# Information-Driven Interaction-Oriented Programming: BSPL, the Blindingly Simple Protocol Language

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

singh@ncsu.edu

## ABSTRACT

We present a novel approach to interaction-oriented programming based on declaratively representing communication protocols. Our approach exhibits the following distinguishing features. First, it treats a protocol as an engineering abstraction in its own right. Second, it models a protocol in terms of the information that the protocol needs to proceed (so agents enact it properly) and the information the protocol would produce (when it is enacted). Third, it naturally maps traditional operational constraints to the information needs of protocols, thereby obtaining the desired interactions without additional effort or reasoning. Fourth, our approach naturally supports *shared nothing* enactments: everything of relevance is included in the communications and no separate global state need be maintained. Fifth, our approach accommodates, but does not require, formal representations of the meanings of the protocols. We evaluate this approach via examples from the literature.

## Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems*; D.2.1 [**Software Engineering**]: Requirements Specifications—*Methodologies*; H.1.0 [**Information Systems**]: Models and Principles—*General*

## General Terms

Theory, Design

## Keywords

Business process modeling, business protocols

## 1. INTRODUCTION

Interaction-oriented programming or IOP is concerned with the engineering of systems comprising two or more autonomous and heterogeneous components or *agents*. Such systems arise commonly in IT applications such as cross-organizational business processes and scientific collaboration. The key idea of IOP is that treating interactions as first-class concepts helps create systems whose participating agents can be independently designed and operated, with correctness judged largely based on their effective participation in the specified interactions. We model interactions as communications (messages sent by one agent to another), even though in some cases they may involve physical actions such as delivering a package or controlling an ocean glider. We specify interactions abstractly as arising between *roles*, each role a discrete conceptual party instantiated by one or more agents.

A *protocol* is a specification of (presumably, conceptually cohesive and suitably structured) communications among two or more agents that neglects the internal reasoning of the agents involved [3]. Two aspects of a protocol are relevant: (i) *operations*, to do with message occurrence and order; and, (ii) *meaning*, to do with the (business) import of the messages. Traditional approaches capture the operations procedurally but disregard the meaning. Procedural approaches are generally over-specified, rigid, and difficult to maintain, but yield obvious directives for agents and, hence, can be easy to realize. The declarative operational approaches support asserting operational constraints in logic, and offer increased clarity and flexibility. However, neither kind of approach handles the challenges of distributed computing well, especially to determine correct local enactments.

In contrast, a *business protocol* is one where we primarily or exclusively state the meanings of its messages and only indirectly any operational constraints on them [12]. Emphasizing meaning improves flexibility and maintainability. However, the meaning-based approaches rely upon a suitable characterization of the operations to unambiguously assign meanings to communications. Thus, existing approaches, whether procedural or declarative, end up specifying interactions in terms of the allowed orderings of messages.

*Contributions.* Our main motivation is to provide a simple declarative foundation for the operational underpinnings of IOP. We propose a novel declarative approach that (i) simplifies the operational details and (ii) cleanly separates operations from meanings, yet supports specifying meanings overlaid on operations. We claim that our approach is extremely simple: it has only two main constructs: (i) defining a message schema and (ii) composing existing protocols. Accordingly, we have dubbed it *BSPL*, the *Blindingly Simple Protocol Language*. BSPL states no constraints on the ordering or occurrence of messages, deriving any such constraints from the information specifications of message schemas. It treats interaction as first class and supports protocol compo-

**Cite as:** Information-Driven Interaction-Oriented Programming: BSPL, the Blindingly Simple Protocol Language, Munindar P. Singh, *Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, Tumer, Yolum, Sonenberg and Stone (eds.), May, 2–6, 2011, Taipei, Taiwan, pp. XXX-XXX.

sition at its core. BSPL forces specifiers to be clear about the essential constraints, thereby helping them exclude spurious constraints that plague traditional approaches. It supports clear well-formedness conditions under which the specified protocols can be shown to be enactable. We show how BSPL captures a variety of common and subtle protocols.

*Organization.* Section 2 identifies some of the key principles that guide our approach. Section 3 introduces BSPL, including its motivation, syntax, intended semantics and several examples illustrating specification and composition of protocols. Section 4 suggests how a suitable semantics of BSPL may be formalized. Section 5 compares BSPL with two existing approaches based on protocols from the financial domain, demonstrating its advantages. Section 6 discusses related work and some directions for future research.

## 2. PRINCIPLES OF BSPL

The following principles distinguish our approach to IOP.

**Information orientation.** In information systems, agents interact with each other because they wish to obtain or convey information. Thus we specify each protocol as involving not only two or more roles but also one or more *parameters*, which stand in for the actual information items to be exchanged among the agents playing the roles during enactment.

**Explicit causality.** The flow of causality is reflected in the flow of information: there are no hidden flows of causality because there are no hidden flows of information. Indeed, if there were any hidden flows, then the very idea of protocols as a basis for IOP would be called into question.

**No state.** We need no global repository of state. All the relevant information affecting the notional *social state* of an interaction is explicit within the protocol—in the values of the parameters of the messages exchanged. No agent's private business logic is relevant, which is a key motivation for IOP.

