Journal of Artificial Intelligence Research 69 (2020) 1351-1393 Submitted 06/2020; published 12/2020

An Evaluation of Communication Protocol Languages for Engineering Multiagent Systems

Amit K. Chopra AMIT.CHOPRA@LANCASTER.AC.UK
Lancaster University
Lancaster, LA1 4WA, UK

Samuel H. Christie V SCHRIST@NCSU.EDU
North Carolina State University
Raleigh, NC 27695, USA
Lancaster University
Lancaster, LA1 4WA, UK

Munindar P. Singh SINGH@NCSU.EDU
North Carolina State University
Raleigh, NC 27695, USA

Abstract

Communication protocols are central to engineering decentralized multiagent systems. Modern protocol languages are typically formal and address aspects of decentralization, such as asynchrony. However, modern languages differ in important ways in their basic abstractions and operational assumptions. This diversity makes a comparative evaluation of protocol languages a challenging task.

We contribute a rich evaluation of diverse and modern protocol languages. Among the selected languages, Scribble is based on session types; Trace-C and Trace-F on trace expressions; HAPN on hierarchical state machines, and BSPL on information causality. Our contribution is four-fold. One, we contribute important criteria for evaluating protocol languages. Two, for each criterion, we compare the languages on the basis of whether they are able to specify elementary protocols that go to the heart of the criterion. Three, for each language, we map our findings to a canonical architecture style for multiagent systems, highlighting where the languages depart from the architecture. Four, we identify design principles for protocol languages as guidance for future research.

1. Introduction

We understand a multiagent system (MAS) as a decentralized system of autonomous agents, each of whom represents a real-world entity such as a person or an organization. In particular, a MAS is not a separate computational entity but is realized purely through its member agents. When a MAS models a collective such as an institution, the institution can be viewed as an entity that is itself a member of the MAS (Singh, 2013).

Agents in a MAS coordinate their computations while retaining loose coupling in their construction and decision making. To accommodate such a conception of a MAS, it is crucial that (1) agents coordinate their computations via arms-length communication, that is, via *asynchronous messaging*, and (2) the coordination requirements be clearly specified and support a programming model that facilitates the construction of agents.

The foregoing twin requirements have motivated an extensive study of languages for specifying *communication protocols*. Broadly, a communication protocol specifies the coordination in a MAS by specifying two or more interacting roles, the messages (that is, message schemas) exchanged by these roles, and the conditions under which agents playing those roles may send (instances of) the various messages to one another. Further, a protocol yields a programming interface (skeleton) for every agent such that if an agent implements its interface, then the agent is *compliant*. Compliance with the stated protocol is the primary correctness criterion for individual agents: If all agents complied, then their computations would be correctly coordinated.

For concreteness, Use Case 1 describes a protocol informally.

Use Case 1 (Purchase) *A buyer requests an item from a seller, who responds with an offer. The buyer may accept or reject the offer. If the buyer accepts the seller's offer, the seller delivers the specified item to the buyer, following which the buyer sends the specified payment to the seller.*

For Use Case 1, the roles would be BUYER and SELLER. The messages would be *Request*, *Offer*, *Reject*, *Accept*, *Payment*, and *Deliver*. BUYER may send *Request*, *Accept*, *Reject*, and *Payment*, and SELLER may send *Offer* and *Deliver*. Each message contains the information relevant to that message (presumably capturing what that message connotes in the protocol). For example, *Request* contains the item and *Offer* contains the price. We identify additional constraints from Use Case 1: *Offer* concerns the same item as specified in *Request*; and, *Payment* specifies the same amount as the price in *Offer*.

The notion of protocol is naturally a foundational one for multiagent systems. Both Hewitt (1991) and Gasser (1991) identify protocols as one of the central challenges for MAS. Protocols are central to work on agent communication languages (FIPA, 2002; Vieira, Moreira, Wooldridge, & Bordini, 2007); on institutions, e.g., (d'Inverno, Luck, Noriega, Rodriguez-Aguilar, & Sierra, 2012); and on agent-oriented software engineering (AOSE) methodologies such as Gaia (Zambonelli, Jennings, & Wooldridge, 2003), Tropos (Bresciani, Perini, Giorgini, Giunchiglia, & Mylopoulos, 2004), and Prometheus (Padgham & Winikoff, 2005). The centrality of protocols has spurred work on protocol specification languages and approaches. Agent UML (AUML) (Odell, Parunak, & Bauer, 2001), an early graphical notation for specifying protocols that extends UML, is applied in Tropos and Prometheus and by FIPA for specifying interaction protocols (FIPA, 2003). Inspired from problems in telecommunications networks and UML, Message Sequence Charts (ITU, 2004) was another early standardization of a protocol notation.

1.1 Problem

From modest beginnings in informal notations based on UML, work on protocol languages has grown to encompass a number of sophisticated—and in some cases, complicated—formal approaches that boast diverse basic abstractions and operational assumptions. For example, the abstractions employed include state machines (Baldoni, Baroglio, Martelli, & Patti, 2006; Winikoff, Yadav, & Padgham, 2018), logic-based constraints (Baldoni, Baroglio, Marengo, & Patti, 2013), action descriptions (Desai & Singh, 2008a), trace expressions (Castagna, Dezani-Ciancaglini, & Padovani, 2012; Ferrando, Winikoff, Cranefield, Dignum,

& Mascardi, 2019), session types (Yoshida, Hu, Neykova, & Ng, 2013), and information constraints (Singh, 2011a). Operational assumptions range from synchronous communication (Winikoff et al., 2018), to asynchronous but pairwise FIFO communication (Castagna et al., 2012; Yoshida et al., 2013), to unordered asynchronous communication (Singh, 2011a).

