Pippi: Practical Protocol Instantiation

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

Amit K. Chopra
Lancaster University
Lancaster, UK
amit.chopra@lancaster.ac.uk

Munindar P. Singh
North Carolina State University
Raleigh, NC, USA
mpsingh@ncsu.edu

ABSTRACT

A protocol specifies interactions between roles, which together constitute a multiagent system (MAS). Enacting a protocol presupposes that agents are bound to the its roles. Existing protocol-based approaches, however, do not adequately treat the practical aspects of how roles bindings come about.

Pippi addresses this problem of MAS instantiation. It proposes the notion of a metaprotocol, enacting which instantiates a MAS suitable for enacting a given protocol. Pippi demonstrates the subtleties involved in instantiating MAS arising from protocol composition, correlation, and decentralization. To address these subtleties and further support practical application patterns, we introduce an enhanced protocol language, with support for parameter types (including role and protocol typed parameters, for metaprotocols), interface flexibility, and binding constraints. We discuss the realization of our approach through an extended agent architecture, including the novel concept of a MAS adapter for contact management. We evaluate Pippi's expressiveness by demonstrating common patterns for agent discovery.

1 INTRODUCTION

Multiagent systems (MASs) can be specified from multiple perspectives, including organizations, commitments, and interactions. Yet, from a practical standpoint, how do agents come together to instantiate a MAS?

We focus on multiagent systems specified through interaction protocols [15], which are characterized by the messages that the agents send to and receive from one another and the constraints on the ordering and occurrence of those messages. For such a specification, the agents take on *roles* in a protocol, which specifies the messages each role may send and receive. Most protocol specification languages focus on the interaction itself and the order in which messages may be sent and received, with little consideration of how the roles are bound to agents. From a theoretical standpoint, this is a reasonable assumption; role binding can be left as an implementation detail. But for practical applications of interaction protocols, it is important to consider how each agent learns of its role and how each agent learns the bindings of the other roles it must interact with.

Composition is a longstanding and challenging theme for practical applications [21]. Protocol composition presents crucial challenges for both modeling and verification [10] because protocols focus on interaction and interaction in a decentralized setting is nontrivial to deal with. Traditional protocol languages rigidly specify the sequences of events that agents can observe, making composition difficult. Information-based protocol languages such as Splee [7] and BSPL [31] support composition but have their own limitations. BSPL protocols have fixed conditions for completion, meaning that protocols only have two states, complete and incomplete, and must therefore hide any nuances. BSPL and Splee protocol key constraints require a composition to have a common key, imposing on their constituent protocols that would otherwise have their own keys. Finally, current implementations propagate meaning from the outermost composition into each constituent protocol by substituting the parameter names; from a conceptual point of view, this is correct (the meaning of each constituent does change according to its use in the composition), but for modularity each agent should only need to know the protocols it supports, not the compositions they are used in.

We adopt the name Pippi (after the famous character in children's literature) for our approach because of its similarity with PPI (for Practical Protocol Instantiation).

1.1 Scenario: Wedding

To illustrate some of these concepts, we adopt a marriage scenario. The scenario starts with one agent, the proposer, who would like to get married to another. The proposer is initialized as part of a broader MAS context and can select potential partners from that context. Once the proposer has selected a partner, they then propose marriage; that is, they propose that the two agents enact a ceremony (protocol) together that will produce a social result of marriage. When the proposee has agreed to go through the ceremony (possibly selecting which form of ceremony they will enact), they can begin; note that this agreement to enact the ceremony is *not* the same thing as committing to the marriage.

As they enact the marriage ceremony, they will need to bring in other agents to participate; for example, in a court wedding, they need a judge to officiate and a witness to sign the license. It should be possible to ask someone to witness at the last second, and the witness shouldn't need to know any details about the rest of the ceremony.

1.2 Objectives and Novelty

The scenario above presents challenges, which we address:

Dynamic Role Binding. How do agents come together at runtime to enact a protocol?

In the wedding scenario, the proposer starts by inviting their prospective spouse, a judge, and a witness. The proposer must communicate not only each invitee's role in the protocol but also the roles of the other agents they must interact with. The proposer may prefer to use *late binding* to invite those agents only when their role becomes necessary, to protect privacy for all parties, and avoid overhead in case the enactment is terminated early.

Composition. How can we ensure that exactly the right amount of information is shared between the constituent protocols in a composition?

The wedding scenario can be decomposed into subprotocols, e.g., a specific wedding ceremony and a generic witness protocol. How do we ensure *independence*, that is, that each agent need only be aware of the protocol it is enacting? A witness participating in a wedding ceremony need only be aware of the generic witness protocol and should not need to implement support for (or even be aware of) the broader ceremony.

1.3 Contribution: Our Solution Conceptually

To negotiate which protocol to enact and which agents to play the roles, agents need the ability to discuss these objects. We introduce the concept of *metaprotocols*, which are protocols whose parameters refer to the elements of another protocol.

Although metaprotocols can be implemented using normal protocols with special interpretation for some of the parameters, we introduce protocol language extensions to make those meanings explicit. We borrow role parameters from Splee [7], and add support for parameter type declarations. Parameters with the type `protocol` may be used as variables for more flexible composition.

2 THE PROTOCOL LANGUAGE

BSPL [30], the foundational information protocol language, did not address role binding, assuming that roles were bound prior to protocol instantiation. Splee [7] supported treating roles as parameters for dynamic binding but did not address practical instantiation challenges.

We now introduce the core aspects of our protocol language, illustrated through specifications of the scenario.

2.1 Syntax

The formal syntax of our language, derived from Splee and BSPL, is given in Table 1, and described in detail below. A superscript of + indicates one or more repetitions, superscript ∗ indicates zero or more, and $\lfloor$ and $\rfloor$ delimit expressions, which are optional when without a superscript.

L$_1$. A specification document consists of a preamble and one or more protocols (which may be individual messages). The preamble contains key/value pairs used to declare common information (such as author or default prefix) or terms for substitution in the document (such as namespaces and IRI abbreviations).

L$_2$. A protocol declaration consists of a name, a public parameter expression, optional private parameters, and references to constituent protocols or messages. The public parameters with the key qualifier form this declaration's key. The private parameters can be surrounded by parentheses to spread them across multiple lines.