**Separating structure from meaning.** The structure of a protocol is completely characterized by the names of the entities in its specification; the meaning relies upon the values exchanged during enactment. A protocol ought not to constrain the values directly since realizing such constraints would depend upon the implementations of agents. Instead, we would place any relevant constraints in the meaning layer, usually in terms of the commitments of the participants [12].

Putting the above observations together leads us to an extensive treatment of parameters. Parameters on messages are obvious, but we expand them to apply on protocols generally. Crucially, we adorn each parameter of a protocol with a specification of whether the parameter must be an input or an output of the protocol. Importantly, the adornments of the parameters are interpreted with respect to the protocol itself, not with respect to any role of the protocol. Thus, the adornment ⌜in⌝ means that the associated parameter must be instantiated so as to enact the protocol, i.e., "exogenously" or "externally" to the protocol. And, ⌜out⌝ means that the associated parameter must be instantiated in the course of enacting the protocol—i.e., it is instantiated "endogenously" or "internally" to the protocol. Consider a quote message as part of a price discovery protocol that includes an item description and a price, and may be sent in response to a request for quotes for a particular item. Clearly, for the quote message to be sent, its sender must instantiate all of its parameters. However, from the standpoint of the protocol, the item description is provided from outside the protocol and the price is provided by the protocol to the outside. Thus we would adorn the item description with ⌜in⌝ and the price with ⌜out⌝.

As indicated above, parameters and their adornments apply not only to individual messages but to entire protocols. For example, we might model *Shipping* as involving a parameter address with adornment ⌜in⌝. This indicates that *Shipping* does not determine the address, but must be "told" it. Conversely, the *Window Shopping* protocol (involving roles SHOPPER and STOREFRONT) would have parameters what and howmuch, each with an adornment ⌜out⌝. This indicates that *Window Shopping* would yield information about what the STOREFRONT is selling for how much.

Sometimes, the interaction explicitly demands a flow of information. For example, it is not possible for a bank to transfer funds without knowing how much and to what account. Therefore, a transfer message must follow a request that specifies the amount and the account. At other times, we may impose an ordering for "conventional" reasons. For example, although payment and delivery may occur concurrently once the item and price are determined, we may impose an ordering arbitrarily. For example, in a fast-food restaurant the customer pays first and in a traditional restaurant the customer pays at the end. We capture such conventions in our causal ordering by explicitly introducing suitable parameters, e.g., we may model the *payment* message as having a token adorned ⌜out⌝ that is adorned ⌜in⌝ on *delivery*. Another example of a convention is where a buyer must show a proof of age before buying an alcoholic drink. We can introduce a parameter ageProof to handle this case.

Our information orientation leads us to make three crucial assumptions, all based on treating a protocol as a conceptual entity or relation [5] whose instances or tuples are its enactments. Protocols are meant to be instantiated multiple times. For example, many agents would use *Purchase* to buy and sell many items, yielding a distinct instance (tuple) for each enactment. (In passing, we note that although we talk of relations here, practical BSPL settings would often use XML and parameters might be bound to XML documents.)

**Uniqueness.** We introduce the notion that some or all of a protocol's parameters define a *key*, which characterizes the expected uniqueness of the enactment: at most one enactment instance may occur one per key binding.

**Integrity.** Analogous to NOT NULL constraints on relations, each required public parameter in a complete protocol enactment must be bound. Otherwise, that means the enactment viewed as a tuple is incomplete.

**Immutability.** Each enactment viewed as an entity instance is immutable. That is, the parameters can be bound multiply often, but the different bindings arise in different enactments. Immutability provides robustness against asynchrony, because it ensures bindings are never ambiguous or out of date.

Together, the above assumptions delimit an enactment: it must proceed sufficiently to generate at least one tuple of parameter bindings; any additional duplication of parameter values is superfluous. This provides a general, principled basis for termination that contrasts both with procedural approaches (explicit enumeration of terminal states) and meaning-based approaches (termination as achieving a requisite meaning state, such as having no pending unconditional commitments). The latter treats termination the same as never beginning, which is operationally false. Further, it rejects protocols whose main purpose is to create unconditional commitments: thus it would formulate a negotiation protocol as producing conditional commitments, but then all or nearly all of its states would be terminal.

Protocols may be composed [3]. For example, we may define *Purchase* as a composition of *Order*, *Payment*, and *Shipping*. A fundamental tenet of our approach is that we make no distinction between individual or composite protocols. A composite protocol is expressed, assigned a formal semantics, and enacted the same way as any other protocol. The only difference might be that the constituent protocols generally exist before the design episode and the composite protocols are created during the design episode. As we remarked above, a message is the unit of interaction. Thus a single message is an atomic protocol.

The main benefit of composing protocols is to facilitate the reuse of designs and implementations by supporting multiple ways to compose the same protocols. Further, protocol composition offers a principled basis for adapting cross-organizational process models to capture evolving requirements, as in Desai et al.'s [2] approach.

