Bliss: Specifying Declarative Service Protocols

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

***Abstract*—BSPL, the Blindingly Simple Protocol Language, is a recent approach for declaratively expressing service communication protocols that involves only two main constructs: a way to specify a message as an atomic protocol and a way to compose protocols. BSPL supports the Local State Transfer architectural style for decentralized service enactment. BSPL offers significant gains in expressing protocols (i.e., specifications) that decouple participants in service engagements (i.e., agents) as much as possible given the causal constraints induced from the information exchanged by them. Importantly, BSPL relies exclusively on how appropriate information flows are induced from the specification. This paper proposes *Bliss*, a conceptual model for interaction that is based on information flow. The idea behind Bliss is to incrementally develop the information needed to complete the social object that a protocol computes. Bliss yields simple steps to help ensure that the resulting protocol adequately captures the given requirements with respect to the social object.**

***Index Terms*—Business process modeling; Business protocols**

I. Introduction

We take as our point of departure the recently proposed Blindingly Simple Protocol Language—or *BSPL*, for short [1], [2], [3]. BSPL is an approach for interaction-oriented programming: it grants first-class status to protocols as specifications of interactions among roles. The key features of BSPL for our present purposes are that it (i) provides a simple functional syntax through which a protocol can be referenced from another protocol; (ii) gives central status to the parameters of a protocol distinguishing between parameters that the protocol assumes as inputs (to instantiate its interactions) and those it computes as outputs (from its interactions). Importantly, BSPL makes the entire causal nature of a distributed enactment explicit in the parameter bindings, dispensing with conventional control flow features such as sequencing and choice.

Singh [1] previously showed how BSPL can capture a number of patterns simply for which previous approaches are complex and over-specified. Moreover, BSPL composes protocols in a manner that preserves encapsulation in settings where traditional approaches fail to do so. Singh [2] proposed a new architectural style called *Local State Transfer* or *LoST*, which supports purely distributed enactments and thus realizes the intuitions of BSPL. These useful properties make BSPL a valuable topic for further investigation from the standpoint of engineering multiagent and service-oriented systems.

An important intuition behind BSPL is that protocols ought to be enacted in a purely distributed manner wherein the agents playing various roles exchange messages asynchronously. The constraints on enactment reflect the essential causality of message flow and the knowledge locally available to the agents by which they determine the contents of their messages. The protocols should be enactable by myopic agents, who work based on what they happen to know at a given moment, and do not look ahead. Further, there are no hidden communication channels. Despite such flexibility, the enactment should produce consistent results. In particular, different agents ought not to interfere with each other. Hence, a protocol should be specified such that invalid enactments are not possible.

A. Contributions

Previous work on BSPL shows how to verify and enact BSPL protocols. However, it does not address the challenge of developing valid BSPL protocols. Given the novelty of BSPL with respect to traditional software engineering and service computing, it is crucial that we develop a suitable methodology for BSPL: it simply would not be feasible to adapt any of the existing methodologies (e.g., for AUML or commitment protocols), which all lack BSPL's information focus.

Accordingly, this paper proposes *Bliss*, a methodology for specifying BSPL protocols. Bliss begins from a significant observation regarding BSPL, namely, that although it does not deal directly with the realm of meanings of interactions, it is designed with an emphasis on how to support perspicuously representing meanings of interactions. The idea is that by removing all control flow constructs, BSPL renders communication purely declaratively. Thus meanings can be associated with each message without regard to where it might occur in the protocol or the ensuing enactment. A BSPL protocol enactment fleshes out a notional social object that encapsulates the progressing social state comprising the meanings of the messages exchanged by agents enacting the given protocol.

The key idea behind Bliss is to describe how to incrementally develop the information needed to complete that social object. The main ingredients of Bliss are (i) a new conceptual model overlaying BSPL; (ii) steps that help systematically specify a protocol that would compute that social object; and (iii) steps that ensure that the protocol is fully general given the top-level requirement to compute that social object.

The main benefits of Bliss are that it better captures interaction requirements and helps avoid errors and over-constrained solutions that traditional approaches might produce. We demonstrate Bliss via the well-known NetBill protocol and evaluate it via a Service Request protocol that is specified using UML sequence diagrams and used in a real-life service computing effort for cyberinfrastructure.

B. Organization

The rest of this paper is organized as follows. Section II follows Singh [1] in describing the essential background on BSPL. Section III introduces the new conceptual model for BSPL that underlies Bliss and helps organize BSPL parameters according to their functions in a protocol specification. Section IV introduces the Bliss methodology. Section V introduces scientific collaboration services as a case study. Section VI evaluates the benefits of Bliss on an independently described real-life service protocol. Section VII concludes with a discussion of the literature and important future directions.

II. Background and Running Example

We use the *NetBill* protocol [4], [5] to illustrate our approach (writing the message names *slanted*). This protocol begins with a customer requesting a quote for some digital goods from a merchant (*rfq*). The merchant sends an *offer* (quote), which the customer may *accept*. If the customer accepts the offer, the merchant delivers the *goods*. The customer then *pays*. Upon receiving the payment, the merchant sends a *receipt* to the customer. The protocol specifies a (linear) sequence of steps, as previously critiqued, e.g., [5], because it limits the flexibility of the agents enacting the protocol.

