Splee: A Declarative Information-Based Language for Multiagent Interaction Protocols

Amit K. Chopra
School of Computing and Communications
Lancaster University
Lancaster, UK
a.chopra1@lancaster.ac.uk

Samuel H. Christie V
Department of Computer Science
North Carolina State University
Raleigh NC, USA
schrist@ncsu.edu

Munindar P. Singh
Department of Computer Science
North Carolina State University
Raleigh NC, USA
singh@ncsu.edu

ABSTRACT

The Blindingly Simple Protocol Language (BSPL) is a novel information-based approach for specifying interaction protocols that can be enacted by agents in a fully decentralized manner via asynchronous messaging. We introduce *Splee*, an extension of BSPL. The extensions fall into two broad categories: *multicast* and *roles*. In Splee, a role binding is information that is dynamically generated during protocol enactment, potentially as the content (payload) of communication between two agents. Multicast communication is the idea that a message is sent to a set of agents. The two categories of extensions are interconnected via novel features such as *set roles* (the idea that a role binding can be a set of agents) and *subroles* (the idea that agents playing a role must be a subset of agents playing another role). We give the formal semantics of Splee and give *small model* characterizations of the safety and liveness of Splee protocols. We also introduce the pragmatic idea of *query attachments* for messages. Query attachments take advantage of Splee's information-orientation, and can help restrict the information (parameter bindings) communicated in a message.

1. INTRODUCTION

A protocol is a first-class abstraction for specifying multiagent systems [4, 19, 31]. Work in multiagent protocols addresses two important concerns: the (normative) *meanings* of the operations performed by the participants and the ordering and occurrence constraints on the operations, which we refer to together as *operational constraints*. To see the distinction between meanings and operational constraints, consider the familiar setting of buyer-seller interactions. An operational constraint in this setting would be that a quote for an item cannot be sent by a seller unless a request for quotes for that item has been received from the buyer. The meaning of the quote message in this setting would be that it commits the seller to delivering the item to the buyer if the buyer pays the quoted price. In decentralized settings, the operations would correspond to communicative acts realized via asynchronous messaging between agents.

Meaning-based protocols, especially commitment protocols [22, 35, 39, 42], have been extensively studied in multiagent systems. In contrast to previous approaches, we are concerned primarily with the specification of *operational protocols*, that is, protocols that specify operational constraints. Below, we use the term *protocol* to mean operational protocols. An important challenge is designing protocol languages that can capture common interaction patterns and support the decentralized enactment of those patterns by asynchronous messaging between agents. We address interaction patterns involving subtleties of roles and multicast communication that have not been systematically addressed in protocol languages that support decentralized enactments.

We take the Blindingly Simple Protocol Language (BSPL) [36, 37, 38] as our point of departure. In contrast to other protocol languages (discussed in Section 6), BSPL is a formal, declarative, and information-based approach for specifying protocols. It naturally supports correct decentralized enactments of protocols via asynchronous messaging between agents. Although BSPL represents an important innovation in interaction protocol languages, it lacks the expressiveness for specifying certain common interaction patterns. One, in a BSPL protocol, roles are not information parameters. The effect is that BSPL does not support dynamic role bindings. Further, in a BSPL protocol, a role may not be played by more than one agent. Two, BSPL supports only point-to-point communication between agents; it does not support multicast, where an agent may send the same message instance to a set of agents.

To highlight the need for more sophisticated models, let us consider auctions as an illustrative application. In any enactment (instance) of an auction protocol, multiple agents may play bidder (not supported in BSPL), specifically, the winner must be selected dynamically from among the bidders (not supported in BSPL). Further, the winner is the bidder who has bid the highest (not supported in BSPL). Integrity requires that in a particular (single-item) auction enactment, the same item information is multicast to all bidders (not supported in BSPL). Analogous challenges would arise in specifying the Contract Net [18] (more than one agent may play contractor) as well as protocols for insurance claims, where the claims adjuster is determined on the fly from among multiple inspectors [10].

This paper seeks to overcome the shortcomings of BSPL by proposing an extended language that we dub *Splee*. The following are the main novel ideas in Splee.

Roles as information parameters. Role bindings are dynamically produced during protocol enactment.

Appears in: *Proc. of the 16th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2017), S. Das, E. Durfee, K. Larson, M. Winikoff (eds.), May 8–12, 2017, São Paulo, Brazil.*

Set roles. A role is adopted by a set of agents.

Subroles. One role may be a subrole of another, indicating that any agent who plays the first also plays the second.

Multicast. An agent may send a message to the set of agents playing a role.

Mapping keys. A protocol's key consisting of two or more parameters may be mapped into a key of one parameter. This mapping facilitates natural specification and composition.

Query attachments. A query restricts the binding generated for a parameter to specific values.

Our contributions are the syntax of Splee along with characterizations of important correctness properties, namely, liveness and safety, for Splee protocols. Notably, for a BSPL protocol, these properties may be determined from its specification alone. However, this is not necessarily so for Splee, as role bindings are dynamic and effectively require consideration of infinitely many *universes of discourse*. To overcome this challenge, we propose a "small model" characterization of these properties.

2. BACKGROUND: BSPL

BSPL is declarative: it has no control-flow abstractions. A BSPL protocol is a bag of protocols, which bottom out in messages. Instead of specifying control flow, in BSPL, one specifies information flow. Specifically, a BSPL protocol (and messages, since they are protocols as well) has *information parameters* and *causality* is explicitly specified in terms of information flow (via messaging) between agents. In other words, the messages an agent may send at any point in an enactment depend on the parameter bindings known to that agent at that point. Notably, an agent can receive any message at any point. Two, BSPL supports integrity constraints via explicit *key parameters* that functionally determine [23] other parameters and capture the idea that in any protocol enactment, an agent may not send or receive conflicting information. Causality and integrity form the bases for safety and liveness [38], which are important correctness properties pertaining to decentralized enactments of BSPL protocols.

2.1 Illustrating BSPL

Listing 1 illustrates BSPL's main concepts.

- The listing declares *Request Quote* as the name of the protocol; two roles (both public, in this case): M (merchant) and C (customer); and three parameters (all public, in this case): ID, item, and price. Parameter ID is annotated key, meaning that ID functionally determines the other parameters. All parameters are adorned ⌜out⌝, meaning that their bindings are generated by enacting the protocol, i.e., enacting the messages declared in it. The public parameters of a protocol serve as its interface and facilitate composition.
- *Request Quote* declares two message schemas (the order of their listing is irrelevant). By convention, any key parameter of the protocol is a key parameter for any message in which it appears, though a message may have additional key parameters. Thus, *request* is directed from customer to merchant and its key is ID.
- The message *request* has two parameters ID and item, each annotated ⌜out⌝, meaning that a customer can generate bindings for them when it sends an instance of *request*. In *quote*, ID and item are ⌜in⌝, meaning that a merchant may send an instance of *quote* with some bindings for ID and item, only if it has received an instance of *request* with those bindings.
- Bindings for ID identify enactments of *Request Quote*. That is, distinct tuples of bindings as allowed by the key constraints correspond to distinct enactments. A *complete* enactment of *Request Quote* corresponds to a tuple of bindings for all its public parameters.

