

Interaction Protocols as Design Abstractions for Business Processes

Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh, *Senior Member, IEEE*

Abstract—Business process modeling and enactment are notoriously complex, especially in open settings, where business partners 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. We propose (business) protocols as components for developing business processes. A protocol is an abstract, modular, publishable specification of an interaction among different partner roles. When instantiated with the participants’ internal policies, protocols yield concrete business processes. Protocols are reusable and refinable, thus simplifying business process design. We show how protocols and their composition are theoretically founded in the π -calculus.

Index Terms—Multiagent systems, Software reuse, Interaction-Based modeling, Software design methodologies, rule-based processing, π -calculus

I. INTRODUCTION

BUSINESS processes in open settings typically involve complex interactions among autonomous and heterogeneous *business partners*. Conventionally, business processes are modeled as centralized flows, specifying exact steps for each participant. However, because of the exceptions and opportunities that arise in open environments, business relationships cannot be preconfigured to full detail. Flow-based models are inherently ill-suited to development and maintenance in the face of evolving requirements. Also, flows are not amenable to reuse via refinement or aggregation. Further, instead of treating all participants equally, conventional models classify the participants as clients and servers, thereby compromising their autonomy.

We apply multiagent systems to model business interactions. Agents mirror the autonomy and heterogeneity of real-world businesses and support rich interaction models. This paper describes a novel, agent-based approach for business process modeling and enactment. The key idea is to capture meaningful interactions as protocols. Protocols involve two or more roles and address specific purposes such as ordering, payment, shipping, and so on. Protocols emphasize the essence of the interactions and omit local details. Such abstract protocol specifications are publishable as components and reusable in different settings.

Crucially, we give protocols a semantics in terms of *commitments*. Commitments among roles provide a basis for modeling

the state of the interaction, thus allowing a variety of possible executions depending on the specific circumstances of the parties involved. Thus, commitments help capture the essence of the interaction supported by a protocol.

A business process reflects a composition of a set of protocols, to be enacted by agents representing real-world partners. Whereas protocols specify the interaction from a global perspective, (*role*) *skeletons* specify the interaction from the perspective of a particular role. During enactment of a process, each agent adopts a role in the corresponding composite protocol, setting its role skeleton accordingly, and integrates the skeleton with its *policies*. A policy is a (typically, private) description of an agent’s business logic that controls its participation in a process. For example, a protocol may allow a role to choose among multiple actions; the agent playing that role would select an action based on its policy. Policies help determine the parameters of the messages being sent, typically based on the parameters of the messages previously received.

Our theoretical contributions are developing the concept of commitment protocols and applying the π -calculus to formalize certain aspects of protocols and policies. We show how to formally construct composite protocols and derive processes by integrating protocols and policies. The formalization enables us to reason about properties of protocols such as their equivalence, the correctness of protocol compositions, and completeness of a composite protocol with respect to its components.

Our practical contributions include a language and tools for protocols, called OWL-P. OWL-P, captured as an ontology expressed in the Web Ontology Language (OWL) [1], provides the primitives to specify protocols, such as roles, the messages exchanged between them, and declarative rules describing the effects of messages in terms of commitments. Our approach includes (1) specifying protocols, (2) composing protocols, (3) generating role skeletons from a protocol, (4) compiling protocols into executable rules, (5) instantiating protocols, and (6) enacting processes. This paper focuses on the first four steps.

The key benefits of this approach are (1) a separation of concerns between interaction (protocols) and local decision making (policies); (2) reusability of protocols across processes; (3) evolution and refinement of processes by protocol composition; and (4) flexible process enactment that respects local policies while adapting continually.

This research was partially supported by NSF grant DST-0139037 and DARPA contract F30603-00-C-0178.

The authors are with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206, USA.

E-mail: {nvdesai,aumallya,akchopra,singh}@ncsu.edu

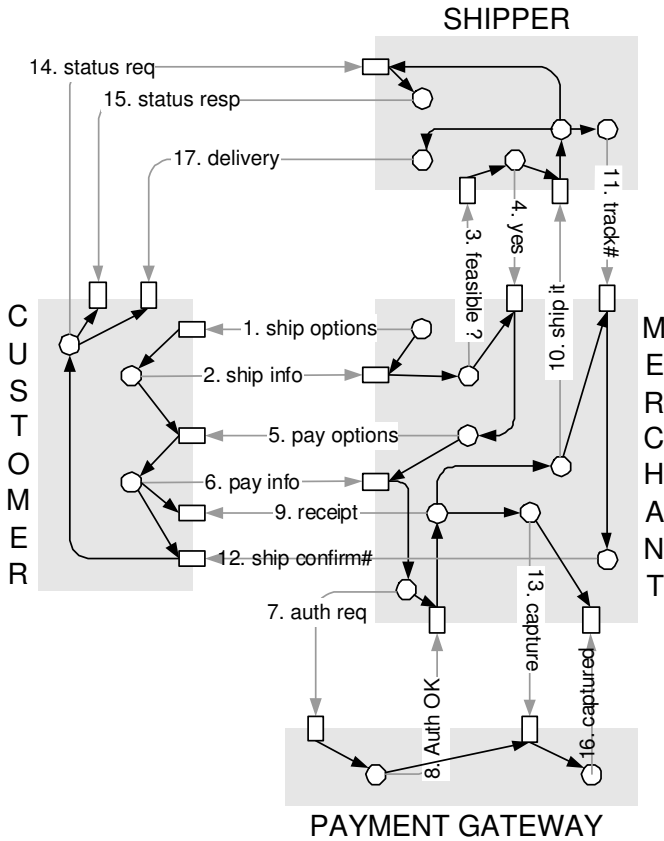


Fig. 1. A purchasing process

A. Case Study

Let's consider the common business process involving a Customer who wants to procure items, a Merchant who supplies items, a Shipper who is a logistics provider, and a Payment Gateway who authorizes payments. The payment-related interactions are based on the Secure Electronic Transactions (SET) standard [2]. Figure 1 depicts a variant where the items and price have been agreed upon. Each participant is shown as a separate shaded region, the graph within it denoting its *local process*. Circular nodes represent a participant's internal business logic or policies, e.g., to decide the parameters of an outbound message. Rectangular nodes represent external interfaces through which a participant receives messages. All out-edges from a node are taken concurrently. The messages are labeled with numbers to indicate a possible order in which they might occur. For example, after message 9, the Merchant could send message 10 and message 13 in any order. Just one possible scenario is shown; a realistic process would involve multiple scenarios.

B. Shortcomings of Traditional Approaches

The above process can be captured via a traditional flow-based approach such as BPEL [3]. Such a representation would be functionally correct, but inadequate from the perspective of software engineering in open environments.

Lack of Reusable Components Local processes are monolithic in nature, and formed by *ad hoc* intertwining of internal business logic and external interactions. Since

the business logic is proprietary, the local processes of one partner are not usable by another. For instance, if a new customer were to participate in this market, its local process would have to be developed from scratch.

Lack of Semantics Traditional approaches expose low-level interfaces, but associate no semantics with the participants' actions. This precludes flexible enactment (as needed to handle exceptions) as well as compliance checking. Without semantics, we cannot determine if a deviation from an expected sequence of steps is significant.

Inadequate Evolvability Suppose the merchant wishes to change the way it interacts with its customers, maybe because of new service features. Say the goods can be shipped before the payment is received. Due to an update in the merchant's process, the customer agent may no longer be able to interact correctly. It is not clear what updates must be made to the process and where.

Organization

Section II introduces key concepts and terminology. Section III describes our protocol specification language and its semantics. Section IV discusses protocol composition and demonstrates its applicability in exception handling. Section V shows how augmenting policies with protocols can be used to develop processes. Section VI presents a π -calculus formalization. Section VII discusses the relevant research. Section VIII outlines directions for future research.

II. CONCEPTS AND TERMINOLOGY

Key intuitions underlying protocols and processes and a deeper background were introduced in [4]. A finite automata representation of protocols and their grounding in BPEL [3] were presented in [5]. A state-based model of commitment protocols and a protocol algebra for refinement and aggregation were presented in [6].

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 can be published and reused. They correspond to service specifications in service-oriented computing (SOC). We specify the *protocol logic* via rules that describe interactions among the participating *roles*. Roles are abstract, and are adopted by agents to enable concrete computations.

Whereas a protocol specifies interactions from the global perspective, a skeleton specifies interactions from a participant's perspective. To participate in multiple protocols, an agent would adopt a role in each of them. A *composite skeleton* can be constructed by combining the adopted 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 an item-sender in a shipping protocol for sending goods to the retailer. A skeleton for such a supplier would combine the skeletons for a trading merchant and a shipping item-sender.

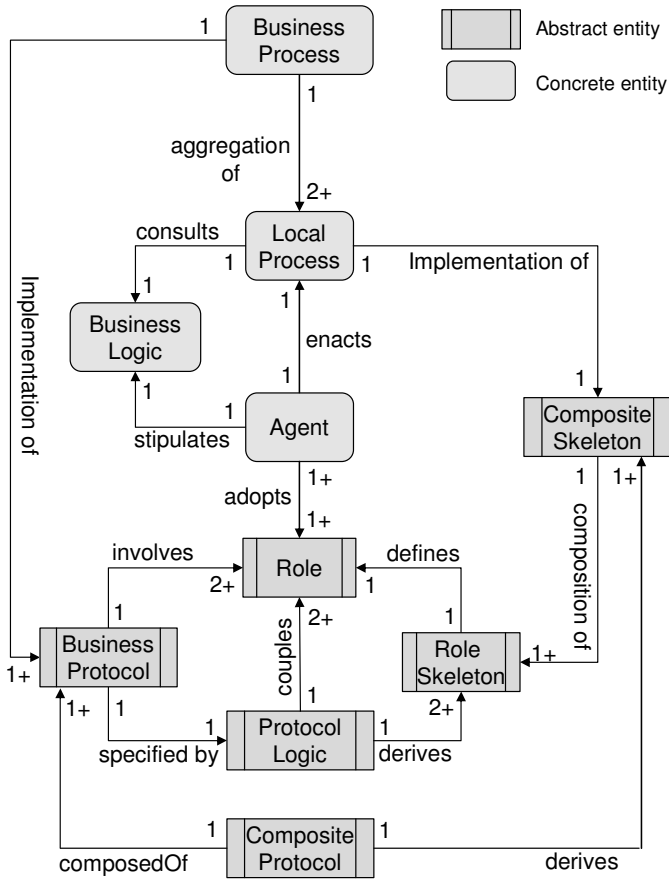


Fig. 2. Conceptual model: business processes based on protocols and policies

The resultant composite skeleton could be published and thus used by multiple suppliers. Alternatively, as here, a *composite protocol* for supply chain can be composed from component protocols such as trading and shipping, and the supplier's skeleton derived from the composite protocol.