Table 1: Pippi Syntax

L$_1$	Spec	$\longrightarrow \lfloor Name\!:\!Value \rfloor^* \lfloor Protocol \vert Message \rfloor^+$
L$_2$	Protocol	$\longrightarrow Name(ParamExpr)$
		$\{ \lfloor$ private $Parameter^+ \rfloor\ Reference^+ \}$
L$_3$	ParamExpr	$\longrightarrow ParamClause \lfloor, ParamClause \rfloor^* (ParamExpr)$
L$_5$	Parameter	$\longrightarrow \lfloor \lfloor Qualifier \rfloor\ Name^+ \lfloor: Constraint^+ \rfloor \rfloor^+$
L$_6$	Reference	$\longrightarrow Name(ParamExpr)\ \vert\ Message$
L$_7$	Message	$\longrightarrow \lfloor Ad \rfloor\ Name \mapsto \lfloor Ad \rfloor\ Name^+: Name[ParamExpr]$
L$_8$	Ad	$\longrightarrow$ in $\vert$ nil $\vert$ out $\vert$ any $\vert$ opt
L$_9$	Qualifier	$\longrightarrow \lfloor$key$\rfloor\lfloor$local$\rfloor\lfloor$set$\rfloor$
L$_{10}$	Name	$\longrightarrow \lfloor Prefix: \rfloor (S) tring$
L$_{11}$	Constraints	$\longrightarrow \lfloor Type \rfloor\lfloor \subseteq \vert \in \vert = \vert < \vert > Name \vert Value \rfloor$
L$_{12}$	Type	$\longrightarrow$ role $\vert$ protocol $\vert$ Name

L$_3$. A parameter expression is a conjunction of parameter clauses, written as a (possibly parenthesized) comma-separated list.

L$_4$. A parameter clause is a disjunction of parameter expressions or an adorned parameter. If an adornment is not provided, it should be inferred in a reference or default to ⌜in⌝.

L$_5$. A parameter has a name and optional qualifiers and constraints. Multiple names can be included, separated by commas, as shorthand for repeating the same adornments, qualifiers, and constraints, provided all the names are on the same line or parenthesized.

L$_6$. A reference to a protocol may consist of a name appended by a parameter expression matching the protocol's declaration. The name must be the name of a protocol or the name of a parameter whose type is protocol.

L$_7$. A message schema consists of a name, sending role and receiving role(s), and a parameter expression.

L$_8$. An adornment is usually either ⌜in⌝ or ⌜out⌝; ⌜nil⌝ indicates an unknown parameter, ⌜any⌝ means either ⌜in⌝ or ⌜out⌝, and ⌜opt⌝ means the parameter is optional.

L$_9$. Qualifiers indicate if a parameter is any combination of key, local, and set. The order of the qualifiers is irrelevant.

L$_{10}$. Names are strings with an optional namespace prefix separated by a colon and an optional abbreviation in parentheses (the location of the abbreviation in the string does not matter). Disallowed characters include: [(){}[]:;,] (whitespace, brackets, and separators)

L$_{11}$. Constraints describe the possible values that a parameter binding may have, expressed as a semicolon-separated list. Example constraints include type, enumerated values, relationships with other parameters, and so on.

L$_{12}$. The type of a parameter may be role, protocol, or some name that can be resolved using the context and prefixes declared in the specification.

2.2 Illustration

Listing 1: Generic Witness Protocol

```
1 Witness(in J, W: role, in key cID, out sig or out objection) {
2   private req
3   J -> W: RequestApproval[in cID, out req]
4   W -> J: Approve[in cID, in req, out sig]
5   W -> J: Object[in cID, in req, out objection]
6 }
```

The *Witness* protocol, given in Listing 1, shows a simple protocol involving five parameters: J (Judge) and W (Witness) are roles, cID is the key that uniquely identifies each enactment (which is adorned ⌜in⌝ and therefore bound externally), and sig (signature) and objection are two parameters that may be bound during an enactment. The public parameters declare the completion requirements for an enactment: when all public parameters are bound, the enactment is complete. In *Witness*, we demonstrate syntax for complex Boolean completion formulas; only one of sig or objection needs to be bound to complete the enactment.

The body of the protocol lists the subprotocols it is composed of; a message is an elementary protocol. The *RequestApproval* message on line 3 is sent by J (the role before the arrow) to W. Its payload contains two parameters, cID and req; these parameters represent the information conveyed by the message. The parameter cID is adorned ⌜in⌝, and so must be bound before *RequestApproval* can be sent. Conversely, req is ⌜out⌝, and so must *not* already be bound; it is bound when sending *RequestApproval*. Parameter bindings are functionally dependent on their keys; req may have only one binding for each value of cID.

Messages may be sent when they are enabled by the satisfaction of their causality constraints as specified by their parameter adornments. Only emissions are constrained; a message may be received any time after it has been sent.

Listing 2: Court Wedding

```
1 CourtWedding(in Propose(R), Propose(E), (J)udge, (W)itness: role,
2   any key cID,
3   (out vowR, out vowE, out license) or out objection) {
4   private signature
5   J -> R, E: Ask[any cID, out questions]
6   E -> J, R: EVow[in cID, in questions, out vowE]
7   R -> J, E: RVow[in cID, in questions, out vowR]
8   Witness(J, W, cID, signature or objection)
9   J -> E, R: Marry[in cID, in vowE, in vowR, in signature, out
          license]
10 }
```

Listing 2 gives a protocol describing a court wedding. Note that all of the role parameters are adorned ⌜in⌝; they must be bound outside the context of the protocol. The key cID is adorned ⌜any⌝, which means it can be either ⌜in⌝ or ⌜out⌝. In addition to the features used in *Witness*, *CourtWedding* references *Witness*, and handles the two outcomes separately. If the witness provides a signature, then the judge can perform the marriage; otherwise, the protocol terminates with an objection. *CourtWedding* exhibits the role abbreviation syntax, e.g., identifying R as the abbreviation for Proposer.

3 DYNAMIC ROLE BINDING

All that is required to enact a protocol is a group of agents willing and able to play all its roles. In most existing work on protocols, the role bindings are assumed; perhaps the MAS was hard-coded or statically configured with the agents' roles and endpoints.

For greater flexibility and to properly model real-life systems, agents need to discover new peers and negotiate protocols with them. Such dynamic role binding is useful even for simple systems, such as where one seller enacts the same purchase protocol with

multiple customers. Explicit semantics for how the seller is introduced to each customer and how those role bindings relate to the enactment information is necessary for correct implementation.