Listing 1: A simple quote protocol in BSPL.

```
Request Quote {
  roles M, C //Merchant, Customer
  parameters out ID key, out item, out price

  C ↦ M: request [out ID, out item]
  M ↦ C: quote [in ID, in item, out price]
}
```

Singh [38] formalizes liveness and safety for BSPL and gives verification techniques. A protocol is *live* iff every enactment can complete and there is at least one enactment. *Request Quote* is live (there is only one enactment). A protocol that cannot be enacted is not live.

A protocol is *safe* iff it is not possible to produce conflicting bindings for a parameter for a given key binding in any enactment. A trivial safety violation would be if a customer sent *request* with ID "1" and item "flower," and then sent ID "1" and item "phone." This violation is trivial because it can be avoided by the customer based solely on its local knowledge. (Resending ⟨"1", "flower"⟩ though would not be a violation.) A message is *viable* iff sending it does not cause any local violation. Safety is characterized as there being no violation assuming each message emission is viable. *Request Quote* is safe. *Request Quote Unsafe* in Listing 2, which modifies *Request Quote* by adding message *desire* from C to M (indicating the price the customer desires the item for), is unsafe. Both merchant and customer can concurrently produce bindings for price; that is, a nonlocal conflict exists.

Listing 2: An unsafe protocol in BSPL.

```
Request Quote Unsafe{
 ...
 C ↦ M: desire [in ID, in item, out price]
}
```

In BSPL, roles are not parameters: they are adopted externally, not during enactment. In effect, *Request Quote* defines a multiagent system of two agents, whose names might as well be M and C.

We use auctions as our running example. A seller requests that an auctioneer put up an item for auction. The auctioneer sends a call for bids including item and starting amount (samt) to a set of bidders. One or more bidders may submit bids. The auctioneer announces the highest bidder as the winner.

The BSPL *Call for Bids* in Listing 3 is inadequate because B can be played by at most one agent in any enactment.

Listing 3: A call for bids in BSPL.

```
BSPL Call for Bids {
  roles S, A, B //Seller, Auctioneer, Bidders
  parameters out aID key, out item, out samt

  S ↦ A: request [out aID, out item]
  A ↦ B: call [in aID, in item, out samt]
}
```

2.2 Decentralization

BSPL realizes LoST (Local State Transfer) [37], which is a decentralized architecture style for multiagent systems. We summarize the basic requirements of LoST. Agents interact via *asynchronous* messaging; in particular, there is no global repository of state (*shared-nothing* setting). Each agent has a *local* (public) state that consists of the messages it has sent and received. Further, agents are *autonomous*, meaning that they may or may not send messages. An agent can send any message for which it has the requisite information.

LoST requires that no assumptions be made about message delivery order. LoST requires being able to work over lossy infrastructures that support message retransmission. In such a loosely coupled architectural setting, LoST imposes the correctness requirement that agents should have a consistent view of their interactions. Safety for BSPL, as described above, in effect, captures the consistency requirements. Liveness ensures that consistency is not maintained trivially, that is, by not acting at all.

3. SPLEE SYNTAX AND MOTIVATION

The formal syntax of Splee is given next. A superscript of + indicates one or more repetitions, and ⌊ and ⌋ delimit expressions, which are optional when without a superscript.

L_1. A protocol declaration consists of a name, public parameters, optional private parameters, and references to constituent protocols or messages. Public parameters are adorned. Both public and private parameters may have one or more optional qualifiers. The public parameters marked key form this declaration's key. In Splee, roles are parameters; qualifier role indicates if a parameter is a role.
Protocol ⟶ *Name* { *Public* ⌊*Private*⌋ *Reference*$^+$ }

L_2. Public parameters are adorned with information flow constraints and qualifiers.
Public ⟶ public ⌊*Adorned* ⌊*Qualifier*⌋⌋$^+$

L_3. Private parameters are not adorned as they are not part of a protocol's interface.
Private ⟶ private ⌊*Name Qualifier*⌋$^+$

L_4. A reference to a protocol (from a declaration) may consist of the name of the protocol appended by as many parameters as it declares.
Reference ⟶ *Name* (*Adorned*$^+$)

L_5. A reference may be a message schema, and consists of exactly one name, sending role and receiving roles, and one or more parameters. The receiving role is adorned.
Reference ⟶ *Name* ↦ *Adorned* : *Name*[*Adorned*]$^+$

L_6. *Adorned* ⟶ *Adornment Name*

L_7. Qualifiers indicate if a parameter is a key, a singleton role (role but not set), a nonsingleton role (role and set), if an agent playing a role is a member of a set of agents playing another role (subrole), and whether a parameter has a map constraint. The order of the qualifiers is irrelevant.
Qualifier ⟶ ⌊key⌋⌊role⌋⌊set⌋⌊subrole *Name*⌋ ⌊map(*Name*$^+$)⌋

L_8. An adornment is usually either ⌜in⌝ or ⌜out⌝; ⌜nil⌝ indicates an unknown parameter.
Adornment ⟶ in | nil | out

We gradually work our way up to a desirable specification of an auction protocol. Below, we use $X \to Y$ to mean that the set of parameters X *functionally determines* each parameter in the set Y [23]. We highlight five innovations below.

First innovation: *roles as information.* Supporting multiple bidders implies being able to bind multiple agents (i.e., their identifiers) with the role B, in essence, treating B as a parameter. *Multiple Bidders* in Listing 4 treats all roles as parameters. Bindings for A and S are generated by enacting *Multiple Bidders*; moreover, they are both key parameters of the protocol (in addition to aID). Role B is a private role parameter of the protocol. The sender is not adorned because its binding must be fixed when the message is sent. Notice the presence of ⌜in⌝ S in *call*: since {S, A, aid} → {item}, it cannot be omitted.

The reason *Multiple Bidders* supports multiple bidders is that B is a key parameter for *call* (following the convention that a message parameter that is a key parameter for the protocol is a key parameter for the message). Or, given an enactment of *Multiple Bidders*—by specifying for A, S, and aID–there can be many bindings for B, and therefore many call messages (one to each bidder).

Listing 4: An attempt to support multiple bidders by treating roles as information parameters.

```
Multiple Bidders {
  public out aID key, out item, out samt, out S role
       key, out A role key
  private B role key

  S ↦ out A: request [out aID, out item]
  A ↦ out B: call [in aID, in S, in item, out samt]
}
```

Second innovation: *key mapping* functions that preserve functional dependencies. We represent a mapping function as an injective function $map : 2^{String} \to String$ that has the property that if $X \to Y$, then $\{map(X)\} \to Y$. String concatenation and perfect hashes are examples of mapping functions. Listing 5 uses key mapping to transform {A, S, aID} to {nID}. That is, {nID} →{A, S, aID, item}. Key mapping facilitates specification and composition, and may help obscure the bindings involved (e.g., to hide the seller's identity from bidders). Notice that in Listing 5, S and A are no longer annotated as key parameters because nID suffices.

