Semantics and Verification of Information-Based Protocols

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

singh@ncsu.edu

ABSTRACT

Information-Based Interaction-Oriented Programming, specifically as epitomized by the Blindingly Simple Protocol Language (BSPL), is a promising new approach for declaratively expressing multiagent protocols. BSPL eschews traditional control flow operators and instead emphasizes causality and integrity based solely on the information models of the messages exchanged. BSPL has been shown to support a rich variety of practical protocols and can be realized in a distributed asynchronous architecture wherein the agents participating in a protocol act based on local knowledge alone. The flexibility and generality of BSPL mean that it needs a strong formal semantics to ensure correctness as well as automated tools to help develop protocol specifications.

We provide a formal semantics for BSPL and formulate important technical properties, namely, enactability, safety, and liveness. We further describe our declarative implementation of the BSPL semantics as well as of verifiers for the above properties using a temporal reasoner. We have validated our implementation by verifying the correctness of several protocols of practical interest.

Categories and Subject Descriptors

H.1.0 [**Information Systems**]: Models and Principles—*General*; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems*

General Terms

Theory, Verification

Keywords

Business protocols, agent communication

1. INTRODUCTION

We take as our point of departure Singh's [13] recent work on Information-Based Interaction-Oriented Programming and especially on the Blindingly Simple Protocol Language (BSPL). The main innovation of BSPL is that it specifies multiagent protocols without the use of any control flow constructs. Instead, it relies purely on the specifications of the information schemas of the messages exchanged among the defined roles. From the message schemas, consisting of parameters (adorned in a specific manner we explain below), BSPL characterizes the relevant (1) causal dependencies and (2) integrity constraints. Causality provides a basis for ordering requirements that traditional languages address via control flow operators. Integrity constraints provide a basis for exclusion requirements that traditional languages address via choice operators.

Singh [13] explains the generality of BSPL in handling a variety of practical protocols and its ability to support the composition of protocols, but without violating encapsulation as previous approaches, e.g., AUML [11] and MAD-P [3], do. Further, Singh [14] shows how BSPL can be realized in a fully distributed, asynchronous architectural style wherein the participating agents can act based solely on local knowledge. The agents are always ready to receive any incoming message and are not prevented from emitting any message whose information prerequisites they can meet.

How can we be sure that a protocol is correct? Will it lead to erroneous enactments? Will it deadlock? Notice that BSPL merely forces the protocol designer to be explicit about causality, but it does not add to the challenges of correctness. Specifically, any approach that supports distributed decision making faces these problems. Traditional approaches insert arbitrary rigidness whereas BSPL offers an opportunity to achieve correctness and flexibility.

What is needed is, first, a rigorous formalization of the BSPL semantics that would capture the inherent distribution of a BSPL enactment along with expressing the local views of the (agents playing the) roles involved. Second, what is also needed is a clear formulation of important correctness properties of protocols and tools that would verify protocols with respect to those properties.

Contributions. The main contribution of this paper is precisely to fill the above gaps. To do so requires new technical results. One, we formulate a formal semantics for BSPL that incorporates a notion of *viability* and respects the locality of each role, the flow of causality across roles, and the asynchrony of the communications between them. Two, we capture the semantic requirements purely declaratively so that a logic-based reasoner can compute with them. Three, we formalize correctness properties. We realize the semantics and properties in a verification tool for BSPL protocols.

Organization. Section 2 follows Singh [13] in describing BSPL. Section 3 provides intuitions about the BSPL semantics as well as the correctness properties of interest. Section 4 provides a semantics of BSPL based on local enactments and observations by agents playing the roles in a protocol. Section 5 introduces a temporal language and shows how to formalize the causal structure of a protocol along with each property. It shows how we can verify each property by checking the (un)satisfiability of the conjunction of the causal structure and a property-specific formula. Section 6 discusses the related literature and some directions for future research.

Appears in: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, Conitzer, Winikoff, Padgham, and van der Hoek (eds.), 4-8 June 2012, Valencia, Spain.

2. BACKGROUND ON BSPL

Listing 1 presents a protocol to help illustrate 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 *Purchase* protocol.

```
Purchase {
 role B, S, Shipper
 parameter out ID key, out item, out price, out
     outcome

 B ↦ S: rfq[out ID, out item]
 S ↦ B: quote[in ID, in item, out price]
 B ↦ S: accept[in ID, in item, in price, out
     address, out response]
 B ↦ S: reject[in ID, in item, in price, out
     outcome, out response]
 S ↦ Shipper: ship[in ID, in item, in address]
 Shipper ↦ B: deliver[in ID, in item, in address,
     out outcome]
}
```

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. Each message instance must bind a proper value for each ⌜in⌝ and each ⌜out⌝ parameter, and a ⌜nul⌝ value for each ⌜nil⌝ parameter. For brevity, we avoid ⌜nil⌝ parameters in our examples.

Consider the quote message in *Purchase*, which includes an item description and a price, and may be emitted in response to a request for quotes for a particular item. Clearly, for the quote message to be emitted, its sender must instantiate all of its parameters. However, from the standpoint of the *quote* message, the item description is provided from the outside into the protocol and the price is provided by the protocol to the outside. Thus we adorn item with ⌜in⌝ and price with ⌜out⌝.

A message instance must provide a binding for each ⌜in⌝ and ⌜out⌝ parameter with the difference being that the ⌜out⌝ binding has declarative force [2]. For example, an agent emitting a price quote is not merely reporting a price previously computed in the conversation but declaring it to be the *definitive* price in this conversation. One can imagine such a message carrying the weight of a commitment, although we deemphasize commitments here.

All of *Purchase*'s parameters are adorned ⌜out⌝, indicating that *Purchase* provides them to any protocol that composes *Purchase*. The *rfq* and *quote* messages help generate a price offer. Here, the BUYER (B) generates item and the SELLER (S) generates price, since these parameters are adorned ⌜out⌝ in messages emitted by these roles. Any message that takes some ⌜in⌝ parameters can be enacted only if referenced from another protocol.

Notice that the *ship* message is irrelevant from the parameter standpoint since all its parameters are adorned ⌜in⌝, indicating that *ship* creates no new information. However, *ship* is clearly essential from the role perspective: it ensures that the SHIPPER learns of the parameter bindings that make the SHIPPER's emission of *deliver* viable. In general, BSPL separates and addresses the two concerns of the interplay of information with (1) interactions and (2) roles.

An enactment corresponds to a binding of public parameters. BSPL requires some of the parameters being declared as forming the *key* of a protocol enactment. Thus multiple concurrent enactments of the same protocol do not interfere with each other. Every protocol and message must have a key: for brevity, the key of a message equals the protocol key parameters that feature in it. An enactment is *complete* when all its public ⌜in⌝ and ⌜out⌝ parameters are bound. Specifically, an enactment of *Purchase* must create a tuple of bindings for its four public parameters but may omit address and response, which are private.