The *local process* of an agent is an executable realization of its skeleton instantiated with its policies or *business logic*. A *business process* aggregates the local processes of the agents participating in it. Equivalently, a business process implements the constituent business protocols.

A. Protocols and Commitments

Commitments capture a variety of contractual relationships, while enabling manipulations such as delegation and assignment, which are essential for open systems [7]. For example, a customer would commit to the merchant to pay an item's price upon delivery. Violations of commitments can be detected; in some important settings, violators can be penalized. Verifying compliance is essential in open settings [8].

Definition 1: A commitment $C(x, y, p)$ denotes that agent x is obliged to agent y for bringing about 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.

Commitments can be *conditional*, denoted by $CC(x, y, p, q)$, meaning that x is committed to y to bring about q if p holds

where, p is called the precondition of the commitment. For example, $CC(c, b, goods(g), pay(a))$ means that customer c is committed to pay the bookstore b an amount a if b delivers the book g to c . When b delivers the book, i.e., $goods(g)$ holds, $CC(c, b, goods(g), pay(p))$ yields the base-level commitment $C(c, b, pay(p))$.

The following operations describe how a commitment c is created, satisfied, and manipulated [7].

- 1) $CREATE(x, c)$ establishes c , if directly performed by c 's debtor, or because of a conditional commitment.
- 2) $CANCEL(x, c)$ terminates c as an action of c 's debtor x . Generally, cancellation is constrained and would lead to new commitments.
- 3) $RELEASE(y, c)$ terminates c by releasing c 's debtor x . 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 is terminated by x fulfilling it.

A commitment is *active* if it has been created but not yet terminated. Commitments are transformed via the rules below:

- 1) $\frac{C(x, y, p) \wedge p}{discharge(x, C(x, y, p))}$
- 2) $\frac{CC(x, y, p, q) \wedge p}{create(x, C(x, y, q)) \wedge discharge(x, CC(x, y, p, q))}$
- 3) $\frac{CC(x, y, p, q) \wedge q}{discharge(x, CC(x, y, p, q))}$

III. PROTOCOL SPECIFICATION

Let's now consider how protocols are specified based on the meaning associated with a message.

A. OWL-P

Figure 3 shows key OWL-P concepts (nodes) and their properties (edges: the end with a rectangle is the domain). Figure 3 reflects the conceptual model of Section II. The core of a protocol is specified using implication rules, as in the Semantic Web Rule Language (SWRL) [9]. A *Message* has a

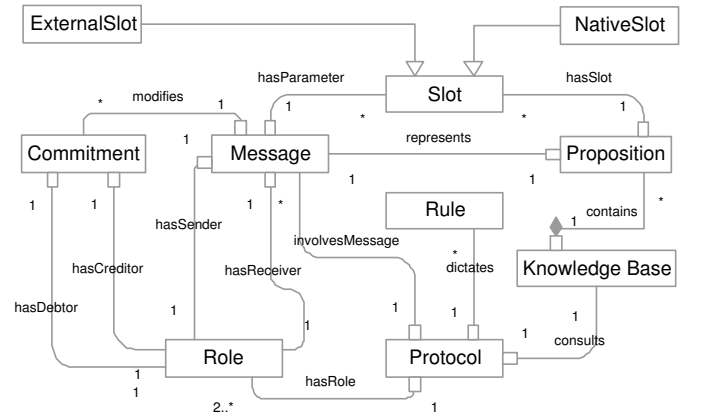


Fig. 3. Basic OWL-P ontology

sender, a receiver, and one or more parameters as *Slots*. The semantics of a message is given by its effects on commitments, expressed via operations. Slots function as data variables. A slot is *defined* when it is assigned a value and *used* when its value is assigned to another slot. A *native slot* is typed and

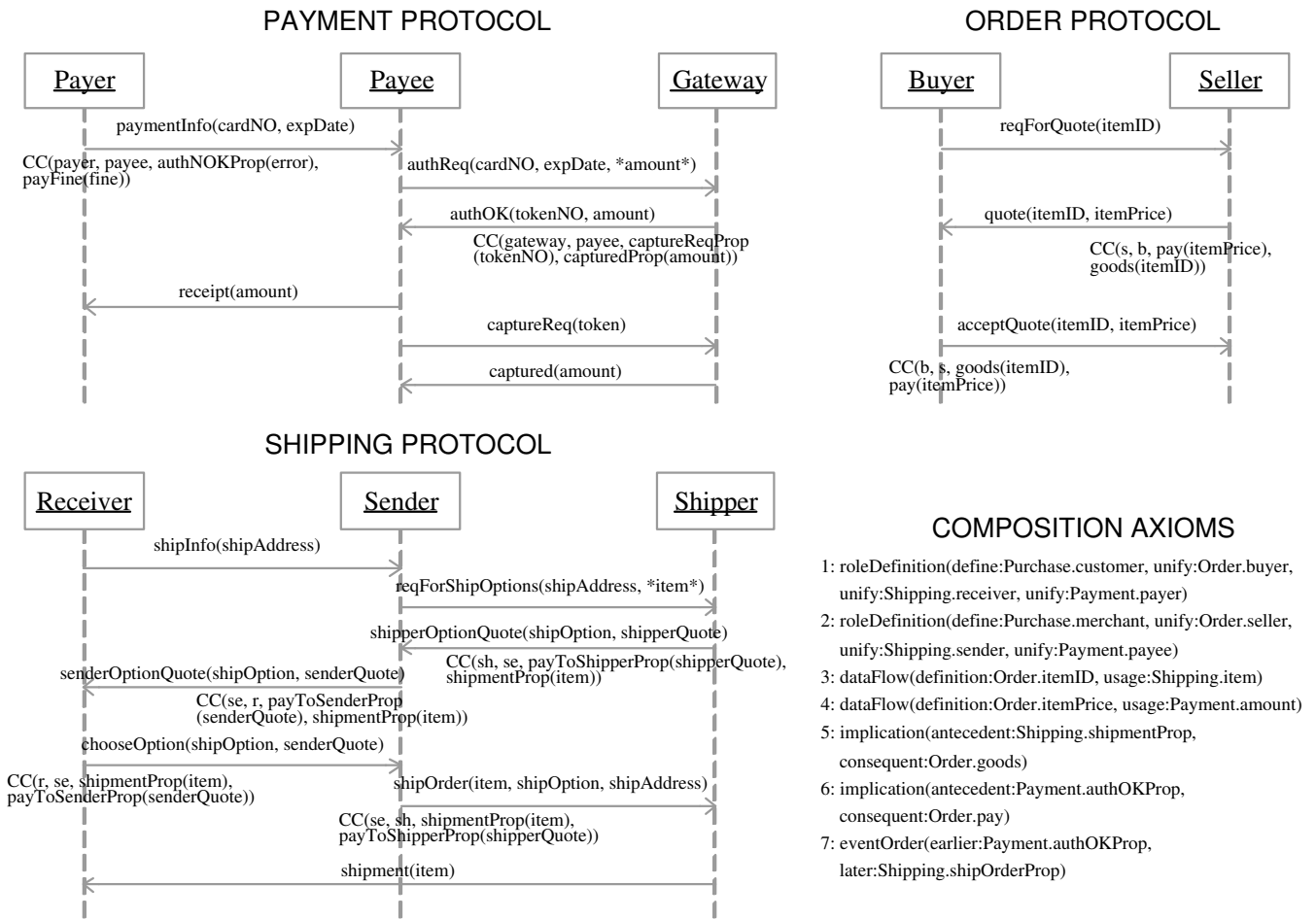


Fig. 5. Example: Order, Shipping, and Payment protocols and their composition

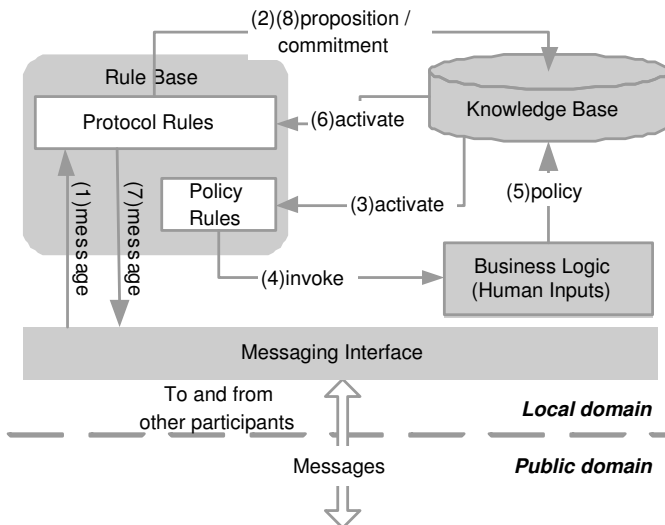


Fig. 4. Agent architecture: protocol and policy interplay

defined within the protocol. An *external slot* is assigned a value produced by another protocol.

A *protocol* dictates rules that govern the interaction and consults a *knowledge base*, which consists of a set of *propositions*. Propositions are represented as slotted facts in a rule-

based system. Propositions express messages sent or received, active commitments, and domain facts.

Figure 5 shows possible scenarios for protocols. A leading and trailing *, as in **amount**, identify an external slot. Consider the *Order* protocol. The buyer requests a quote for an item; the seller responds with a quote. The semantics of quote is that it creates a commitment from the seller to the buyer guaranteeing delivery if the buyer pays the quoted price. In this scenario, the buyer accepts the quote. The semantics of accept is to create a commitment from the buyer to the seller to pay the quoted price if it receives the requested item. Below are the rules for *Order* in the “antecedent \Rightarrow consequent” notation.