The rich diversity of languages for specifying protocols raises an important question: how may we compare and evaluate them? Today, we lack generally clear evaluation criteria and use cases for protocol languages that would enable us to evaluate them on conceptual grounds. And yet, there is a general belief in the research community that existing languages can largely tackle the challenges of building multiagent systems.

1.2 Contributions, Novelty, and Significance

We posit that current languages, illustrating the established paradigms, largely do not support the engineering of decentralized multiagent systems. In support, we contribute a conceptual evaluation of protocol languages with respect to decentralization. Specifically, our contributions in this paper are the following.

First, we provide evaluation criteria for protocol languages that are informed by decentralization. The criteria have to do with how well a language supports specifying flexible interactions; whether a language enables expressing the appropriate information constraints; and the assumptions (demands) a language makes of a MAS's operational environment.

Second, we undertake a comparative evaluation of selected protocol languages against the aforementioned criteria. The selected languages are modern, diverse, and prominent. An important feature of our evaluation is our reliance on "minimal" use cases for protocols that help bring forth the distinctions between the languages. Specifically, for each criterion, we take realistic use cases that go to its heart and specify them as best possible in each of the selected languages. We then analyze each resulting protocol specification for validity according to the semantics of the language it is specified in. Following this methodology, we show that several of the selected languages fall short of what is required to support decentralization.

Third, we identify the architectural assumptions underlying the selected languages and discuss how they map to MAS architecture.

Fourth, we posit principles for engineering MAS and discuss how the languages fare against them. *No unitary perspective* states that a protocol must not specify orderings of events from a unitary perspective. *Noninterference* and the *end-to-end principle for protocols* both concern layering. Noninterference states that a protocol must not interfere with agent reasoning. The end-to-end principle states that a protocol can be fully and correctly implemented only in agents, not in the infrastructure. Relying on infrastructure for correctness (e.g., via FIFO message delivery) may be inadequate and unnecessary for correctness. The end-to-end principle for protocols derives from the more general end-to-end principle for system design (Saltzer, Reed, & Clark, 1984).

This paper's *significance* lies in bringing forth the information models, semantics, and architectural assumptions underlying protocol languages as they relate to decentralization. Doing so not only provides a conceptual framework for understanding protocols and protocol languages but also clarifies requirements for MAS and yields guidance on research into protocol languages. Its *novelty* arises from the absence, currently, of such a framework.

Notably, this paper focuses on essential representational criteria for protocols and plays down contingent features such as current tool support and popularity.

We use the following notational conventions throughout (except in listings and figures). We use SMALL CAPS for role names; *Slant* for protocol and message names; sans serif for parameter names; and `teletype` for parameter values.

1.3 Organization

The rest of this paper is organized as follows. Section 2 introduces protocols as an architectural abstraction for MAS. Section 3 introduces the languages we analyze at depth in this paper. Section 4 evaluates the languages for concurrency and extensibility, as important aspects of flexibility; Section 5 for protocol instances, integrity, and social meaning, as important aspects related to the information exchanged in protocol enactments; and Section 6 for assumptions about the operational environment for protocol enactment. Section 7 teases out the architectural models underlying the various languages and compares them to the canonical MAS architecture, as presented earlier. It also presents broad principles for protocol languages and evaluates the selected languages against them. Section 8 summarizes the overall evaluation in the context of alternative evaluations in the literature. It ends with a discussion of future directions.

2. Multiagent Systems Architecture

A protocol is an architectural abstraction and therefore any evaluation of protocol languages must start with a clear understanding of MAS architecture. Below, we present a canonical architectural style (Shaw & Garlan, 1996) for MAS, clearly indicating its components, assumptions, and constraints. The idea is that any concrete instantiation of the architecture must satisfy constraints but without making stronger assumptions.

Strictly speaking, agents and roles are distinct categories (an agent may play several roles and a role may be played by several agents). For expository convenience, and to focus on other concerns in this paper, we assume that any role is played by a single agent and distinct roles are played by distinct agents. This assumption enables us to identify an agent with the role it plays. From here on, we talk primarily of agents and deemphasize roles.

As Figure 1 depicts, each agent represents an autonomous *principal*, for example, a human or an organization. An agent internally encodes the private decision making of its principal, including any private knowledge bases that it relies upon for decision making. We elide principals in the subsequent figures.

In general, to achieve interoperation, the interoperating parties must agree at multiple levels (Singh & Huhns, 2005). Here, the protocols focus on the operational level, which concerns the exchange of information (i.e., reflected in constraints on the ordering and occurrence of messages). In particular, we set aside both low-level concerns such as how the information is encoded and high-level concerns as to the meaning of the information exchanged—though we insist that protocols be able to support a flexible representation of meaning.