Furthermore, we would like to support not only explicit role binding but also *late* role binding, where some agents begin to play their roles after the protocol has already been initiated. For example, in a rideshare setting, riders might want to request a different driver if the first is not coming without having to resubmit their destination. In addition, it is reasonable to delay the involvement of additional parties for efficiency and privacy reasons. If a customer's bank is notified every time the customer considers a purchase regardless of their final choice, it unnecessarily exposes their actions and wastes the bank's resources.

To address the above concerns, we introduce two patterns of protocol design: metaprotocols and self-contained protocols.

3.1 Metaprotocols

Metaprotocols are protocols that specify communication about the elements of another protocol, such as its structure and role bindings. Roles in a protocol may be represented by parameters in a metaprotocol without any special semantics.

Late role binding is enabled in a metaprotocol by interleaving its enactment with that of its target protocol. Then, messages binding specific roles need not be sent until they are relevant in the target protocol, though, of course, they may be sent earlier.

A metaprotocol capturing the proposal phase of the marriage scenario might be implemented as follows:

Listing 3: Proposal Metaprotocol

```
1 Proposal(out r0, r1: role, out key mID, out acceptance or out
        rejection) {
2   private (r2, r3: role,
3     cID, R, E, J, W,
4     ceremony: protocol)
5   out r0 -> out r1: Propose[out mID, out ceremony, out R, out E]
6   r1 -> r0: Accept[in mID, in ceremony, out acceptance]
7   r0 -> r1: Plan[in mID, in ceremony, in accept, out cID]
8   ceremony(R, E, J, W, cID, out vowR, out vowE, out result)
9   r0 -> out r2: Schedule[in mID, in cID, in E, out J, out date]
10  r0 -> out r3: Invite[in mID, in cID, in J, out W, out
        invitation]
11  r1 -> r0: Reject[in mID, in ceremony, out rejection]
12 }
```

The *Proposal* metaprotocol, given in Listing 3, sets up the roles necessary for enacting a wedding ceremony such as *CourtWedding*. First, the roles in this protocol are numbered because they have no meaning in themselves; they are placeholders for the purpose of inviting agents to perform the ceremony. Thus, r0 sends *Propose* to r1, suggesting that they take on the roles R and E in some ceremony—not necessarily in that order. The ceremony is specified by binding it to the ceremony parameter. If r1 accepts, r0 may proceed to invite other agents. Note that *all* of the roles from *CourtWedding* are bound in this protocol, yet late binding is possible because r0 may delay inviting r3 to perform the W role until it becomes relevant.

Metaprotocols can be used to implement protocol negotiation, as *Proposal* demonstrates by using a parameter to reference a protocol: r0 can propose the ceremony to enact, provided it matches the same interface. However, not all metaprotocols involve negotiation; a simple metaprotocol could simply communicate the relevant role bindings to each participant.

Metaprotocols cleanly separate discussion about the protocol from discussion of its enactment. They are fully compatible with BSPL even without our syntax enhancements and may be easily written for existing protocols. Metaprotocols do not reduce flexibility but enable it. Without a metaprotocol, role binding is not flexible but merely unspecified, leading to tight coupling and hidden complexity in a multiagent system.

3.2 Metaprotocol Generation

A simple algorithm for generating a metaprotocol from an existing protocol is as follows:

(1) Compute a contact graph among the roles, with directed edges following the message transmissions.
(2) Identify the initiating role, possibly with user input.
(3) Using the contact graph compute the rank of each role (distance of each role from the initiator).
(4) Generate a message from the initiator that binds a metaprotocol ID, and another binding the protocol ID.
(5) Generate messages from each agent to their contacts of higher rank (flowing from the initiator), inviting them to the protocol and naming any mutual contacts they have invited.

The above algorithm generates introduction messages for each role according to the contact graph of the original protocol; in this way, agents are responsible for inviting only agents they contact. The rank-based invitation approach produces a minimal metaprotocol; that is, the smallest number of single-recipient messages that cover all of the invitations. As such, the complexity of the metaprotocol is no worse than the original protocol; it has approximately one message per role.

The resulting metaprotocol can be customized to support domain requirements. For example, the metaprotocol for a purchase protocol can be customized so that the seller chooses the transaction ID, even though the customer initiates the interaction. Or, in the wedding example, the metaprotocol can specify that the proposer will invite the witness even though the judge is the only agent who contacts the witness during the ceremony.

Using the above algorithm on *CourtWedding* and arbitrarily selecting R as the initiator, we can derive the following:

(1) R sends messages to J and E; J sends to R, E, and W; E sends messages to R; W sends to J
(2) R has rank 0 (as the selected initiator), J and E have rank 1, and W has rank 2

In general, the initiating agent could be identified from the protocol by looking at which messages declare the primary keys of the protocol or depend only on external parameters. However, in this particular example, the initiator for the metaprotocol (R) is not the same as the first agent to communicate in the CourtWedding protocol (J), so it must be explicitly selected as the initiator.

With the above contact graph and rank information, we can derive the comprehensive metaprotocol shown in Listing 4. The *Generated* protocol in Listing 4 is simple because the rank sorting prevents J from inviting E and vice-versa. E is arbitrarily invited first in, e.g., alphabetical order. Our example algorithm is far from the only possibility; alternative algorithms could generate a multicast message that simultaneously invites multiple agents or generate a more complex metaprotocol to invite them in either order.

Listing 4: Generated Metaprotocol for CourtWedding

```
1  Generated(out r0, r1, r2, r3: role,
2    out R, E, J, W: role,
3    out key mID, out cID,
4    out vowR, vowE, out license or out objection) {
5    out r0 -> out r1: InviteE[out mID, out cID, out R: role=r0,
          out E: role=r1]
6    r0 -> out r2: InviteJ[mID, cID, R, E, out J: role=r2]
7    r2 -> out r3: inviteW[mID, cID, R, E, out W: role=r3]
8  }
```

We used type and value constraints (e.g., 'role=r2', where 'role' is a type constraint and '=r2' is a value constraint) for the ⌜out⌝ role parameters; both are optional but make the protocol clearer.

3.3 Self-Contained Protocols

Although the previous approach to protocol configuration is conceptually simple and compatible with existing BSPL semantics, it can be somewhat impractical because every nontrivial protocol must have a corresponding and cumbersome metaprotocol. Those metaprotocols can add many messages as overhead solely for communicating contact information.