Each protocol name is unique within our universe of discourse. Each protocol defines a scope (a unique namespace) within which its roles, parameters, and messages are also uniquely named. The roles and parameters of a protocol identify its public interface. In logical terms, a role acts as a kind of parameter, but semantically, they are quite different: a role corresponds to an executing agent whereas a parameter corresponds to a data item.

# 3. BSPL: LANGUAGE AND PATTERNS

Based on the foregoing, we define a protocol as consisting of exactly one message schema (template) or of the composition of two or more protocols. Although this is mathematically satisfactory, for practical convenience, we define the syntax of BSPL in a somewhat more conventional manner. The following is the syntax along with brief explanations. Superscripts of $+$ and $*$ indicate one or more and zero or more repetitions, respectively. Below, $\lfloor$ and $\rfloor$ delimit expressions, considered optional if without a superscript. For simplicity, we state cardinality restrictions informally.

For readability, we include some syntactic sugar tokens. In listings, we write reserved keywords in small sans serif, capitalize role names, and write parameters in camel case. In the text, we write message and protocol names *Slanted*, roles in SMALL CAPS, and parameters in sans serif. We insert ⌜ and ⌝ as delimiters, as in ⌜Self ↦ Other: hello[ID, name]⌝.

$L_1$. A protocol declaration consists of exactly one name, two or more roles, one or more parameters, and one or more references to constituent protocols or messages. A nilable parameter, ignored here for brevity, can remain unbound. All the parameters marked key (cannot be nilable) together form the key of this declaration.

*Protocol* ⟶ *Name* { role *Role*$^+$
parameter ⌊*Parameter*⌊key|⌊nilable⌋⌋$^+$ *Reference*$^*$ }

$L_2$. A reference consists of the name of a protocol appended by the same number of roles and parameters as in its declaration. At least one of the parameters of the reference must be a key parameter of the declaration in which it occurs: this ensures the enactments of the reference relate to those of the current declaration.

*Reference* ⟶ *Name* ( *Role*$^+$ *Parameter*$^+$ )

$L_3$. Alternatively, a reference is a message schema, and consists of exactly one name along with exactly two roles, one or more parameters (at least one a key parameter of the declaration), and optionally a meaning.

*Reference* ⟶ *Role* ↦ *Role* : *Name* [ *Parameter*$^+$ ]
⌊means *Expression*⌋

$L_4$. Each parameter consists of a name and an optional adornment.

*Parameter* ⟶ ⌊*Adornment*⌋ *Name*

$L_5$. An adornment is generally either ⌜in⌝ or ⌜out⌝. It may be ⌜nil⌝ in a reference to indicate that the adorned parameter is unknown, which can be crucial in some cases.

*Adornment* ⟶ in | out | nil

Now, we introduce a series of examples of BSPL as a way to informally describe its semantics. We omit the means clauses in our development but revisit them in Section 6.

## 3.1 Simple Protocol Declarations

Listing 1 demonstrates BSPL to define the simple *Pay* protocol. *Pay* consists of two roles and one message consisting of two parameters, ID and amount. We adorn both parameters ⌜in⌝ to indicate that they must be known (supplied) to enact *Pay*. In other words, a multiagent system must enact *Pay* in combination with some one or more other protocols that determine ID and amount. Because ID is the key, at most one payment may be made for a given ID value.

**Listing 1: The *Pay* protocol.**

```
Pay {
 role Payer, Payee
 parameter in ID key, in amount

 Payer ↦ Payee: payM[in ID, in amount]
}
```

Listing 2 shows *Offer*, which serves as a way to generate a price offer. Notice that both item and price are adorned ⌜out⌝, indicating that *Offer* would compute these parameters endogenously. Here, the BUYER generates item and the SELLER generates price, since these parameters are adorned ⌜out⌝ in messages to be sent by these roles. Thus, *Offer* can generate the amount (if identified with price) needed to enact *Pay*. Conversely, ID is adorned ⌜in⌝, meaning that *Offer* can only be used in combination with another protocol in which ID (suitably renamed if necessary) is adorned ⌜out⌝.

**Listing 2: The *Offer* protocol.**

```
Offer {
 role Buyer, Seller
 parameter in ID key, out item, out price
```

```
 Buyer ↦ Seller: rfq[in ID, out item]
 Seller ↦ Buyer: quote[in ID, in item, out
     price]
}
```

Listing 3 shows *Order*, which as formalized here repeats the entire *Offer*. Section 3.2 shows how to avoid such redundancy through composition. Notice that Listing 3 includes a parameter rID, which is adorned ⌜out⌝ in the protocol's interface as well as in the *accept* and *reject* messages. Each of these placements has an important ramification. First, if we were to omit rID from *Order*'s interface, its enactments would complete as soon as *quote* was sent, because all its parameters would then be bound. In other words, the enactment would complete prematurely. BSPL does not support separately requiring *accept* or *reject*, because doing so would violate the principle of explicit causality. Second, the presence of rID in both *accept* and *reject* indicates mutual exclusion of those two messages: simply because a parameter cannot be bound more than once in any enactment.