Listing 5: Applying a dependency-preserving key map.

```
Key Mapping {
  public out nID key map(S, A, aID), out item, out samt,
       out S role, out A role
  private B key role

  S ↦ out A: request [out aID, out item, out nID]
  A ↦ out B: call [in nID, in item, out samt]
}
```

Third and fourth innovation: *set roles* and *multicast.* Listing 5 has an important shortcoming: the auctioneer can send different bindings for samt to different bidders, which would be wrong in a (single-item) auction enactment. Different bindings are possible because B is a key parameter in *call*;

that is, {nID, B}→ {samt}. Recall that we made B key for the express purpose of supporting multiple bidders. So how can we have multiple bidders but prevent different samt bindings from being communicated to them? The solution is to treat the binding of B as a set of agents, as in Listing 6. In effect, because B is no longer key, {nID}→{samt,B}.

Listing 6: *call* is a multicast message; B is set, meaning that it can take a set of more than one agent as its binding.

```
Multicast {
  public out nID key map(S, A, aID), out item, out samt,
       out S role, out A role
  private B role set

  S ↦ out A: request [out aID, out item, out nID]
  A ↦ out B: call [in nID, in item, out samt]
}
```

Fifth and sixth innovations: role binding *selection*, in which an agent performs a role binding and communicates it to other agents, and *subroles*. Listing 7 introduces bidding and selection of a winner. Any agent playing an individual bidder (I) is indicated to belong to the set of agents playing B. Such agents may send instances of *bid* with an amount bamt, which may be different for each bidder. The auctioneer selects the winner (W) in *wins*; an agent playing W must also be a member of B. Recall that in the actor model of computation, an actor may send another actor a third actor's name [25]. In contrast, in Splee, an agent may not only send an agent's name to another but may also send a role binding, in essence changing the type of the agent.

Listing 7: Bidding and selecting winner. For brevity, we omit the qualifier role when subrole is present.

```
Winner {
  public out nID key map(S, A, aID), out item, out samt,
       out S role, out A role, out W subrole B, out wamt
  private B role set, I key subrole B //Individual bidder

  S ↦ out A: request [out aID, out item, out nID]
  A ↦ out B: call [in nID, in item, out samt]
  I ↦ in A: bid [in nID, in item, out bamt]
  A ↦ in B: wins [in nID, out W, out wamt]
}
```

Splee protocols are subject to certain well-formedness criteria. Only role parameters may be set; if a role X is subrole of role Y, then both X and Y are role and Y is set; the sender and receiver roles in any message declaration are both role, the sender is not a set, and receiver is not adorned ⌜nil⌝; and at least one public parameter is declared key. Further, because nonkey parameters have no meaning when separated from their key, the key (or its mapped form) with which the binding of a parameter is produced must be used in any reference where the parameter is used.

4. SEMANTICS

We formalize the above intuitions, enhancing Singh's [38] semantics. The significant difference in our treatment arises from the following fact. For a BSPL protocol, the relevant *universe of discourse* consists of roles and messages in the protocol. In contrast, for a Splee protocol, there are infinitely many universes of discourse, each including a finite set of agents and the roles each agent plays in the protocol.

We give a brief overview of the formalization before laying out its details. First, we formalize the structure of a protocol. We explain how a message schema is itself an elementary protocol. We then relate a message instance to its schema. This enables us to define the history of an agent as a set of message instances. Then we define a viable message for a history as one that satisfies causality, integrity, role, and map constraints. A multiagent enactment is modeled by a history vector whose components are histories of individual agents. Viable history vectors are those that satisfy the physical constraint that a message must have been sent before it is received. To capture enactments for specific roles, message schemas, agents, and the roles they play, we introduce the idea of a universe of discourse (UoD). This enables us to define the universe of enactments of a UoD. This enables us to define enactments (instances) of a protocol, the enactment of a message schema being the base case.

For convenience, we fix the symbols by which we refer to finite lists (mostly, treated as sets) of public parameters ($\vec{p}$), public key parameters ($\vec{k} \subseteq \vec{p}$), private parameters ($\vec{q}$), private key parameters ($\vec{l} \subseteq \vec{q}$), public roles ($\vec{x} \subseteq \vec{p}$), private roles ($\vec{y} \subseteq \vec{q}$), roles that are qualified set ($\vec{z} \subseteq \vec{x} \cup \vec{y}$), subrole and mapping constraints ($\vec{c}$), ⌜in⌝ parameters ($\vec{p_I} \subseteq \vec{p}$), ⌜out⌝ parameters ($\vec{p_O} \subseteq \vec{p}$), ⌜nil⌝ parameters ($\vec{p_N} \subseteq \vec{p}$), 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 $\vec{p_N} \cap \vec{p_O} = \emptyset$. And, p refers to an individual parameter. For brevity, we rename private parameters to be distinct in each protocol, and the public parameters of a reference to match the declaration in which they occur. Throughout, we use $\downarrow_g$ to project a list to those of its elements that belong to g.

Definition 1 states that a Splee protocol (via any of its parameters) may reference another protocol (via its public parameters). The references bottom out at message schemas.

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

We treat a message schema with name m, parameters $\vec{p}$, sender role ($s \in \vec{p}$), singleton receiver role ($r \in \vec{p}$), key parameters $\vec{k}$, and constraints $\vec{c}$ as an atomic protocol with exactly two public roles (sender and receiver) and no references: $\langle m, \vec{p}, \vec{k}, \emptyset, \emptyset, \{s, r\}, \emptyset, \emptyset, \vec{c}, \emptyset\rangle$. If the receiver is qualified set, then we write it as $\langle m, \vec{p}, \vec{k}, \emptyset, \emptyset, \{s, r\}, \emptyset, \{r\}, \vec{c}, \emptyset\rangle$. For brevity, we write them as $\ulcorner m : \vec{p}(\vec{k}, s \mapsto r)\vec{c}\urcorner$ and $\ulcorner m : \vec{p}(\vec{k}, s \Mapsto r)\vec{c}\urcorner$, respectively. Also, where we do not care whether the role takes a singleton set, we simply write $\ulcorner m : \vec{p}(\vec{k}, s \rightsquigarrow r)\vec{c}\urcorner$. Usually, $\vec{k}$ and $\vec{c}$ are understood from the protocol in which the schema is referenced. Specifically, $\vec{k}$ equals the intersection of $\vec{p}$ with the key parameters of the protocol declaration and $\vec{c}$ contains any subrole constraints that pertain to either s or r and relevant *map* constraints.

Below, let $\text{roles}(\mathcal{P}) = \vec{x} \cup \vec{y} \cup \bigcup_i \text{roles}(F_i)$; $\text{params}(\mathcal{P}) = \vec{p} \cup \vec{q} \cup \bigcup_i \text{params}(F_i)$; $\text{msgs}(\mathcal{P}) = \bigcup_i \text{msgs}(F_i)$ and $\text{msgs}(\ulcorner m : \vec{p}(\vec{k}, s \rightsquigarrow r)\vec{c}\urcorner) = \{m\}$. Definition 2 assumes that message instances are unique up to the key, as in their schema.

