts in Web services. Commitment-based protocols inherently specify part of the contractual arrangements. Commitments are, in principle, enforceable and partners can be held responsible for violation of commitments. It is reasonable to expect that not all parts of a complicated contract will be reflected in a protocol. A contract is more like the context in which the protocol is executed. Our work with scoping will help in dealing with contracts.

7. REFERENCES

[1] K. Baïna, B. Benatallah, F. Casati, and F. Toumani. Model-driven web service development. In *Proceedings of Advanced Information Systems Engineering: 16th International Conference, CAiSE*, June, 2004.

[2] BPEL. Business process execution language for web services, version 1.1, May 2003. www-106.ibm.com/developerworks/webservices/library/ws-bpel.

[3] Çağdaş Evren Gerede, R. Hull, O. Ibarra, and J. Su. Automated composition of e-services: Lookaheads. In *Proceedings of the International Conference on Service Oriented Computing*, 2004.

[4] DAML-S. DAML-S: Web service description for the semantic Web. In *Proceedings of the 1st International Semantic Web Conference (ISWC)*, July 2002. Authored by the DAML Services Coalition, which consists of (alphabetically) Anupriya Ankolekar, Mark Burstein, Jerry R. Hobbs, Ora Lassila, David L. Martin, Drew McDermott, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terry R. Payne and Katia Sycara.

[5] H. Davulcu, M. Kifer, and I. Ramakrishnan. CTR-S: A logic for specifying contracts in semantic web services. In *Proceedings of the 13th International World Wide Web Conference*, May 2004.

[6] E. J. Friedman-Hill. Jess, the Java expert system shell, 1997. herzberg.ca.sandia.gov/jess.

[7] B. N. Grosof and T. C. Poon. SweetDeal: Representing agent contracts with exceptions using XML rules, ontologies, and process descriptions. In *Proceedings 12th International Conference on the World Wide Web*, 2003.

[8] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A semantic web rule language combining OWL and RuleML, May, 2004 (W3C Submission). http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/.

[9] JADE. Java agent development framework, 2004. http://jade.tilab.com/.

[10] A. U. Mallya and M. P. Singh. A semantic approach for designing commitment protocols. In *Proceedings of the AAMAS-04 Workshop on Agent Communication*, July 2004. To appear.

[11] T. W. Malone, K. Crowston, and G. A. Herman, editors. *Organizing Business Knowledge: The MIT Process Handbook*. MIT Press, Cambridge, MA, 2003.

[12] OWL. Web ontology language, Feb 2004. http://www.w3.org/TR/2004/REC-owl-features-20040210/.

[13] Protégé. The protégé ontology editor and knowledge acquisition system, 2004. http://protege.stanford.edu/.

[14] RosettaNet. Home page, 1998. www.rosettanet.org.

[15] SET. Secure electronic transactions (SET) specifications, 2003. http://www.setco.org/ set_ specifications.html.

[16] M. P. Singh. Multiagent systems as spheres of commitment. In *Proceedings of the International Conference on Multiagent Systems (ICMAS) Workshop on Norms, Obligations, and Conventions*, Dec. 1996.

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

[18] M. Solanki, A. Cau, and H. Zedan. Augmenting semantic web service descriptions with compositional specification. In *Proceedings of the International World Wide Web Conference*, pages 544–552, 2004.

[19] M. Venkatraman and M. P. Singh. Verifying compliance with commitment protocols: Enabling open Web-based multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 2(3):217–236, Sept. 1999.

[20] WSCI. Web service choreography interface 1.0, July 2002. wwws.sun.com/ software/ xml/ developers/ wsci/ wsci-spec-10.pdf.

[21] WSDL. Web Services Description Language, 2002. http://www.w3.org/TR/wsdl.