- 1: startProp \Rightarrow reqForQuote(?itemID)
- 2: reqForQuoteProp(?itemID) \Rightarrow quote(?itemID, ?itemPrice) \wedge CC(S, B, pay(?itemPrice), goods(?itemID))
- 3: quoteProp(?itemID, ?itemPrice) \Rightarrow acceptQuote(?itemID, ?itemPrice) \wedge CC(B, S, goods(?itemID), pay(?itemPrice))
- 4: quoteProp(?itemID, ?itemPrice) \Rightarrow rejectQuote(?itemID, ?itemPrice)

A knowledge base contains startProp at the start of a protocol. Here, reqForQuote, quote, and acceptQuote are OWL-P messages, and their corresponding propositions are

reqForQuoteProp, quoteProp, and acceptQuoteProp. pay and goods are also propositions. ?itemID and ?itemPrice are native slots. The ?itemID in the first rule is not assigned any value by the antecedent, meaning that the rule is abstract and not executable. As we see in Section V-B, it can be augmented with business logic that produces such values. The OWL-P ontology and example protocols are available at <http://research.csc.ncsu.edu/mas/OWL-P/>.

B. Operational Semantics

Protocols are specified from the global perspective with an assumption of an abstract global knowledge base. Later sections show how the abstraction of a global knowledge base maps to the perspectives of the participants' local knowledge bases. In OWL-P, rules are forward-chained. A message in the consequent of a rule indicates a message exchange. A commitment in the consequent of a rule indicates its creation. By the inference rules in Section II-A, a commitment is discharged automatically when its condition holds. Other operations on commitments are explicitly specified.

Figure 4 shows an abstract rule machine that provides operational semantics to the rules. For now, ignore steps 3, 4, and 5 dealing with policy rules. When a message is received, it is checked against the protocol rules to see if it may be consumed. If so, a corresponding proposition is asserted and any activated rules are executed. Doing so may activate other rules, resulting in further propositions being asserted and messages being sent.

IV. COMPOSITE PROTOCOLS

Protocols are rarely used in isolation and must be composed so as to address multiple business goals. A second motivation for composition is that it enables modifying protocols. When a desired modification is captured as a set of rules, it is definitionally identical to a protocol. Thus a protocol can be modified by composing it with a given modification. Let's now discuss how we can construct composite protocols and how they facilitate reusability and refinement.

A. Construction of Composite Protocols

Figure 6 describes key entities and properties dealing with protocol composition. A *composite protocol* aggregates (component) protocols and is defined by a *composition profile*. A composition profile stipulates several *composition axioms* to constrain the composite protocol. Figure 5 depicts three protocols, *Order*, *Shipping*, and *Payment*, and a set of composition axioms. We denote the composition axioms as $axiom_1(property_1:range_1, property_2:range_2, \dots)$, where $axiom_1$ identifies the axiom class, the $property_i$ are the properties of $axiom_1$, and the $range_i$ are the values of those properties. For brevity, we replace irrelevant slot names by '.'.

A *role definition* axiom defines a (unified) role in the composite protocol in terms of the roles in the component protocols to which it corresponds. In Figure 6, the first axiom states that customer in *Purchase* unifies buyer in *Order*, payer in *Payment*, and receiver in *Shipping*. Similarly, the second

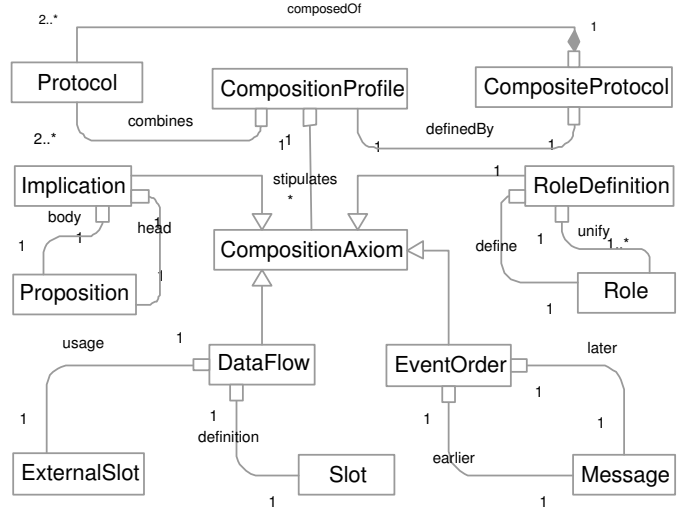


Fig. 6. OWL-P composition classes and properties

axiom defines a merchant in *Purchase*. Typically, the non-unified roles are played by different agents.

A *data flow* axiom captures a data flow between the protocols. It defines an external slot in one protocol as using an external or a native slot of another. In Figure 6, the fourth axiom states that slot ?amount in *Payment* gets its value from slot ?itemPrice in *Order*. Such a dependency implicitly ensures that none of the rules using a slot can fire before the slot is assigned a value by the defining rule. Formally:

```
axiom: dataFlow(definition:Order.itemPrice,
               usage:Payment.amount)
def   : reqForQuoteProp(.) ⇒ quote(.,?itemPrice) ∧ ...
use0  : paymentInfoProp(.,.) ⇒ authReq(.,?amount)
use1  : ...
usen  : captureReqProp(.) ⇒ captured(?amount)
```

The resulting rules:

```
use0  : paymentInfoProp(.,.) ∧ quoteProp(.,?itemPrice) ⇒
       authReq(.,?amount)
use1  : ...
usen  : captureReqProp(.) ∧ quoteProp(.,?itemPrice) ⇒
       captured(?itemPrice)
```

The ordering is achieved by adding the definition of the slot as a premise for the usage of the slot. Since there might be multiple uses of a slot, premises of all of them need to be updated. The usage slot takes the name of the defining slot and the external slot is not needed in the composite protocol.

An *implication* axiom states that proposition X in a protocol implies proposition Y in another protocol. In Figure 6, the sixth axiom states that an assertion of authOKProp in *Payment* means an assertion of pay in *Order*. This is easily achieved by adding an implication rule to the composite rule base. Formally:

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

The resulting rule:

```
rule : authOKProp(.,.) ⇒ pay(.)
```

An *event order* axiom specifies an ordering among selected

messages of the component protocols. In Figure 6, 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 ensured by making the rule for the later event depend on the rule for the earlier event, that is:

```
axiom :eventOrder(earlier:Payment.authOK,
                 later:Shipping.shipOrder)
earlier:authReqProp(·,·) ⇒ authOK(·,·) ∧ CC(·,·,·)
later :chooseOptionProp(·,·) ⇒ shipOrder(·,·) ∧ CC(·,·,·)
```

The resulting rule:

```
later :chooseOptionProp(·,·) ∧ authOKProp(·,·) ⇒
      shipOrder(·,·) ∧ CC(·,·,·)
```

Composition axioms must be specified by a designer. There may be several ways of composing a protocol. Specifically, if the component protocols are mutually independent, no axioms need be specified; the protocols can simply be unioned, yielding the specification of a composite. If necessary, additional subclasses of `CompositionAxiom` may be introduced by specifying their syntactic properties and operational semantics. A composite protocol is like any other protocol, and can be composed further.

How can we determine whether additional component protocols are needed to obtain an enactable protocol? A protocol may not be enactable due to *nonlocal choice*, which means that an agent's decision depends on information it doesn't have locally [10]. For example, protocols with rules of the form $a \Rightarrow b$, where proposition a is known only to roles ρ_1 and ρ_2 , and message b is sent by ρ_3 , are not enactable as ρ_3 cannot observe a . This problem can arise because a protocol is a global specification whereas role skeletons are partial views. The following properties of a protocol P avoid this problem and ensure that P is enactable. Here, r denotes a rule, a , b , p , and q are propositions or messages as applicable, and P is interpreted as a set of rules.

Prop1. $\forall r \in P, \forall a \in r.body, a \neq start \rightarrow \exists r' \in P a \in r'.head$.

The propositions in the antecedent of a protocol rule must be asserted by some protocol rule.

Prop2. $\forall r \in P, \forall a \in r.head, (start \rightsquigarrow a) \in P$. All asserted propositions are reachable from *start*. Here, $p \rightsquigarrow q$ means q is reachable from p . Formally, $p \rightsquigarrow q \stackrel{def}{=} p \Rightarrow q \vee (p \Rightarrow q' \wedge q' \rightsquigarrow q)$.

Prop3. $\forall r \in P, \forall a \in r.body, (\exists b \in r.head \wedge \exists \rho = sender(b) \rightarrow \rho = sender(a) \vee \rho = receiver(a))$. If a message exchange b has an antecedent a then, the sender ρ of b must be able to verify a . That is, ρ must receive a message corresponding to a or must send a message corresponding to a , i.e., ρ observes a or causes a .

A designer's goal is to obtain an enactable protocol by repeated applications of composition. Observe that in Figure 5, the rules of *Order* do not assert propositions `pay` and `goods` necessary for discharging the commitments created. The rules of *Payment* do not assert the proposition `pay(fineAmount)` and `amount` is an external slot. The rules of *Shipping* do not assert `payToSenderProp` and `payToShipperProp`, and `item` is an external slot. The rules of the composite *Purchase* do not assert the propositions `pay(fineAmount)`, `payToSenderProp`, and `payToShipperProp`. A designer would choose protocols

ADJUSTMENT PROTOCOL

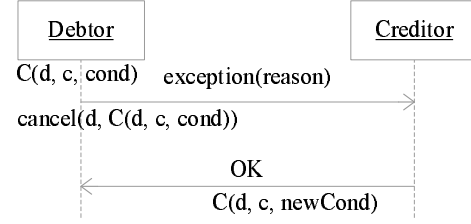


Fig. 7. Handling exceptions by composition

that assert the missing propositions and compose them with *Purchase* to obtain an enactable protocol.

B. Refinement by Composition

Business protocols evolve continually as new requirements and new features arise. Therefore, the ability to systematically refine protocols is valuable. In the composite *Purchase*, 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. 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 the payment and release the merchant from the commitment, provided the customer agrees to it. We can achieve this flexibility by combining *Purchase* with *Adjustment* as specified below (Figure 7):

```
C(d, c, cond) ⇒ exception(?reason) ∧ cancel(d, C(d, c, cond))
exceptionProp(?reason) ⇒ OK() ∧ C(d, c, newCond)
exceptionProp(?reason) ⇒ NOK() ∧ C(d, c, oldCond)
```

The axioms below yield *New*, a more flexible protocol:

- 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:Adjustment.C(d, c, cond))
- 4: Implication(body:Adjustment.C(d, c, newCond), head:Purchase.C(m, c, refund))
- 5: Implication(body:Adjustment.C(d, c, oldCond), head:Purchase.C(m, c, goods(itemID)))

Adjustment allows cancellation of the merchant's commitment if the customer deems it reasonable. The semantics of `OK` specifies the creation of a new commitment for compensation. The creditor may not agree, send a `NOK`, and retain the old commitment. Similar protocols for assigning, delegating, and releasing commitments can be defined.