Definition 2: A *message instance* $m[\vec{p}, \vec{v}]$ associates a message schema $\ulcorner m : \vec{p}(\vec{k}, s \rightsquigarrow r)\vec{c}\urcorner$ with a list of values, where $|\vec{v}| = |\vec{p}|$, $\vec{v} \downarrow_s$ is a singleton set of agents, and $\vec{v} \downarrow_r$ is a set of

agents, and $\vec{v} \downarrow_p = \ulcorner \mathsf{nil} \urcorner$ iff $p \in \vec{p_N}$. If the schema specified $\mapsto$, then $\vec{v} \downarrow_r$ would be a singleton set as well. For convenience, we may make more schema elements explicit, e.g., by writing $m[\vec{p}(\vec{k}), \vec{v}]$ or $m[\vec{p}(s, r, \vec{k}, \vec{c}), \vec{v}]$, as necessary.

Definition 3 captures the idea of a *history* of an agent as a sequence (equivalent to a set in our approach) of all and only the message instances emitted or received by the agent. Thus, H^α captures the local view of agent α during a protocol enactment. A history may be infinite in general. However, we assume each enactment in which a tuple of parameter bindings is generated is finite. Notice that the emission of a message instance that has more than one receiver corresponds to the emission of a set of *physical* message tokens of identical contents, one to each receiver. We assume infrastructure supports sending such message tokens.

Definition 3: A *history* of an agent α, written H^α, is given by a sequence of zero or more message instances $m_1 \circ m_2 \circ \ldots$ ($\circ$ means sequence). Each m_i is of the form $m[\vec{p}(s, r), \vec{v}]$ and either $\alpha \in \vec{v} \downarrow_s$ or $\alpha \in \vec{v} \downarrow_r$.

Definition 4 captures the idea that what an agent knows at a history is exactly given by what the agent has seen so far in terms of incoming and outgoing messages. Here, 2(i) ensures that the message instance under consideration does not violate the uniqueness of the bindings; 2(ii) ensures that the agent knows the binding for each $\ulcorner \mathsf{in} \urcorner$ parameter and not for any $\ulcorner \mathsf{out} \urcorner$ or $\ulcorner \mathsf{nil} \urcorner$ parameter; 2(iii) ensures that the instance satisfies *map* constraints; and 2(iv) ensures that the subrole constraints are satisfied.

Definition 4: A message instance $m[\vec{p}(s, r, \vec{k}, \vec{c}), \vec{v}]$ is *viable* at H^α iff (1) $\alpha \in \vec{v} \downarrow_r$ (reception) or (2) $\alpha \in \vec{v} \downarrow_s$ (emission) and (i) $(\forall m_i[\vec{p_i}, \vec{v_i}] \in H^\alpha$ if $\vec{k} \subseteq \vec{p_i}$ and $\vec{v_i} \downarrow_{\vec{k}} = \vec{v} \downarrow_{\vec{k}}$ then $\vec{v_i} \downarrow_{\vec{p} \cap \vec{p_i}} = \vec{v} \downarrow_{\vec{p} \cap \vec{p_i}})$, (ii) $(\forall p \in \vec{p} : p \in \vec{p_I}$ iff $(\exists m_i[\vec{p_i}, \vec{v_i}] \in H^\alpha$ and $p \in \vec{p_i}$ and $\vec{k} \subseteq \vec{p_i}))$, (iii) every map constraint in $\vec{c}$ is satisfied in $\vec{v}$, and (iv) (for every constraint s $\mathsf{subrole}$ g in $\vec{c}$, $\exists m_i[p_i, v_i] \in H^\alpha$ such that $\alpha \in \vec{v_i} \downarrow_g)$, and (for every constraint r $\mathsf{subrole}$ g in $\vec{c}$, $\exists m_i[p_i, v_i] \in H^\alpha$ such that $\vec{v} \downarrow_r \subseteq \vec{v_i} \downarrow_g)$.

Definition 5 captures that a *history vector* for a protocol is a vector of histories of agents that together are causally sound: a message instance is received only if it has been emitted [30].

Definition 5: A *history vector* for a finite set of agents $\mathcal{A}$ is $[H^1, \ldots, H^{|\mathcal{A}|}]$ such that ($\forall a \in \mathcal{A}$: H^α is a component in the vector and ($\forall m[\vec{p}(s, r), \vec{v}] \in H^\alpha$: if $\alpha \in \vec{v} \downarrow_r$, $\beta \in \vec{p} \downarrow_s$, then $m[\vec{p}(s, r), \vec{v}] \in H^\beta$)).

A history vector records the progression of a protocol enactment. Under the above causality restriction, a vector that includes a reception must have progressed from a vector that includes the corresponding emission. Further, we avoid the FIFO assumption about message delivery. The viability of the messages emitted by any agent ensures that the progression is epistemically correct with respect to each agent.

Definition 6: A history vector for a finite set of agents $\mathcal{A}$, $[H^1, \ldots, H^{|\mathcal{A}|}]$, is *viable* iff either (1) each of its component 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 instance by one of its agents, i.e., $(\exists i, m_j : H^i = H'^i \circ m_j$ and $[H^1, \ldots, H'^i, H^{|\mathcal{A}|}]$ is viable).

Definition 7 introduces a *universe of discourse (UoD)*. Definition 8 defines viable history vectors relative to a UoD. 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 8 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 under our assumptions. We include private parameters in the intension to support compositionality. In the last stage, we project the intension to the public parameters.

Definition 7: A *UoD* is a tuple $\langle \mathcal{R}, \mathcal{M}, \mathcal{A}, \mathcal{X} \rangle$ 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 $\mathcal{R}$; $\mathcal{A}$ is a finite set of agents; and $\mathcal{X} \subseteq \mathcal{R} \times \mathcal{A}$. Tuple $(\rho, a) \in \mathcal{X}$ means that agent a plays role ρ.

Definition 8: A *viable history vector for a UoD* $\langle \mathcal{R}, \mathcal{M}, \mathcal{A}, \mathcal{X} \rangle$ is a viable history vector $[H^1, \ldots, H^j]$ such that (1) exactly j distinct agents appear in $\mathcal{X}$ and each such agent has a history in the vector, and (2) each message instance in any history in the vector instantiates a schema in $\mathcal{M}$. The *universe of enactments* for that UoD, $\mathcal{U}_{\mathcal{R},\mathcal{M},\mathcal{A},\mathcal{X}}$, is the set of viable history vectors for that UoD.

Definition 9 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 instance of the schema occurs in both its sender and each receiver's histories.