We propose further extensions to BSPL to support more practical, progressive protocol instantiation at the cost of some complexity.

Listing 5: Self-Contained Wedding Protocol

```
1  Self-Contained-Wedding(out Propose(R), Propose(E): role
2    out key cID, out rejection or
3      (out vowR, out vowE, out license or out objection)) {
4    private (J)udge: role, (W)itness: role, signature
5
6    out R -> out E: Propose[out cID]
7    E -> R: Accept[in cID, out acceptance]
8    E -> R: Reject[in cID, out rejection]
9    R -> out J: Schedule[in cID, in E, in acceptance, out date]
10   J -> R, E: Ask[in cID, out questions]
11   E -> J, R: EVow[in cID, in questions, out vowE]
12   R -> J, E: RVow[in cID, in questions, out vowR]
13   R -> out W: Invite[in mID, in cID, in J, out invitation]
14   Witness(J, W, cID, out signature or out objection)
15   J -> E, R: Marry[in cID, in vowE, in vowR, in signature, out
          license]
16 }
```

The *Self-Contained-Wedding* protocol in Listing 5 does not rely on a separate metaprotocol but instead specifies the process and propagation of role binding internally.

Self-contained protocols are not superior to metaprotocols, but may be an optimization if they can, e.g., communicate role bindings alongside other information in the protocol.

4 COMPOSITION

Composition is another important aspect of information protocols. Implicitly, agents can participate in multiple protocols at the same time and must keep them separate. Protocols can be explicitly composed into a larger protocol specification. Provided all the agents know the composite protocol, composition adds no complexity to an implementation. However, the following concerns arise with respect to instantiation: (1) enabling interface flexibility, so that protocols may share or hide information as necessary with the broader composition (2) agents should not be required to know all compositions containing the protocols they support (3) enabling composition of protocols without needing a common key.

4.1 Interface Flexibility

Existing information protocol languages BSPL and Splee have rigid protocol interfaces; all public parameters must be bound for an enactment to complete. Hence, protocol designers must encode multiple termination states—such as accepting or rejecting a marriage proposal—in a single result parameter.

Listing 6: Information Hiding Example

```
1  OpaqueOffer(in B,S: role, in key oID, out item, amount, result) {
2    private acceptance, rejection
3    S -> B: Offer[oID, out item, out amount]
4    B -> S: Accept[oID, item, amount, out result]
5    B -> S: Reject[oID, item, amount, out result]
6  }
7
8  Purchase(out B, S: role, out key pID, out transaction) {
9    private request, item, amount
10   out B -> out S: RFQ[out pID, out request]
11   OpaqueOffer(B, S, pID, item, result)
12   S -> B: Ship[pID, result, out transaction]
13   // no way to distinguish outcomes in composition
14   S -> B: Cancel[pID, result, out transaction]
15 }
```

OpaqueOffer in Listing 6 has only a single result parameter in its public parameter line, hiding the actual response of B. If used in a composition such as *Purchase* in the same listing, there is no way to distinguish the outcomes except through internal agent logic applied to the parameter bindings.

However, when protocols are used in a composition the limitation becomes clearer; what if some subsequent actions should be available only after acceptance, and others only after rejection? For example, the wedding ceremony should be enacted only if the proposee accepts the proposal. In this case, the ceremony has an explicit dependency on the outcome of the proposal, so it should not be hidden.

Our proposed solution is to enhance protocol interfaces to support more complex Boolean expressions over the protocol's parameters. Where before the interface was a conjunction over the public parameter bindings, now parameters can be grouped into disjoint clauses.

Listing 7: Flexible Interface

```
1  TransparentOffer(in B, S: role, in key oID, out item, amount,
       out acceptance or rejection) {
2    private acceptance, rejection
3    S -> B: Offer[oID, out item, out amount]
4    B -> S: Accept[oID, item, amount, out acceptance]
5    B -> S: Reject[oID, item, amount, out rejection]
6  }
7
8  Purchase(out B,S:role,out key pID,out package or out canceled) {
9    private request, item, amount, acceptance, rejection
10   out B -> out S: RFQ[out pID, out request]
11   TransparentOffer(B, S, pID, item, acceptance or rejection)
12   S -> B: Ship[pID, acceptance, out package]
13   S -> B: Cancel[pID, rejection, out canceled]
14 }
```

TransparentOffer in Listing 7 shows how Boolean parameter expressions enable more transparent interfaces that expose multiple termination states, which can then be handled separately in a composition. The parameters acceptance and rejection are normal parameters joined into a disjunctive clause by the or keyword; only one of the two bindings is necessary for completion. Thus, an

enactment of *TransparentOffer* is complete when the parameters oID, item, amount, and either acceptance or rejection are bound.

Note that because messages are elementary protocols, they may use Boolean parameter expressions. However, this flexibility can obscure the meaning of the messages and so should be used with care. For example, in a wedding or purchase protocol, sending an explicit *Accept* message forms a commitment unambiguously and can be interpreted at the event level without inspection. By contrast, interpreting a more abstract Decide[ID, out acceptance or out rejection] requires examining the payload.

4.2 Constituent Protocol Independence

Semantically speaking, when a protocol is referenced in a composition, its parameters are substituted by the parameters passed in the reference. Thus, a bank transfer protocol may involve a parameter named amount that is used to transfer payment in a purchase protocol or refund in a protocol for processing returns. Proper encapsulation and modularity require that the bank is able to enact its role in the transfer without being aware of the broader purpose. This requirement is violated by current implementations because the bank would need to interpret the substituted parameter names instead of the expected amount.

Listing 8: Dependent Representation

```
1  // (D)ebtor, (C)reditor, (B)ank
2  // original: D -> B: Transfer[in ID, out amount, C]
3  {"ID": <uuid>, "amount": 100, "C": "Creditor"}
4  // (B)uyer, (S)eller, Ban(K)
5  //composed: B -> K: Transfer[in pID, out payment, S]
6  {"pID": <uuid>, "payment": 50, "S": "Seller"}
```

Listing 8 shows two versions of a message schema. The first schema (line 2) is the generic transfer expected by the bank with the parameter amount specifying the amount to be transferred. Its representation in JSON is given on the following line, with the expected parameter names as dictionary keys. Using the transfer protocol in a purchase composition would result in substituting the parameters, as shown on line 5. The subsequent JSON representation uses the parameter names from the composition, which may not be understood by the bank.