Listing 1 illustrates the main features of BSPL. For readability, in the listings, we write reserved keywords in sans serif, and capitalize role names. 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]⌝.

Listing 1. The *NetBill* protocol.

```
NetBill Original {
 role C, M
 parameter out ID key, out item, out price, out
      done
 private confirmation, document, payment

 C ↦ M: rfq[out ID, out item]
 M ↦ C: offer[in ID, in item, out price]
 C ↦ M: accept[in ID, in item, in price, out
      confirmation]
 M ↦ C: goods[in ID, in item, in confirmation,
      out document]
 C ↦ M: pay[in ID, in price, in document, out
      payment]
 M ↦ C: receipt[in ID, in item, in payment, out
      done]
}
```

BSPL distinguishes three main adornments on the parameters of a message: ⌜in⌝, meaning the binding must come from some other message; ⌜out⌝, meaning that the binding originates in this message (presumably based on private computations of the sender); and ⌜nil⌝, meaning that no binding is known to the sender at the time of emission. The ⌜out⌝ binding indicates declarative force [6]; e.g., an agent sending a price quote is not merely reporting a price but declaring it to be the *definitive* price in this protocol enactment (instance).

Each message instance must bind a proper value for each ⌜in⌝ and each ⌜out⌝ parameter, and a ⌜nul⌝ value for each ⌜nil⌝ parameter. (The private line is syntactic sugar to help catch typographical errors in parameter names by making sure that all parameters used in the protocol are listed explicitly.) A parameter adorned ⌜out⌝ in message m_1 and ⌜in⌝ in message m_2 creates a causal dependency from m_1 to m_2.

We introduce a key parameter ID propagated to each message: this improves on the existing formalizations [4], [5] by clarifying the information objects exchanged by the participants. The first message, *rfq*, has ID adorned ⌜out⌝ to initiate the transaction; the remaining messages have it adorned ⌜in⌝. The messages generate additional information through the ⌜out⌝ parameters item, price, confirmation, document, payment, and receipt, respectively. In *NetBill*, the messages are causally chained so only a sequential enactment is possible.

An enactment corresponds to a binding of public parameters. BSPL requires some of the parameters being declared as forming the *key* to ensure that multiple concurrent enactments of the same protocol would not interfere with each other. Every protocol and message must have a key: for brevity, by default, the key of a message equals the set of the protocol's key parameters that feature in it. An enactment is *complete* when all its public ⌜in⌝ and ⌜out⌝ parameters are bound. Specifically, an enactment of NetBill completes when ID, item, price, done are bound. Its private parameters are not relevant for judging completion.

Formal Syntax

The following BSPL syntax and explanations are simplified from Singh [1]. 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 omit cardinality restrictions and parameter types.

L$_1$. A protocol declaration consists of a name, two or more roles, one or more parameters, and one or more references to constituent protocols or messages. The parameters marked key together form this declaration's key.
$\textit{Protocol} \longrightarrow \textit{Name}\ \{\ \textsf{role}\ \textit{Role}^{+}\ \textsf{parameter}\ \lfloor \textit{Parameter}\lfloor\textsf{key}\rfloor\rfloor^{+} \textit{Reference}^{*}\ \}$

L$_2$. A reference to a protocol (from a declaration) consists of the name of the protocol appended by as many roles and parameters as it declares. At least one parameter of the reference must be a key parameter of the declaration in which it occurs.
$\textit{Reference} \longrightarrow \textit{Name}\ (\ \textit{Role}^{+}\ \textit{Parameter}^{+}\ )$

L$_3$. Alternatively, a reference is a message schema, and consists of exactly one name, exactly two roles, and one or more parameters (at least one of which must be a key parameter).
$\textit{Reference} \longrightarrow \textit{Role} \mapsto \textit{Role} : \textit{Name}\ [\ \textit{Parameter}^{+}\ ]$

L$_4$. Each parameter consists of an adornment and a name.
$\textit{Parameter} \longrightarrow \textit{Adornment}\ \textit{Name}$

L$_5$. An adornment is usually either ⌜in⌝ or ⌜out⌝; ⌜nil⌝ indicates an unknown parameter; ⌜opt⌝ (occurs only in a declaration) indicates optionality: could be ⌜nil⌝ or ⌜out⌝.
$\textit{Adornment} \longrightarrow \textsf{in} \mid \textsf{out} \mid \textsf{nil} \mid \textsf{opt}$

III. Bliss: Conceptual Model

BSPL's parameters capture all that is essential to a protocol's semantics. The focus on parameters leads to a clear protocol semantics that enhances flexibility of enactment. Previous works [3] provide the formal semantics and a tool for BSPL but did not provide a conceptual model for how BSPL can capture interactions as distributed computations. This section provides a simple yet effective conceptual model underlying Bliss that lends structure to how parameters are applied in BSPL, i.e., based on the parameters' functions in BSPL. The benefit of the conceptual model is to help us formulate protocols that are correct: in essence, to validate a protocol with respect to requirements without over-constraining it.