Listing 1 involves a private parameter response that is ⌜out⌝ in both *accept* and *reject*. Since BSPL models each enactment as producing a tuple of parameter bindings, the existence of the same parameter with an ⌜out⌝ adornment in two messages indicates an integrity violation: thus the two messages cannot both occur: if they did the binding would be conceptually undefined. That is, *accept* and *reject* conflict on response. And, outcome is ⌜out⌝ in both *reject* and *deliver*, thereby causing a conflict between them.

Syntax

The following BSPL syntax and explanations are simplified from Singh [13]. Superscripts of $+$ and $*$ indicate one or more and zero or more repetitions, respectively. Below, ⌊ and ⌋ 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.
Protocol ⟶ *Name* { role *Role*$^+$ parameter
⌊*Parameter*⌊key⌋⌋$^+$*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.
Reference ⟶ *Name* (*Role*$^+$ *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).
Reference ⟶ *Role* ↦ *Role* : *Name* [*Parameter*$^+$]

L_4. Each parameter consists of an adornment and a name.
Parameter ⟶ *Adornment Name*

L_5. An adornment is usually either ⌜in⌝ or ⌜out⌝. A ⌜nil⌝ in a reference indicates an unknown parameter.
Adornment ⟶ in | out | nil

3. INTUITIONS ON BSPL SEMANTICS

A protocol describes an interaction by specifying messages to be exchanged between specific roles, and by (indirectly, though effectively) 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. A message instantiates a message schema and is precisely described by its name, its sender, receiver, and bindings for each of its parameters.

BSPL is characterized by the interplay between parameters and messages. In describing interactions declaratively, we are concerned with tuples of parameter bindings. The keys determine the units of enactment. And, parameter bindings are immutable in any enactment. The parameter adornments determine how information is propagated through them. Interactions are realized exclusively through the exchange of messages: everything of relevance to the interaction is visible in a message emission and reception.

For example, *quote* (for a given ID, its key) may occur only after ID and item are bound. An enactment of a protocol may begin only when at least one of its messages is enabled. However, an enactment begins and proceeds to completion only if the agents involved decide autonomously to do their respective parts.

3.1 Knowledge and Viability

We distinguish between an agent's local and internal states. The *local* state is public, though limited to the role's view of the protocol enactment. The *internal* state depends on the agent implementation and is not visible to any other agent. We consider only the local states of roles. The history of a role maps naturally to its local state: each message emitted or received advances the local state.

Figure 1 illustrates the impact on a sender and receiver's knowledge with respect to a parameter adorned ⌜in⌝, ⌜out⌝, or ⌜nil⌝. Further, because of the immutability of parameter bindings, the knowledge of a role increases monotonically as its history extends. Thus these are the only three possible adornments of a parameter.

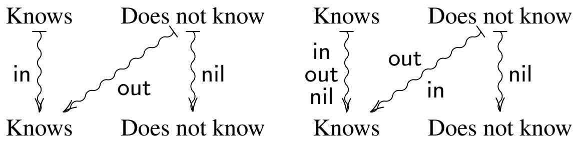


Figure 1: Viability and knowledge effects for an adorned parameter on the sender (left) and receiver (right) of a message.

As a result, an ⌜in⌝ emission must be preceded by an ⌜in⌝ or an ⌜out⌝ emission or reception and an ⌜out⌝ or a ⌜nil⌝ emission must not be preceded by an ⌜in⌝ or an ⌜out⌝ emission or reception. Notice that to send a message with an ⌜out⌝ parameter, the sender agent computes the parameter (through its internal business logic) but the only role states, i.e., local states, in which the message can be emitted are those where the role does not know already what it is. A correctly implemented agent would not compute a binding for an ⌜out⌝ parameter that (for the given keys) is already known to its role. And, for an ⌜in⌝ parameter, the role must know its binding through a previous message. Such parameter-based causality constraints underlie the semantics of BSPL.

Figure 1 identifies all the *viable* message emissions and receptions given a role's state of knowledge with respect to a parameter in a message. A message reception is always viable. In a practical system, we would validate incoming messages, as the LoST middleware [14] does, but in the abstract semantics we assume that the local state is never corrupted. For a message emission, the sender must already know the bindings of the ⌜in⌝ parameters and *not* know the bindings for any of the other parameters.

3.2 Causal Structures

Because BSPL incorporates a flexible description of interactions, it matches well with our computational approach of generating a *causal structure* as a set of declarative (temporal) constraints that capture the flow of causality within and across roles. A causal structure identifies partial states for each role that describe some parameters as known, some as unknown, and leave others indeterminate. It supports reasoning to verify various properties.

BSPL's flexibility does not accord well with detailed graphical representations of possible enactments, which get unwieldy fast. Specifically, a causal structure may be mapped to a finite state machine but with an explosion in states and transitions. However, to convey some intuitions, we show some *informal* pictures below.

Consider protocol *Sequential*. Because ID is ⌜out⌝ in *initial* and ⌜in⌝ in *additional*, *additional* causally depends on *initial*.

```
Sequential { ... // Omitting roles
 parameter out ID key, out answer, out more
 B ↦ S: initial [out ID, out answer]
 B ↦ S: additional [in ID, out more]
}
```

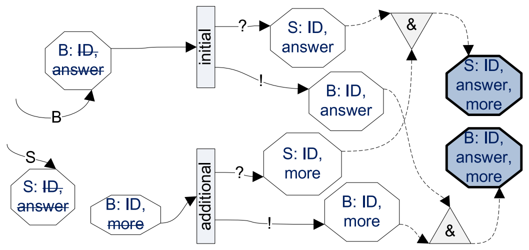


Figure 2: *Sequential* is enactable and safe.

Sequential's causal structure (Figure 2) represents the partial states of each role. It shows each message along with a precondition partial state in which it might be emitted and the effects it would have on the states of its sender and receiver. The precondition specifies what parameters its sender must know (⌜in⌝ parameters) and must not know (⌜out⌝ and ⌜nil⌝ parameters) before emitting the message. The effects specify what parameters its sender and receiver must know after it occurs (⌜out⌝ and ⌜in⌝ parameters). The solid transitions capture the intuitions of Figure 1. (The dashed lines show logical relationships.) A message can be emitted in any state that matches its precondition: the sender should know *all* parameters written plain and know *none* of the parameters written with a strikethrough line.

In *Sequential*, role B can send only *initial* at the outset. Upon emitting and receiving this message, respectively, B and S's states change as specified by the ? and ! edges from the message node. Thus, B knows answer and so cannot send *initial* but it would be superfluous here anyway. Also, B knows ID and can send *additional*, resulting in changing B and S's states further. When both messages are emitted and received, each ⌜out⌝ parameter of *Sequential* is known to at least one role, i.e., the enactment completes.

Figure 3 shows the causal structure for *Purchase*. The buyer B emits an *rfq*, which enables the seller S to send a *quote*. At this point, B has a choice about whether to *accept* or *reject*. In case of *reject*, all public parameters are bound so the enactment completes. In case of *accept*, S may send *ship* to the SHIPPER, who can DELIVER the item to B, thereby completing the enactment.