An agent sends and receives messages via a communication infrastructure. An agent's *observations* are its message emissions and receptions. For simplicity and in accordance with the literature, we assume that agents make observations serially (Agha, 1986; Fagin,

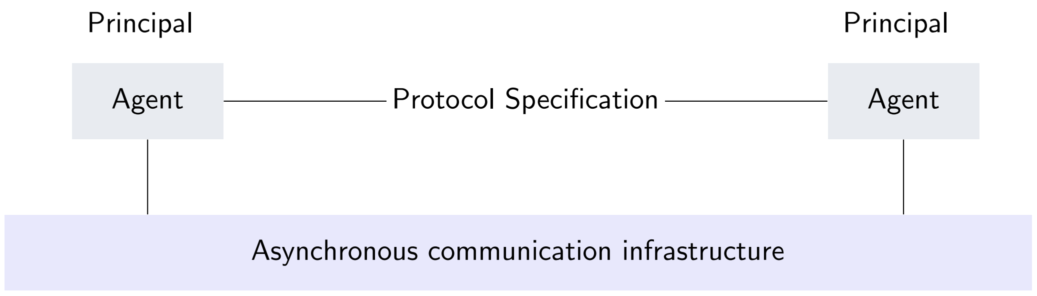


Figure 1: Minimal MAS architecture. Agents implement the protocol and communicate via asynchronous messaging. The communication infrastructure provides no message delivery guarantees other than that it is noncreative (delivers only those messages that were sent).

Halpern, Moses, & Vardi, 1995; Hewitt, 1977). An agent's *history* is the set of its observations. Technically, an agent complies with a protocol if and only if all of its observations are correct with respect to the protocol. Constraints 1 and 2 address the correctness of emissions and receptions, respectively.

Constraint 1 (Emission correctness) *The correctness of a message emission by an agent may be determined from the agent's history.*

Constraint 1 rules out reliance on any kind of state other than the history for purposes of determining the correctness of emissions. In particular, it rules out reliance on (1) the global state, which may include what the agent has not observed; (2) the future state of an agent, because any decision should respect causation; and (3) an agent's internal state, for example, as encoded in its beliefs (Singh, 1998).

Constraint 2 (Reception correctness) *The reception of any message that was emitted correctly is correct.*

Constraint 2 captures the intuition of respecting the structure of causality. Specifically, if the reception of a message could be incorrect, then that message ought never to have been sent. Otherwise, the recipient would enter a "corrupted" state from which there is no recourse. The only alternative would be for the infrastructure to intervene and prevent a message reception that would be erroneous, but doing so would customize the infrastructure to include application-specific details, in contravention of the famous end-to-end principle (Saltzer et al., 1984), which advocates generality in the infrastructure.

Constraints 1 and 2 imply that agents need no more than an asynchronous communication infrastructure, as captured by Constraints 3 and 4.

Constraint 3 (Asynchrony: Nonblocking emission) *Sends are nonblocking, meaning that when an agent sends a message, it does not synchronize with the receiver on the sending action.*

Constraint 4 (Asynchrony: Anytime reception) *An agent receives a message when it is delivered by the infrastructure. That is, message reception is nondeterministic.*

Of course, an agent being autonomous may choose not to act on a message it has received but the reception itself occurs due to the infrastructure.

Asynchrony promotes loose coupling between agents. Notably, programming paradigms for building distributed systems such as the actor model (Agha, 1986; Hewitt, 1977; Hewitt, Bishop, & Steiger, 1973) give prominence to asynchronous messaging for organizing decoupled computations. Practical communication infrastructures such as the Internet support asynchronous messaging. In fact, asynchrony is the only viable option in the important setting of the Internet of Things (IoT) (OASIS, 2014; Shelby, Hartke, & Bormann, 2014; XMPP, 2015).

Assumption 1 (Infrastructure guarantees) *The infrastructure is noncreative; that is, only sent messages are received. Further, the infrastructure does not deliver corrupt messages.*

Notice that Assumption 1 does not say that a sent message be also received. Indeed, messages may be lost in transit. Some applications of MAS may require messages to be delivered; the analysis in this paper however does not rely upon such an assumption. Also notice that no message delivery order was assumed. Constraint 2 means no such assumption is required for purposes of correctness.

Constraint 5 (Fullness) *A protocol fully specifies a multiagent system at the operational level.*

Conceptually, as stated above, a protocol specifies the constraints on the operational level, i.e., on the information exchange. Constraint 5 states that nothing else besides a protocol is needed to characterize a MAS at the operational level. That is, this constraint rules out reliance on extra-protocol mechanisms for coordination, e.g., agreements about when to send or not send certain messages. Such extra-protocol mechanisms would amount to hidden coupling between the agents: Agents developed to interoperate solely on the basis of the protocol would not interoperate with agents that relied on extra-protocol mechanisms. Constraint 5 means that Figure 1 captures a MAS fully from the standpoint of coordination.

Figure 2 elaborates on the architecture of Figure 1 by refining an agent into two components: *protocol filter* and *reasoner*.

An agent's protocol filter ensures compliance. The filter interfaces with the communication infrastructure to send and receive messages. And it interfaces with the reasoner to notify the reasoner of observations of interest and to accept message emission requests. The filter materializes the agent's history and uses it to check the correctness of any message that the reasoner requests it to send. If the message is correct, the filter sends the message on the infrastructure (and records the emission as an observation in the history). If the message is not correct, the filter discards the message with the indication of an exception to the reasoner (and does not change the history). The filter is a form of generic protocol-based *control* on the reasoner (Banihashemi, De Giacomo, & Lespérance, 2016, 2018).