V. PROCESSES

To enact its local process, a participant extracts its skeleton from an enactable protocol and instantiates it with its policies.

A. Role Skeletons

OWL-P describes a protocol assuming a global state. As in all distributed systems, the state of a protocol as seen by a role changes only when a message is sent or received by that role. This observation forms the basis of skeleton-generation as in Algorithm 1, where P is a protocol and ρ is a role.

Algorithm 1: $\text{deriveSkeleton}(P, \rho)$: Generate the skeleton for role ρ

```

1  $\rho.rules \leftarrow \text{getPertinentRules}(P, \rho)$ ;
2 foreach  $r \in \rho.rules \wedge m \in r.head \wedge \rho \neq \text{sender}(m)$  do
3   foreach  $a \in r.body$  do
4      $hist \leftarrow \{\}$ ;
5     Replace  $a$  with  $\text{replace}(P, \rho, a)$ ;
6 Procedure  $\text{getPertinentRules}(P, \rho)$ : Get rules for  $\rho$ 
7  $rules \leftarrow \{\}$ ;
8 foreach  $r \in P.rules$  do
9   foreach  $m \in r.head$  do
10    if  $\rho = \text{receiver}(m)$  then
11      Replace  $m$  with  $\text{RECEIVE}(m)$ ;
12    if  $\rho = \text{sender}(m)$  then
13      Replace  $m$  with  $\text{SEND}(m)$ ;
14    $rules \leftarrow rules \cup \{r\}$ ;
15 return  $rules$ ;
16 Procedure  $\text{replace}(P, \rho, a)$ : If  $a$  isn't asserted in
    $\rho.rules$ , replace it by something that is
17 if  $a \in hist$  then
18   return "FALSE";
19 if  $a = "start" \vee (rule \ r \in \rho.rules \wedge a \in r.head)$  then
20   return  $a$ ;
21  $hist \leftarrow hist \cup a$ ;
22  $disj \leftarrow "FALSE"$ ;
23 foreach  $r \in P.rules \wedge a \in r.head$  do
24    $conj \leftarrow "TRUE"$ ;
25   foreach  $b \in r.body$  do
26      $conj \leftarrow conj \cdot " \wedge " \cdot \text{replace}(P, \rho, b)$ ;
27    $disj \leftarrow disj \cdot " \vee " \cdot conj$ ;
28  $hist \leftarrow hist - a$ ;
29 return  $disj$ ;
   algocf

```

The algorithm gathers all the rules from the specification for P that have ρ receiving or sending a message. Next, the algorithm invokes $\text{replace}(P, \rho, a)$, defined from line 16 to line 29. If a proposition a enables a rule $r \in \rho.rules$ but a is not asserted by any rule in $\rho.rules$, it means that a was not observed by ρ . This procedure replaces a with the last proposition that ρ did observe, i.e., the proposition a' that was asserted in $\rho.rules$ and leads to a being asserted. Such a' will be found due to the properties of an enactable protocol.

As an example, we show a rule in *Shipping* in Figure 5, and the same rule in the receiver's skeleton. 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.

Protocol rule:

$\text{shipperOptionQuoteProp}(\cdot, \cdot) \Rightarrow \text{senderOptionQuote}(\cdot, \cdot) \wedge$
 $\text{CC}(\text{Se}, \text{Re}, \text{payToSenderProp}(\cdot), \text{shipmentProp}(\cdot))$

Receiver skeleton rule:

shipInfoProp(\cdot) $\Rightarrow \text{receive}(\text{senderOptionQuote}(\cdot, \cdot)) \wedge$
 $\text{CC}(\text{Se}, \text{Re}, \text{payToSenderProp}(\cdot), \text{shipmentProp}(\cdot))$

Protocol computations are given in terms of the message exchanges m , whereas skeletons computations are given in terms of $\text{send}(m)$ and $\text{receive}(m)$. We now prove Algorithm 1 correct.

Theorem 1 (Soundness): If condition $cond$ enables $\text{send}(m)$ and $\text{receive}(m)$ in the skeletons generated from protocol P , then $cond$ enables m in P .

Proof: Let $cond$ hold in the skeletons, enabling $\text{send}(m)$ and $\text{receive}(m)$. As $cond$ enables $\text{send}(m)$, there must exist a role $\rho = \text{sender}(m)$ such that $(cond \Rightarrow \text{send}(m)) \in \rho.rules$. For this rule to be in $\rho.rules$, it is necessary that a rule $(cond \Rightarrow m) \in P.rules$, as line 13 in Algorithm 1 is the only way to add such a rule to $\rho.rules$. However, if $(cond \Rightarrow m) \in P.rules$ then the message exchange m is enabled in P . ■

Theorem 2 (Completeness): If condition $cond$ enables a message exchange m in protocol P , then $cond$ also enables $\text{send}(m)$ and $\text{receive}(m)$ in the skeletons generated from P .

Proof: Let $cond$ hold in P , enabling m . This implies a rule $(cond \Rightarrow m) \in P.rules$. By line 13 in Algorithm 1, this means that there exists a role $\rho_s = \text{sender}(m)$ such that $(cond \Rightarrow \text{send}(m)) \in \rho_s.rules$. If such a rule exists in $\rho.rules$, and if $cond$ holds, then $\text{send}(m)$ must be enabled.

Also, By line 11, there exists a role $\rho_r = \text{receiver}(m)$ such that $cond' \Rightarrow \text{receive}(m) \in \rho_r.rules$, where $cond'$ is the result of line 5. However, by Lemma 1 (Appendix D), $cond \rightarrow cond'$. Hence, $\text{receive}(m)$ must be enabled. ■

B. Policies

As mentioned earlier, a rule in a protocol may be abstract, meaning that values of some of its native slots must be produced via the role's business logic. That is, a role skeleton must be augmented with business logic to obtain a local process. To capture this intuition, we define a skeleton as *concrete* if all of its native slots are defined, and as *abstract* otherwise.

Definition 2: A local process is a concrete skeleton derived from an enactable protocol.

An agent's business logic is specified as local policy rules. The skeleton of the seller in *Order* augmented with the policy rules of the seller agent as shown below. The policy rule invokes a business logic operation to decide what price to quote. The operation asserts quotePolicy , which activates the second skeleton rule.

Seller skeleton rules:

```

1:  $\text{startProp} \Rightarrow \text{receive}(\text{reqForQuote}(\text{?itemID}))$ 
2:  $\text{reqForQuoteProp}(\text{?itemID}) \wedge \text{quotePolicy}(\text{?itemPrice}) \Rightarrow$ 
    $\text{send}(\text{quote}(\text{?itemID}, \text{?itemPrice})) \wedge$ 
    $\text{CC}(\text{S}, \text{B}, \text{pay}(\text{?itemPrice}), \text{goods}(\text{?itemID}))$ 
3:  $\text{quoteProp}(\text{?itemID}, \text{?itemPrice}) \Rightarrow$ 
    $\text{receive}(\text{acceptQuote}(\text{?itemID}, \text{?itemPrice})) \wedge$ 
    $\text{CC}(\text{B}, \text{S}, \text{goods}(\text{?itemID}), \text{pay}(\text{?itemPrice}))$ 

```

```
4 : quoteProp(?itemID, ?itemPrice) =>
    receive(rejectQuote(?itemID, ?itemPrice))
```

Seller's policy for deciding quote:

```
1 : reqForQuoteProp(?itemID) =>
    call(policyDecider, quotePolicy(?itemID))
```

This pattern of augmenting policy rules is general and applies to rules where the agent has to make a decision and respond. It also assigns a value to native slots that are not defined. Figure 4 shows the interplay between the protocol rules and the policy rules of an agent. Steps 3, 4, and 5 show policy rules in action. The business logic could involve looking up a legacy database or waiting for human input.

C. Usage

Here, we show how our model of business processes can be grounded in Web services. Figure 8 summarizes our methodology with a scenario involving a customer interested in purchasing goods online. Software designers design protocols and register them with protocol repositories. They may also construct composite protocols by specifying composition axioms and reusing the existing component protocols from the repository. *Composer* is our tool for generating the composite protocol. A merchant wishing to supply goods online looks up *Purchase* in a repository. It generates the skeleton for the merchant role, augments it with its local policies, and deploys the result as a service. The profile for this service contains an OWL-P description of *Purchase*, and may be published to a UDDI registry. If a customer wishes to procure 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 skeleton with its local policies and starts interacting with the merchant. Our tools support these development scenarios and a prototype implementation based on the agent architecture of Figure 4.

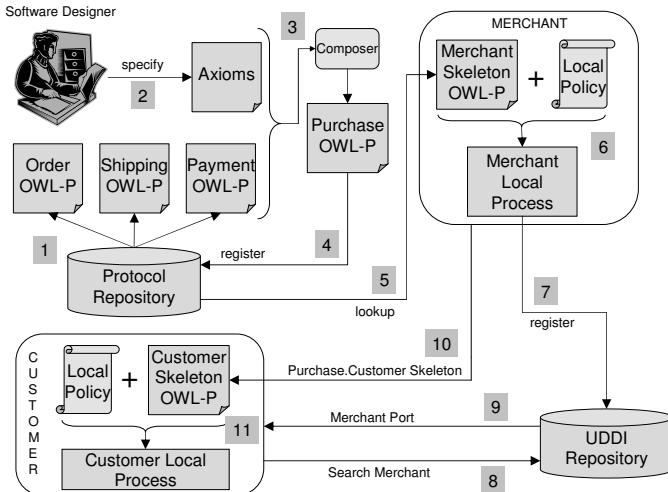


Fig. 8. A usage scenario

VI. π -CALCULUS FORMALIZATION

The foregoing showed how our interaction-based treatment facilitates reusability and evolvability. It would help to reason about the properties of the protocols. To do so, we first introduce the π -calculus, show how protocols and their composition can be specified in it, and then describe some properties of protocols established using the π -calculus formalization.