Our proposed conceptual model places BSPL parameters into five major categories. Recall that a BSPL protocol refers both to a composite protocol and an atomic protocol (i.e., an individual message). The purpose of a BSPL protocol is, when enacted, to compute a social object. And the computation is entirely driven by the information parameters carried by messages emitted and received by the agents enacting a protocol. The parameters, therefore, capture conceptually distinct though related purposes. Some parameters (i) characterize the identity of the social object; (ii) some describe its ultimate meaning, which falls outside BSPL's scope; (iii) some help realize the computation despite the decentralized and myopic decision-making by the agents; and (iv) some ensure that the integrity of the social object is preserved.

a) Key parameters: These capture the notion of conceptual object identity that is essential to BSPL. In many commonly occurring protocols, which represent a single interactive transaction, the key would be a singleton. However, a protocol can potentially represent an aggregation or association of multiple independent transactions, in which case the key would have as many parameters as there are independent transactions. In a protocol that initiates a business transaction, the key parameters must be ⌜out⌝. If a key is ⌜out⌝, the remaining parameters cannot be ⌜in⌝—there being no basis for their bindings. The key of a message occurring in a protocol must not be disjoint with the key of the protocol.

b) Payload parameters: These feature in the meanings associated with the protocol, which correspond to transitions in the social state of the parties enacting the protocol. For example, a *quote* may signify that the sender (a merchant) is creating a commitment to sell the specified item for the specified price: the item description and price must be described by the payload of the price quote message. Likewise, an *accept* may signify creating a commitment to pay the specified price for the specified item. The above examples happen to involve commitments. In other applications, other social constructs may be relevant. For example, a party *invitation* may create an expectation of "friendship," viewed as a social construct.

c) Completion parameters: These determine when a protocol completes. A general rule is that we need to ensure that some parameters are adorned ⌜out⌝: such parameters indicate that an enactment must involve a definitive action by (an agent playing) one of the roles, thereby exercising its autonomy. In this manner, only when the appropriate completion parameters are bound can we interpret the payload parameters correctly as characterizing a transition in the social state of the participants. For example, assuming the item description and price are already bound, the mere existence of bindings for these parameters does not signify that the merchant or customer, respectively, make or accept an offer. That is, we must include a parameter with an ⌜out⌝ adornment that signifies that the sender autonomously conveys the appropriate meaning. However, if one of the payload parameters is adorned ⌜out⌝, there is no need for a separate completion parameter.

The completion parameters for a composite protocol depend on when the corresponding social object is completed. For an individual message, since the meaning would be simpler, it is usually a matter of verifying whether the payload parameters are adequate and, if not, inserting a single ⌜out⌝ completion parameter. In essence, completion corresponds to the declaration or *illocution* [6] that the specified element of the social object is complete. This declaration would occur because of the actions of one participant, who generates the corresponding element. This declaration may in addition *constitute* [7] a request for quotes at the application level and may potentially create a commitment, or presume a suitable commitment (or other relevant relationship) already exists.

d) Integrity parameters: These ensure that the social state of the entire enactment is not corrupted. For example, we would ensure that a customer doesn't both accept and reject an offer by using an integrity parameter that would be adorned ⌜out⌝ in both *accept* and *reject*, thereby ensuring that no more than one of them occurs. Likewise, if there were two ways in which the merchant could offer to sell an item for a price, we would have to make sure that no more than one of those ways could be realized in the same enactment. The same situation holds when two or more agents are involved. The verification challenges are harder in such a case—to ensure that different agents acting myopically do not violate the integrity of the social object. In effect, the integrity parameters capture constraints on the mutual nonoccurrence of messages that would violate an integrity property.

e) Control parameters: These capture properties for which there is no direct justification based solely on the payload. In effect, these parameters encode ordering properties. In general, such parameters arise only because a modeler attempts to force fit an existing over-constrained protocol into BSPL. Such gratuitous control parameters can also include what would normally be integrity parameters, simply as a way of preventing two messages from both being sent.

The above kinds of parameters apply both to a protocol as a whole and to the messages within it. A protocol might only need the first three kinds, which are the most important from the standpoint of describing the state of a social object. Integrity parameters are needed only when there is the possibility of the social state being corrupted. Control parameters would not be used unless there is a need to constrain computations beyond what is needed to compute the social object.

TABLE I
BLISS METHODOLOGY: MAIN STEPS. BLISS ITERATES OVER THESE STEPS.

Step	description	Input	Output Artifact
1	Identify the roles needed in a protocol	Interaction purpose	Roles
2	Identify the conceptual social object computed	Interaction purpose	Protocol parameters
3	Identify the messages (or, recursively, subprotocols) to compute the social object	Roles	Conceptual schema
4	Identify each message as a component of the social object and any additional constraints	Messages	Message parameters
5	Introduce polymorphism of messages to support flexible sourcing of parameter bindings	Messages	Protocol

IV. BLISS: METHODOLOGY

We now describe the Bliss methodology for specifying a BSPL protocol. To specify any software artifact requires identifying the stakeholders and understanding their requirements. In the case of a protocol to be enacted by autonomous agents, the natural stakeholders are the business partners whose agents would enact the protocol being specified. Not all stakeholders need be present at design time but the requirements of the business partners ought to be accounted for—or else, they would not adopt roles in, and enact, the protocol.