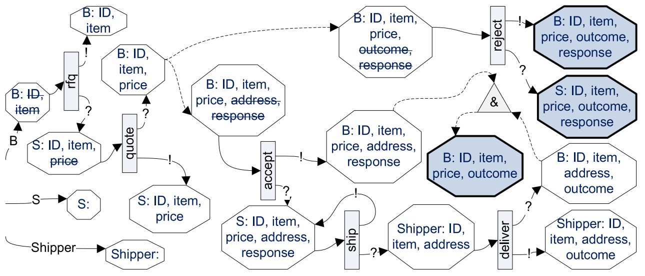

B: ID, item
!
rfq
B: ~~ID,~~ ~~item~~
?
B
S: ID, item, ~~price~~
quote
?
B: ID, item, price
!
S: ID, item, price
S
S:
Shipper
Shipper:
B: ID, item, price, ~~address,~~ ~~response~~
B: ID, item, price, ~~outcome,~~ ~~response~~
reject
!
B: ID, item, price, outcome, response
?
S: ID, item, price, outcome, response
&
accept
!
B: ID, item, price, address, response
?
B: ID, item, price, outcome
S: ID, item, price, address, response
!
ship
?
Shipper: ID, item, address
deliver
?
B: ID, item, address, outcome
!
Shipper: ID, item, address, outcome

Figure 3: Causal structure for *Purchase*.

3.3 Enactability

The intuition behind enactability simply is that a protocol should provide a clear path to completion, i.e., generating a tuple for all ⌜in⌝ and ⌜out⌝ public parameters. *Sequential* and *Purchase* are enactable, as the enactments described above demonstrate. Let's consider *Local Conflict*, whose causal structure is in Figure 4.

```
Local Conflict { ...
 parameter out ID key, out answer, out alternative
 B ↦ C: one [out ID, out answer]
 B ↦ C: two [out ID, out alternative]
}
```

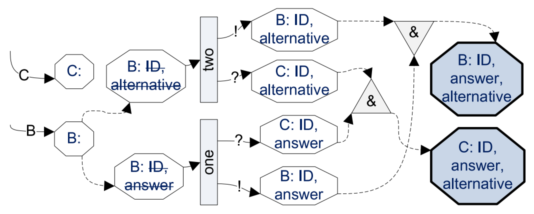


Figure 4: *Local Conflict* is not enactable, but is safe.

Local Conflict is not enactable. Because messages *one* and *two* conflict on ID, emitting *one* disables *two* and vice versa. Thus, at most one of them can be emitted for a given binding of ID. Hence, either answer or alternative would necessarily remain unbound.

3.4 Safety

A useful protocol must be safe, meaning that each parameter must have no more than one binding for the same key in any enactment. *Sequential* is safe because each of its parameters is ⌜out⌝ in at most one message.

Safety requires that at most one message instance with the same parameter adorned ⌜out⌝ occurs for any key. This condition is easy to ensure for a single role. Specifically, when a role emits a message with an ⌜out⌝ parameter, its knowledge changes and it may send no subsequent messages with the same ⌜out⌝ parameter. The BSPL semantics requires this local constraint and the LoST middleware [14] enforces it. In essence, the agent should decide internally which, if any, of the conflicting messages to send. *Local Conflict* is safe because B is the sender of both conflicting messages.

But no such reasoning applies *across* senders because each maintains separate local state information. For example, *Abrupt Cancel* below is not safe because it involves a race between B and S. Consider *Abrupt Cancel* and its causal structure (Figure 5).

```
Abrupt Cancel { ...
 B ↦ S: order [out ID, out item]
 B ↦ S: cancel [in ID, in item, out outcome]
 S ↦ B: goods [in ID, in item, out outcome]
}
```

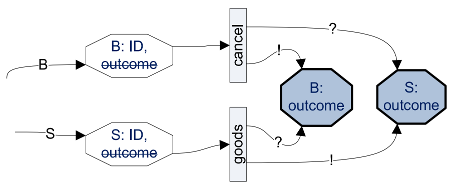


Figure 5: *Abrupt Cancel* is enactable but not safe.

Abrupt Cancel is enactable because it is possible to go from its initial states to where all its public parameters bound. However, because messages *goods* and *cancel* may both be emitted, outcome may be bound twice. Thus *Abrupt Cancel* is not safe.

To verify safety involves checking that the protocol prevents a situation where two roles can both be enabled to send conflicting messages. In essence, if there are two conflicting execution paths, they must have a prior branching point controlled by the same role. For example, in Listing 1, the conflict between *accept* and *reject* on private parameter response means that at most one of these two messages may occur. The same sender, B, is involved, so *Purchase* is safe. Further, because *deliver* can only occur if *ship* occurs previously (to convey address) and *ship* can occur only if *accept* occurs prior (to produce a binding for address). Thus *deliver* presupposes *accept* but *accept* and *reject* conflict. Therefore, the mutual exclusion between *deliver* and *reject* is guaranteed.

Listing 2 shows *Purchase Unsafe* based on *Purchase*, and which dispenses with the private parameter response. Thus *accept* and *reject* no longer conflict, and B may send both of them. Thus *deliver* and *reject* may both occur, violating safety for outcome.

Listing 2: An unsafe variant of *Purchase*.

```
Purchase Unsafe{ // Same as Purchase except these
 B ↦ S: accept[in ID, in item, in price, out
      address]
 B ↦ S: reject[in ID, in item, in price, out
      outcome]
}
```

In *Abrupt Cancel*, the inconsistency is obvious because different roles make mutually inconsistent decisions. *Purchase Unsafe* is more insidious because the inconsistent decisions by B and SHIPPER are gated by a decision by B. Our intuition may indicate that B acts in an odd manner when it emits both *accept* and *reject*. However, the semantics of a protocol depends *only* on the protocol specification, not on any imagined internal policies of the agents adopting its various roles. The protocol specification is the only means by which we constrain such policies: there is no hidden additional specification. (Of course, an autonomous agent may violate any protocol but the semantics tells us clearly what is a violation. Notice that LoST [14] helps an agent both respect a given protocol itself and ensure that others are respecting the protocol too.)

In general, when different roles are the senders of two conflicting messages, consistency is enforceable only if there is a causally prior conflict produced by the same role. Specifically, safety holds precisely when no causal path on which an ⌜out⌝ parameter is bound once may have the same ⌜out⌝ parameter bound again.

3.5 Liveness

Consider the following variant of *Purchase*.

```
Purchase No Ship { ...
//Same as Purchase but with ship deleted
}
```

Purchase No Ship remains enactable because if B emits *reject*, all its public parameters are bound. Despite this, however, if B emits *accept* the protocol enactment cannot complete: outcome is never bound because B can no longer send *reject* and the SHIPPER never becomes enabled to send *deliver*. Notice that we can never require that an agent send any message. And, in some cases, the protocol semantics prevents an agent from legally emitting a message.