**Listing 3: The *Order* protocol.**

```
Order {
 role B, S
 parameter in ID key, out item, out price, out rID

 B ↦ S: rfq[in ID, out item]
 S ↦ B: quote[in ID, in item, out price]

 B ↦ S: accept[in ID, in item, in price, out rID]
 B ↦ S: reject[in ID, in item, in price, out rID]
}
```

## 3.2 Composing a Protocol

The power of protocols in modeling arises from the fact that they can be readily composed [3]. BSPL makes no distinction between a protocol that happens to be composed and one that is not. Indeed, each message can be viewed as a protocol in its own right. Listing 4 expresses the message ⌜From ↦ To: aMessage[in one, out two]⌝ as a protocol.

**Listing 4: A message viewed as a protocol.**

```
Message-as-Protocol {
 role From, To
 parameter in one key, out two key

 From ↦ To: aMessage[in one, out two]
}
```

Taking the same idea further, Listing 5 expresses *Order* as a composition of *Offer* and two protocols corresponding to the other messages defined in Listing 3. Even if we changed the role and parameter names, Listings 3 and 5 would remain semantically identical.

**Listing 5: *Order* expressed as a composition.**

```
Order {
 role B, S
 parameter in ID key, out item, out price, out rID

 Offer(S, B, in ID, out item out price)
 acceptProt(B, S, in ID, in item, in price, out
     rID)
 rejectProt(B, S, in ID, in item, in price, out
     rID)
}
```

## 3.3 More on Parameters in Protocols

Parameters are crucial to BSPL. We adorn the parameters not only in a declaration but also in each reference, including the individual messages. In general, such adornments are essential for capturing any constraints on what parameter bindings to propagate in what direction.

Listing 2 demonstrates two important well-formedness requirements on protocols. One, a parameter that is adorned ⌜in⌝ in a declaration must be ⌜in⌝ throughout its body. For brevity, we may sometimes omit such ⌜in⌝ adornments, but such parameters can take no adornment other than ⌜in⌝. Two, a parameter that is adorned ⌜out⌝ in the declaration must be ⌜out⌝ in at least one reference. At run time, at most one reference with an ⌜out⌝ adornment for a parameter may be enacted: thus such references are mutually exclusive.

If a parameter is adorned in a protocol declaration $P$, then any reference to $P$ must apply the same adornment to that parameter. Otherwise, it would not be clear what propagation was appropriate. But we can leave some or all of $P$'s parameters unadorned in a declaration or reference, thus signifying that propagation in both directions is permissible for that declaration or reference. We can think of this as the in–out adornment. When we refer to $P$ from another protocol declaration, we may choose adornments for any of such unadorned parameters of $P$ as a simple way to disambiguate the direction of information propagation.

Importantly, a top-level protocol declaration—one that stands alone and is ready to be enacted—must adorn all its parameters ⌜out⌝. Another way to think of this is as follows. For enactment, every parameter adorned ⌜in⌝ must have its value supplied through some other protocol, such as a message to one of the enacting agents, which would indicate that the given protocol omits relevant communications and therefore is not enactable in itself.

## 3.4 Common Specification Patterns

We now present some examples demonstrating the main concepts and typical usage of BSPL.

### 3.4.1 Duplicating a Parameter

Sometimes a role that needs to obtain a parameter binding might not be receiving it. Listing 6 shows how the originator of the binding can send a duplicate copy to another role. Because *Duplicating* has an ⌜in⌝ parameter, it is not enactable by itself. Here, we presume the ORIGINATOR produces a prior message in which aParameter is adorned ⌜out⌝.

**Listing 6: *Duplicating* a parameter.**

```
Duplicate-Parameter {
 role Originator, Consumer
 parameter in aParameter key

 Originator ↦ Consumer: share[in aParameter]
}
```

### 3.4.2 Generating an Identifier

A consequence of the information basis of BSPL is that the correctness of a protocol depends upon its keys. To facilitate composition in multiple contexts, it is convenient to define protocols that adorn an identifying parameter as ⌜in⌝, which means that such protocols cannot be enacted standalone. Listing 7 shows a simple protocol that generates an identifier, which can thus drive other protocols.

**Listing 7: Generating an identifier.**

```
Generate-Identifier {
 role Authority, Subject
 parameter out ID key

 Authority ↦ Subject: announce[out ID]
}
```

### 3.4.3 Local Parameters

A *local* parameter occurs within a protocol declaration but is not exposed in its public interface. Often, such a parameter may be essential for carrying out the desired interaction and thus would be included in underlying messages, but may not feature as an essential public interface of a protocol. Listing 8 shows a variant of *Purchase* in which the destination address is computed and used, but not deemed relevant for exposing from the overall interaction. Hence, if we were to refer to this *Purchase* variant from another declaration, we would not be able to refer to address.

**Listing 8: Variant of *Purchase* with hidden address.**

```
Purchase {
 role B, S, Shipper
 parameter in ID key, out what, out howmuch

 Order(B, S, in ID, out what, out howmuch)
 Decide-Address(B, S, in ID, out address)
 Ship(S, Shipper, in ID, in what, in address)
}
```

A local parameter must be adorned ⌜out⌝ in exactly one reference and ⌜in⌝ in all the rest. Hiding a parameter has consequences on the semantics. Because only the public parameters become part of the public interface, uniqueness applies only to tuples constructed from the public parameters. Thus, although the local parameters may take on multiple values, they would have no direct effect on the outcome as defined by the protocol. Consequently, a designer should hide only the parameters that are irrelevant to the intended outcome of the interaction, and should expose all the others.