Table I summarizes Bliss. The first two steps of Bliss call for a collection and analysis of the requirements by the protocol designers: first, to determine the roles needed in the protocol and, second, to determine the social object that enactments of the protocol would compute. If we could magically compute the social object, the problem would be solved. However, in a distributed system, the social object must be computed incrementally through the emission and reception of messages among the participants playing roles, respecting the semantic constraints of causality and integrity, as in BSPL. The subsequent steps flesh out the remaining elements of the BSPL conceptual model to ensure that a valid enactment according to BSPL semantics would produce the target social object.

Let us illustrate Bliss by exercising the steps of Table I on *NetBill* to produce our improved variant of it.

1) Roles: CUSTOMER (C) and MERCHANT (M).

2) Protocol Parameters: We classify these as follows. (i) Keys: we arbitrarily choose a parameter ID to serve as the protocol key, reflecting the business transaction it carries out. (ii) Payload: We choose parameters item and price as the payload, since they correspond to the meaning we associate with the protocol, i.e., its business transaction. (iii) Completion: We determine what constitutes the completion of a *NetBill* protocol enactment. That is, what are the components of the conceptual object whose instances will be computed by enactments of the *NetBill* protocol? This step represents the essence of the information-centric conceptualization of BSPL. Clearly, the item and price would have been settled and, respectively, delivered and paid. A receipt would have been provided by the MERCHANT.

3) Conceptual Schema: Figure 1 shows a simple conceptual schema, omitting the parameters on the messages. The messages are as named in Section II. We propagate the same key to each message within the protocol, since the entire protocol carries out one business transaction. Since each message must occur no more than once in any business transaction, we need no additional key parameter on any of the messages.

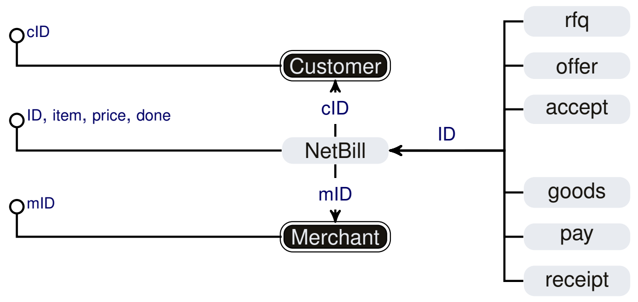

Fig. 1. A conceptual schema for *NetBill*. Edge direction indicates a foreign key dependence. A dark background signifies a runnable entity with double boxes identifying the roles; a light background signifies a conceptual social object, i.e., a protocol (including a message). The parameters shown for each entity (including those on the edges emanating from it) jointly form its key.

4) Message Parameters: We consider each message in turn.

- *rfq*. Since *rfq* initiates the business transaction, it includes the protocol's key parameter, ID, with an adornment of ⌜out⌝. Also, *rfq* needs to specify the item that the CUSTOMER wishes to purchase. This leads to a parameter that describes the item, and which we call item. Moreover, the CUSTOMER determines the item, so item is adorned ⌜out⌝.
- *offer*. For an *offer* to be meaningful, it must specify the price the MERCHANT is offering and for what item. The price is provided definitively by the MERCHANT, hence price is adorned ⌜out⌝. In *NetBill*, since *offer* and the remaining messages do not introduce a new business transaction, they all use the protocol's key parameter, ID, with an adornment of ⌜in⌝. Here, item is ⌜in⌝ since *rfq* specifies it.
- *accept*. The CUSTOMER accepts an offer with the price as specified in the offer. To capture the CUSTOMER's autonomy in making the illocution, we adorn confirmation as ⌜out⌝.
- *goods*. The MERCHANT provides goods and thus definitively specifies what they are. In *NetBill*, the goods are treated as digital artifacts, which instantiate the item description employed in *rfq* and subsequent messages. Hence, *goods* includes a parameter document that is adorned ⌜out⌝.
- *pay*. The CUSTOMER provides and definitively specifies the funds. (More realistically, the CUSTOMER provides a digital certificate that the MERCHANT can encash for funds.) Hence, *pay* includes a parameter payment adorned ⌜out⌝.
- *receipt*. The MERCHANT definitively specifies that the pay-

ment has been received from the CUSTOMER. Hence, *receipt* includes a parameter done that is adorned ⌜out⌝.

Message parameters not featuring in the protocol declaration are private, indicating they don't contribute to the social object.

5) Polymorphism: This can apply to each message; we discuss some of Yolum and Singh's [5] variations for illustration.

- For an *offer* to mean the creation of a commitment, it must specify the price and item. However, as Yolum and Singh [5] explain, item need not come from the CUSTOMER—e.g., to support unsolicited offers. Thus, we introduce a schema of *offer* that adorns item as ⌜out⌝. However, because of integrity, we cannot have two bindings for item for the same ID; because *rfq* and *offer* have different senders, the protocol would fail safety [3]. Therefore, the new schema specifies ID as ⌜out⌝, which would ensure a unique binding for ID by initiating a fresh transaction for an unsolicited offer.
- For *accept*, we introduce a schema with ⌜out⌝ price. As explained above, this schema must adorn ID as ⌜out⌝ and, therefore, also item as ⌜out⌝. This schema needs confirmation only because that parameter is needed for *goods*.
- For *goods*, the MERCHANT can provide the goods prior to receiving a message carrying the confirmation. Hence, we introduce a schema with ⌜nil⌝ confirmation. In the same spirit as above, it might make business sense to allow the MERCHANT to open a new transaction, i.e., with ⌜out⌝ ID, ⌜out⌝ item, and ⌜nil⌝ confirmation by providing the goods.
- For *pay*, the CUSTOMER merely needs to know the price to pay and can do so prior to receiving the document. Hence, we introduce a schema with ⌜nil⌝ document.