Liveness requires that no matter what messages any of the agents has emitted, it should always be possible for a protocol enactment to legally complete. Liveness does not mean that the completion is necessarily a "happy" one from the application standpoint, just that the enactment terminates. In this sense, *Purchase* is live as are *Purchase Unsafe* and *Abrupt Cancel*. *Local Conflict* is not live.

Liveness entails enactability but not the other way around. Liveness is the more fundamental property. However, during design, enactability can help catch errors that are easier to fix, whereas to make corrections to ensure liveness can be more demanding.

4. FORMALIZING BSPL SEMANTICS

We now formalize the above intuitions. For convenience, we fix the symbols by which we refer to finite lists (mostly, treated as sets) of roles ($\vec{t}$), public roles ($\vec{x}$), private roles ($\vec{y}$), public parameters ($\vec{p}$), key parameters ($\vec{k} \subseteq \vec{p}$), ⌜in⌝ parameters ($\vec{p_I} \subseteq \vec{p}$), ⌜out⌝ parameters ($\vec{p_O} \subseteq \vec{p}$), ⌜nil⌝ parameters ($\vec{p_N} \subseteq \vec{p}$), private parameters ($\vec{q}$), and parameter bindings ($\vec{v}, \vec{w}$). Here, $\vec{p} = \vec{p_I} \cup \vec{p_O} \cup \vec{p_N}$, $\vec{p_I} \cap \vec{p_O} = \emptyset$,

$\vec{p}_I \cap \vec{p}_N = \emptyset$, and $\vec{p}_N \cap \vec{p}_O = \emptyset$. Also, t and p refer to an individual role and parameter, respectively. To reduce notation, we rename private roles and parameters to be distinct in each protocol, and the public roles and parameters of a reference to match the declaration in which they occur. Throughout, we use $\downarrow_x$ to project a list to those of its elements that belong to x.

Definition 1 captures BSPL protocols. A protocol (via any of its parameters) may reference another protocol (via its public parameters). The references bottom out at message schemas. Above, *Purchase* references *accept*. And, if a protocol were to reference *Purchase*, it would be able to reference (from its public or private parameters) only the public parameters of *Purchase*, not address.

DEFINITION 1. A *protocol* $\mathscr{P}$ is a tuple $\langle n, \vec{x}, \vec{y}, \vec{p}, \vec{k}, \vec{q}, F\rangle$, where n is a name; $\vec{x}$, $\vec{y}$, $\vec{p}$, $\vec{k}$, and $\vec{q}$ are as above; and F is a finite set of f references, $\{F_1, \ldots, F_f\}$. $(\forall i : 1 \leq i \leq f \Rightarrow F_i = \langle n_i, \vec{x}_i, \vec{p}_i, \vec{k}_i\rangle$, where $\vec{x}_i \subseteq \vec{x} \cup \vec{y}$, $\vec{p}_i \subseteq \vec{p} \cup \vec{q}$), $\vec{k}_i = \vec{p}_i \cap \vec{k}$, and $\langle n_i, \vec{x}_i, \vec{p}_i, \vec{k}_i\rangle$ is the public projection of a protocol $\mathscr{P}_i$ (with roles and parameters renamed).

DEFINITION 2. The *public projection* of a protocol $\mathscr{P} = \langle n, \vec{x}, \vec{y}, \vec{p}, \vec{k}, \vec{q}, F\rangle$ is given by the tuple $\langle n, \vec{x}, \vec{p}, \vec{k}\rangle$.

We treat a message schema $\ulcorner s \mapsto r : m\ \vec{p}(\vec{k})\urcorner$ as an atomic protocol with exactly two roles (sender and receiver) and no references: $\langle m, \{s, r\}, \emptyset, \vec{p}, \vec{k}, \emptyset, \emptyset\rangle$. Here $\vec{k}$ is the set of key parameters of the message schema. Usually, $\vec{k}$ is understood from the protocol in which the schema is referenced: $\vec{k}$ equals the intersection of $\vec{p}$ with the key parameters of the protocol declaration.

Below, let $\text{roles}(\mathscr{P}) = \vec{x} \cup \vec{y} \cup \bigcup_i \text{roles}(F_i)$; $\text{params}(\mathscr{P}) = \vec{p} \cup \vec{q} \cup \bigcup_i \text{params}(F_i)$; $\text{msgs}(\mathscr{P}) = \bigcup_i \text{msgs}(F_i)$ and $\text{msgs}(s \mapsto r : m\ \vec{p}) = \{m\}$. Definition 3 assumes that the message instances are unique up to the key specified in their schema.

DEFINITION 3. A *message instance* $m[s, r, \vec{p}, \vec{v}]$ associates a message schema $\ulcorner s \mapsto r : m\ \vec{p}(\vec{k})\urcorner$ with a list of values, where $|\vec{v}| = |\vec{p}|$, where $\vec{v}\downarrow_p = \ulcorner\text{nil}\urcorner$ iff $p \in \vec{p}_N$.

Definition 4 introduces a *universe of discourse (UoD)*. Definition 5 captures the idea of a *history* of a role as a sequence (equivalent to a set in our approach) of all and only the messages the role either emits or receives. Thus H^ρ captures the local view of an agent who might adopt role ρ during the enactment of a protocol. A history may be infinite in general but we assume each enactment in which a tuple of parameter bindings is generated is finite.

DEFINITION 4. A *UoD* is a pair $\langle \mathcal{R}, \mathcal{M}\rangle$, where $\mathcal{R}$ is a set of roles, $\mathcal{M}$ is a set of message names; each message specifies its parameters along with its sender and receiver from $\mathcal{R}$.

DEFINITION 5. A *history* of a role ρ, H^ρ, is given by a sequence of zero or more message instances $m_1 \circ m_2 \circ \ldots$. Each m_i is of the form $m[s, r, \vec{p}, \vec{v}]$ where $\rho = s$ or $\rho = r$, and $\circ$ means sequencing.

Definition 6 captures the idea that what a role knows at a history is exactly given by what the role has seen so far in terms of incoming and outgoing messages. Here, 2(i) ensures that $m[s, r, \vec{p}(\vec{k}), \vec{v}]$, the message under consideration, does not violate the uniqueness of the bindings. And, 2(ii) ensures that ρ knows the binding for each $\ulcorner\text{in}\urcorner$ parameter and not for any $\ulcorner\text{out}\urcorner$ or $\ulcorner\text{nil}\urcorner$ parameter.

DEFINITION 6. A message instance $m[s, r, \vec{p}(\vec{k}), \vec{v}]$ is *viable* at role ρ's history H^ρ iff (1) $r = \rho$ (reception) or (2) $s = \rho$ (emission) and (i) $(\forall m_i[s_i, r_i, \vec{p}_i, \vec{v}_i] \in H^\rho$ if $\vec{k} \subseteq \vec{p}_i$ and $\vec{v}_i\downarrow_{\vec{k}} = \vec{v}\downarrow_{\vec{k}}$ then $\vec{v}_i\downarrow_{\vec{p}\cap\vec{p}_i} = \vec{v}\downarrow_{\vec{p}\cap\vec{p}_i})$ and (ii) $(\forall p \in \vec{p} : p \in \vec{p}_I$ iff $(\exists m_i[s_i, r_i, \vec{p}_i, \vec{v}_i] \in H^\rho$ and $p \in \vec{p}_i$ and $\vec{k} \subseteq \vec{p}_i))$.