### 3.4.4 Standing Offer

This is a common business situation where we need to generate multiple messages tied to the same standing offer. Desai et al. [2] describe such a situation in the insurance domain but, lacking a proper treatment of parameters, cannot formalize it. Once an insurance policy is created, it forms a standing offer: the insurance VENDOR would process howsoever many claims the SUBSCRIBER makes. Listing 9 shows how BSPL can naturally accommodate such a protocol.

**Listing 9: The *Insurance Claims* protocol (from [2]).**

```
Insurance-Claims {
 role Vendor, Subscriber
 parameter out policyNO key, out reqForClaim key,
     out claimResponse

 Vendor ↦ Subscriber: createPolicy[out
     policyNO, out details]
 Subscriber ↦ Vendor: serviceReq[in policyNO,
     out reqForClaim]
 Vendor ↦ Subscriber: claimService[in
     policyNO, in reqForClaim, out
     claimResponse]
}
```

Each claim refers to a unique policy and has a unique response; one policy may lead to multiple claims. Hence, we make policyNO and reqForClaim jointly the key. If necessary, we can include additional parameters to describe the policy, including its termination, in greater detail. The remaining protocols given by Desai et al. [2] involve simpler structures, such those demonstrated in the preceding sections.

## 3.5 Subtle Specification Patterns

The following patterns demonstrate the power of BSPL.

### 3.5.1 Flexible Sourcing of out Parameters

Listing 10 shows *Buyer or Seller Offer*, in which either the BUYER or the SELLER may generate the price. This protocol illustrates the distinction between a parameter being endogenous to a protocol versus being generated by one or another of the agents playing its roles. *Buyer or Seller Offer* involves two variants of *rfq* and *quote*, with differences in adornments of their parameters. We overload the message names since informally the names relate to meaning: the same commitment would be associated with a *quote* whether the price was ⌜in⌝ or ⌜out⌝ in it. We could equally well use different names. Notice that the interface of *Buyer or Seller Offer* is the same as that of *Offer* (Listing 2) since it has the same roles and parameters (with the same adornments).

**Listing 10: The *Buyer or Seller Offer* protocol.**

```
Buyer-or-Seller-Offer {
 role Buyer, Seller
 parameter in ID key, out item, out price

 Buyer ↦ Seller: rfq[in ID, out item, nil price]
 Buyer ↦ Seller: rfq[in ID, out item, out price]

 Seller ↦ Buyer: quote[in ID, in item, out
     price]
 Seller ↦ Buyer: quote[in ID, in item, in price]
}
```

In Listing 10, both *quote* variants rely upon the BUYER having provided item. As a result, the BUYER speaks first. The BUYER may announce the price or not, by choosing the appropriate variant of *rfq*. The two variants of *rfq* are mutually exclusive because they have incompatible adornments for price: thus at most one of them can be sent. Likewise, the two variants of *quote* are mutually exclusive. In essence, the choice is the BUYER's and the SELLER follows along. This is the reason we introduce the ⌜nil⌝ adornment on price in *rfq*. Upon receiving a ⌜nil⌝ price, the SELLER would not be able to send *quote* without generating the price locally.

### 3.5.2 in-out Polymorphism

Let us consider *Flexible Offer*, which can apply both where the price is exogenous (supplied) and where it is endogenous (computed by the protocol). We do so by omitting the adornment on price. Then, as Listing 11 shows, we need to provide alternatives so that each of the possible adornments of price is enactable.

If a reference to *Flexible Offer* adorns price ⌜in⌝, the only possible enactment is when B sends an *rfq* specifying the price to S, who responds with a *quote*. Alternatively, if a reference adorns price ⌜out⌝, B must send an *rfq* without specifying the price to S, who responds with a *quote* that specifies the price. That is, price is determined either from the reference (when it is referenced as ⌜in⌝) or by S (when it is referenced as ⌜out⌝). And, qID helps ensure that the enactment remains incomplete until *quote* occurs.

**Listing 11: *Flexible Offer*: price as in or out.**

```
Flexible-Offer {
 role B, S
 parameter in ID key, out item, price, out qID

 B ↦ S: rfq[ID, out item, nil price]
 B ↦ S: rfq[ID, out item, in price]

 S ↦ B: quote[ID, in item, out price, out qID]
 S ↦ B: quote[ID, in item, in price, out qID]
}
```

Listing 12 defines *Offer* as a simple variant of *Flexible Offer*. It illustrates a way to restrict the adornments of parameters without changing the logical structure of protocols. Even if we do not adorn the parameters of *Flexible Offer* as referenced from within *Offer*, there is no ambiguity, because those parameters are adorned in the declaration of *Offer*. Thus, when *Flexible Offer* is enacted, the parameters would be treated as in the declaration of *Offer* (qID is not used).