We decide not to allow *receipt* to occur without a prior *pay*. Listing 2 shows the resulting specification in BSPL. It retains Listing 1's message schemas: hence the enactments supported originally remain available.

Listing 2. *NetBill* protocol via Bliss (Partial).

```
NetBill Bliss Simple {
 role C, M
 parameter out ID key, out item, out price, out
     done
 private confirmation, document, payment

 C ↦ M: rfq[out ID, out item]

 M ↦ C: offer[in ID, in item, out price]
 M ↦ C: offer[out ID, out item, out price]

 C ↦ M: accept[in ID, in item, in price, out
     confirmation]
 C ↦ M: accept[out ID, out item, out price, out
     confirmation]

 M ↦ C: goods[in ID, in item, in confirmation,
     out document]
 M ↦ C: goods[in ID, in item, nil confirmation,
     out document]
 C ↦ M: pay[in ID, in price, in document, out
     payment]
 C ↦ M: pay[in ID, in price, nil document, out
     payment]
 M ↦ C: receipt[in ID, in item, in payment, out
     done]
}
```

The benefit of the protocol of Listing 2 is that it remains as robust as the finite-state machine version while supporting greater flexibility of enactment on part of the roles, thereby promoting their autonomy. Yolum and Singh [5] identify some but not all of the variations that we can systematically produce. In addition, they fail to capture the information flows precisely: for example, when the MERCHANT sends an offer prior to receiving a corresponding rfq, the MERCHANT must initiate an independent transaction—which we can capture.

Oddly, the original *NetBill* does not include a *reject* message. We can enhance it so the MERCHANT and CUSTOMER iterate about the price [8]. The main point of this example from the standpoint of Bliss is that it involves messages with more than one key parameter. The *offer* message introduces a key parameter, say, ⌜out⌝ offerID. All the other messages would include ⌜in⌝ offerID to identify the specific offer.

V. Applying Bliss on a Complex Scenario

This scenario arises in the domain of scientific collaboration for oceanography [9]. Scientists own various resources that they do not fully use in their own investigations. These resources might be expensive or uniquely located, as a result of which sharing them can prove highly valuable. For example, suppose an oceanographic chemist, Alice, has a buoy in Chesapeake Bay that she would like to share with colleagues.

We envisage a resource-sharing community as including a unique moderator and zero or more resource contributors and users. We assume each participant has a suitably obtained unique account in the community. The same person may be a contributor and a user for different resources. An owner sharing a resource merely makes it visible to others; each specific usage episode needs a separate permission. When a resource is not in use, its owner can withdraw the resource from the community. A resource has a location (e.g., Chesapeake) and type (e.g., buoy with salinity sensor).

The following facts and steps are relevant: (i) multiple topical communities exist; (ii) the moderator can admit a scientist upon request; (iii) a member can contribute a resource to the community resource directory; (iv) the owner of a contributed resource can withdraw it from the community resource directory; (v) a member can search for a resource in the community resource directory, potentially specifying its location and type; and (vi) a member can request to borrow a resource from its owner, specifying the purpose.

The proposed solution is built from the conceptual schema of the resource-sharing scenario, as in Figure 2. This schema shows the domain entities, including the roles involved. The conceptual social objects capture the four main interactions that take place in this scenario, and map to protocols in a straightforward manner. The key of each object includes as foreign keys the keys of the entities it associates, as well as a key parameter specific to the social object. We suppress the identifiers for the roles participating in a given protocol.

Each resulting protocol has one completion parameter—named outcome in each case. The payload parameters are few: request for *Community Membership* and location and rType for both *Resource Contribution* and *Resource Discovery*. *Resource Negotiation* needs no explicit payload. None of these protocols needs any additional control parameters. Note that the protocols could be combined into a single large protocol but the proposed design is modular and thus easier to read and validate. (We omit *withdraw* for brevity.)

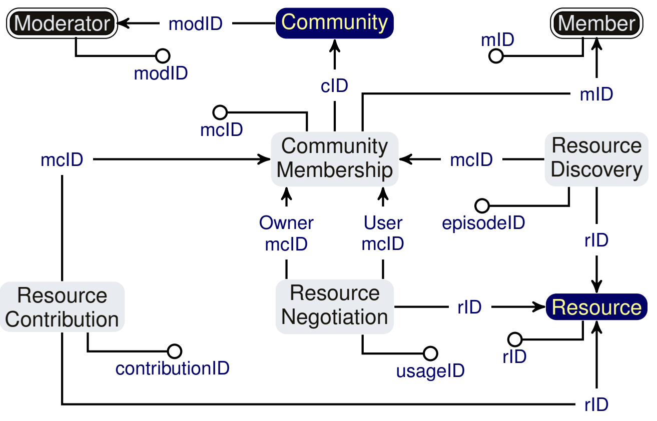

Fig. 2. A conceptual schema for resource sharing. Community and Resource are runnable entities but not active within the protocol. MODERATOR is a role but its key does not feature in any protocol (or message), because we assume a single moderator per community.