Definition 7 captures that a *history vector* for a protocol is a vector of histories of role that together are causally sound: a message is received only if it has been emitted [8].

DEFINITION 7. Let $\langle \mathcal{R}, \mathcal{M}\rangle$ be a UoD. We define a *history vector* for $\langle \mathcal{R}, \mathcal{M}\rangle$ as a vector $[H^1, \ldots, H^{|\mathcal{R}|}]$, such that $(\forall s, r : 1 \leq s, r \leq |\mathcal{R}| : H^s$ is a history and $(\forall m[s, r, \vec{p}, \vec{v}] \in H^r : m \in \mathcal{M}$ and $m[s, r, \vec{p}, \vec{v}] \in H^s))$.

The progression of a history vector records the progression of an enactment of a multiagent system. Under the above causality restriction, a vector that includes a reception must have progressed from a vector that includes the corresponding emission. Further, we make no FIFO assumption about message delivery. The viability of the messages emitted by any role ensures that the progression is epistemically correct with respect to each role.

DEFINITION 8. A history vector over $\langle \mathcal{R}, \mathcal{M}\rangle$, $[H^1, \ldots, H^{|\mathcal{R}|}]$, is *viable* iff either (1) each of its element histories is empty or (2) it arises from the progression of a viable history vector through the emission or the reception of a viable message by one of the roles, i.e., $(\exists i, m_j : H^i = H'^i \circ m_j$ and $[H^1, \ldots, H'^i, H^{|\mathcal{R}|}]$ is viable).

The heart of our formal semantics is the *intension* of a protocol, defined relative to a UoD, and given by the set of viable history vectors, each corresponding to its successful enactment. Given a UoD, Definition 9 specifies a universe of enactments, based on which we express the intension of a protocol. We restrict attention to viable vectors because those are the only ones that can be realized. We include private roles and parameters in the intension so that compositionality works out. In the last stage of the semantics, we project the intension to the public roles and parameters.

DEFINITION 9. Given a UoD $\langle \mathcal{R}, \mathcal{M}\rangle$, the *universe of enactments* for that UoD, $\mathcal{U}_{\mathcal{R},\mathcal{M}}$, is the set of viable history vectors, each of which has exactly $|\mathcal{R}|$ dimensions and each of whose messages instantiates a schema in $\mathcal{M}$.

Definition 10 states that the intension of a message schema is given by the set of viable history vectors on which that schema is instantiated, i.e., an appropriate message instance occurs in the histories of both its sender and its receiver.

DEFINITION 10. The intension of a message schema is given by: $[\![m(s, r, \vec{p})]\!]_{\mathcal{R},\mathcal{M}} = \{H | H \in \mathcal{U}_{\mathcal{R},\mathcal{M}}$ and $(\exists \vec{v}, i, j : H^s_i = m[s, r, \vec{p}, \vec{v}]$ and $H^r_j = m[s, r, \vec{p}, \vec{v}])\}$.

A (composite) protocol completes if one or more of subsets of its references completes. For example, *Purchase* yields two such subsets, namely, $\{$*rfq*, *quote*, *accept*, *ship*, *deliver*$\}$ and $\{$*rfq*, *quote*, *reject*$\}$. Informally, each such subset contributes all the viable interleavings of the enactments of its members, i.e., the intersection of their intensions. Definition 11 captures the *cover* as an adequate subset of references of a protocol, and states that the intension of a protocol equals the union of the contributions of each of its covers.

DEFINITION 11. Let $\mathscr{P} = \langle n, \vec{x}, \vec{y}, \vec{p}, \vec{k}, \vec{q}, F\rangle$ be a protocol. Let cover $(\mathscr{P}, G) \equiv G \subseteq F | (\forall p \in \vec{p} : (\exists G_i \in G : G_i = \langle n_i, x_i, p_i\rangle$ and $p \in \vec{p}_i))$; and $\mathscr{P}$'s intension, $[\![\mathscr{P}]\!]_{\mathcal{R},\mathcal{M}} = (\bigcup_{\text{cover}(\mathscr{P},G)} (\bigcap_{G_i \in G} [\![G_i]\!]_{\mathcal{R},\mathcal{M}}))\downarrow_{\vec{x}}$.

As an example, consider a message m_1 with a single key parameter p adorned $\ulcorner\text{in}\urcorner$ whose sender is role r_1. The intension of this message with respect to a UoD $\langle \mathcal{R}, \mathcal{M}\rangle$ can be nonempty only if one of the following conditions holds for some $m_2 \in \mathcal{M}$ (m_2 should precede m_1 in role r_1's history):

- m_2's schema involves the parameter p adorned $\ulcorner\text{out}\urcorner$ and r_1 is the sender or receiver of m_2.
- m_2's schema involves the parameter p adorned $\ulcorner\text{in}\urcorner$ and the receiver of m_2 is the same role r_1.

The intension of m_1 would still be empty if the intension of any such m_2 is empty, e.g., if m_2 does not occur on any viable history vector. In general, if message m_1 has message m_2 as an essential prerequisite, then the intension of m_1 must be a subset of the intension of m_2, i.e., $[\![m_1]\!] \subseteq [\![m_2]\!]$.

The UoD of protocol $\mathscr{P}$ consists of $\mathscr{P}$'s roles and messages including its references recursively. For example, *Purchase*'s UoD $U = \langle\{\text{B, S, SHIPPER}\}, \{\mathit{rfq}, \mathit{quote}, \mathit{accept}, \mathit{reject}, \mathit{ship}, \mathit{deliver}\}\rangle$.

DEFINITION 12. The *UoD of a protocol* $\mathscr{P}$, $\text{UoD}(\mathscr{P}) = \langle\text{roles}(\mathscr{P}), \text{msgs}(\mathscr{P})\rangle$.

4.1 Enactability, Safety, and Liveness

Enactability means that we can produce a history vector that generates bindings for all public parameters. Note that a protocol that has a public ⌜in⌝ parameter is not enactable standalone, and must be referenced from another protocol. Safety means that we cannot produce a history vector that generates more than one binding for any parameter. Liveness means that we cannot produce a history vector that deadlocks.

DEFINITION 13. A protocol $\mathscr{P}$ is *enactable* iff $[\![\mathscr{P}]\!]_{\text{UoD}(\mathscr{P})} \neq \emptyset$.

DEFINITION 14. A protocol $\mathscr{P}$ is *safe* iff each history vector in $[\![\mathscr{P}]\!]_{\text{UoD}(\mathscr{P})}$ is safe. A history vector is *safe* iff all key uniqueness constraints apply across all histories in the vector.

DEFINITION 15. A protocol $\mathscr{P}$ is *live* iff each history vector in the universe of enactments $\text{UoD}(\mathscr{P})$ can be extended through a finite number of message emissions and receptions to a history vector in $\text{UoD}(\mathscr{P})$ that is complete.