**Listing 12: *Offer* as a restriction on *Flexible Offer*.**

```
Offer {
 role Buyer, Seller
 parameter in ID key, out what, out howmuch

 Flexible-Offer(Buyer, Seller, in ID, out what,
     out howmuch, out qID)
}
```

## 3.6 Specification Patterns Hinting at Meaning

These patterns demonstrate connections with meaning.

### 3.6.1 Forwarding a Copy

Listing 13 shows a simple protocol that can be used to forward a parameter from one role to another. This protocol is often needed to help make a protocol enactable where a necessary parameter binding would not otherwise be known to a specified role. Notice that the functioning of this protocol relies upon meaning, namely, to ensure that the value of copy equals the value of original.

**Listing 13: Forwarding a parameter value.**

```
Forward {
 role From, To
 parameter in original key, out copy

 From ↦ To: forward[in original, out copy]
}
```

### 3.6.2 Mixed Initiative

Listing 14 shows a protocol that supports either role taking the initiative. This protocol is inspired by the formalization of the Enhanced NetBill by Yolum and Singh [12]. In this protocol, the BUYER and the SELLER can exchange as many messages as they like with the SELLER repeatedly sending *quote* messages and the BUYER *accept* messages. Each of them has the initiative and can work independently of the other. If necessary, we can combine mixed initiative with polymorphism, as introduced in Section 3.5.2.

**Listing 14: The *Mixed Initiative Offer* protocol.**

```
Mixed-Initiative-Offer {
 role B, S
 parameter in ID key, out qID key, out aID key
     out qItem, out qPrice, out aItem, out aPrice

 S ↦ B: quote[in ID, out qID, out qItem, out
     qPrice]
 B ↦ S: accept[in ID, out aID, out aItem, out
     aPrice]
}
```

The meaning layer would capture that the quoted and accepted items and prices are equal. Each message would correspond to the creation of a suitable commitment to provide a specified item if paid a specified amount and to pay a specified amount if provided a specified item. Assuming each party needs a commitment from the other in order to proceed, progress will occur only when they produce their respective commitments for the same item and price. This example shows that though BSPL captures the operational aspects in a declarative manner, it seeks neither to obstruct appropriate meaning nor to substitute for meaning.

### 3.6.3 Digressions

Yolum and Singh [12] introduced the idea of a digression where an agent may interact differently from a protocol for some steps but later resynchronize with it. Digression applies primarily to enactments rather than to protocols, although it facilitates the refinement of protocols. BSPL naturally supports digression. As long as the parameter adornments are satisfied, a digression has no impact on the enactment of a protocol. Digressions in the sense of Yolum and Singh depend upon a notion of meaning.

# 4. SKETCH OF A SEMANTICS FOR BSPL

We give an account of how BSPL protocols may be enacted and how to determine their distributed enactability using some mathematical concepts but, for brevity, without any mathematical notation.

A protocol describes an interaction by specifying messages to be exchanged between specific roles, and by imposing a partial order on the messages. An enactment of a protocol involves each of its roles being adopted by an agent, and the agents exchanging messages that the protocol specifies. Therefore, we capture the semantics of a protocol in terms of the enactments it allows. A message instantiates a message schema and is precisely described by its name, sender, receiver, and bindings for each of its parameters.

We define a *history* of a role as a sequence of messages, in each of which the role is either the sender or the receiver. Thus the history captures the local view of an agent who might adopt the role during the enactment of a protocol.

We define a *history vector* for a protocol as a vector each of whose elements is the history of a role mentioned in the protocol. A history vector is *quiescent* provided every message present in a sender's history is also present in its receiver's history. The fundamental causality constraint of distributed computing applies: a (receiving) role's history may contain a message reception only if the (sending) role's history contains the corresponding message emission [8]. However, we need a more sophisticated treatment of causality that captures the nature of parameters in BSPL.

The history of a role maps naturally to its local state. Notice we are interested in the local view of the public interactions, *not* in the internal state of an agent playing this role. Each message sent or received progresses the local state of the role, expressed in terms of the bindings of the parameters that the role knows. Specifically, a message emission is *viable* for a role if the role knows the bindings of all ⌜in⌝

parameters in that message and does not know the bindings of the ⌜out⌝ and ⌜nil⌝ parameters. In essence, it must produce the bindings for the ⌜out⌝ parameters, which it then knows (for future messages). A message reception is always viable and changes the state of the knowledge of the role, affecting the viability of future messages. A history vector is *viable* provided it arises from viable message emissions and receptions by the roles, i.e., by growing their local histories.

Informally, the *intension* of a protocol is given by the set of quiescent viable history vectors that enact it. The intension for a message is the set of quiescent viable history vectors in which it occurs. The intension of a composite protocol is the set of all viable interleavings of the history vectors in the intensions of its references. Only a protocol with an empty set of public ⌜in⌝ parameters may be enacted.

To understand when an enactment is correct, consider two references that occur within the same declaration and involve one or more common parameters, and consider their respective adornments of such a common parameter.

out–in indicates an ordering conflict: a message with ⌜out⌝ (even if nested in a reference) must precede, and the binding must propagate, to a message with ⌜in⌝.

nil–in **or** nil-out indicate a knowledge conflict and as such only apply to the same role: once a role sends or receives a message with ⌜out⌝ or ⌜in⌝, it cannot send a message with ⌜nil⌝.

out–out indicates an occurrence conflict: at most one of the references may occur anywhere in the system.