To better support protocol independence, we propose a simplification of message representations, where each message is encoded as a (name, payload) pair, and the payload is simply an array of unnamed parameters. The name must uniquely identify the message in the MAS; it could be an IRI or a more compact encoding defined by the MAS for efficiency. The payload is interpreted by referring to the message schema.

Replacing named parameters with a uniquely identified message schema minimizes the context sensitivity of the message and simplifies interpretation. That is, referring to the protocol that specifies the message provides the information necessary for decoding the message: which parameters are included in which order. However, all semantic information is left out of the encoding, so each agent can interpret the message according to its understanding of the composition. This is similar to message encoding libraries such as Protocol Buffers [18], which use an external schema to parse the messages, enabling compact encoding and backward compatibility across schema versions (since names can change without changing the encoding).

Listing 9: Independent Representation

```
1  // original: D -> B: Transfer[in ID, out amount, C]
2  [<uuid>,100,"Creditor"]
3  // composed: B -> K: Transfer[in pID, out payment, S]
4  [<uuid>,50,"Seller"]
```

In Listing 9, leaving out the parameter names does not make the schema ambiguous but does make it independent of its use in a composition. The bank receives the information necessary to execute a transfer without being confused by the buyer's intent to use the transfer as payment for a purchase.

4.3 Global and Local Keys

We have added syntax for distinguishing global and local keys. The purpose of the distinction is to enable *invertible* relationships between protocol keys, without which protocols in composition must share a common key to share information.

Listing 10: Noninvertible Key Composition

```
1  Noninvertible(B, S: role, key ID, oID, dID,
2    item, price, package) {
3    out B -> out S: Start[out ID]
4    Order(B, S, ID, oID, item, price)
5    Deliver(B, S, ID, dID, item, package)
6  }
```

As shown in Listing 10, the *Order* and *Deliver* protocols must share an *ID* to correlate the information, which intrusively imposes awareness of the composition on those protocols. If written independently, each would need only its own key.

An invertible relationship exists between keys if they are uniquely associated with each other; that is, an agent receiving a message containing one key can correlate it to an enactment of another protocol using the other key. If a relational data model is used, the agent's adapter can construct queries for enactment information using an association table to join information from the two protocols. Or, the agent's adapter can use a dictionary to map the unique keys to a single shared enactment object.

Listing 11 shows a composition where the keys are universally unique. Because both oID and dID are unique, the *TrackingInfo* message declares an invertible relationship between them, enabling both agents to correlate the messages inside *Deliver* to the broader enactment without needing to propagate a common key.

Listing 11: Invertible Key Composition

```
1  Invertible(B, S: role,
2    key oID, dID // new default of global keys
3    item, price, package) {
4    Order(B, S, oID, item, price)
5    S -> B: TrackingInfo[in oID, out dID]
6    Deliver(B, S, dID, item, package)
7  }
```

Global keys are the default. They are identified by the sole qualifier key, and must be guaranteed unique within the MAS. We include the `local` keyword to cover all other kinds of keys (e.g., sequence IDs or timestamps). Such keys are required to be unique only within the context of the global keys. Thus, a local key must be used in the context of a global key, possibly scoped further by other ⌜in⌝ local keys. If multiple local keys are ⌜out⌝ in the same message, they form a hierarchy based on the order in which they are given.

Listing 12 gives an example of how local keys might be used in a protocol. It specifies a support request protocol, where B opens a ticket, specifying a topic and generating a new unique ID. Then, B and S can continue replying with an indefinite number of messages, each uniquely identified with reply IDs (possibly sequential or timestamped), bound to brID and srID, respectively. These response IDs need only be unique within the context of the support request because they are local keys.

Listing 12: Local Key Example

```
1  Support (B, S: role,
2    key ID, // global key
3    local key brID, srID, // local keys
4    topic, closed) {
5    B -> S: OpenTicket[out ID, out topic]
6    S -> B: SReply[in ID, in topic, out srID, out content]
7    B -> S: BReply[in ID, in topic, out brID, out content]
8    S -> B: CloseTicket[in ID, out closed]
9  }
```

5 REALIZATION

Programming agents to support dynamic role binding and multiple parallel interactions need not be complicated but is more involved than supporting a single protocol.

In previous programming models for information protocols (PoT [13], Bungie [11], Deserv [12]), agents adopt a single role in a single protocol. Architecturally, each agent was instantiated using a protocol adapter configured with the role and protocol that the agent supported. The adapter was configured with static role bindings that were bound outside of the protocol enactment. However, it may be possible to change these bindings; the adapter supported only a single binding for each role and protocol. Supporting additional protocols would require the agent to run multiple adapters with separate endpoints.

5.1 Protocol Adapter

The protocol adapter must support multiple roles across several protocols and metaprotocols. Each protocol is loaded in the context of a specification, which may define terms used in the protocols.

The adapter would support a collection of initial protocols (or metaprotocols), along with the roles it is willing to play in those protocols and handlers for those messages. These initial protocols could be implemented according to patterns such as peer-to-peer introductions or a central registry, among others, which we examine further in Section 6.

During the enactment of the initial protocols, the agent will either propose enactments of protocols as specific roles or receive invitations to take on a role. Fortunately, the agent need not specify special-case handlers for these invitations any different from other messages; the information structure of the protocols is enough to guide further enactment.

Roles have become parameters bound within the context of a protocol or metaprotocol instance key. No longer are roles implicitly part of the message schema. A role must either be explicitly bound in sending a message to a new role or provided by the enactment history. Provided the protocol (or metaprotocol) is verified to be live and enacted without violation or error, the agents will receive the contacts they need to complete their role. How an agent selects or approves the specific role bindings is left to their internal logic.

5.2 MAS Adapter

To better support agent discovery and selection, we propose a new component, the *MAS Adapter*. The MAS adapter is optional; simple agents (such as web services) may handle each enactment independently. Provided each role binding is used only within the context of an enactment, the information management provided by the protocol adapter is sufficient. The MAS adapter is responsible for remembering the contacts and the history of interactions with them beyond a single enactment.