A. The π -calculus

The π -calculus [11] is a process algebra for modeling concurrent processes whose configurations may change as the processes execute. In the π -calculus, the fundamental unit of computation is the transfer of a communication link between two processes. The simplicity of the π -calculus arises from the fact that it includes only two kinds of entities: names and agents (processes). These are sufficient to rigorously define interactional behavior. This paper uses the synchronous π -calculus in which interaction corresponds to a handshake between two processes and involves the output of a link by one process and simultaneous receipt of the link by another process. The language of the π -calculus consists of prefixes and process expressions, as summarized below.

Prefixes are of the following kinds:

Prefixes	$\alpha ::=$	$\bar{a}(x)$	(Output)
		$a(x)$	(Input)
		τ	(Silent)

- The output prefix $\bar{a}(x)$ means that x is sent along the channel a .
- The input prefix $a(x)$ means that the channel a can be used to receive input and binds this input to x .
- The silent τ means that nothing observable happens.

The process expressions are as follows:

Agents	$P ::=$	0	(Nil)
		$\alpha.P$	(Prefix)
		$P + P$	(Sum)
		$P P$	(Parallel)
		$[x = y]P$	(Match)
		$[x \neq y]P$	(Mismatch)
		$(new x)P$	(Restriction)
		$!P$	(Replication)
		$A\langle y_1, \dots, y_n \rangle$	(Identifier)
Definitions	$A\langle x_1, \dots, x_n \rangle$	$\stackrel{\text{def}}{=} P,$	(where $i \neq j$ $\Rightarrow x_i \neq x_j$)

- 0 represents the nil-process.
- $\alpha.P$ does the action represented by prefix α and changes to P .
- $P + Q$ represents the sum-nondeterminism, that is, do either process P or process Q .
- $P|Q$ represents that process P and process Q execute in parallel.
- $[x = y]P$ represents the process that changes to P if $x = y$. Mismatch is the opposite, i.e., it checks $x \neq y$.
- $(new x)P$ means that the variable x is declared as a new name local to process P and bound in P . It is not visible outside of P .

- $!P$ represents an unbounded number of copies of the process P . Formally, $!P \stackrel{def}{=} P|!P$.
- $A\langle y_1, \dots, y_n \rangle$ represents the instantiation of a defined agent.
- $A(x_1, \dots, x_n) \stackrel{def}{=} P$ represents the declaration of a process A in terms of process P . One can think of it as a method declaration in traditional procedural programming.

The input prefix and the *new* operator bind names. For example, in the process $a(x).P$, the name x is bound, but a is not. It helps to use the *polyadic* π -calculus, because it enables sending and receiving tuples of variables [11]. For example, $\bar{x}(a, b, c)$ outputs three names a , b , and c over the channel x . And, $x(p, q, r)$ receives three names over x and binds them to p , q , and r respectively. We introduce a notation for broadcasts. A broadcast on a channel x of a name y is indicated by $\hat{x}(y)$ and is equivalent to $\bar{x}(y)$. As an example, consider the following:

$$P \equiv (\text{new } z)(z(w).\bar{w}(y)|x(u).\bar{u}(v)|\bar{x}(z))$$

The pair $\bar{x}(z)$ and $x(u)$ can interact. P_1 indicates the process left after interaction.

$$P_1 \equiv (\text{new } z)(z(w).\bar{w}(y)|\bar{z}(v)|0)$$

In P_1 , another interaction is possible over the channel z giving process P_2 .

$$P_2 \equiv (\text{new } z)(\bar{v}(y)|0|0)$$

Thus, process P evolves to P_2 over two interactions: $P \rightarrow P_1$ and $P_1 \rightarrow P_2$. We use \rightarrow^* to abbreviate a series of such interactions, thus writing $P \rightarrow^* P_2$.

B. Protocols

Using *Order* of Figure 5 as an example, we show how to formalize a protocol in the π -calculus. Figure 9 shows a model of *Payment* in the π -calculus. Pipes denote the channels, boxes the π -processes, and rounded rectangles the agents. The appendix presents complete formalizations of *Payment*, *Shipping*, and *Purchase*.

An exchange of messages corresponds to *reactions* in the π -calculus. Thus, a protocol can be specified as a π -process in which each message is represented by a channel. Unique names for channels corresponding to each message type are assumed. Each role skeleton is a π -process having the relevant messages as the π -process parameters. The structure of a protocol π -process would be a parallel composition of the skeletons of each of the roles. Thus, *Order* would be formalized as:

$$\begin{aligned} \text{Order}(b, s, rfq, quote, accept, reject, goods, pay) &\stackrel{def}{=} \\ &\text{Buyer}(rfq, quote, accept, reject) \mid \\ &\text{Seller}(rfq, quote, accept, reject) \mid \\ &\text{Comm}(b, s, rfq, quote, accept, reject, goods, pay) \end{aligned}$$

Here, *Comm* is a process that observes the message exchanges and manages commitments according to the message semantics. Each role is also represented as a name, e.g., b and s , and may represent a debtor or a creditor of a

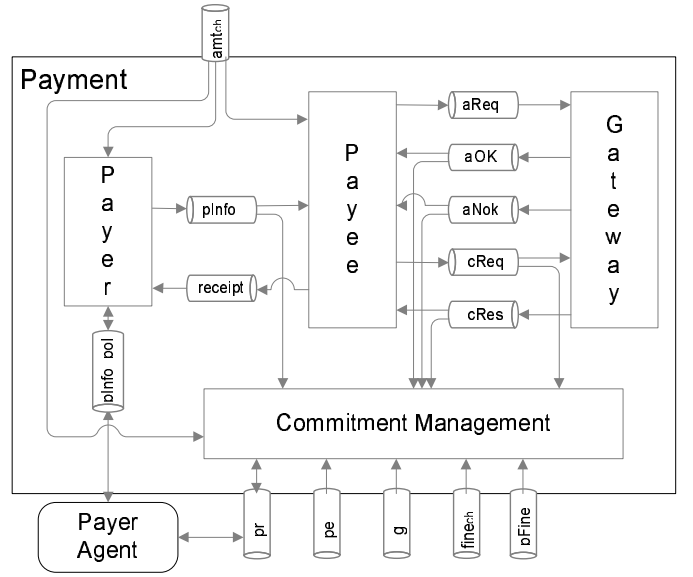


Fig. 9. A π -calculus model of *Payment*

commitment. The protocol messages are channels *rfq*, *quote*, *accept*, and *reject*. The channels *goods* and *pay* represent external messages, as discussed below. Both skeletons are passed all the message channels as both of the roles are involved in all the interactions. Here is the π -process for the seller skeleton:

$$\begin{aligned} \text{Seller}(rfq, quote, accept, reject) &\stackrel{def}{=} (\text{new } quote_pol) \\ &rfq(g).Quote_p.quote(g, p). \\ &(\text{accept}(g', p') + \text{reject}(g', p')) \end{aligned}$$

$$Quote_p \equiv \overline{quote_pol}(g).quote_pol(g, p)$$

The seller receives g (itemID) on *rfq*, consults its policy for quoting and sends the quote price p (itemPrice) on *quote*, and receives either *accept* (acceptQuote) or *reject* (rejectQuote). $Quote_p$ is a stub for interacting with the quoting policy of the seller agent, using the channel *quote_pol*. Such policy stubs output necessary names to the corresponding policy channel, and then receive some new names output by the business logic on the same channel. As the policies are private, the channels for interacting with them are restricted to the skeleton π -process, e.g., *quote_pol*. As g is bound by *rfq*, a new name g' should be bound to the input on *accept* or *reject*. Similarly as p is bound by *quote_pol*, p' binds to the input on *accept* or *reject*. Usually, g and g' would receive the same input but it cannot be guaranteed. Ideally, the seller should check if they match for the purpose of correlation. The following is the π -process for the buyer skeleton:

$$\begin{aligned} \text{Buyer}(rfq, quote, accept, reject) &\stackrel{def}{=} (\text{new } rfq_pol, accept_pol) \\ &RFQ_p.rfq(g).quote(g', p'). \\ &(\text{Accept}_p.\widehat{\text{accept}}(g', p') + \text{Reject}_p.\widehat{\text{reject}}(g', p')) \end{aligned}$$

$$RFQ_p \equiv \overline{rfq_pol}.rfq_pol(g)$$

$$\text{Accept}_p \equiv \overline{\text{accept_pol}}(g', p').\text{accept_pol}$$

$$\text{Reject}_p \equiv \overline{\text{reject_pol}}(g', p').\text{reject_pol}$$

It is easy to see that $(Buyer \mid Seller) \rightarrow^* 0$, provided the business logic is correct. This means that the skeletons of the buyer and the seller role are compatible for interaction. The commitment semantics of the interaction is given by the observer π -process $Comm$ as follows:

$$Comm(b, s, rfq, quote, accept, reject, goods, pay) \stackrel{def}{=} \\ (new\ ch_1, ch_2, ch_3, ch_4) \\ quote(g, p).CC\langle s, b, pay, goods, ch_1, ch_2 \rangle.(\overline{ch_1}(p) \mid \overline{ch_2}(g)) \mid \\ accept(g', p').CC\langle b, s, goods, pay, ch_3, ch_4 \rangle.(\overline{ch_3}(g') \mid \overline{ch_4}(p'))$$

On a message exchange involving a commitment operation, $Comm$ would invoke the corresponding commitment operation π -process. Here, the commitments are created when $quote$ or $accept$ is observed according to their semantics. After calling the π -process CC for a conditional commitment, the necessary message parameters are passed to it. The π -process CC is:

$$CC(deb, cred, cond_1, cond_2, ch_1, ch_2) \stackrel{def}{=} (new\ ch_3, ch_4, ch_5) \\ (ch_1(val_1) \mid ch_2(val_2)).(cond_2(val'_2).([val_2 \neq val'_2] \\ CC\langle deb, cred, cond_1, cond_2, ch_3, ch_4 \rangle.(\overline{ch_3}(val_1) \mid \overline{ch_4}(val_2)) + \\ [val_2 = val'_2]\tau) + cond_1(val'_1).([val_1 = val'_1] \\ C\langle deb, cred, cond_2, ch_5 \rangle.\overline{ch_5}(val_2) + [val_1 \neq val'_1] \\ CC\langle deb, cred, cond_1, cond_2, ch_3, ch_4 \rangle.(\overline{ch_3}(val_1) \mid \overline{ch_4}(val_2))))$$

First, the parameters passed by $Comm$ are received, and then the channels related to the commitment conditions are observed. On an exchange, if the commitment condition is satisfied then the process ends. If the precondition is satisfied, the conditional commitment reduces to a commitment, the π -process C is called, and the necessary parameters passed to it. If neither is satisfied, CC is called again and the commitment remains active. Private channels ch_i are used for passing parameters to avoid interference with the protocol channels. The commitment conditions may involve varying number of parameters and CC needs to be defined accordingly. The definition is for when both the precondition and the condition have one parameter each. Similarly, the π -process C is:

$$C(deb, cred, cond_2, ch_2) \stackrel{def}{=} (new\ ch_3)ch_2(val_2). \\ (cond_2(val'_2).([val_2 = val'_2]\tau + [val_2 \neq val'_2] \\ C\langle deb, cred, cond_2, ch_3 \rangle.\overline{ch_3}(val_2)))$$

The separation of the commitment semantics from the interaction specification simplifies the derivation of skeletons, e.g., the buyer skeleton is the parallel composition of the π -processes $Buyer$, $Comm$, CC , and C . Also, the loose coupling among the commitment semantics, the interaction specification, and the policy definitions (business logic) allows maintainability, reusability, and simplicity of design.