An agent's reasoner encodes the decision making of its principal. The reasoner determines how an agent processes events, both private (e.g., an update to an internal knowledge base) and observations recorded by the filter. The processing of an event may require the emission of a message, for which the reasoner relies on the filter. For example, referring to

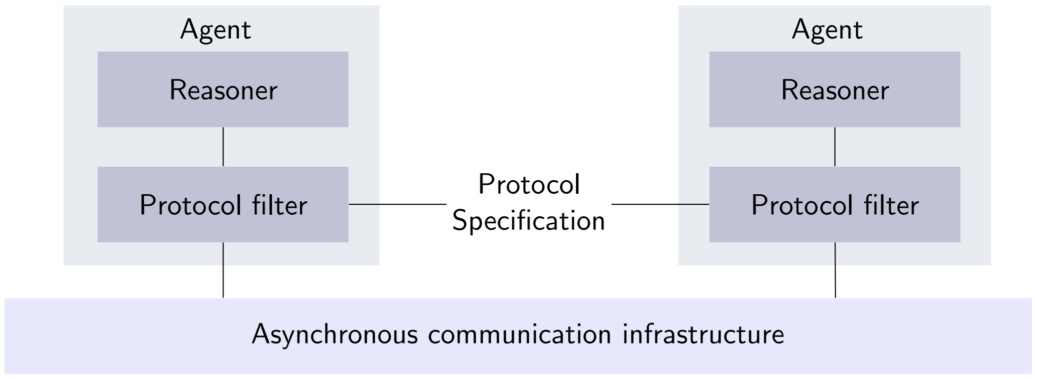


Figure 2: MAS architecture with compliance checking. History is maintained by the protocol filter for purposes of compliance checking.

Use Case 1, BUYER's reasoner, upon being notified by an internal database that a particular item was out of stock, may ask the filter to send a *Request* for that item to SELLER. If the *Request* is correct, the filter sends it to SELLER. SELLER's reasoner, when notified by its filter of the reception of the *Request*, may determine—by looking up an internal price list—that an *Offer* for the requested item should be sent for some price. And so on.

The architecture in Figure 3 further refines the architecture in Figure 2 by introducing a declarative specification of the *social meaning* of an interaction (Singh, 1998) and a runtime for the language in which meaning is specified, namely, the *meaning computer*. The meaning specification takes an agent's observations as the base-level social events and maps combinations of social events to higher-level social events.

Constraint 6 defines what may be considered a social event.

Constraint 6 (Social) *A social event is either an observation or is inferred from other social events (Chopra & Singh, 2016).*

Constraint 6 means that a social event cannot feature any information that does not show up in an observation (of a message, as defined earlier). Internal events that reflect updates to an agent's internal state (e.g., its beliefs) have no effect on the computation of social events (Singh, 1998).

Social meaning is essential to the application-specific correctness of MAS. A MAS for financial loans may model social meaning via abstractions for *debt*, *collateral*, *default*, and so on. For example, from events corresponding to the issuance of a loan and a payment against the loan, a new debt event could be inferred that reflects the outstanding debt. Further, were the outstanding debt to be zero, it could lead to the inference of a new *repaid* event. In like manner, a MAS that supports a community of toy train enthusiasts could model social meaning via abstractions for the *provenance*, *ownership*, and *desirability* of a toy train.

In MAS research, social meaning is often modeled via commitments (Bentahar, Moulin, Meyer, & Chaib-draa, 2004; Dastani, van der Torre, & Yorke-Smith, 2017; Fornara & Colombetti, 2002; Meneguzzi, Magnaguagno, Singh, Telang, & Yorke-Smith, 2018; Telang, Singh,

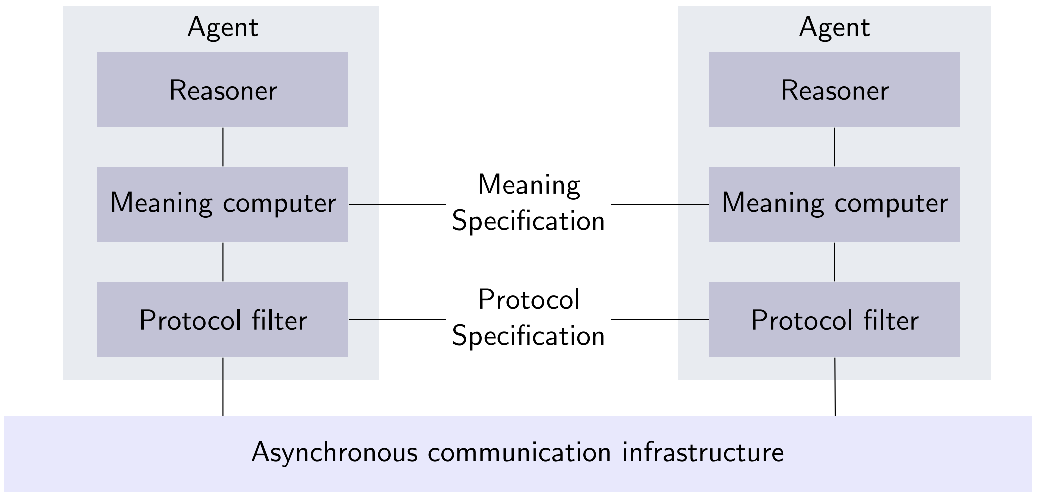


Figure 3: MAS architecture with social meaning. Agents interoperate on the basis of protocols and higher-level social meanings. An agent's meaning computer infers social events from its history.