4.2 Well-Formedness Conditions

Because BSPL relies upon parameter adornments and keys for causality and integrity, it is essential that a protocol meet some elementary syntactic conditions. The main idea is that each key (set of parameters) identifies a logical entity and nonkey parameters express attributes of that entity. Thus the nonkey parameters have no meaning if separated from their keys. As an example, think of taking the age of a person with an identifier and putting the age in a record without the person's identifier. We need to carry the original identifier along. Of course, an agent may copy the contents of one parameter to another (e.g., set price equal to age), but such internal reasoning is not in the purview of BSPL.

The foregoing motivation leads to the following constraints. First, two messages must involve different parameters unless the key of one is a subset of the key of the other. Second, no message m_0 that has an ⌜out⌝ key parameter must have an ⌜in⌝ nonkey parameter p unless: if p occurs as an ⌜out⌝ in a message m_1 then m_0 includes m_1's key; and, if p occurs as an ⌜in⌝ in a message m_2 with an entirely ⌜in⌝ key then m_0 includes m_2's key. Notice that Definition 6 seeks to respect the above constraint. The net result is that if a message has an ⌜out⌝ parameter p then the key with which p is produced must be stated whenever we use p.

5. VERIFYING BSPL PROTOCOLS

To verify BSPL protocols, we express a causal structure as well as the target properties in a temporal language and determine the satisfiability of the resulting expressions. The language we adopt, Precedence, is an extension of Singh's [12] language.

The atoms of Precedence are events. Below, e and f are events. If e is an event, its complement $\overline{e}$ is also an event. Precedence treats e and $\overline{e}$ on par. The terms $e \cdot f$ and $e \star f$, respectively, mean that e occurs prior to f and e and f occur simultaneously. The Boolean operators: '$\vee$' and '$\wedge$' have the usual meanings. The syntax follows conjunctive normal form:

L$_6$. $I \longrightarrow \mathit{clause} \mid \mathit{clause} \wedge I$

L$_7$. $\mathit{clause} \longrightarrow \mathit{term} \mid \mathit{term} \vee \mathit{clause}$

L$_8$. $\mathit{term} \longrightarrow \mathit{event} \mid \mathit{event} \cdot \mathit{event} \mid \mathit{event} \star \mathit{event}$

The semantics of Precedence is given by pseudolinear runs of events (instances): "pseudo" because several events may occur together though there is no branching. Let Γ be a set of events where $e \in \Gamma$ iff $\overline{e} \in \Gamma$. A run is a function from natural numbers to the power set of Γ, i.e., $\tau : \mathbb{N} \mapsto 2^{\Gamma}$. The i^{th} index of τ, $\tau_i = \tau(i)$. The length of τ is the first index i at which $\tau(i) = \emptyset$ (after which all indices are empty sets). We say τ is empty if $|\tau = 0$. The subrun from i to j of τ is notated $\tau_{[i,j]}$. Its first $j - i + 1$ values are extracted from τ and the rest are empty, i.e., $\tau_{[i,j]} = \langle \tau_i, \tau_{i+1} \ldots \tau_{j-i+1} \ldots \emptyset \ldots \rangle$. On any run, e or $\overline{e}$ may not both occur. Events are nonrepeating.

$\tau \models_i E$ means that τ satisfies E at i or later. We say τ is a *model* of expression E iff $\tau \models_0 E$. E is *satisfiable* iff it has a model.

M$_1$. $\tau \models_i e$ iff $(\exists j \geq i : e \in \tau_j)$

M$_2$. $\tau \models_i e \star f$ iff $(\exists j \geq i : \{e, f\} \subseteq \tau_j)$

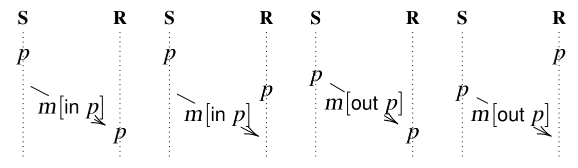
M$_3$. $\tau \models_i r \vee u$ iff $\tau \models_i r$ or $\tau \models_i u$

M$_4$. $\tau \models_i r \wedge u$ iff $\tau \models_i r$ and $\tau \models_i u$

M$_5$. $\tau \models_i e \cdot f$ iff $(\exists j \geq i : \tau_{[i,j]} \models_0 e$ and $\tau_{[j+1,|\tau|]} \models_0 f)$

We capture the local state of each role in a BSPL protocol by defining two kinds of events: (1) a particular message having been observed (one event each for sender and receiver) and (2) a particular parameter having a known binding (one event for each parameter whether emitted or received in any message).

Although our approach is generic and implemented, we describe it via examples based on *Purchase* to simplify the exposition in limited space. From *Purchase*, we first determine events from messages (B: quote, S: quote, ...: total 12) and parameters (B: price, S: price, ...: total 17). Next, we describe how we generate a causal structure, ignoring ⌜nil⌝ adornments for brevity. For protocol $\mathscr{P}$, let $\mathscr{C}_P$ be the conjunction of all clauses of the following types.



(a) ⌜in⌝ parameters (b) ⌜out⌝ parameters

Figure 6: Treatment of ⌜in⌝ and ⌜out⌝ parameters: each has one possible scenario for the sender and two for the receiver.

Reception. If a message is received, it was previously emitted. Specifically, either the receiver role never observes a message or the sender observes it before (six clauses).

$\overline{\text{B: quote}} \vee \text{S: quote} \cdot \text{B: quote}$

Information transmission. For each message, in its sender's view.

See Figure 6(a). Either the message is never emitted or each of its ⌜in⌝ parameters is observed before (14 clauses).

$\overline{\text{S: quote}} \vee \text{S: item} \cdot \text{S: quote}$

See Figure 6(b). Either the message is never emitted or it is observed simultaneously with each of its ⌜out⌝ parameters (eight clauses).

$\overline{\text{S: quote}} \vee \text{S: price} \star \text{S: quote}$

Information reception. For each message, in its receiver's view.

Any ⌜out⌝ or ⌜in⌝ parameter occurs before or simultaneously with the message. In other words, either the message is not observed or each such parameter is observed no later than the message. A parameter may be observed earlier through some other path [14] (22 clauses). See Figure 6.

$\overline{\text{B: quote}} \vee \text{B: price} \cdot \text{B: quote} \vee \text{B: price} \star \text{B: quote}$

Information minimality. For any role, if a parameter occurs, it occurs simultaneously with *some* message emitted or received. Thus, no role observes a parameter noncausally (16 clauses).

$\overline{\mathsf{B}\colon\mathsf{price}} \vee \mathsf{B}\colon\mathsf{price} \star \mathsf{B}\colon\mathsf{quote} \vee \mathsf{B}\colon\mathsf{price} \star \mathsf{B}\colon\mathsf{accept} \vee \mathsf{B}\colon\mathsf{price} \star \mathsf{B}\colon\mathsf{reject}$

Ordering. If a role emits any two messages, it observes them in some order, not simultaneously. This constraint rules out lockstep enactments whereby two messages happen to be emitted magically at the same time to escape our causality constraints (four clauses).