Listing 3. The *Community Membership* protocol. Here, ⌜opt⌝ signifies that the specified parameter is optional or nilable, to capture denial. The parameters cID (the community's ID) and mID (the member's ID) are both adorned ⌜in⌝ to indicate that the community and the member must already exist within the given social environment. Also, mcID denotes the member's membership token for the relevant community.

```
Community Membership {
 role Mod, Mem // Moderator , Member
 parameter in cID key, in mID key, opt mcID, out
     outcome
 private request

 Mem ↦ Mod: requestAdmission[in cID, in mID, out
      request]
 Mod ↦ Mem: admit[in cID, in mID, in request ,
     out mcID, out outcome]
 Mod ↦ Mem: deny[in cID, in mID, in request , nil
     mcID, out outcome]
}
```

Listing 4. The *Resource Contribution* protocol. Here, rID denotes a resource ID; rID is adorned ⌜in⌝ because the resource would already have been commissioned before it is contributed to a particular community by a member. Depending on the nature of the resources of interest, an alternative solution would be to make both rLocation and rTtype public parameters adorned ⌜in⌝ to signify that their bindings are determined elsewhere. Here, we assume that the contributor places a resource at some location and sets the resource's type, and sends a *contribute* message about the resource.

```
Resource Contribution {
 role Mod, Mem // Moderator , Member
 parameter in mcID key, in rID key, out
     contributionID key, out outcome
 private rlocation , rType

 Mem ↦ Mod: contribute[in mcID, in rID, out
     rLocation, out rType, out contributionID]
}
```

Listing 5. The *Resource Discovery* protocol. Here, rID denotes a resource ID; rID is adorned ⌜out⌝ because, although the resource would already have been contributed to a particular community, its key would be generated for the given episodeID for it to feature in a search result. rOwnerID is the mcID of the contributor the given resource. The prospective user's mcID is suppressed.

```
Resource Discovery {
 role Mod, Mem // Moderator , Member
 parameter out episodeID key, out rID key, out
     rOwnerID
 private rlocation , rType

 Mem ↦ Mod: search[out episodeID, out
     rLocation, out rType]
 Mod ↦ Mem: response[in episodeID, in
     rLocation, in rType, out rOwnerID, out rID]
}
```

Listing 6. The *Resource Negotiation* protocol. Here, rID denotes a resource ID; rID is adorned ⌜in⌝ because the resource is fixed for the negotiation before it can feature in a search result.

```
Resource Negotiation {
 role User, Owner
 parameter in rID key, out usageID key, out outcome

 User ↦ Owner: request[in rID, out usageID]
 Owner ↦ User: permit[in rID, in usageID, out
     outcome]
 Owner ↦ User: deny[in rID, in usageID, out
     outcome]
}
```

VI. Evaluation: Service Request

The *Service Request* protocol [10] is used as a crucial protocol in a large cyberinfrastructure project for oceanography. Figure 3 shows a corresponding UML sequence diagram that clarifies the alternatives. Even though this is a simple protocol, it shows errors for a fully distributed enactment, which we can detect through a formalization in BSPL.

Listing 7. Reconstruction in BSPL of the original (unsafe) *Service Request* protocol. To reduce clutter, we fold in parameters replyBy and content into the parameter operation.

```
protocol OOI Service Request Unsafe {
 role R, P
 parameter out ID key, out operation, out result
 private confirmation

 R ↦ P: request[out ID, out operation]
 P ↦ R: accept[in ID, out confirmation]
 P ↦ R: reject[in ID, out confirmation, out
     result]
 R ↦ P: cancel[in ID, out result]
 P ↦ R: fail[in ID, out result]
 P ↦ R: answer[in ID, out result]
}
```

Listing 7 is obtained from Figure 3 by capturing each message with a key parameter and suitable payload parameters based on the documentation. In addition, some of the messages include parameters with suitable adornments to ensure the stated sequencing requirements and the stated mutual exclusion requirements. Applying the BSPL verification tool [3], we

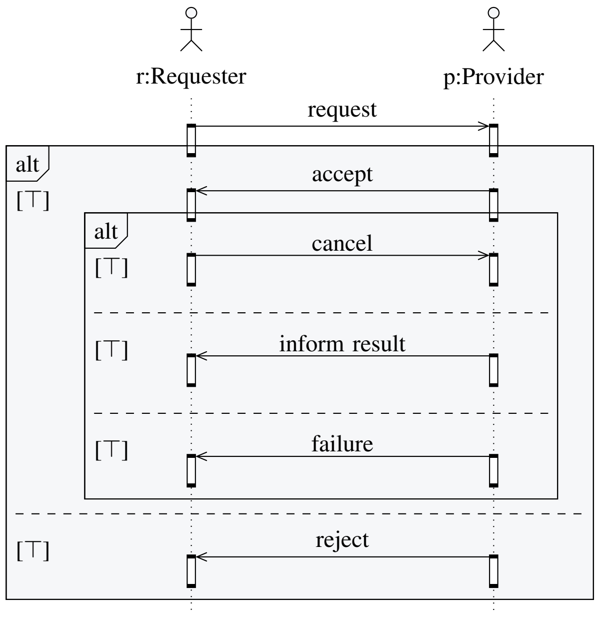

Fig. 3. The *Service Request* protocol. The guard ⊤ on each alternative indicates nondeterministic choice, reflecting the autonomy of the participants.

discover that this protocol is *unsafe*. Specifically, it is subject to a race condition. Both *cancel* and *inform result* may occur, leading to a violation of integrity where the binding of result is not unique among the agents enacting the protocol instance.