& Yorke-Smith, 2019; Winikoff, Liu, & Harland, 2005; Yolum & Singh, 2002) and other norms (Alechina, Halpern, Kash, & Logan, 2018; Artikis, Sergot, & Pitt, 2009; Padget, Vos, & Page, 2018). In the rest of the paper, for reasons of concreteness and familiarity, we use commitments as an exemplar way of modeling social meaning. Use Case 2 illustrates the use of commitments to capture meaning.

Use Case 2 (Deliver-Payment Commitment) *In the context of purchase in Use Case 1, the meaning of an Accept from* SELLER *to* BUYER *for some* item *for some* price *is that it creates a commitment from* BUYER *to* SELLER *that if* SELLER *Delivers the* item *by some deadline, then* BUYER *will make a Payment of the* price *by some deadline.*

3. Overview of Selected Languages

For this evaluation, we select protocol specification languages that are recent, have a formal semantics, and represent diverse doctrines. AUML notably is not in our selection: neither is it recent nor does it have a satisfactory formal semantics. AUML is closely related to UML Sequence Diagrams, which too lacks a formal semantics. Some of the selected languages though adopt important intuitions behind AUML, including the idea of specifying an interaction as a control flow of messages and using a graphical notation. One might argue that some of the languages we discuss below seek to formalize intuitions that undergird AUML.

Below, we introduce the main ideas of the selected languages by specifying Use Case 1.

3.1 Multiparty Session Types: Scribble

Scribble (Yoshida et al., 2013) is a practical instantiation of multiparty session types (Honda, Yoshida, & Carbone, 2016). In Scribble, a protocol is an ordering of constituent protocols (bottoming out at individual message specifications) using constructs such as sequence, choice, and recursion. Scribble assumes that communication between pairs of participants is asynchronous but ordered over FIFO channels.

Listing 1 gives an encoding of Use Case 1 as a Scribble protocol. In the listing, a semicolon (**;**) indicates sequencing.

Listing 1: *Purchase* (Use Case 1) in Scribble.

```
global protocol Purchase(role Buyer, role Seller) {
 Request() from Buyer to Seller;
 Offer() from Seller to Buyer;

 choice at Buyer {
   Accept() from Buyer to Seller;
   Deliver() from Seller to Buyer;
   Payment() from Buyer to Seller;
  } or {
   Reject() from Buyer to Seller;
  }
}
```

Given a protocol, Scribble yields projections, called *local protocols*, for each agent. (We retain the term "projection" to avoid conflict with "protocol.") The idea is that the protocol represents computations from a unitary perspective whereas an agent's projection represents computations from its own local perspective. Scribble's tools (Scribble, 2018) may be used to generate these projections. We have used the tooling to verify all Scribble specifications presented in this paper.

Listing 2 gives the projections for each of the agents in the *Purchase* protocol in Listing 1. BUYER's projection says: send *Request* to SELLER, then receive *Offer* (from SELLER), then send either *Accept* or *Reject*. If *Accept* is sent, then receive *Deliver* and then send *Payment*. SELLER's projection is read in an analogous manner.

Notice that in the protocol, the choice between *Accept* and *Reject* is indicated as BUYER's. Therefore, in the projections, the choice is interpreted as an internal choice for BUYER and as an external choice for SELLER. The agent with an internal choice chooses from the available alternatives autonomously. The agent with an external choice does not choose but follows along. The internal choice determines the external choice. Thus, if BUYER chooses to send *Accept* (alternatively, *Reject*), its reception resolves the SELLER's choice to receive *Accept* (alternatively, *Reject*).

Listing 2: Scribble projections of *Purchase* (Listing 1) for BUYER and SELLER.

```
local protocol Purchase_Buyer(role Buyer, role Seller) {
 Request() to Seller;
 Offer() from Seller;

 choice at Buyer {  //internal choice
```

```
   Accept() to Seller;
   Deliver() from Seller;
   Payment() to Seller
 } or {
   Reject() to Seller;
 }
}

local protocol Purchase_Seller(role Buyer, role Seller) {
 Request() from Buyer;
 Offer() to Buyer;

 choice at Buyer {    //external choice
   Accept() from Buyer;
   Deliver() to Buyer;
   Payment() from Buyer;
 } or {
   Reject() from Buyer;
 }
}
```

The notion of realizability ties together a protocol and its projections. A protocol is *realizable* if and only if the agents acting locally based on their projections jointly realize exactly the computations of the protocol (as we shall see, this is not always the case). The *Purchase* protocol in Listing 1 is realizable.