Definition 9: The intension of a message schema is given by: $[\![\ulcorner m : \vec{p}(s \rightsquigarrow r) \urcorner]\!]_{\mathcal{R},\mathcal{M},\mathcal{A},\mathcal{X}} = \{H | H \in \mathcal{U}_{\mathcal{R},\mathcal{M},\mathcal{A},\mathcal{X}}$ and $(\exists \alpha, \vec{v}, i : H_i^\alpha = m[\vec{p}(s, r), \vec{v}]$ and $\alpha \in \vec{v} \downarrow_s$ and $(\forall \beta \in \vec{v} \downarrow_r \exists j : H_j^\beta = m[\vec{p}(s, r), \vec{v}]))\}$.

As for BSPL, a Splee protocol completes when all its public parameters are bound. A (composite) protocol completes if one or more of subsets of its references completes. Informally, each such subset contributes all the viable interleavings of the enactments of its members, i.e., the intersection of their intensions. Definition 10 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 10: Let $\mathcal{P} = \langle n, \vec{p}, \vec{k}, \vec{q}, \vec{l}, \vec{x}, \vec{y}, \vec{z}, \vec{c}, F \rangle$ be a protocol. Let $\text{cover}(\mathcal{P}, G) \equiv G \subseteq F | (\forall p \in \vec{p} : (\exists G_i \in G : G_i = \langle n_i, p_i, k_i \rangle$ and $p \in \vec{p_i}))$; and $\mathcal{P}$'s intension, $[\![\mathcal{P}]\!]_{\mathcal{R},\mathcal{M},\mathcal{A},\mathcal{X}} = (\bigcup_{\text{cover}(\mathcal{P},G)} (\bigcap_{G_i \in G} [\![G_i]\!]_{\mathcal{R},\mathcal{M},\mathcal{A},\mathcal{X}}))\downarrow_{\vec{p}}$.

Definition 11 define the notion of a UoD of a protocol as involving only $\mathcal{P}$'s roles and messages (including its references recursively). A protocol UoD would vary with the set of agents and the roles they play in the protocol.

Definition 11: A *UoD of a protocol* $\mathcal{P}$, written $\text{UoD}(\mathcal{P})$, is a UoD $\langle \mathcal{R}, \mathcal{M}, \mathcal{A}, \mathcal{X} \rangle$, where $\mathcal{R} = \text{roles}(\mathcal{P})$ and $\mathcal{M} = \text{msgs}(\mathcal{P})$. By $\text{allUoD}(\mathcal{P})$, we refer to the set of all $\text{UoD}(\mathcal{P})$.

4.1 Naïve Properties

Safety means that for any key value, 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. Definitions 12 and 13 characterize these properties *weakly*—meaning they require only the existence of an appropriate UoD (i.e., an appropriate set of agents and their roles bindings).

Definition 12: A protocol $\mathcal{P}$ is weakly safe iff $(\exists U : U \in \text{allUoD}(\mathcal{P})$ and each history vector in $[\![\mathcal{P}]\!]_U \neq \emptyset$ is safe). A

history vector is safe iff all key uniqueness constraints apply across all histories in the vector.

Definition 13*:* A protocol $\mathcal{P}$ is weakly live iff ($\exists U : U \in$ allUoD($\mathcal{P}$) and each history vector in $\mathcal{U}_U$ can be extended through a finite number of message emissions and receptions to a history vector in $\mathcal{U}_U$ that is complete).

Ideally, we would prefer *strong* characterizations where satisfaction did not depend upon a UoD. Definition 14 gives a strong version of liveness, with safety being analogous.

Definition 14*:* A protocol $\mathcal{P}$ is strongly live iff ($\forall U : U \in$ allUoD($\mathcal{P}$) implies each history vector in $[\![\mathcal{P}]\!]_U$ can be extended through a finite number of message emissions and receptions to a history vector in $\mathcal{U}_U$ that is complete).

However, the strong versions would be *too strong*: clearly, there exists a UoD for any protocol in which it is not live. For example, the auction protocol in Listing 7 is not live if no agent plays auctioneer.

Because for any protocol $\mathcal{P}$, allUoD($\mathcal{P}$) is an infinite set, we employ model abstraction techniques to make the problem of verifying safety and liveness for the protocol tractable. Next, we describe a method for statically verifying Splee protocols for liveness and safety.

4.2 Distinct UoDs

In general, an agent may adopt more than one role in a protocol. Liveness and safety of a protocol should not rely upon the same agent adopting two or more roles. In a protocol, if any two roles were meant to be adopted by one agent, then the protocol is ill-conceived (we should simply merge the two roles into one). A *distinct UoD* is one where each role is played by a distinct agent except when the roles are connected by a subrole relationship. Formally, for any UoD of a protocol, we can generate a distinct UoD by substituting new agent identifiers for every agent that agent that plays multiple roles in the original UoD, none of which are connected by a subrole relationship. For example, if agent a plays both roles ρ_1 and ρ_2, we generate new agent identifiers $a\rho_1$ and $a\rho_2$ for the distinct UoD and substitute them for the original agent by binding them to the roles in its place: $a\rho_1$ with ρ_1 and $a\rho_2$ with ρ_2.

4.3 Canonical UoDs

Informally, our strategy is the following. For modularity, we consider only distinct UoDs for a protocol. Later we show how to handle the case where agents play multiple roles.

The first step is to reduce arbitrary (distinct) UoDs to UoDs with a canonical structure. We define a *large* UoD for $\mathcal{P}$ as one where every role is played by some agent and if the protocol has set roles, then at least one of the roles is played by three or more agents. A *canonical* UoD for $\mathcal{P}$ is one where every role is played by some agent and every set role is played by exactly two agents. The intuition behind every set role played by two agents (not fewer) in a canonical UoD is that if correctness relied upon fewer than two agents adopting a set role, then it would be pointless to declare it a set role. Since agent identities are not important, we can generate a unique canonical UoD for a protocol where for any nonset role ρ, its agent identifier is $a\rho$ and for set role ρ', its agent identifiers are $a_0\rho'$ and $a_1\rho'$.

Theorem 1*:* A protocol $\mathcal{P}$ is safe in a large UoD for $\mathcal{P}$ iff $\mathcal{P}$ is safe in the canonical UoD for $\mathcal{P}$.

Proof. In the forward direction, we set up an inductive argument: essentially we show that an additional agent playing a role preserves unsafety. The base case is that of a canonical UoD U_C. Let some role ρ in U_C be played by agents a_0 and a_1. Let U_C^+ be identical to U_C except that a_3 plays ρ as well. Let's assume U_C is unsafe, which means there is a minimal history vector H of $\mathcal{P}$ in U_C that is unsafe. We construct a history vector H' that is identical to H except that a_3's history consists only of receipts of messages (a_3 does not send any message) and some other agent sends all the messages that a_3 sent in H. H' would be unsafe as well. By induction over additions, any large UoD for $\mathcal{P}$ would be unsafe as well.