The MAS adapter implements a registry of known agents. New entries are automatically added as agents are introduced. Each entry contains information about the protocols and roles that the agent is believed to support, based on the introduction. Each entry contains information about the roles that the agent has played and the corresponding enactment histories. Additional information can be added according to domain requirements, such as quality of service ratings. An agent can query its MAS adapter to find agents to enact protocols with.

6 EVALUATION

We now show how the above concepts can be applied toward capturing canonical role-binding patterns in practical decentralized applications.

6.1 Preconfigured Contacts

In peer-to-peer applications, it is common for the peers to be preconfigured with the knowledge of some bootstrap nodes to get the application going. Network applications are typically preconfigured with a DNS server for purposes of resolving domain names.

In our context, preconfigured contacts are those an agent is given before it is initiated. Preconfigured contacts are implemented in the agent internals and used to bind the initial ⌜out⌝ roles of the first protocols the agent enacts. This is a simple pattern, but it is a necessary component of all more complex patterns.

Listing 13: Preconfigured Contacts Example (JSON)

```
1 {"Seller":["http://storeA.com/agent","http://storeB.com/agent"],
2  "Bank": ["http://bank.com/agent"]}
```

Listing 13 shows how contacts might be declared in a JSON file. Preconfigured contacts are not hard coded bindings; they are candidates that are bound to roles dynamically according to the protocol structure and agent logic.

6.2 Central Registry

The next level of complexity from a collection of preconfigured contacts is a central registry provided as a service by another agent. Discovering agents from a central registry is common in services. The Uber application, e.g., uses Uber's (the organization's) central registry of potential drivers to bind as the pickup driver in a particular transaction with a customer.

A central registry is simple in that it means that an agent configuration requires only a single registry connection. In addition, a registry enables more dynamic and scalable peer discovery than preconfiguration. As each agent comes online, they can enact a registration protocol, notifying the registry of their existence and willingness to perform specific roles. To enact a protocol, an agent

can first query the registry to discover potential peers. As the example in Listing 14 shows, the pattern could be implemented as two protocols, where a single well-known agent uses its MAS adapter to remember and recommend contacts to other agents.

Listing 14: Central Registry

```
1  Registration(out A, R: role,
2    out key ID, out endpoint, confirmation
3    out set protocols: protocol) {
4    out A -> out R: Register[out ID, out endpoint, out protocols]
5    R -> A: Confirm[in ID, in endpoint, in protocols, out
          confirmation]
6  }
7
8  Discovery(out Q, R: role,
9    out key ID, out protocol: protocol, out set agents: role) {
10   out Q -> out R: Query[out ID, out protocol]
11   R -> Q: Introduce[in ID, in protocol, out agents]
12 }
```

6.3 Peer Sharing

Peer sharing is characterized by the absence of any distinguished system nodes that support discovery. Peer-to-peer discovery and binding are common in MAS and distributed systems. For example, it is used in referral networks [34], in the Contract Net [32]), and in leader election protocols. Further, mesh networks and IoT-based systems typically invoke the peer-to-peer pattern.

To find a desired peer, each agent checks its own MAS adapter and queries its neighbors. The exact nature of the peer selection and query process is application-specific, but generally, each query will return more peers; either the desired peer will be among them, or they can be queried in turn.

Listing 15: Peer Discovery

```
1  Discover(out P1, P2: role, out key ID,
2    out set protocols: protocol, out set neighbors: role) {
3    out P1 -> out P2: Query[out ID, out protocols]
4    P1 -> P2: Introduce[in ID, in protocol, out neighbors]
5  }
```

In the peer discovery protocol in Listing 15, there is no need for registration; each agent simply needs an initial bootstrap peer to connect to. Each time an agent asks a peer for neighbors, it naturally introduces itself and reveals the protocol(s) it is interested in.

7 DISCUSSION

We conceptualized the problem of instantiating a MAS in terms of binding the roles of the protocol that models the MAS. Although multiagent systems are typically thought of as being open in the sense of agents dynamically joining and leaving the system, MAS approaches have not paid sufficient attention to the operational aspects of the problem. Pippi demonstrates how MAS can be instantiated dynamically by showing how a protocol's roles are bound. Our approach supports dynamic (not hard-coded) and late (just-in-time) role bindings: not all roles in a protocol need to be bound before enacting the protocol. We discussed a possible realization of our approach and demonstrated how commonly used patterns can be captured in our approach.

Dastani et al. [14] consider compatibility between agent and role specifications as a basis for an agent's decision whether or not to play a role. Such internal decision making, although practically relevant, is outside the scope of Pippi. Pippi is concerned with public

aspects of decision making as reflected in protocol enactments. HAPN [33] supports dynamic role binding; e.g., an auction's winner is dynamically bound. It does not give a general metaprotocol-based approach that enables discovery and bindings of roles. Grenna et al. [19] extended JADE to enable agents to enact roles, but within the context of an organization rather than a protocol.

Günay et al. [20] propose a metaprotocol by which agents can reach agreement on the commitment protocol: All agents should accept that they will create the commitments involved. They assume role bindings (e.g., that there are customer and merchant agents). Thus, Pippi complements their work. McGinnis and Robertson [23] propose protocols as first-class abstractions for composing open MAS. Pippi agrees with their intuition and demonstrates how role bindings generated as information in the messages of one protocol (the metaprotocol) can be used as roles (the senders and receivers of messages) in another protocol via composition.

Multiagent systems are conceived of as open systems in the sense that agents can join and leave the MAS. Mazouzi et al. [22] give an early example of an agent wanting to join several groups and show that abstract specifications can be refined depending on requirements. Role binding, as we formalize here, is akin to "joining" a MAS. However, because parameter bindings and hence role bindings are immutable, "leaving" a MAS is not unbinding the role. Leaving would be captured by communicating that the agent has left the MAS. Immutability of information gives the immutability of events, which is necessary for realism. For example, once bound as a partner in some wedding, an agent is always a partner for that wedding. Divorce is not an unbinding of the agent to the partner role but a change in the normative relationships [9] between the partners, perhaps via the creation of a new MAS. For example, two divorced people may have joint custody of their children and may thus function together in a MAS, albeit not the same MAS as when they were married. Such ideas merit further study.

Minsky and Murata [25] discuss the robustness and manageability of a MAS modeled as a law-based society [24]. In considering robustness, Minsky and Murata bring up both static and dynamic role binding in the law, which is realized as a set of Prolog-like rules. The static bindings are given as facts, and the dynamic bindings are established via events referred to in the rules. Our approach is compatible with rule-based specifications (e.g., commitments or other norms [9]) but captures the general operational aspects of role binding in a decentralized system via protocols.