3.2 Trace-C

Castagna et al. (2012) describe a language for specifying protocols that is based upon trace expressions, which we refer to as *Trace-C*. A trace is a sequence of communication events. In Trace-C, each expression maps to a set of traces. The expression $x \xrightarrow{\mathrm{m}} y$ is atomic; it denotes the communication of message m from x to y; and it maps to the (singleton) set of traces $\{m\}$. The **;** operator denotes sequential composition; the expression $e; f$ is the concatenation of the traces of e with the traces of f. The $\vee$ operator denotes choice; the expression $e \vee f$ is the union of traces of e and the traces of f. The $\wedge$ operator denotes the shuffle of its operands; the expression $e \wedge f$ is the set of those traces that represent an interleaving of a trace of e with a trace of f. Like Scribble, Trace-C assumes FIFO-based asynchronous communication.

Listing 3 shows how Use Case 1 may be rendered in Trace-C. Although the Trace-C specification appears more algebraic than Scribble, we can see that they are structurally similar once we realize that the choice operator in Scribble corresponds to the $\vee$ operator in Trace-C.

Listing 3: *Purchase* protocol in Trace-C (and Trace-F).

$$\begin{array}{l}
\text{Buyer} \xrightarrow{\text{Request}} \text{Seller} \;;\; \text{Seller} \xrightarrow{\text{Offer}} \text{Buyer} \;; \\
\quad ((\text{Buyer} \xrightarrow{\text{Accept}} \text{Seller} \;;\; \text{Seller} \xrightarrow{\text{Deliver}} \text{Buyer} \;;\; \text{Buyer} \xrightarrow{\text{Payment}} \text{Seller}) \vee \\
\qquad \text{Buyer} \xrightarrow{\text{Reject}} \text{Seller})
\end{array}$$

Like in Scribble, a Trace-C protocol yields projections for each agent. Listing 4 gives the projections for *Purchase* in Listing 3. In the projections, $\oplus$, $+$, and **;** denote internal choice, external choice, and sequence, respectively; AGENT!*Message* and AGENT?*Message*, respectively, denote the emission of *Message* to AGENT and the reception of *Message* from AGENT. The projections are structurally similar to those of *Purchase* in Scribble, even though the syntax is different. Notice that BUYER's choice is internal and SELLER's external, meaning that although SELLER could receive either *Accept* or *Reject*, the choice of what it receives depends on what BUYER sends. The protocol is realizable.

Listing 4: Trace-C projections of *Purchase* in Listing 3.

```
//Buyer's projection
Buyer: Seller!Request ; Seller?Offer ;
  ((Seller!Accept ; Seller?Deliver ; Seller!Payment) ⊕ Seller!Reject)

//Seller's projection
Seller: Buyer?Request ; Buyer!Offer ;
  ((Buyer?Accept ; Buyer!Deliver ; Buyer?Payment) + Buyer?Reject)
```

3.3 Trace-F

Ferrando et al. (2019) describe a trace expressions-based language for specifying protocols, which we refer to as *Trace-F*. It builds upon earlier work on monitoring decentralized MAS (Ferrando, Ancona, & Mascardi, 2017). Trace-F, like Trace-C, features operators for sequence, choice, and shuffle. In Trace-F, shuffle is represented |; however, for uniformity with Trace-C, we use the Trace-C representation for shuffle, that is, $\wedge$. With this simplification, the protocol in Listing 3 serves as a specification of Use Case 1 in Trace-F.

The projections generated by Trace-F for Listing 3 though are different from the projections produced by Trace-C as shown in Listing 4. Specifically, in Trace-F, the choice in the protocol does *not* reduce to internal and external choice in the projections for BUYER and SELLER. Instead, the choice is preserved in the projection and the distinction between internal and external choice is captured semantically in a decision structure. In general, Trace-F preserves all binary operators used in a protocol in the projections.

Listing 5: Trace-F projections of *Purchase* in Listing 3.

```
//Buyer's projection
Buyer: Seller!Request ; Seller?Offer ;
 ((Seller!Accept ; Seller?Deliver ; Seller!Payment) ∨ Seller!Reject)

//Seller's projection
Seller: Buyer?Request ; Buyer!Offer ;
 ((Buyer?Accept ; Buyer!Deliver ; Buyer?Payment) ∨ Buyer?Reject)
```

Ferrando et al. (2019) introduce two dimensions of variation in reasoning about the realizability (which they term "enactability") of a protocol. One dimension concerns the communication infrastructure—whether it is asynchronous or synchronous and if it is asynchronous what kind of ordered delivery guarantees it offers. Out of the other approaches evaluated in this paper that support asynchrony, none requires stronger ordering guarantees than FIFO delivery. Hence, the interesting cases for Trace-F, for our purposes, are asynchrony without any kind of ordered delivery, which we refer to as *unordered asynchrony*, and asynchrony with FIFO delivery, which we refer to as *FIFO asynchrony*.

The other dimension that Ferrando et al. (2019) introduce (drawing upon (Desai & Singh, 2008b)) concerns how the sequence operator is interpreted in terms of the observations of events. Take the protocol in Listing 6.

Listing 6: A Trace-F protocol.

$$W \xrightarrow{p} X \ ; \ W \xrightarrow{q} Y$$

Under the *send before send* (SS) interpretation, W must send p before W sends q. Under the *send before receive* (SR) interpretation, W must send p before Y receives q. Under the *receive before send* (RS) interpretation, X must receive p before W sends q. And, under the *receive before receive* (RR) interpretation, X must receive p before Y receives q.