In the converse direction, we set up an analogous inductive argument, but this time for deletions from large UoDs. Let U_L be a large UoD of $\mathcal{P}$. Let $\mathcal{P}$ be unsafe in U_L. Then there must be some role ρ that is played agents $a_0, a_1, \ldots, a_n$ for $n \geq 3$. We construct U_L^- that is identical to U_L, except that a_n does not play ρ in U_L^-. If $\mathcal{P}$ is unsafe in U_L, then there must be a minimal history vector that must be unsafe. In U_L^-, there will be a corresponding history vector such that some $a_{(n-1)}$ would have received the same information as a_n and therefore is enabled to act as a_n did. Hence that history vector will be unsafe as well. □

Theorem 2*:* Let $\mathcal{P}$ be safe. Then $\mathcal{P}$ is live in a large UoD for $\mathcal{P}$ iff $\mathcal{P}$ is live in the canonical UoD for $\mathcal{P}$.

Proof. In the forward direction, we consider U_L and U_L^- as usual. Then every maximal (no messages can be sent or received) viable history vector of $\mathcal{P}$ in U_L is complete. In U_L^- (which differs from U_L in that a_n does not play ρ), let's assume there is a maximal history vector H that is not complete. Then it must be because a_n does not play ρ. However, any message that was received and sent by a_n can be received and sent by $a_{(n-1)}$. In fact, a_n's history can be interleaved with $a_{(n-1)}$ as long as the interleaving is consistent with the individual histories. This is because the protocol is assumed safe: there were no nonlocal conflicts and merging histories cannot produce local conflicts, which would block the enactment short of completion. Therefore, our assumption that H is a maximal incomplete history vector must be incorrect. By induction over deletions, the canonical UoD is live as well.

In the converse direction, we consider a canonical UoD U_C and U_C^+. Let us assume that U_C is live but U_C^+ is not live, that is, there exists a maximal viable history H of $\mathcal{P}$ in U_C^+ that is not complete. Then, it must be incomplete because of a_3 (the additional agent) playing ρ. That means a_3 has sent or received a message that is causing the agents to block. However, any message that a_3 can send or receive, a_2 can send or receive as well because of our assumption of safety. That means our assumption that U_C^+ is not live must be false. By induction, any larger U_L must be live as well. □

Finally, we discharge our assumption of using only distinct UoD in the above theorems.

Theorem 3*:* $\mathcal{P}$ is safe in a UoD iff it is safe in the UoD's distinct UoD.

Proof. Let U be a UoD. Let U' be U's distinct UoD. In the forward direction, we are essentially splitting (projecting) each agent's history in any enactment in U to multiple histories in an enactment in U' (one for each role it plays in U), which will not introduce a new safety violation. In the converse direction, if U is distinct we are done. If U is not

distinct, then there is an agent a that plays two roles ρ and ρ' that were played by distinct agents in U'. Combining histories can only reduce conflicts; so if there were no conflicts in any history vector of $\mathcal{P}$ in U', then there are no conflicts in any history of $\mathcal{P}$ in U. □

Theorem 4*:* Assume $\mathcal{P}$ is safe in a UoD U. $\mathcal{P}$ is live in U iff it is live in U's distinct UoD U'.

Proof. In the forward direction, the projection to multiple histories (as described above) will preserve information flow. Hence liveness will be preserved. In the converse direction, if U is distinct we are done. If U is not distinct, then there is an agent a that plays two roles ρ and ρ' that were played by distinct agents in U'. If $\mathcal{P}$ is live in U' but not live in U, that means upon combining the agents into one in U, some nonlocal choice that was causing the conflict is now manifesting as a local choice in the combined history. However, according to our assumption, $\mathcal{P}$ is safe in the UoD; therefore there cannot be such a nonlocal choice. Hence, our assumption that $\mathcal{P}$ is not live in U must be false. □

4.4 Reducing Splee to BSPL

In BSPL, the UoD for a protocol that contains $\mathcal{R}$ roles and $\mathcal{M}$ messages is given by $\langle \mathcal{R}, \mathcal{M} \rangle$ [38]. Notably, a BSPL-UoD does not refer to agents, which makes static checking of liveness and safety possible for BSPL protocols. For Splee, we have mapped the problem of checking liveness and safety for arbitrary UoDs to distinct "small-model" canonical UoDs. However, we cannot yet perform static checking for Splee protocols because their canonical UoDs refer to agents. We now give a method for mapping a protocol's canonical UoD to a BSPL-UoD. Recall that the canonical model has one agent for every nonset role and two agents for every set role. Let U be $\mathcal{P}$'s canonical model. For a Splee protocol $\mathcal{P}$, we construct a BSPL protocol $\mathcal{B}$ that is identical to $\mathcal{P}$ except that each occurrence of a set role in $\mathcal{P}$ is substituted by two new roles in $\mathcal{P}$. The idea is that each of the new roles identifies an agent playing the original role in U. For a message in $\mathcal{P}$, where the set role occurs, this means we replace it by two copies of the message (with distinct names) in $\mathcal{B}$, one for each of the new roles. The BSPL-UoD of $\mathcal{B}$ contains only the roles and messages that occur in $\mathcal{B}$. We refer to this construction as *Splee-to-BSPL* mapping.

Theorem 5*:* A protocol $\mathcal{P}$ is live (analogously, safe) in a canonical UoD iff its Splee-to-BSPL mapping $\mathcal{B}$ is live (analogously, safe).

Proof. By the construction above, which gets rid of agents but in doing so introduces a role for each agent and suitably modifies the messages as well so that they refer to the new roles. Hence, the set of viable history vectors in both UoDs is identical. □

With this step (Theorem 5), we have given a method for statically verifying the liveness and safety of a Splee protocol in large UoDs by mapping it to a BSPL verification problem. Verification for strictly smaller UoDs (e.g., one where a set role is adopted by only one agent) can be performed by reduction to BSPL verification problems, as described above. However, smaller UoDs are oddities in the sense that set roles would have only singleton bindings or nonset roles would have no bindings.

5. QUERY ATTACHMENTS

Listing 7 does not constrain how the winner is determined—in our auction, the binding for W must be the highest bidder and the binding for wamt must be its bid. This constraint could be captured as a norm specification, e.g., the auctioneer commits to choosing the bindings as described [13]. That is, at the Splee level, the auctioneer may announce anyone as the winner; however, at the meaning-level, the auctioneer's commitment could be violated. Such a formulation would expand agent autonomy but diminish error checking.

Alternatively, to enhance error checking, we can constrain the bindings of ⌜out⌝ parameters via *query attachments*. Since each message schema corresponds to a database relation, the parameters could be bound based upon the result of queries on message instances an agent has recorded in its (local) database [37]—any other bindings would be illegal. The queries can be written in a language compiles easily into SQL or another practical language. In Listing 8, wamt is constrained to be maximum bamt and W is constrained to the bidder who bid wamt in the SQL-like query attachment.

Listing 8: Building upon Listing 7, an auction protocol in Splee; winner is determined via a query attachment.

```
Auction {
  .../* All of Winner (Listing 7) up to and including
       schema bid*/
  A ↦ in B: wins [in nID, out W, out wamt]
   :: select this.wamt = ifnull(max(bamt),'No bid'),
             this.W= ifnull(I,'No bid')
      from bid where nID = this.nID
      group by nID;
}
```

This tension between autonomy and error checking is the tension between *regulation* and *regimentation* [28], with error checking representing a higher degree of regimentation. The degree of regimentation is a design choice for protocol designers. In some situations, a designer may find it appropriate to introduce Splee query attachments so that certain kinds of errors are ruled out and desirable implementations are more easily constructed.