Our semantics addresses ordering conflicts through causality and knowledge conflicts through each role's view. For occurrence conflicts, there is no general solution, but we can analyze a BSPL specification to make sure that the same role controls which of the conflicting references occurs.

# 5. EVALUATION: CASE STUDY

We consider foreign exchange transactions, as formalized by Desai et al. [1]. *Bilateral Price Discovery* or *BPD* involves a TAKER sending a *priceRequest* to a MAKER, who responds with a *priceResponse*. Each message specifies a number of parameters, which for clarity we reduce to two parameters: query and result. Listing 15 shows the strikingly simple BSPL formalization of *BPD*.

**Listing 15: The *Bilateral Price Discovery* protocol.**

```
BPD {
 role Taker, Maker
 parameter out reqID key, out query, out result

 Taker ↦ Maker: priceRequest[out reqID, out
     query]
 Maker ↦ Taker: priceResponse[in reqID, in
     query, out result]
}
```

Desai et al. identify constraints under which a *priceResponse* message may not occur. These complicate Desai et al.'s specification, but Listing 15 captures them naturally: (1) "No priceRequest with a matching reqID has happened" (BSPL: adorn reqID (and query) as ⌜in⌝ on *priceResponse* and as ⌜out⌝ on *priceRequest*); (2) "A priceResponse with identical parameters has happened" (BSPL: automatic since repetitions are superfluous); (3) "A priceResponse with the same ID but a different result ... is happening simultaneously" (BSPL: mark reqID as a key); and (4) like #3 above but for other messages (BSPL: handle as above).

Consider Desai et al.'s [1] discussion of *multilateral price discovery (MPD)*, in which a TAKER interacts with a MAKER via an intermediary EXCHANGE. Intuitively, it makes sense that MPD is a composition of BPD with itself. Odell et al. [10] informally discuss the concept of *nesting* in AUML, wherein an agent playing a role in one protocol may participate in additional protocols in the middle. In Odell et al.'s terms, the EXCHANGE would be a MAKER in one copy of BPD and nest the second copy of BPD. There are two shortcomings with Odell et al.'s nesting. First, it draws a false hierarchy between two protocols, placing one as subservient to the other, whereas the interactions are conceptually peers. Two, and more fundamentally, nesting is a matter of how an agent is implemented. For all that anyone knows, even in the plain BPD, a MAKER might be shopping for deals in the background, possibly acting as a TAKER in another copy of BPD. But such internally driven behaviors are not public interactions and thus are not part of the given protocol.

Desai et al. [1] offer a better solution than nesting by explicitly composing BPD with itself to produce MPD. They assert data flow axioms whereby a query parameter in one copy of *priceRequest* is passed to the second copy, and likewise in the reverse direction for the result from *priceResponse*. However, Desai et al.'s approach violates encapsulation: it opens up each copy of BPD so as to enable stating constraints on the constituent messages of each copy in order to compose them as desired.

In BSPL, MPD can be expressed in a remarkably simple manner. Listing 16 uses in-out polymorphism (Section 3.5.2) to define a *Generalized BPD* or GBPD, in which query and res are not adorned. We can produce a specification of BPD equivalent to Listing 15 exactly as Listing 12 defines *Offer*.

**Listing 16: *Generalized Bilateral Price Discovery*.**

```
GBPD {
 role T, M
 parameter reqID key, query, res

 T ↦ M: priceRequest[out reqID, out query]
 T ↦ M: priceRequest[in reqID, in query]

 M ↦ T: priceResponse[in reqID, in query, out
     res]
 M ↦ T: priceResponse[in reqID, in query, in
     res]
}
```

Next, Listing 17 specifies MPD as an almost trivial composition of GBPD with itself. The adornments of the parameters in the two references to GBPD are different, and ensure that the composition is correct. Notice that the encapsulation is not broken (the GBPD declaration is not revealed here) and we are not specifying the internals of any role.

**Listing 17: *Multilateral Price Discovery*.**

```
MPD {
 role Taker, Exchange, Maker
 parameter out reqID key, out query, out res

 GBPD(Taker, Exchange, out reqID, out query, in
     res)
 GBPD(Exchange, Maker, in reqID, in query, out
     res)
}
```

# 6. DISCUSSION

What do we gain from an interaction-oriented approach wherein protocols are first-class entities? Although an agent-oriented approach, which focuses on the roles, is more familiar, it limits the modeling unnecessarily. By focusing on interactions, we can capture constraints from a public, as opposed to a role, perspective. In particular, when roles are introduced during composition, such new roles would automatically view any relevant constraint as satisfied.

Although we suppress the means clauses above, BSPL is geared toward providing the undergirding for any effective treatment of meanings. Listing 18 shows an example based on Listing 9 to give the reader a flavor of a means clause. Here C indicates a commitment [2, 12], and the expression states that the VENDOR commits to providing claim service to the SUBSCRIBER whenever the SUBSCRIBER sends a request under the specified claim. The benefit of BSPL here is that the operational basis for the meanings in terms of causality and information is taken care of automatically.