C. Composite Protocols

A composite protocol is a parallel composition of the component protocols and the composition axioms, e.g. $Purchase$ can be composed as follows (details in the Appendix):

$$Purchase(c, m, g, sh, \\ \dots\ order\ channels\ \dots, \\ \dots\ payment\ channels\ \dots, \\ \dots\ shipping\ channels\ \dots) \stackrel{def}{=}$$

$$Order\langle c, m, \dots \rangle \mid Payment\langle c, m, g, \dots \rangle \mid \\ Shipping\langle c, m, sh, \dots \rangle \mid Axiom4\langle accept, amt_{ch} \rangle \mid \\ Axiom6\langle aOK, pay \rangle \mid Axiom7\langle aOK, shipment \rangle \mid \\ \dots\ other\ axioms\ \dots$$

$$Axiom4(msg, channel) \stackrel{def}{=} msg(g, p).\widehat{channel}(p) \\ Axiom6(msg_1, msg_2) \stackrel{def}{=} msg_1(tNO, amt).\overline{msg_2}(amt) \\ Axiom7(msg_1, msg_2) \stackrel{def}{=} msg_1(\cdot, \cdot).\overline{msg_2}(\cdot, \cdot)$$

The channels c , m , g , and sh are the roles of $Purchase$. Notice how passing these channels to $Order$, $Payment$, and $Shipping$ accomplishes role unification; role definition axioms are not needed. The axiom numbers relate to the axioms in Figure 5.

A *Data Flow* axiom is a π -process with two parameters: the channel that defines a name, and the channel that uses the defined name. The axiom receives on the defining channel and forwards it to the using channel. For example, $Axiom4$ receives the price on the $accept$ channel and outputs the price to the amt_{ch} , i.e., the channel on which the users of amt in $Payment$ listen for the definition. (Broadcast is needed as there might be multiple listeners.)

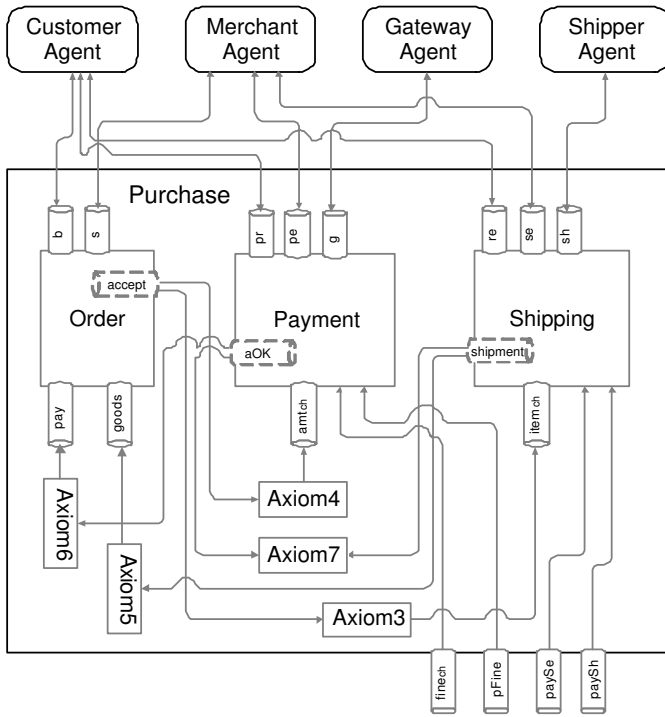
An *Implication* axiom is a π -process with two parameters. The first parameter is the antecedent of the implication and the second parameter is the consequent. The axiom listens on the antecedent channel, and on the message exchange, signals by outputting to the consequent channel. For example, $Axiom6$ listens on aOK to see if the payment is authorized and, signals so to pay when it is authorized.

An *Event Order* axiom is a π -process with two parameters: the channels that should exchange messages first and last, respectively. For example, $Axiom7$ expresses the condition that pay should do its exchange before $shipment$. We take a passive view of event order: monitoring the events to determine if the ordering is satisfied, but not attempting to enforce the condition through planning.

Figure 10 shows a model of $Purchase$. The dotted pipes are the protocol channels and the axioms listen on them. Agent business logics and the policy channels are not shown. It is encouraging to see how the protocols can be coupled with each other through their external channels via composition axioms. Clearly, $Purchase$ itself can be a component protocol in another composition.

D. Properties of Protocols

We describe some inferences about protocols that are supported by our formalization. Below, \mathcal{P} is any π -process expression, $Prot_i$ is a protocol π -process definition, $Role_{ij}$ is the skeleton definition of role j in $Prot_i$, and ch_{ik} is an external channel k for $Prot_i$. Subscripts i, j, k, m, n and so on range over the cardinality of the respective entity, e.g., $Role_{mn}$ could represent a skeleton π -process for role n in a protocol $Prot_m$. Also, \mathcal{S} and \approx are the weak simulation and bisimulation relations, respectively. We assume correct functioning of the business logic by replacing each $Policy_p$ by τ .

Fig. 10. A π -calculus model of *Purchase*

1) *Incorrectness*: A protocol specification $Prot_1$ causes a commitment violation if $Prot_1 \mid \overline{ch_{11}} \mid \dots \mid \overline{ch_{1k}} \rightarrow^* C \mid \mathcal{P}$. The debtor of C is responsible for the violation.

A protocol specification $Prot_1$ causes an interaction violation if $Prot_1 \mid \overline{ch_{11}} \mid \dots \mid \overline{ch_{1k}} \rightarrow^* Role_{1j} \mid \mathcal{P}$.

A protocol specification $Prot_1$ causes an event ordering violation if $Prot_1 \mid \overline{ch_{11}} \mid \dots \mid \overline{ch_{1k}} \rightarrow^* Axiom_i \mid \mathcal{P}$, and $Axiom_i$ is an event order axiom.

A protocol specification $Prot_1$ is incorrect if it causes either of the above violations. Note that correctness of a specification can only be verified relative to requirements. The definition of incorrectness given here is general and applies regardless of the requirements.

2) *Compatibility*: Role skeletons $Role_{ij}$ and $Role_{mn}$ are compatible if $Role_{ij} \mid Role_{mn} \mid \overline{ch_{11}} \mid \dots \mid \overline{ch_{pq}} \rightarrow^* 0$. Although this definition is intuitive, in case of iterative interactions, the process would never reduce to a *nil* process as suggested by Canal *et al.* [12]. In that case, Skeletons $Role_{ij}$ and $Role_{mn}$ are compatible if $Role_{ij} \mid Role_{mn} \mid \overline{ch_{11}} \mid \dots \mid \overline{ch_{pq}}$ continues making τ transitions without interacting with its environment.

3) *Equivalence*: Protocols $Prot_1$ and $Prot_2$ are equivalent if $Prot_1 \approx Prot_2$. Due to the presence of policy stubs in the protocol specification, the equivalence is defined by the weak bisimulation with policy interactions modeled as τ transitions.

4) *Flexibility*: A protocol $Prot_1$ is more flexible than a protocol $Prot_2$ if $(Prot_2, Prot_1) \in \mathcal{S}$ and $(Prot_1, Prot_2) \notin \mathcal{S}$. Note that this also implies $Prot_2$ can be substituted by $Prot_1$. Canal *et al.* [12] give a more rigorous treatment of component substitutability.

Other properties of interest might be commitment equivalence of the protocols to see if two protocols are equivalent as

far as the commitment semantics of the interaction goes. Also, a relation between a protocol and its refined version can be defined to see if one of them is a specialization of the other. For example, *payment by credit card* is a specialized payment protocol. We believe the formalization is rich enough to allow such inferences.

VII. RELATED WORK

The uniqueness of our approach arises from (1) our commitment-based semantics for interactions, (2) A focus on a methodology that enables reuse, refinement, and aggregation, and (3) Practical grounding of our concepts along with their theoretical foundations. Several relevant research efforts are summarized next.

Component behavior modeling: Plasil *et al.* [13] propose a language based on regular expressions for describing the behavior of software components using protocols. They define software components as agents and the proposed language allows expressions of method invocations over components. This behavior protocols are similar to our protocols but do not support commitments. Similarly, Canal *et al.* [12] use π -calculus to describe protocols over the methods provided of CORBA objects. They reason about interaction compatibility and object substitutability, i.e., if an object can be successfully substituted by another. Studying architectural connectors, Allen *et al.* [14] express similar intuitions about interactions being a fundamental abstraction for complex systems. Their components correspond to our agents, connectors to protocols, and attachments to role adoption. They formalize components and connectors in CSP [15], and have abstractions of roles and connector composition analogous to ours.

These approaches seek to verify interaction specifications for correctness, compatibility, deadlock freedom, and substitutability for components, whereas our thrust is to exploit protocols to develop processes, thus improving the quality of modeling and reuse of interactions. Previous approaches view interactions as sequences of send and receive actions without considering commitment-based semantics.

Scenario-based behavior modeling: This area develops visual formalisms for modeling interaction scenarios of entities (agents, components, or processes). Popular formalisms include AUML [16], Message Sequence Charts [17], and UML sequence diagrams [18].