$\overline{\mathsf{S}\colon\mathsf{quote}} \vee \overline{\mathsf{S}\colon\mathsf{ship}} \vee \mathsf{S}\colon\mathsf{quote} \cdot \mathsf{S}\colon\mathsf{ship} \vee \mathsf{S}\colon\mathsf{ship} \cdot \mathsf{S}\colon\mathsf{quote}$

The correctness of our decision procedures relies on the following result, whose proof follows from construction.

THEOREM 1. Given a well-formed protocol $\mathscr{P}$, for every viable history vector, there is a model of $\mathscr{C}_P$ and vice versa.

Proof Sketch. In the forward direction, we can proceed by induction to find a pseudolinear run that includes each message emission and reception that occurs in any history in the history vector H. An empty history vector corresponds to an empty run. Inductively assume that a run τ exists for history vector H. We can extend H via the emission of a message only if the message is viable in H, meaning that its sender has locally observed its ⌜in⌝ parameters but not its ⌜out⌝ or ⌜nil⌝ parameters. Using the information transmission clauses, we can construct a run τ' that extends τ with the message emission event. We can extend H via the reception of the message by a role r. By Definition 7, H must include a message emission event for some role sending to r. Thus we can construct τ' as τ appended with the message reception. In the reverse direction, given a run τ, we simply construct each member history of the vector from τ by appending the message emission and reception events (and ignoring all others) involving a role in sequence to that role's history. The clauses in $\mathscr{C}_P$ ensure that the resulting vector is viable. □

5.1 Verifying Enactability

We generate clauses that together indicate completion of a protocol enactment. For each public parameter we identify all the messages in which it occurs and specify a clause that is the disjunction of the receivers of those messages observing that parameter. For example, for outcome, we have $\mathsf{B}\colon\mathsf{outcome} \vee \mathsf{S}\colon\mathsf{outcome}$ because B and S are the receivers of the two messages in which outcome occurs. For protocol $\mathscr{P}$, let $\mathscr{E}_P$ be the conjunction of all such clauses. Our decision procedure is simply to check if $\mathscr{C}_P \wedge \mathscr{E}_P$ is satisfiable.

THEOREM 2. A well-formed protocol $\mathscr{P}$ is *enactable* if and only if $\mathscr{C}_P \wedge \mathscr{E}_P$ is satisfiable.

Proof Sketch. In the forward direction, assume protocol $\mathscr{P}$ has a nonempty intension. Then, by Definition 11, it has a cover of references that yield bindings for all its public parameters, indicating that $\mathscr{E}_P$ holds for each vector. Further from Theorem 1, we know that $\mathscr{C}_P$ holds in the corresponding run. In the reverse direction, from any run that satisfies $\mathscr{C}_P \wedge \mathscr{E}_P$ we can identify a cover with a nonempty intension. □

5.2 Verifying Safety

Safety means that for each parameter (public or private) adorned ⌜out⌝ in two or more messages, no more than one of those "competing" messages may be emitted. To this end, we generate clauses expressing that two or more of the competing messages of some parameter are observed by their sender. For *Purchase*, outcome and response are the relevant parameters. Therefore, the resulting two clauses are $(\mathsf{B}\colon\mathsf{accept} \vee \textsc{shipper}\colon\mathsf{deliver})$ and $(\mathsf{B}\colon\mathsf{reject})$. This clause says that both reject and either accept or deliver is emitted, which signifies an inconsistency. For protocol $\mathscr{P}$, let $\mathscr{S}_P$ be the conjunction of all the property clauses. Our decision procedure is simply to check if $\mathscr{C}_P \wedge \mathscr{S}_P$ is unsatisfiable.

THEOREM 3. A well-formed protocol $\mathscr{P}$ is *safe* if and only if $\mathscr{C}_P \wedge \mathscr{S}_P$ is *not* satisfiable.

Proof Sketch. Let τ satisfy $\mathscr{C}_P \wedge \mathscr{S}_P$. Then, by Theorem 1, we can construct a history vector in which at least two conflicting messages occur. We can extend such a vector to one where at least two roles generate bindings for the same parameter, thus violating integrity (we cannot prove that they will generate the same bindings since we have no access to internal reasoning). Conversely, if $\mathscr{C}_P \wedge \mathscr{S}_P$ is *not* satisfiable, there is no history vector in the intension of $\mathscr{P}$ that violates integrity. □

5.3 Verifying Liveness

Notice that a specific protocol enactment being incomplete does not entail that some role is blocked. A enactment may fail to complete even though the protocol may be live: (1) one or more agents may decide not to send messages or (2) one or more messages may be lost—causality requires only that receptions are preceded by emissions, not that emissions are always followed by receptions.

To avoid situations where some agents may decide not to send any messages, we restrict attention to models that are *maximal* in the sense that they have no message left unemitted that could be emitted based on the parameters that feature in it. That is, in a maximal model, if the sender has observed the ⌜in⌝ and not observed the ⌜out⌝ parameters of a message, then the sender must also observe the message. The following says that either S emits *quote* or it does not observe ID or item or it observes price, i.e., $(\mathsf{S}\colon\mathsf{quote} \vee \overline{\mathsf{S}\colon\mathsf{ID}} \vee \overline{\mathsf{S}\colon\mathsf{item}} \vee \mathsf{S}\colon\mathsf{price})$.

To avoid situations where the communication infrastructure may drop messages, we constrain our model to those where every message emitted is delivered, e.g., $(\overline{\mathsf{S}\colon\mathsf{quote}} \vee \mathsf{B}\colon\mathsf{quote})$. Third, the enactment is incomplete, which means that at least one of the public ⌜out⌝ parameters remains unbound at each role. The above condition yields a clause $(\overline{:\mathsf{ID}} \vee \overline{:\mathsf{item}} \vee \overline{:\mathsf{price}} \vee \overline{:\mathsf{outcome}})$ constructed from protocol-level literals, for which no role is relevant. For each such literal, we assert two clauses $(:\mathsf{price} \vee \overline{\mathsf{B}\colon\mathsf{price}} \vee \overline{\mathsf{S}\colon\mathsf{price}})$ and $(\overline{:\mathsf{price}} \vee \mathsf{B}\colon\mathsf{price} \vee \mathsf{S}\colon\mathsf{price})$, meaning that the protocol-level literal is true exactly if at least one role observes the parameter.

If a maximal, nonlossy enactment can be incomplete that means the protocol is not live. For protocol $\mathscr{P}$, let $\mathscr{L}_P$ be the conjunction of all the above property clauses. Our decision procedure is to check if $\mathscr{C}_P \wedge \mathscr{E}_P$ is satisfiable and $\mathscr{C}_P \wedge \mathscr{L}_P$ is unsatisfiable.