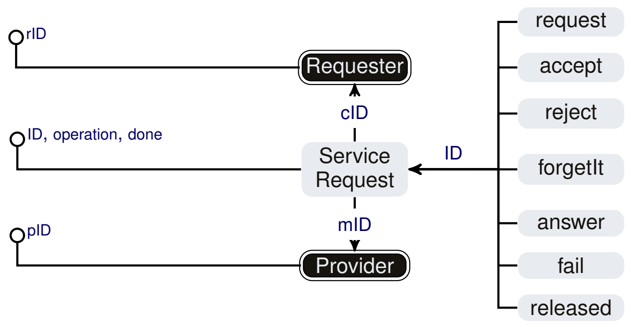

Fig. 4. A conceptual schema for *Service Request*.

Instead, if we apply Bliss to developing a service request protocol given the requirements, we end up with Listing 8, a safe and live design [3]. Below, "same" refers to the given element being the same here as in Listing 7. Notice that the way this protocol is set up (to satisfy the documentation) no polymorphism is appropriate. Here, unlike in our variant of *NetBill*, the provider does not take the initiative and generate an answer for which the requester has not sent a request.

The parameters are these: (i) Key (same): ID; (ii) Payload (same): operation and result; (iii) Completion (same): none needed; (iv) Integrity (same): none needed; and (v) Control (same): none needed. The messages are these:

- *request.* Same.
- *accept.* Same in principle, though repeating the parameters from the *request*. (Section VII discusses the ReSTful style.)
- *reject.* Same.
- *forgetIt.* Uses an ⌜out⌝ parameter releaseToken instead of result so as to avert an integrity violation. Here, releaseToken is a control parameter in Bliss. This message has a weaker connotation than the *cancel* message in the original.
- *answer.* Like *accept*, repeats the parameters from *request*. Uses the control parameter releaseToken to avoid an occurrence conflict that would cause an integrity violation.
- *fail.* Uses the control parameter releaseToken to avoid an occurrence conflict to prevent an integrity violation.
- *released.* A message introduced to achieve completion (liveness [3]) that uses the control parameter releaseToken to create an ordering dependency from *forgetIt*, so the PROVIDER cannot release the REQUESTER unilaterally. The completion parameter result serves double duty as a control parameter preventing *fail* and *released* from both being sent.

Listing 8. The *Service Request* protocol, corrected.

```
protocol OOI Service Request Corrected {
 role R, P
 parameter out ID key, out operation , out result
 private confirmation , releaseToken

 R ↦ P: request[out ID, out operation]
 P ↦ R: accept[in ID, in operation , out
     confirmation]
 P ↦ R: reject[in ID, in operation , out
     confirmation , out result]
 R ↦ P: forgetIt[in ID, in operation , in
     confirmation , out releaseToken]
 P ↦ R: answer[in ID, in operation , in
     confirmation , nil releaseToken , out result]
 P ↦ R: fail[in ID, in operation , in
     confirmation , nil releaseToken , out result]
 P ↦ R: released[in ID, in operation , in
     releaseToken , out result]
}
```

VII. Discussion

Existing work on business protocols, e.g., [11], [12], concentrates on verifying their correctness with respect to technical properties. Previous works on BSPL [2], [3] address its architecture and formal semantics. Bliss complements them by addressing the important challenge of ensuring that the protocols specified are valid with respect to the social object we wish them to compute and flexible in how they do so.

In general, over-specifying a protocol is easy—by encoding a concrete "happy" path. Existing protocols tend to fall into this trap. Thinking about alternative enactments takes more careful thinking. Bliss provides a conceptual model and methodology based on the information content of social objects that guides the requisite thinking and analysis to capture flexible protocols. Consequently, it provides a robust basis for declaratively capturing meanings of interactions.

Often, multiple formalizations are possible for the payload of a message. In a ReSTful style, we would place all parameters needed to express the meaning of a message into the message explicitly, which simplifies the connection between social meaning and the information carried by the parameters. Notice that these would be the parameters needed to compute the meaning, not all the parameters that might be sent. An alternative formalization is to include only some of the parameters needed for a message's meaning explicitly in the message. The included parameters would, of course, be the key parameters of the message as well as the necessary parameters that are not already bound to the key parameters through some other, causally prior, messages involving the sender and receiver of the current message. The remaining, elided parameters would be recovered via a database join. The benefit would be reducing communication redundancy to save bandwidth or energy. For simplicity, we adopt the ReSTful approach here, and thus ensure that each message includes all the parameters needed to formulate its meaning. A post-processing step may optimize protocols generated using Bliss for message transmission costs.

A. Literature

The topic of design methodologies for service-oriented and multiagent systems has garnered a lot of attention. Existing methodologies [13], [14], [15], [16] consider high-level abstractions such as goals and commitments and formulate steps to create a service-oriented or multiagent system beginning from an understanding of the requirements of stakeholders. Despite their use of high-level concepts, they usually produce protocols expressed as UML Sequence Diagrams [17] or a similar procedural notation that limits the flexibility of the participants. Given BSPL's novelty in capturing interactions purely via information, the existing methodologies do not apply, which motivates us to propose Bliss.