Whittle *et al.* [19] present an approach for creating a design model (specified as a statechart) from a collection of interaction scenarios using a state-identification approach to merge scenarios. Uchitel *et al.* [20] present an MSC language to explicitly specify the assumptions under which scenarios are merged. Whereas the above approaches merge behaviors to produce one protocol, our approach merges several protocols to produce a composite protocol. Further, our methodology incorporates commitments to better capture business requirements.

Uchitel *et al.* [21] elaborate behavior models by deriving implied scenarios from an existing model, categorizing them as desired or not, and updating the model. Implied scenarios arise as a result of the partial nature of scenario-based specifications.

Protocols are naturally extensible, especially open protocols, which necessarily need to be fleshed out to become enactable.

Service composition: BPEL [3] is a language for specifying the static composition of Web services. However, it mixes interaction activities with business logic, thus reducing reusability. WS-CDL [22] specifies the conversational behavior of a service using control flow constructs. However, these specifications lack a flexible semantics, which makes them difficult to compose and reuse. OWL-S [23], which includes a process model for Web services, uses semantic annotations to enable dynamic composition. A composed service is produced at runtime based on stated constraints. Dynamic service composition assumes a perfect markup of the services being composed, and involves ontological matching between inputs and outputs. OWL-S formalizes process models of the ilk of BPEL, but doesn't address protocols as such. An end to end service creation approach separates a logical composition (service type) from a physical one (service instances) [24]. This approach combines a service matchmaking similar to OWL-S and a BPEL based service implementation. Due to its close ties with OWL-S and BPEL, it inherits their limitations mentioned above.

Some approaches treat service composition via formal specifications to support verifiability. Solanki *et al.* [25] 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 yields better compositionality than other approaches. Gerede *et al.* [26] model services as activity-based finite automata to study the decidability of composability and existence of a lookahead delegator for a set of services. However, these approaches consider neither the autonomy of the partners, nor the flexibility of composition.

Agent-oriented software engineering: Agent-oriented software methodologies, e.g., Gaia, KAOS, MaSE, and SADDE [27], apply software engineering principles to agent-based systems. Tropos [28] includes an early requirements stage in the process. Gaia [29] describes roles in the software system being developed, identifies their processes, and the safety and liveness conditions for the processes. It incorporates protocols in its interaction model and can be used with commitments. Baïna *et al.* [30] advocate a model-driven Web service development approach to ensure compliance between a service's implementation and its external protocol specifications. By contrast, our objective is achieving protocol reusability by separating protocols and policies, and by composing protocols.

Component catalogs: Our methodology enables using protocols as building blocks of business processes. Protocols are reused by refining them to yield improved protocols. RosettaNet has [31] put protocols in practice. About 100 published protocols, termed Partner Interface Processes (PIPs), account for billions of dollars of business. However, RosettaNet is limited to two-party request-response PIPs, which lack a formal semantics. The MIT Process Handbook [32] classifies business processes. For example, *sell* is a generic business process, which can be qualified by *sell what*, *sell to whom*, and so on.

VIII. CONCLUSIONS

The contributions of this paper include an approach to protocol composition, its grounding in OWL-P, and a formalization of the approach. The charm of this approach is that it recognizes the fundamental interactive nature of open environments where the autonomy of the participants must be preserved. Interactions are treated as first class. Commitments provide a semantics that combines rigor with flexibility. The presented approach will enable the development of an ever-increasing set of protocols for critical business functions.

We demonstrated the practicality of our approach by grounding it in a framework for specifying protocols. Some useful prototype tools supporting this methodology are available. We also demonstrated how our approach is theoretically founded in the π -calculus.

An important direction is to validate this approach in practice on a real-life problem with respect to metrics of reusability, scalability, and cost-effectiveness [33]. Another direction is to adapt protocols or skeletons to the context in which the business partners are interacting.

REFERENCES

- [1] OWL, "Web ontology language," Feb 2004, <http://www.w3.org/TR/2004/REC-owl-features-20040210/>.
- [2] SET, "Secure electronic transactions (SET) specifications," 2003, http://www.setco.org/set_specifications.html.
- [3] BPEL, "Business process execution language for web services, version 1.1," May 2003, www-106.ibm.com/developerworks/webservices/library/ws-bpel.
- [4] M. P. Singh, A. K. Chopra, N. Desai, and A. U. Mallya, "Protocols for processes: programming in the large for open systems," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 73–83, 2004.
- [5] N. Desai and M. P. Singh, "Protocol-based business process modeling and enactment," in *Proceedings of the International Conference on Web Services*, 2004, pp. 35–42.
- [6] A. U. Mallya and M. P. Singh, "A semantic approach for designing commitment protocols," in *Developments in Agent Communication*, LNAI, R. V. Eijk, Ed. Berlin: Springer, 2005, vol. 3396, pp. 37–51.
- [7] M. P. Singh, "An ontology for commitments in multiagent systems: Toward a unification of normative concepts," *Artificial Intelligence and Law*, vol. 7, pp. 97–113, 1999.
- [8] M. Venkatraman and M. P. Singh, "Verifying compliance with commitment protocols: Enabling open Web-based multiagent systems," *Autonomous Agents and Multi-Agent Systems*, vol. 2, no. 3, pp. 217–236, Sept. 1999.
- [9] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosz, 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/>.
- [10] P. B. Ladkin and S. Leue, "Interpreting message flow graphs," *Formal Aspects of Computing*, vol. 7, no. 5, pp. 473–509, 1995.
- [11] M. Robin, *Communicating and Mobile Systems: the pi-Calculus*. Cambridge University Press, 1999.
- [12] C. Canal, L. Fuentes, E. Pimentel, J. M. Troya, and A. Vallecillo, "Adding roles to corba objects," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 242–259, March 2003.
- [13] F. Plasil and S. Visnovsky, "Behavior protocols for software components," *IEEE Transactions on Software Engineering*, vol. 28, no. 11, pp. 1056–1076, 2002.
- [14] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Transactions on Software Engineering and Methodology*, July 1997.
- [15] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.
- [16] B. Bauer, J. Müller, and J. Odell, *Agent UML: A Formalism for Specifying Multiagent Interaction*, P. Ciancarini and M. Wooldridge, Eds. Springer-Verlag, 2001.

- [17] ITU, *Message Sequence Charts, Recommendation z.120*. International Telecommunication Union, Telecommunication Standardization Sector, 1996.
- [18] I. Jacobson, J. Rumbaugh, and G. Booch, *The Unified Software Development Process*. Addison Wesley, 1999.
- [19] J. Whittle and J. Schumann, "Generating statechart designs from scenarios," in *Proceedings of International Conference on Software Engineering*, 2000, pp. 314–323.
- [20] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of behavioral models from scenarios," *IEEE Transactions on Software Engineering*, vol. 29, no. 2, pp. 99–115, February 2003.
- [21] —, "Incremental elaboration of scenario-based specifications and behavior models using implied scenarios," *ACM Transactions on Software Engineering and Methodology*, vol. 13, no. 1, pp. 37–85, January 2004.
- [22] N. Kavantzaz, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon, "Web services choreography description language," December, 2004 (W3C Working Draft 1.0), <http://www.w3.org/TR/ws-cdl-10/>.
- [23] 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.
- [24] V. Agarwal, K. Dasgupta, N. Karnik, A. Kumar, A. Kundu, S. Mittal, and B. Srivastava, "A service creation environment based on end to end composition of web services," in *Proceedings of the International World Wide Web Conference*. ACM Press, 2005, pp. 128–137.
- [25] M. Solanki, A. Cau, and H. Zedan, "Augmenting semantic web service descriptions with compositional specification," in *Proceedings of the International World Wide Web Conference*, 2004, pp. 544–552.
- [26] C. E. 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.
- [27] F. Bergenti, M.-P. Gleizes, and F. Zambonelli, Eds., *Methodologies and Software Engineering for Agent Systems*. Kluwer, 2004.
- [28] P. Bresciani, A. Perini, P. Giorgini, F. Guinchiglia, and J. Mylopolous, "Tropos: An agent-oriented software development methodology," *Journal of Autonomous Agents and Multi-Agent Systems*, vol. 8, no. 3, pp. 203–236, May 2004.
- [29] F. Zambonelli, N. R. Jennings, and M. Wooldridge, "Developing multi-agent systems: The gaia methodology," *ACM Transactions on Software Engineering Methodology*, vol. 12, no. 3, pp. 317–370, 2003.
- [30] K. Bařna, 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.
- [31] RosettaNet, "Home page," 1998, www.rosettanet.org.
- [32] T. W. Malone, K. Crowston, and G. A. Herman, Eds., *Organizing Business Knowledge: The MIT Process Handbook*. Cambridge, MA: MIT Press, 2003.
- [33] P. Vitharana, H. Jain, and F. M. Zahedi, "Strategy-based design of reusable business components," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 34, no. 4, pp. 460–474, November 2004.



Ashok U. Mallya is a Ph.D. student in the Department of Computer Science at NCSU. His research interests are in agent communication, business process modeling, and the semantic web.



Amit K. Chopra is a Ph.D. student in the Department of Computer Science at NCSU. His interests span service-oriented computing, business processes, and multiagent systems.



Munindar P. Singh is a professor at North Carolina State University. His research interests include multiagent systems and Web services, where he specifically addresses the challenges of trust, service discovery, and business processes and protocols in large-scale open environments. Munindar's recent books include the coauthored *Service-Oriented Computing* and the edited *Practical Handbook of Internet Computing*.



Nirmitt Desai is a Ph.D. student in the Department of Computer Science at NCSU. His research interests include software engineering for open environments, business process modeling, and organization modeling. He received an M.S. in computer science from NCSU in 2003, and a B.E. in information technology from Gujarat University, India in 2001.

APPENDIX

Here, the π -calculus formalizations of *Payment* and *Shipping* of Figure 5 are presented followed by the formalization of the composite *Purchase*.