**Listing 18: Meaning for *Insurance Claims*.**

```
Insurance-Claims { ...

 Vendor ↦ Subscriber: createPolicy[out
     policyNO] means C(Vendor, Subscriber,
     serviceReq[policyNO, reqForClaim],
     claimService[policyNO, reqForClaim,
     claimResponse])
}
```

## 6.1 Literature

Increasing recognition of the importance of interaction has led to work on *choreographies* [11], which too capture the operational aspects of protocols as studied here. However, a choreography is typically specified procedurally, usually in a language such as message-sequence charts (MSCs) [6] or an analogous notation, such as WS-CDL [11].

AUML [10] is an important notation for protocols (many of its features were assimilated into UML 2.0). AUML's sequence diagram notation takes a strong procedural stance for describing interactions. Thus, it emphasizes explicit constraints on how messages are ordered. In contrast, our parameter adornments force clarification of the arrow of causality, making it correspond to the flow of information.

Recently, Miller and McGinnis [9] proposed $\mathcal{RASA}$, a language for protocols based on the proposition dynamic logic. Some of this language refers to agent reasoning and some to interaction. BSPL can capture the latter parts of it. In particular, in BSPL, iteration arises from the possible bindings of a protocol's parameters, and is limited only by the size of the cross-product of the domains of the key parameters. And, our semantics limits choice to guarded choice. $\mathcal{RASA}$ describes first-class protocols, i.e., those that an agent can inspect and reason about. BSPL, in addition, treats protocols as a first-class modeling concept for ready composition.

Desai and Singh [4] identify several challenges to the enactability of a protocol. BSPL avoids all the ordering problems they identify as varieties of *blindness*, because the only way to capture an ordering constraint in BSPL is to do so in a causally sound way: from a reference with an ⌜out⌝ adornment of a parameter to a reference with an ⌜in⌝ adornment of the same parameter. The problematic enactments cannot arise. The well-known problem of *nonlocal choice* [7] arises when correct behavior by a role depends on actions of another role. BSPL does not automatically avoid nonlocal choice. However, we can analyze a BSPL specification to determine that it is not at risk of nonlocal choice.

Traditional work on service composition primarily considers orchestrations where a conceptually central party controls two or more services. A strength of this work lies in its formalization of service behaviors and in its use of planning and constraint reasoning to construct appropriate service compositions. Although our present setting is quite different, we imagine that many of the techniques of service composition may be expanded and applied in our setting.

## 6.2 Future Work

A useful direction would be enhancing the treatment of the information model. For instance, it might be appropriate to entertain multiple keys for a protocol. Further, it would be useful to understand how important properties such as enactability may be verified in a compositional manner. Some natural extensions to BSPL that we will be considering include (1) a principled treatment of multicast, where multiple agents playing the same role receive a message and (2) accommodating discovery protocols, where the roles are bound late during enactment.

## Acknowledgments

This work was partially supported by the OOI Cyberinfrastructure program, which is funded by NSF contract OCE-0418967 with the Consortium for Ocean Leadership via the Joint Oceanographic Institutions. Thanks to Matthew Arrott, Amit Chopra, and Kohei Honda for helpful discussions.

# 7. REFERENCES

[1] N. Desai, A. K. Chopra, M. Arrott, B. Specht, and M. P. Singh. Engineering foreign exchange processes via commitment protocols. In *SCC*, pp. 514–521, 2007.

[2] N. Desai, A. K. Chopra, and M. P. Singh. Amoeba: A methodology for modeling and evolution of cross-organizational business processes. *ACM TOSEM*, 19(2):6:1–6:45, Oct. 2009.

[3] N. Desai, A. U. Mallya, A. K. Chopra, and M. P. Singh. Interaction protocols as design abstractions for business processes. *IEEE TSE*, 31(12):1015–27, 2005.

[4] N. Desai and M. P. Singh. On the enactability of business protocols. In *AAAI*, pp. 1126–1131, 2008.

[5] R. Elmasri and S. Navathe. *Fundamental of Database Systems*. Benjamin Cummings, second edition, 1994.

[6] ITU. Formal description techniques (FDT)—Message Sequence Chart (MSC). Document Z.120, Apr. 2004.

[7] P. B. Ladkin and S. Leue. Interpreting message flow graphs. *Formal Aspects Comput.*, 7(5):473–509, 1995.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[9] T. Miller and J. McGinnis. Amongst first-class protocols. In *ESAW 2007*, *LNCS* 4995, pp. 208–223. Springer, 2008.

[10] J. Odell, H. V. D. Parunak, and B. Bauer. Representing agent interaction protocols in UML. In *AOSE 2000*, *LNCS* 1957, pp. 121–140. Springer, 2001.

[11] WS-CDL. Web Services Choreography Description Language. www.w3.org/TR/ws-cdl-10/, Nov. 2005.

[12] P. Yolum and M. P. Singh. Commitment machines. In *ATAL 2001*, *LNAI* 2333, pp. 235–247. Springer, 2002.