Ferrando et al. [17] consider the enactability [16] of protocols specified in a language for specifying execution traces, where the constraints specify message ordering. They evaluate the enactability of protocols under different infrastructure assumptions, e.g., with or without FIFO delivery. Ferrando et al. do not consider the problem of role binding. Other early work on formally specifying [29] and enacting [28] multiagent interactions also doesn't address role binding as an explicit concern.

Pippi's protocols can be enacted with the minimal set of infrastructure assumptions, namely that only sent messages be delivered. Further, Pippi's protocols can be interleaved without requiring to be composed (this is a property of information protocols [8]), which is not a possibility with trace-based approaches. Role binding for a protocol can be interleaved with the enactment of the protocol, as shown in the *Proposal* metaprotocol in Listing 3.

Chocron and Schorlemmer [6] consider the problem in open MAS where agents know they are participating in the same interaction (specified in temporal logic) but have different message vocabularies. They propose a method by which agents can dynamically align their vocabularies, that is, learn the mappings between their vocabularies. Pippi currently assumes a shared vocabulary (whatever is in the shared information protocols) but would benefit from alignment techniques in scenarios where the assumption does not hold. Arguably, aligning the "same" information protocol with different vocabularies is a more natural and challenging problem to address since information protocols can be more flexible than message ordering constraints specified in temporal logic.

Rocha and Brandão [27] apply multiagent systems to model dynamism in Internet of Things applications. They model devices via agents that enter and leave the system. Pippi enables realizing such dynamism in a general way with clean, modular representations.

JADE [2] supports the discovery of agents via a directory facilitator that provides a yellow pages service. JADE supports FIPA interaction protocols by providing endpoint implementations (classes and methods) that agents can use to implement interactions with others. However, JADE is not equipped to enable building multiagent systems off protocol specifications as Pippi's adapter and programming model enable. Further, discovery, although important, differs from role binding. In the wedding scenario, an agent can discover other agents looking for prospective partners via a directory service but would need to *bind* one of them to Proposee to enact Proposal. Briola et al. [4] demonstrate how JADE can be used to discover agents and implement a peer-to-peer system. As the foregoing examples demonstrate, Pippi supports both discovery and role binding and thus enables the realization of arbitrary applications as a peer-peer system.

Carriero and Gelernter's [5] tuple space approach for coordinating processes has been influential in multiagent systems. It features as the underlying coordination mechanism in CArtAgO [26], which itself is part of JaCaMo [3], which Baldoni et al. [1] use to support the implementation of commitment-based business processes. Tuple spaces (like logic programming) are attractive for their information-based abstractions (one works with tuples of information). However, they represent a shared-memory approach, which is not suitable for building decentralized MAS. It is widely argued that a tuple space decouples the readers and writers of information, but that argument holds only where the writers don't care who reads the information (as is the case in the classic "readers and writers" synchronization problem). In MAS, agents communicate with particular agents, e.g., the Proposer wants to communicate different information to the Proposee and to the Judge. Such coupling between agents is specified in the protocol. It would, however, be interesting to investigate if tuple spaces can be used to implement an agent's local state since it provides information-based abstractions.

ACKNOWLEDGMENTS

REFERENCES

[1] Matteo Baldoni, Cristina Baroglio, Federico Capuzzimati, and Roberto Micalizio. 2019. Process Coordination with Business Artifacts and Multiagent Technologies. *Journal on Data Semantics* 8, 2 (June 2019), 99–112. https://doi.org/10.1007/s13740-019-00100-8

[2] Federico Bergenti, Giovanni Caire, Stefania Monica, and Agostino Poggi. 2020. The First Twenty Years of Agent-Based Software Development with JADE. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 34, 2 (2020), 36. https://doi.org/10.1007/s10458-020-09460-z

[3] Olivier Boissier, Rafael H. Bordini, Jomi Fred Hübner, Alessandro Ricci, and Andrea Santi. 2013. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming* 78, 6 (June 2013), 747–761. https://doi.org/10.1016/j.scico.2011.10.004

[4] Daniela Briola, Daniela Micucci, and Leonardo Mariani. 2019. A Platform for P2P Agent-Based Collaborative Applications. *Software – Practice and Experience* 49, 3 (2019), 549–558. https://doi.org/10.1002/spe.2657

[5] Nicholas Carriero and David Gelernter. 1992. Coordination Languages and their Significance. *Communications of the ACM (CACM)* 35, 2 (Feb. 1992), 97–107. https://doi.org/10.1145/129630.376083

[6] Paula Daniela Chocron and Marco Schorlemmer. 2020. Vocabulary Alignment in Openly Specified Interactions. *Journal of Artificial Intelligence Research (JAIR)* 68 (May 2020), 69–107. https://doi.org/10.1613/jair.1.11497

[7] Amit K. Chopra, Samuel H. Christie V, and Munindar P. Singh. 2017. Splee: A Declarative Information-Based Language for Multiagent Interaction Protocols. In *Proceedings of the 16th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, São Paulo, 1054–1063. https://doi.org/10.5555/3091125.3091274

[8] Amit K. Chopra, Samuel H. Christie V, and Munindar P. Singh. 2020. An Evaluation of Communication Protocol Languages for Engineering Multiagent Systems. *Journal of Artificial Intelligence Research (JAIR)* 69 (Dec. 2020), 1351–1393. https://doi.org/10.1613/jair.1.12212

[9] Amit K. Chopra and Munindar P. Singh. 2016. From Social Machines to Social Protocols: Software Engineering Foundations for Sociotechnical Systems. In *Proceedings of the 25th International World Wide Web Conference*. ACM, Montréal, 903–914. https://doi.org/10.1145/2872427.2883018

[10] Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. 2018. Compositional Correctness in Multiagent Interactions. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Stockholm, 1159–1167. https://doi.org/10.5555/3237383.3237868

[11] Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. 2021. Bungie: Improving Fault Tolerance via Extensible Application-Level Protocols. *IEEE Computer* 54, 5 (May 2021), 44–53. https://doi.org/10.1109/MC.2021.3052147

[12] Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. 2021. Deserv: Decentralized Serverless Computing. In *Proceedings of the 19th IEEE International Conference on Web Services (ICWS)*. IEEE Computer Society, Virtual, 51–60. https://doi.org/10.1109/ICWS53863.2021.00020