A. Payment

$Payment(pr, pe, g, pInfo, aReq, aOK, aNOK, receipt, cReq, cRes,$
 $amt_{ch}, pFine, fine_{ch}) \stackrel{def}{=} Payer(pInfo, receipt, amt_{ch}) \mid$
 $Payee(pInfo, aReq, aOK, aNOK, receipt, cReq, cRes, amt_{ch})$
 $\mid Gateway(aReq, aOK, cReq, cRes) \mid$
 $Comm\langle pr, pe, g, pInfo, aReq, aOK, aNOK, receipt, cReq, cRes,$
 $amt_{ch}, pFine, fine_{ch} \rangle$

$Payer(pInfo, receipt, amt_{ch}) \stackrel{def}{=} (new\ pInfo_pol)$
 $amt_{ch}(amt).PInfo_p.pInfo(cNO, exp).receipt(amt')$

$Payee(pInfo, aReq, aOK, aNOK, receipt, cReq, cRes, amt_{ch}) \stackrel{def}{=} (new\ aReq_pol, receipt_pol, cReq_pol)$
 $pInfo(cNO', exp').amt_{ch}(amt).AReq_p.aReq(cNO', exp', amt).$
 $(aOK(tNO, amt')).(Receipt_p.receipt(amt') \mid$
 $CReq_p.cReq(tNO).cRes(amt'')) + aNOK(tNO, error))$

$Gateway(aReq, aOK, cReq, cRes) \stackrel{def}{=} (new\ aOK_pol, aNOK_pol, cRes_pol)aReq(cNO, exp, amt).$
 $(AOK_p.aOK(tNO, amt').cReq(tNO').CRes_p.cRes(amt'')) +$
 $ANOK_p.aNOK(error))$

$Comm(pr, pe, g, pInfo, aReq, aOK, aNOK, receipt, cReq, cRes,$
 $amt_{ch}, pFine, fine_{ch}) \stackrel{def}{=} (new\ ch_1, ch_2, ch_3, ch_4)$
 $(pInfo(cNO, exp).CC\langle pr, pe, aNOK, pFine, ch_1, ch_2 \rangle.$
 $(amt_{ch}(amt) \mid fin_{ch}(fine)).(\overline{ch_1}(cNO, exp, amt) \mid \overline{ch_2}(fine)))$
 $\mid (aOK(tNO, amt').CC\langle g, pe, cReq, cRes, ch_3, ch_4 \rangle.$
 $(\overline{ch_3}(tNO) \mid \overline{ch_4}(amt'))))$

$PInfo_p \equiv \overline{pInfo_pol}(amt).pInfo_pol(cNO, exp)$
 $AReq_p \equiv \overline{aReq_pol}.aReq_pol$
 $AOK_p \equiv \overline{aOK_pol}(cNO, exp, amt).aOK_pol(tNO, amt')$
 $ANOK_p \equiv \overline{aNOK_pol}(cNO, exp, amt).aNOK_pol(error)$
 $Receipt_p \equiv \overline{receipt_pol}.receipt_pol$
 $CReq_p \equiv \overline{cReq_pol}.cReq_pol$
 $CRes_p \equiv \overline{cRes_pol}(tNO', amt').cRes_pol(amt')$

B. Shipping

$Shipping(re, se, sh, shInfo, rfso, shoq, seq, choose, shOrder,$
 $shipment, item_{ch}, paySh, paySe) \stackrel{def}{=} Receiver\langle shInfo, seq, choose, shipment, item_{ch} \rangle \mid$
 $Sender\langle shInfo, rfso, shoq, seq, choose,$
 $shOrder, shipment, item_{ch} \rangle \mid$
 $Shipper(rfso, shoq, shOrder, shipment) \mid$
 $Comm\langle re, se, sh, shInfo, rfso, shoq, seq, choose, shOrder,$
 $shipment, item_{ch}, paySh, paySe \rangle$

$Receiver(shInfo, seq, choose, shipment, item_{ch}) \stackrel{def}{=} (new\ shInfo_pol, choose_pol) item_{ch}(item).$
 $ShInfo_p.shInfo(shAdd).seq(shOp, seq).$
 $Choose_p.choose(shOp, seq).shipment(item')$

$Sender(shInfo, rfso, shoq, seq, choose, shOrder,$
 $shipment, item_{ch}) \stackrel{def}{=} (new\ rfso_pol, seq_pol, shOrder_pol)$
 $shInfo(shAdd).item_{ch}(item).Rfso_p.rfso(shAdd, item).$
 $shoq(shOp, shq).Seq_p.seq(shOp, seq).choose(shOp, seq).$
 $shOrder_p.shOrder(item, shOp, shAdd)$

$Shipper(rfso, shoq, shOrder, shipment) \stackrel{def}{=} (new\ shoq_pol, shipment_pol) rfso(shAdd, item).$
 $Shoq_p.shoq(shOp, shq).shOrder(item', shOp, shAdd').$
 $Shipment_p.shipment(item')$

$Comm(re, se, sh, shInfo, rfso, shoq, seq, choose, shOrder,$
 $shipment, item_{ch}, paySh, paySe) \stackrel{def}{=} (new\ ch_1, ch_2, ch_3, ch_4, ch_5, ch_6, ch_7, ch_8)$
 $(shoq(shOp, shq).CC\langle sh, se, paySh, shipment, ch_1, ch_2 \rangle.$
 $item_{ch}(item).(\overline{ch_1}(shq) \mid \overline{ch_2}(item))) \mid$
 $(seq(shOp, seq).CC\langle se, re, paySe, shipment, ch_3, ch_4 \rangle.$
 $item_{ch}(item).(\overline{ch_3}(seq) \mid \overline{ch_4}(item))) \mid$
 $(choose(shOp, seq).CC\langle re, se, shipment, paySe \rangle.$
 $item_{ch}(item).(\overline{ch_5}(item) \mid \overline{ch_6}(seq))) \mid$
 $(shOrder(item, shOp, shAdd).$
 $CC\langle se, sh, shipment, paySh, ch_7, ch_8 \rangle.$
 $\overline{ch_7}(item) \mid \overline{ch_8}(shq))$

$ShInfo_p \equiv \overline{shInfo_pol}(item).shInfo_pol(shAdd)$
 $Rfso_p \equiv \overline{rfso_pol}.rfso_pol$
 $Shoq_p \equiv \overline{shoq_pol}(shAdd, item).shoq_pol(shOp, shq)$
 $Seq_p \equiv \overline{seq_pol}(shOp, shq).seq_pol(shOp, seq)$
 $Choose_p \equiv \overline{choose_pol}(shOp, seq).choose_pol$
 $ShOrder_p \equiv \overline{shOrder_pol}.shOrder_pol$
 $Shipment_p \equiv \overline{shipment_pol}.shipment_pol$

C. Purchase

$Purchase(c, m, g, sh,$
 $rfq, quote, accept, reject, goods, pay, pInfo,$
 $aReq, aOK, aNOK, receipt, cReq, cRes, payFine, amt_{ch},$
 $fine_{ch}, shInfo, rfso, shoq, seq, choose,$
 $shOrder, shipment, item_{ch}, paySh, paySe) \stackrel{def}{=} Order\langle c, m, rfq, quote, accept, reject, goods, pay \rangle \mid$
 $Payment\langle c, m, g, pInfo, aReq, aOK, aNOK, receipt,$
 $cReq, cRes, payFine, amt_{ch}, fine_{ch} \rangle \mid$
 $Shipping\langle c, m, sh, shInfo, rfso, shoq, seq, choose,$
 $shOrder, shipment, item_{ch}, paySh, paySe \rangle \mid$
 $Axiom3\langle accept, item_{ch} \rangle \mid$
 $Axiom4\langle accept, amt_{ch} \rangle \mid$
 $Axiom5\langle shipment, goods \rangle \mid$
 $Axiom6\langle aOK, pay \rangle \mid$
 $Axiom7\langle aOK, shipment \rangle$

$Axiom3(msg, channel) \stackrel{def}{=} msg(g, p).\widehat{channel}(g)$
 $Axiom4(msg, channel) \stackrel{def}{=} msg(g, p).channel(p)$
 $Axiom5(msg_1, msg_2) \stackrel{def}{=} msg_1(item).\overline{msg_2}(item)$
 $Axiom6(msg_1, msg_2) \stackrel{def}{=} msg_1(tNO, amt).\overline{msg_2}(amt)$
 $Axiom7(msg_1, msg_2) \stackrel{def}{=} msg_1(tNO, amt).msg_2(item, shOp, shAdd)$

D. Lemma

Lemma 1: $\forall r \in P$, if P is enactable, r is of the form $p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow m$, where each p_i is an atomic proposition, $\rho_r = receiver(m)$, and $p'_1 \wedge p'_2 \wedge \dots \wedge p'_n \Rightarrow receive(m) \in \rho_r.rules$, where each p'_i is a proposition, then $\forall i\ p_i \rightarrow p'_i$

Proof: Without loss of generality, assume that for p_1 , following are the only rules in P that assert p_1 :

- 1: $a_1^1 \wedge a_1^2 \wedge \dots \wedge a_1^n \Rightarrow p_1$
- 2: $a_2^1 \wedge a_2^2 \wedge \dots \wedge a_2^n \Rightarrow p_1$
- ⋮
- k: $a_k^1 \wedge a_k^2 \wedge \dots \wedge a_k^n \Rightarrow p_1$

When, Algorithm 1 is invoked for ρ , the rule $p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow m$ would be encountered at line 2, and a call to `replace` will be made for p_1 . There are two possibilities in `replace`:

Case 1: The condition on line 19 is satisfied. In that case, $p'_1 = p_1$. As this can be established for any p_i , $p_i \rightarrow p'_i$.

Case 2: The condition on line 23 is satisfied. In that case, line 23 loops over each of the rules 1 to k above and line 25 loops over each of the a_k^i in a rule k . Assuming that for each recursive call to `replace` for a_k^i , the condition on line 19 is satisfied, `Disj` of line 29 would be the disjunction of the bodies of the rules 1 to k above. As there is no other rule that asserts p_1 , $p_1 \rightarrow \text{Disj}$. That is, for p_1 to hold, `Disj` must hold. But, according to line 29, $p'_1 = \text{Disj}$. This means that $p_1 \rightarrow p'_1$. As this can be established for any p_i , $p_i \rightarrow p'_i$.

If in a recursive call to `replace`, some a_k^i did not satisfy the condition on line 19, then in the construction of `Disj` the algorithm will eventually replace a_k^i with a b_k^i such that b_k^i satisfies the condition on line 19. By the argument presented above, $a_k^i \rightarrow b_k^i$. Hence, again $p_i \rightarrow p'_i$.

As P is enactable, either of the above cases must hold due to the properties Prop1 and Prop2 (Section IV-A). ■