Notice that *wins* can be sent when no bids have been received. In such a case, the query binds both W and wamt to the value No bid. Query attachments are not a way to control the flow of message instances. Instead, they should be regarded as a way to *narrow* the set of possible bindings for message instances an agent is allowed to emit. Intuitively, query attachments affect the viability of messages (Definition 4) in the same way that *map* constraints do. However, a formalization of attachments is out of our present scope.

6. CONCLUSION

Splee is a generalization of BSPL in the direction of dynamic role bindings and multicast. A fundamental enhancement is to make roles themselves information parameters that take agents as values. We introduced associated enhancements such as set roles and subroles, without which specifying important interaction patterns such as auctions would be impossible. We emphasize that although auctions have been specified and implemented in various multiagent languages and approaches e.g., [20, 33, 35], those are not protocol languages that support decentralized enactment. Finally, we introduced the idea of query attachments in order to constrain generated parameter bindings to specific values. The benefit of attachments is to simplify and make explicit the computation that agent developers would oth-

erwise specify in every agent implementation. The queries here could be applied automatically in protocol adapters.

Splee is a significant enhancement of BSPL. Whereas, in BSPL, roles are the locus of causality and integrity, in Splee, agents are the locus of causality and integrity (see Definition 4). This generalization, however, makes static verification of liveness and safety of Splee protocols challenging. The challenge arises from the fact that information about agents and their bindings to roles is not part of the protocol specification. To address this challenge, we introduced small model characterization of liveness and safety for Splee protocols and gave a mapping from these to the liveness and safety of BSPL protocols, which can be verified statically. Small model abstractions have been studied for other multiagent verification problems [29, 32] but not for operational protocols.

Splee treats multicast messages instances as an integral unit. Following BSPL, Splee's declarative semantics and the use of $\ulcorner\texttt{out}\urcorner$ captures Austin's [5] idea of the constitutive meaning of a communicative act by "declaring" a specified parameter binding. Under Austin's doctrine, a copy of a message or its information content carries a different and weaker meaning than the original message. As such, multicast cannot be properly simulated by generating message copies because only the first would have constitutive force.

An agent's state may be considered as consisting of two distinct components: internal and local ("public") [19, 37]. The local component consists of an agent's communications, both sent and received. A query attachment may refer only to the local component; thus computing it does not constitute an agent's internal reasoning. For example, calculating the winner involves aggregating over bids, which are communications. Referring to any element of internal state in a query would be unsound as it would introduce false coupling between the protocol and specific agent implementations, not only making the protocol nonreusable but also making the determination of compliance impossible.

Operational protocols. AUML [33], Message Sequence Charts [26], $\mathcal{RASA}$ [31], HAPN [41], and WS-CDL [40] are all languages for specifying operational protocols. In these languages, a protocol specification takes the form of a control-flow specification. Correctly enacting a protocol specified in terms of control flow is challenging in decentralized settings with multiple loci of control. Therefore, these languages may only be used to specify highly synchronous protocols. AUML notably supports annotations of cardinality of a role or a message; however, the annotations, like the rest of AUML, are informal.

Alternative approaches for specifying protocols—in logic [2, 21], process algebras [3], organization-oriented languages [20], and AOSE methodologies such as Prometheus [34]—typically do not address decentralization, as Splee does.

Meaning-based protocols. Operational protocols and meanings are distinct concerns. To see this, we need only realize that the same operations may feature in multiple meaning-based protocols. For example, a *price* message may mean an offer to sell in e-commerce or last price in stock markets. Operational protocols specify causality and integrity constraints whereas meaning-based protocols specify how operations progress meanings. Traditional work on meaning-based protocols mixes concerns of operation and meaning in the same specification (e.g. [7, 35, 42]). Recent work on meaning though [12, 11, 13] has begun to consider operational aspects as outside its scope, effectively modularizing meaning-based protocols with respect to operational protocols. BSPL and Splee provide declarative bases for operations to enable declarative meaning-based protocols.

Baldoni et al. [6] present a platform that has a central commitment store for coordination among agents. Chopra and Singh [12] present a decentralized architecture for enacting commitments. An important direction is to understand how decentralized meaning-based protocols may be realized via Splee protocols.

Orchestration. An alternative to protocol-based coordination of agents is *orchestration*. In these approaches, an agent orchestrates its interactions with other agents to achieve some desired outcome. Examples of such approaches include Reo [16] and Orc [15]. Orchestration is conceptually different from protocols. Orchestration is grounded in agents whereas protocols, especially as we model them, abstract away from agents entirely. Another way to look at the difference is that in protocol-based approaches, the units of composition are protocols, whereas in orchestration-based approaches, the units of composition are agents (potentially complex, representing entire organizations). In general, composing protocols leads to enhanced reusability.

Agents and roles. The idea of a role being adopted by multiple agents is supported in a number of multiagent specification languages, e.g., [20]. However, these languages do not address decentralized enactment.

Some works [1, 17, 24, 27] formalize compliance between agent specifications and role, protocol, or organizational specifications. Although valuable, verifying compliance is outside our present scope. Notice though that protocols modularize verification since agents need not be verified with each other; each agent need only be verified with the protocol.

Programming. Ultimately, specifying protocols makes the development of agents easier. This is because for any application, the protocol abstracts out the interactive part of the application logic that would otherwise have to be implemented in each agent in that application. For instance, in the absence of dynamic role bindings, agent developers would have to either hard code them in each agent or rely on some external discovery mechanism. In the absence of multicast support in protocol languages, developers would have to implement multicast correctly in each agent. Modeling applications via protocols is the essence of Interaction-Oriented Software Engineering (IOSE) [14].

In programming frameworks such as JaCaMo [8], agents communicate via a *shared* CArtAgo environment. Splee by contrast is *shared nothing*; it conceptually avoids entities such as an environment. In decentralized settings with one CArtAgo component per agent with asynchronous messaging, one would need Splee to specify how those components interoperate. Supporting Splee in frameworks such as JaCaMo, potentially by incorporating a middleware in the form of protocol adapters [37], would considerably enhance the capabilities of the frameworks.

Acknowledgments

Thanks to the anonymous reviewers for helpful comments. Chopra was supported by the EPSRC grant EP/N027965/1 (*Turtles*). Christie and Singh were partially supported by the NCSU Laboratory of Analytic Sciences.

REFERENCES

[1] Y. Abushark, J. Thangarajah, T. Miller, and J. Harland. Checking consistency of agent designs against interaction protocols for early-phase defect location. In *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 933–940. IFAAMAS, 2014.

[2] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Verifiable agent interaction in abductive logic programming: The SCIFF framework. *ACM Transactions on Computational Logic*, 9(4), 2008.