Whether a protocol is realizable depends on the communication infrastructure and the interpretation of the sequence operator. For concreteness, let us consider the protocol in Listing 6 under unordered asynchrony. The protocol is realizable with either SS (W being the sender of both p and q can ensure that p is sent before q) or SR (from the facts that the protocol is realizable under SS and the emission of a message must be prior to its reception). However, the protocol is realizable neither under RS (W has no way of knowing when X has received p, so it cannot ensure that q will be sent after the reception of p) nor under RR (since the receivers are different, there is no way to ensure that q will be received after the reception of p). Changing the interpretation to FIFO asynchrony makes no difference (because the receivers of p and q are different).

To see how the choice of communication infrastructure makes a difference, consider the protocol in Listing 7. Notice that both p and q are messages from W to X. Under unordered asynchrony and with the RR interpretation, the protocol is unrealizable (there being no way to guarantee that p will be received before q). However, under FIFO asynchrony and the RR interpretation, the protocol is realizable (p is sent before q, so p is also received before q).

Listing 7: A Trace-F protocol.

$$W \xrightarrow{p} X \ ; \ W \xrightarrow{q} X$$

3.4 HAPN

HAPN (Winikoff et al., 2018) is a graphical language that enables specifying a protocol as a nested state machine in a manner similar to statecharts (Harel, 1987). As Figure 4 shows, nodes represent states or reference other protocols to compose those protocols. Edges can have complex annotations, supporting the specification of message transmissions, guard expressions, and changes to the state. HAPN has been implemented in tooling that animates a state machine. Although Winikoff et al. acknowledge the importance of realizability, they do not give a method for projecting HAPN protocols nor do they discuss the specific infrastructure assumptions under which a HAPN protocol would be realizable.

HAPN provides methods to flatten a hierarchical protocol into simple protocols and finite state machines for verification.

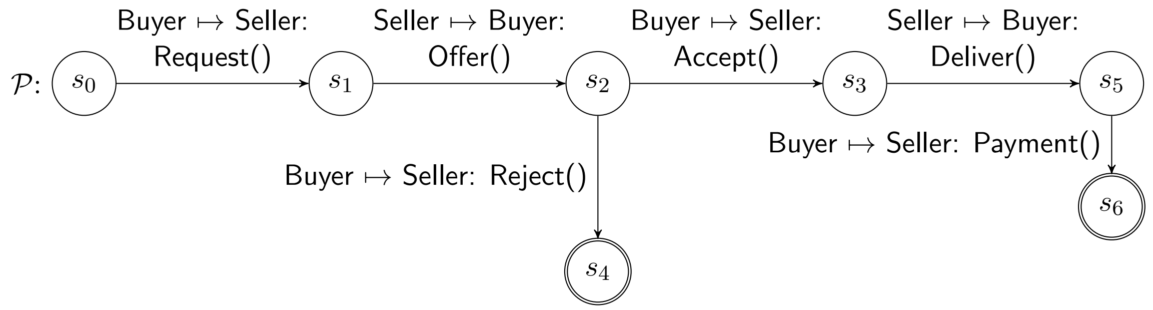


Figure 4: *Purchase* in HAPN, starting from s_0.

3.5 BSPL

BSPL (Singh, 2011a, 2012), the Blindingly Simple Protocol Language, and Splee (Chopra, Christie, & Singh, 2017), which extends BSPL, are exemplars of information-based languages. Instead of specifying the control flow between messages, BSPL specifies information causality and integrity constraints.

Listing 8 shows the *Purchase* protocol in BSPL. It specifies a set of roles, a set of parameters, and a set of messages. In *Purchase*, the roles are BUYER and SELLER; the parameters are ID, item, price, decision, and OK; and message schemas are *Request*, *Offer*, and so on. *Request* is a message from BUYER to SELLER and has parameters ID and item. BSPL is declarative; the order in which the messages appear in a protocol is irrelevant to how the protocol may be enacted.

Listing 8: *Purchase* in BSPL.

```
Purchase {
 role Buyer, Seller
 parameter out ID key, out item, out price, out decision, out OK

 Buyer ↦ Seller: Request[out ID, out item]
 Seller ↦ Buyer: Offer[in ID, in item, out price]
 Buyer ↦ Seller: Accept[in ID, in item, in price, out decision, out
    address]
 Buyer ↦ Seller: Reject[in ID, in item, in price, out decision, out OK]
 Seller ↦ Buyer: Deliver[in ID, in item, in address, out dropOff]
 Buyer ↦ Seller: Payment[in ID, in price, in dropOff, out OK]
}
```

A BSPL protocol may be viewed as an information object as described by the protocol parameters, at least one of which is annotated key. The key parameters enable identifying *instances* of the protocol: distinct bindings for the key parameters identify distinct *instances* of the protocol. *Purchase* specifies ID as its key parameter. A key parameter of the protocol is also a key parameter of the messages in which it appears and enables identifying distinct instances of messages. Thus, parameter ID is key for all messages in *Purchase*. Protocol instances are related to protocol enactment: A protocol instance is a *view* over correlated (by bindings of common keys) message instances. For example, an emission of *Request* with ID `1` and item `fig` yields a *Purchase* instance with ID `1` and item `fig`.