Bliss is distinguished from other early-stage design methodologies by its emphasis on first capturing the notional social object computed by a protocol enactment and then seeking parameter descriptions that help specify such an object. The more typical approach is to proceed bottom up by identifying what messages the participants might exchange before describing what those messages jointly compute.

Parunak [18] advocates an approach for eliciting requirements wherein designers standing in for different stakeholders imagine individual enactments (specifically, sequences of messages exchanged); annotate the messages with their relationships to one another and compute conversational structures from those relationships. Kalia and Singh's [19] Muon approach goes further by using commitments as a basis for detecting and handling business exceptions. Bliss benefits most from an understanding of the meaning of a protocol. It does not require, but is assisted by, prior knowledge of the messages exchanged and constraints on their orderings. However, Bliss seeks to capture causally significant relationships as evinced in the information transmitted in those messages. The focus on meaning and causality helps avoid over-specification by producing protocols that are not restricted to the particular enactments that the designers happen to have imagined.

B. Directions

A possible way to enhance Bliss is to incorporate the classical top-down stepwise refinement style. For example, for *Purchase*, we may determine it involves *Order Placement*, *Payment*, and *Shipping*. In the next step, one would refine each of these protocols, thus identifying their constituent messages. Then, the designer can think of any causality or consistency constraints, and represent each constraint through suitably adorned parameters. The designer can finally determine the crucial requirements for each protocol. Bliss could benefit from tool support that would facilitate deployment and extensive empirical evaluation in practical settings.

Acknowledgments

This work was partially supported by the NSF under grant 0910868 and by NSF contract OCE-0418967 with the Consortium for Ocean Leadership. I would like to thank Matthew Arrott and Amit Chopra for helpful discussions.

References

[1] M. P. Singh, "Information-driven interaction-oriented programming: BSPL, the Blindingly Simple Protocol Language," in *Proc. AAMAS*, 2011, pp. 491–498.

[2] ——, "LoST: Local State Transfer—An architectural style for the distributed enactment of business protocols," in *Proc. ICWS*, 2011, pp. 57–64.

[3] ——, "Semantics and verification of information-based protocols," in *Proc. AAMAS*, 2012, pp. 1149–1156.

[4] B. Cox, J. D. Tygar, and M. Sirbu, "NetBill security and transaction protocol," in *Proc. USENIX Wkshp. Elect. Comm.*, 1995, pp. 77–88.

[5] P. Yolum and M. P. Singh, "Commitment machines," in *Proc. ATAL 2001*, LNAI 2333, Springer, 2002, pp. 235–247.

[6] J. L. Austin, *How to Do Things with Words*. Clarendon Press, 1962.

[7] A. K. Chopra and M. P. Singh, "Constitutive interoperability," in *Proc. AAMAS*, 2008, pp. 797–804.

[8] M. P. Singh, "Formalizing communication protocols for multiagent systems," in *Proc. IJCAI*, 2007, pp. 1519–1524.

[9] M. Arrott,A. Chave, C. Farcas, E. Farcas,J. Kleinert, I. Krueger, M. Meisinger, J. Orcutt, C. Peach, O. Schofield, M. P. Singh, and F. Vernon, "Integrating Marine Observatories into a System-of-Systems," in *Proc. MTS-IEEE Oceans*, 2009, pp. 1–9.

[10] OOI, "Generic service request protocol," 2011, https://confluence.oceanobservatories.org/display/syseng/CIAD+COI+OV+Service+Request+Protocol.

[11] T. Miller and P. McBurney, "Propositional dynamic logic for reasoning about first-class agent interaction protocols," *Comp. Intell.*, 27(3):422–457, 2011.

[12] P. Yolum, "Design time analysis of multiagent protocols," *Data & Know. Eng.*, 63(1):137–154, 2007.

[13] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos, "Tropos: An agent-oriented software development methodology," *J. JAAMAS*, 8(3):203–236, 2004.

[14] T. Juan, A. Pearce, and L. Sterling, "ROADMAP: Extending the Gaia methodology for complex open systems," in *Proc. AAMAS*, 2002, pp. 3–10.

[15] L. Padgham and M. Winikoff, "Prometheus: A practical agent-oriented methodology," in *Agent-Oriented Methodologies*, B. Henderson-Sellers and P. Giorgini, Eds. Idea Group, 2005, ch. 5, pp. 107–135.

[16] M. Wooldridge, N. R. Jennings, and D. Kinny, "The Gaia methodology for agent-oriented analysis and design," *J. AAMAS*, 3(3):285–312, 2000.

[17] Object Management Group, UML 2.0 Superstructure Specification, 2004, http://www.omg.org/spec/UML/2.0/Superstructure/PDF/

[18] H. V. D. Parunak, "Visualizing agent conversations," in *Proc. ICMAS*, 1996, pp. 275–282.

[19] A. K. Kalia and M. P. Singh, "Muon: Designing Multiagent Communication Protocols from Interaction Scenarios," *J. AAMAS*, 2014, pp. 1–32. In press.