[3] D. Ancona, D. Briola, A. Ferrando, and V. Mascardi. Global protocols as first class entities for self-adaptive agents. In *Proceedings of the Fourteenth International Conference on Autonomous Agents and Multiagent Systems*, pages 1019–1029, Istanbul, 2015. IFAAMAS.

[4] F. Arbab. Puff, the magic protocol. In *Formal Modeling: Actors, Open Systems, Biological Systems*, pages 169–206. Springer, 2011.

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

[6] M. Baldoni, C. Baroglio, and F. Capuzzimati. A commitment-based infrastructure for programming socio-technical systems. *ACM Transactions on Internet Technology*, 14(4):23:1–23:23, 2014.

[7] M. Baldoni, C. Baroglio, E. Marengo, V. Patti, and F. Capuzzimati. Engineering commitment-based business protocols with the 2CL methodology. *Autonomous Agents and Multi-Agent Systems*, 28(4):519–557, 2014.

[8] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 78(6):747–761, June 2013.

[9] A. Bond and L. Gasser, editors. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann, San Francisco, 1988.

[10] S. Browne and M. Kellett. Insurance (motor damage claims) scenario. Document Identifier D1.a, CrossFlow consortium, 1999.

[11] A. K. Chopra and M. P. Singh. Cupid: Commitments in relational algebra. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 2052–2059, 2015.

[12] A. K. Chopra and M. P. Singh. Generalized commitment alignment. In *Proceedings of the Fourteenth International Conference on Autonomous Agents and Multiagent Systems*, pages 453–461. IFAAMAS, 2015.

[13] A. K. Chopra and M. P. Singh. Custard: Computing norm states over information stores. In *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1096–1105, Singapore, May 2016. IFAAMAS.

[14] A. K. Chopra and M. P. Singh. From social machines to social protocols: Software engineering foundations for sociotechnical systems. In *Proceedings of the 25th International World Wide Web Conference*, pages 903–914, Montréal, 2016.

[15] W. Cook and J. Misra. Computation orchestration: A basis for wide-area computing. *Software and Systems Modeling*, 6(1):83–110, March 2007.

[16] M. Dastani, F. Arbab, and F. S. de Boer. Coordination and composition in multi-agent systems. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 439–446, Utrecht, The Netherlands, July 2005. ACM Press.

[17] M. Dastani, V. Dignum, and F. Dignum. Role-assignment in open agent societies. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 489–496, 2003.

[18] R. Davis and R. G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63–109, 1983. Reprinted in [9].

[19] N. Desai, A. U. Mallya, A. K. Chopra, and M. P. Singh. Interaction protocols as design abstractions for business processes. *IEEE Transactions on Software Engineering*, 31(12):1015–1027, Dec. 2005.

[20] M. d'Inverno, M. Luck, P. Noriega, J. A. Rodriguez-Aguilar, and C. Sierra. Communicating open systems. *Artificial Intelligence*, 186:38–94, July 2012.

[21] U. Endriss, N. Maudet, F. Sadri, and F. Toni. Protocol conformance for logic-based agents. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 679–684, 2003.

[22] N. Fornara and M. Colombetti. Operational specification of a commitment-based agent communication language. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 535–542. ACM Press, July 2002.

[23] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Pearson, 2nd edition, 2008.

[24] A. Günay, M. Winikoff, and P. Yolum. Dynamically generated commitment protocols in open systems. *Autonomous Agents and Multi-Agent Systems*, 29(2):192–229, 2015.

[25] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[26] ITU. Message sequence chart (MSC), Apr. 2004. http://www.itu.int/ITU-T/2005-2008/com17/languages/Z120.pdf.

[27] A. S. Jensen, V. Dignum, and J. Villadsen. A framework for organization-aware agents. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, pages 1–36, Jan. 2016. Online.

[28] A. J. I. Jones and M. J. Sergot. On the characterisation of law and computer systems: The normative systems perspective. In J.-J. C. Meyer and R. J. Wieringa, editors, *Deontic Logic in Computer Science: Normative System Specification*, chapter 12, pages 275–307. John Wiley and Sons, Chichester, UK, 1993.

[29] P. Kouvaros and A. Lomuscio. Parameterised verification for multi-agent systems. *Artificial Intelligence*, 234:152–189, May 2016.

[30] L. Lamport. Time, clocks, and the ordering of events

in a distributed system. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[31] T. Miller and J. McGinnis. Amongst first-class protocols. In *Proceedings of the 8th International Workshop on Engineering Societies in the Agents World (ESAW 2007)*, volume 4995 of *Lecture Notes in Computer Science*, pages 208–223. Springer, 2008.

[32] M. Montali, D. Calvanese, and G. D. Giacomo. Verification of data-aware commitment-based multiagent system. In *Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems*, pages 157–164, Paris, May 2014. IFAAMAS.

[33] J. Odell, H. V. D. Parunak, and B. Bauer. Representing agent interaction protocols in UML. In *Proceedings of the 1st International Workshop on Agent-Oriented Software Engineering (AOSE 2000)*, volume 1957 of *Lecture Notes in Computer Science*, pages 121–140, Toronto, 2001. Springer.

[34] L. Padgham and M. Winikoff. Prometheus: A practical agent-oriented methodology. In B. Henderson-Sellers and P. Giorgini, editors, *Agent-Oriented Methodologies*, chapter 5, pages 107–135. Idea Group, Hershey, Pennsylvania, 2005.

[35] J. Pitt, L. Kamara, and A. Artikis. Interaction patterns and observable commitments in a multi-agent trading scenario. In *Proceedings of the 5th International Conference on Autonomous Agents*, pages 481–488, 2001.

[36] M. P. Singh. Information-driven interaction-oriented programming: BSPL, the Blindingly Simple Protocol Language. In *Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems*, pages 491–498, 2011.

[37] M. P. Singh. LoST: Local State Transfer—An architectural style for the distributed enactment of business protocols. In *Proceedings of the 9th IEEE International Conference on Web Services (ICWS)*, pages 57–64, Washington, DC, 2011. IEEE Computer Society.

[38] M. P. Singh. Semantics and verification of information-based protocols. In *Proceedings of the 11th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 1149–1156, Valencia, Spain, June 2012. IFAAMAS.

[39] M. Winikoff, W. Liu, and J. Harland. Enhancing commitment machines. In *Proceedings of the 2nd International Workshop on Declarative Agent Languages and Technologies (DALT)*, volume 3476 of *LNAI*, pages 198–220, Berlin, 2005. Springer-Verlag.

[40] WS-CDL. Web services choreography description language version 1.0, Nov. 2005. www.w3.org/TR/ws-cdl-10/.

[41] N. Yadav, M. Winikoff, and L. Padgham. HAPN: Hierarchical Agent Protocol Notation. In *Proceedings of the International Workshop on Coordination, Organisation, Institutions and Norms in Multi-Agent Systems (COIN@IJCAI)*, 2015.

[42] P. Yolum and M. P. Singh. Flexible protocol specification and execution: Applying event calculus planning using commitments. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 527–534, Bologna, July 2002. ACM Press.