[13] Samuel H. Christie V, Daria Smirnova, Amit K. Chopra, and Munindar P. Singh. 2020. Protocols Over Things: A Decentralized Programming Model for the Internet of Things. *IEEE Computer* 53, 12 (Dec. 2020), 60–68. https://doi.org/10.1109/MC.2020.3023887

[14] Mehdi Dastani, Virginia Dignum, and Frank Dignum. 2003. Role-assignment in open agent societies. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. ACM Press, Melbourne, 489–496. https://doi.org/10.1145/860575.860654

[15] Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. 2005. Interaction Protocols as Design Abstractions for Business Processes. *IEEE Transactions on Software Engineering* 31, 12 (Dec. 2005), 1015–1027. https://doi.org/10.1109/TSE.2005.140

[16] Nirmit Desai and Munindar P. Singh. 2008. On the Enactability of Business Protocols. In *Proceedings of the 23rd Conference on Artificial Intelligence (AAAI)*. AAAI Press, Chicago, 1126–1131.

[17] Angelo Ferrando, Michael Winikoff, Stephen Cranefield, Frank Dignum, and Viviana Mascardi. 2019. On Enactability of Agent Interaction Protocols: Towards a Unified Approach. In *Proceedings of the 7th International Workshop on Engineering Multi-Agent Systems (EMAS) (Lecture Notes in Computer Science, Vol. 12058)*. Springer, Montréal, 43–64. https://doi.org/10.1007/978-3-030-51417-4_3

[18] Google. 2022. Protocol Buffers. https://developers.google.com/protocol-buffers/.

[19] Roberto Grenna, Matteo Baldoni, Guido Boella, Leendert van der Torre, Mauro Dorni, Andrea Mugnaini, and Valerio Genovese. 2008. Adding Organizations and Roles as Primitives to the JADE Framework. In *Programming Multi-Agent Systems (Dagstuhl Seminar Proceedings, 08361)*, Rafael Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany. http://drops.dagstuhl.de/opus/volltexte/2008/1639

[20] Akın Günay, Michael Winikoff, and Pınar Yolum. 2015. Dynamically Generated Commitment Protocols in Open Systems. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 29, 2 (March 2015), 192–229. https://doi.org/10.1007/s10458-014-9251-7

[21] Michael N. Huhns, Nigel Jacobs, Tomasz Ksiezyk, Wei-Min Shen, Munindar P. Singh, and Philip E. Cannata. 1992. Enterprise Information Modeling and Model Integration in Carnot. In *Enterprise Integration Modeling: Proceedings of the First International Conference*, Charles J. Petrie, Jr. (Ed.). MIT Press, Hilton Head, South Carolina, 290–299. https://doi.org/10.7551/mitpress/2768.003.0036

[22] Hamza Mazouzi, Amal El Fallah Seghrouchni, and Serge Haddad. 2002. Open Protocol Design for Complex Interactions in Multi-Agent Systems. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. ACM Press, Bologna, 517–526. https://doi.org/10.1145/544862.544866

[23] Jarred McGinnis and David Robertson. 2005. Dynamic and Distributed Interaction Protocols. In *Proceedings of the Workshop on Adaptive Agents and Multi-Agent Systems (Lecture Notes in Computer Science, 3394)*. Springer, Melbourne, 167–184. https://doi.org/10.1007/978-3-540-32274-0_11

[24] Naftaly H. Minsky. 1991. The Imposition of Protocols over Open Distributed Systems. *IEEE Transactions on Software Engineering* 17, 2 (1991), 183–195. https://doi.org/10.1109/32.67599

[25] Naftaly H. Minsky and Takahiro Murata. 2003. On Manageability and Robustness of Open Multi-agent Systems. In *Software Engineering for Multi-Agent Systems II, Research Issues and Practical Applications (SELMAS) (Lecture Notes in Computer Science, Vol. 2940)*. Springer, Portland, Oregon, 189–206. https://doi.org/10.1007/978-3-540-24625-1_11

[26] Alessandro Ricci, Michele Piunti, Mirko Viroli, and Andrea Omicini. 2009. Environment Programming in CArtAgO. In *Multi-Agent Programming, Languages, Tools and Applications*, Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni (Eds.). Springer, Dordrecht, Netherlands, Chapter 8, 259–288. https://doi.org/10.1007/978-0-387-89299-3_8

[27] Vladimir Rocha and Anarosa Alves Franco Brandão. 2019. A Scalable Multiagent Architecture for Monitoring IoT Devices. *Journal of Network and Computer Applications* 139 (Aug. 2019), 1–14. https://doi.org/10.1016/j.jnca.2019.04.017

[28] Munindar P. Singh. 1998. A Customizable Coordination Service for Autonomous Agents. In *Intelligent Agents IV: Proceedings of the 4th International Workshop on Agent Theories, Architectures, and Languages (ATAL-97) (Lecture Notes in Computer Science, 1365)*. Springer, Providence, Rhode Island, 93–106. https://doi.org/10.1007/BFb0026752

[29] Munindar P. Singh. 2003. Distributed Enactment of Multiagent Workflows: Temporal Logic for Web Service Composition. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. ACM Press, Melbourne, 907–914. https://doi.org/10.1145/860575.860721

[30] Munindar P. Singh. 2011. Information-Driven Interaction-Oriented Programming: BSPL, the Blindingly Simple Protocol Language. In *Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Taipei, 491–498. https://doi.org/10.5555/2031678.2031687

[31] Munindar P. Singh. 2012. Semantics and Verification of Information-Based Protocols. In *Proceedings of the 11th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Valencia, Spain, 1149–1156. https://doi.org/10.5555/2343776.2343861

[32] Reid G. Smith. 1980. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Trans. Comput.* C-29, 12 (1980), 1104–1113.

[33] Michael Winikoff, Nitin Yadav, and Lin Padgham. 2018. A New Hierarchical Agent Protocol Notation. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 32, 1 (Jan. 2018), 59–133.

[34] Pınar Yolum and Munindar P. Singh. 2005. Engineering self-organizing referral networks for trustworthy service selection. *IEEE Transactions on Systems, Man, and Cybernetics, Part A* 35, 3 (May 2005), 396–407. https://doi.org/10.1109/TSMCA.2005.846401