THEOREM 4. A protocol $\mathscr{P}$ is *live* if and only if $\mathscr{C}_P \wedge \mathscr{E}_P$ is satisfiable and $\mathscr{C}_P \wedge \mathscr{L}_P$ is *not* satisfiable.

Proof Sketch. Let τ satisfy $\mathscr{C}_P \wedge \mathscr{L}_P$. Then, by Theorem 1, we can construct a viable history vector that cannot be extended by a message emission (maximality) or by message reception (lossless transmission), and yet is incomplete. That is, such a history vector belongs to the universe of enactments of $\mathscr{P}$ but is neither complete nor can be completed. Thus, by Definition 15, $\mathscr{P}$ is not live.

Conversely, if a protocol $\mathscr{P}$ is *live*, we know there is a history vector in the universe of enactments of $\mathscr{P}$ that is complete. From Theorem 2, that vector satisfies $\mathscr{C}_P \wedge \mathscr{E}_P$. We also know that each history vector in the universe of enactments of $\mathscr{P}$ is either complete or can be finitely extended to a complete history vector. Thus if $\mathscr{P}$ is live, $\mathscr{C}_P \wedge \mathscr{L}_P$ is *not* satisfiable. □

6. DISCUSSION

We employ a satisfiability (SAT) solver for Precedence built using a propositional SAT solver—an established technology for logical reasoning. Our approach expresses and solves temporal constraints directly instead of building an explicit state machine representation. Table 1 shows the numbers of Precedence clauses for selected protocols.

Table 1: Counts of Precedence clauses for our properties.

Protocol	$\lvert\mathscr{C}_P\rvert$	$\lvert\mathscr{E}_P\rvert$	$\lvert\mathscr{S}_P\rvert$	$\lvert\mathscr{L}_P\rvert$	Total
Abrupt Cancel	14	2	4	9	29
Purchase	70	4	4	21	99
Purchase No Reject	56	4	0	19	79
Purchase No Ship	62	4	4	19	89
Purchase Unsafe	64	4	2	21	91
FIPA Request [6]	121	3	56	27	207

A technically correct protocol may have design flaws: (1) it may have deadwood messages, which can never be enacted; (2) parameters that may never be bound (e.g., a private parameter that is not ⌜out⌝ in any message); (3) some parameters that may never be used (e.g., a private parameter that is not adorned ⌜in⌝ in any message). Such checks are valuable because they indicate other problems.

We elide other easy checks that help validate a protocol but which are not critical to our verifier. A protocol that has a public ⌜in⌝ parameter cannot be enacted standalone: its intension is empty. A protocol that lacks a message involving any of its public ⌜out⌝ parameters is also not enactable.

Traditional notations for protocols such as AUML [11], UML 2.0 message-sequence charts (MSCs), and choreography description languages take a procedural stance for describing interactions. Thus they emphasize explicit constraints on how messages are ordered. Desai and Singh [4] identify several challenges to the enactability of a protocol: ordering problems termed *blindness* and occurrence problems termed *nonlocal choice* [7]. Traditional approaches formalize properties such as safety and liveness but those are understood purely procedurally and the underlying model does not sustain a declarative information-based model as BSPL does. In contrast, BSPL's parameter adornments force clarity in terms of causality and the flow of information. In this way, BSPL avoids both blindness and nonlocal choice: each of them yields an empty intension and is thus deemed nonenactable. A designer can correct an unsound protocol by inserting suitable messages.

Miller and McGinnis [9] propose RASA, a language for expressing protocols. RASA takes a procedural stance on capturing protocols and takes its semantics from propositional dynamic logic (PDL). RASA does not have a notion of parameter adornment as in BSPL and its semantics does not capture the ideas of maximizing concurrency and interaction orientation. Like BSPL, RASA supports protocols that an agent can inspect and reason about.

LoST [14] focuses on the architectural aspects of realizing a language such as BSPL. That work describes the functioning of a suitable middleware using conceptually relational data stores. It does not describe the formal semantics as introduced in this paper.

Several researchers have developed approaches for analyzing protocols [1, 5, 10, 15] that by and large consider higher-level aspects of interaction than BSPL. It would be interesting and useful to see how such approaches can be applied on top of BSPL. A potential advantage, and one that motivates BSPL, is that by guaranteeing the integrity of distributed enactments, BSPL can facilitate expressing high-level, declarative meanings of protocols, thereby facilitating analyses carried out in the above works. Thus the reasoning they perform on commitments, delegation, and other normative constructs could have even better effect than presently, in particular, obtaining a distributed rendition for free and thus the opportunity to apply in an asynchronous information-driven environment. Exploring such connections is an important theme for future research.

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 Claudiu Farcas for discussions. Thanks to Amit Chopra, Neil Yorke-Smith, and the anonymous reviewers for comments on previous versions.

7. REFERENCES

[1] A. Artikis, M. J. Sergot, and J. Pitt. An executable specification of a formal argumentation protocol. *Artificial Intelligence*, 171(10–15):776–804, 2007.

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

[3] N. Desai and M. P. Singh. A modular action description language for protocol composition. In *Proc. 22nd Conference on Artificial Intelligence (AAAI)*, pp. 962–967, Vancouver, July 2007.

[4] N. Desai and M. P. Singh. On the enactability of business protocols. In *Proc. 23rd Conference on Artificial Intelligence (AAAI)*, pp. 1126–1131, Chicago, July 2008.

[5] M. El Menshawy, J. Bentahar, H. Qu, and R. Dssouli. On the verification of social commitments and time. In *Proc. AAMAS*, pp. 483–490, Taipei, May 2011.

[6] FIPA. FIPA interaction protocol specifications, 2003. FIPA: The Foundation for Intelligent Physical Agents, http://www.fipa.org/repository/ips.html.

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

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

[9] T. Miller and J. McGinnis. Amongst first-class protocols. In *Proc. 8th International Workshop on Engineering Societies in the Agents World (ESAW 2007)*, *LNCS* 4995, pp. 208–223. Springer, 2008.

[10] T. J. Norman and C. Reed. A logic of delegation. *Artificial Intelligence*, 174(1):51–71, Jan. 2010.

[11] J. Odell, H. V. D. Parunak, and B. Bauer. Representing agent interaction protocols in UML. In *Proc. 1st International Workshop on Agent-Oriented Software Engineering (AOSE 2000)*, *LNCS* 1957, pp. 121–140. Springer, 2001.

[12] M. P. Singh. Distributed enactment of multiagent workflows: Temporal logic for service composition. In *Proc. AAMAS*, pp. 907–914, Melbourne, July 2003.

[13] M. P. Singh. Information-driven interaction-oriented programming: BSPL, the blindingly simple protocol language. In *Proc. AAMAS*, pp. 491–498, Taipei, May 2011.

[14] M. P. Singh. LoST: Local state transfer—An architectural style for the distributed enactment of business protocols. In *Proc. 7th IEEE International Conference on Web Services (ICWS)*, pp. 57–64, Washington, DC, 2011.

[15] P. Yolum. Design time analysis of multiagent protocols. *Data and Knowledge Engineering Journal*, 63:137–154, 2007.