A protocol instance must satisfy *integrity*, which is the idea that no two message instances that are correlated with the protocol instance may conflict on (that is, have different bindings for) any parameter—this is the meaning of a key. Thus for example, a *Request* with ID `1` and item `fig` and an *Offer* with ID `1` and item `jam` would violate integrity: ID `1` may either be associated with item `fig` or item `jam`, but not both.

For any instance, *causality* constraints specify information flow and are expressed via ⌜in⌝ and ⌜out⌝ adornments on parameters (we omit discussion of the ⌜nil⌝ adornment since it does not feature in any BSPL specification in the present paper). Ordering between messages falls out of these constraints. To see how, consider *Request*. In *Request*, both ID and item are adorned ⌜out⌝, meaning that in sending a *Request*, BUYER *produces* bindings for ID and item. When SELLER receives the *Request*, it comes to know those bindings unless it knew them already. In *Offer*, both ID and item are adorned ⌜in⌝, meaning that SELLER needs to *know* these parameters before SELLER can send *Offer*. This means that if SELLER has seen *Request* before, it can send *Offer* by producing a binding for price. When BUYER receives *Offer*, it may send either *Accept* or *Reject* since it knows the bindings of ID, item, and price and it may produce a binding for address (which features in *Accept* but not *Reject*), decision and OK (which features in *Reject* but not *Accept*). It cannot send both *Accept* and *Reject* though because both messages produce a binding for decision, and integrity requires a parameter to have at most one binding. When SELLER receives *Accept*, it knows address and therefore may send *Deliver* by producing a binding for dropOff. Upon reception of *Deliver*, BUYER knows dropOff, and therefore, it can send *Payment* by producing a binding for OK.

A tuple of bindings for a protocol's parameters corresponds to a *complete* protocol instance. That is the motivation for *Purchase* being designed such that *Reject* features OK but *Accept* does not. On the *Accept* branch, the protocol completes with *Payment*.

Unlike the languages introduced earlier, BSPL does not give the computations of a protocol from a unitary perspective. Instead, it takes an inherently decentralized perspective. Any computation of a protocol is a vector comprised of a history for each agent. However, the vector is conceptual (not materialized anywhere). To be able to correctly enact a protocol, an agent needs no more than its history. Therefore projections are trivial in BSPL.

BSPL works with asynchronous communication without any ordering guarantees.

4. Flexibility

Does a language support specifying flexible protocols? Being able to interact flexibly is supportive of autonomy (Yolum & Singh, 2002). However, flexibility is in tension with decentralization: Independently-constructed agents deciding locally must still be able to interoperate. Below, we discuss *concurrency* and *extensiblity*, two aspects of flexibility.

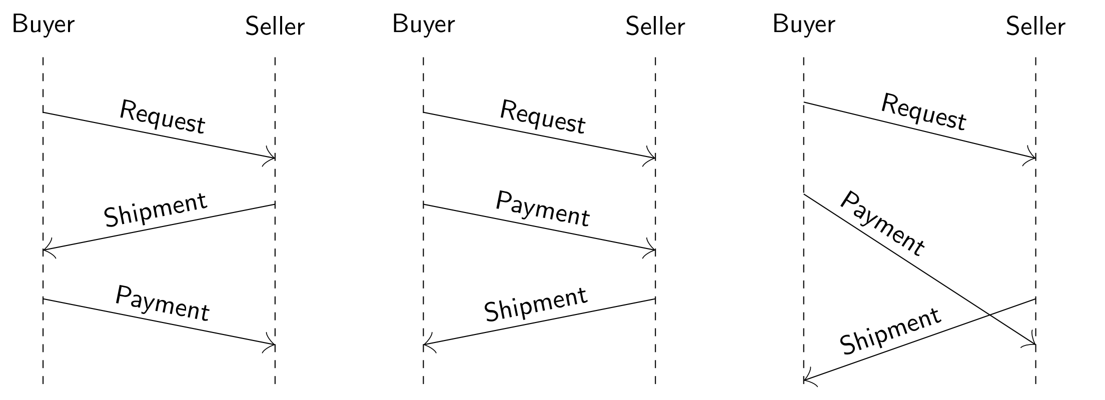
Below, we denote enactments via sequence diagrams, as in Figure 5, where each agent's lifeline captures its history.

4.1 Concurrency

Does a language support specifying protocols in which agents may emit and receive messages concurrently? Consider Use Case 3.

Use Case 3 (Flexible purchase) BUYER *sends Request to* SELLER *to ship some item. After sending Request,* BUYER *may send Payment. After receiving Request, Seller may send Shipment. That is, Payment and Shipment are not mutually ordered.*

Figure 5 shows some possible enactments for Use Case 3.



(a) Shipment first. (b) Payment first. (c) Concurrent.

Figure 5: Three possible enactments of Flexible purchase (Use Case 3).

Listing 9 serves as a protocol specification in both Trace-C and Trace-F that prima facie captures Use Case 3 by not mutually ordering *Payment* and *Shipment*.

Listing 9: Flexible purchase (Use Case 3) in Trace-C and Trace-F.

```
//Flexible purchase
Buyer $\xrightarrow{Request}$ Seller; (Buyer $\xrightarrow{Payment}$ Seller ∧ Seller $\xrightarrow{Shipment}$ Buyer)
```

To understand what enactments are supported by the protocol in Listing 9, following Trace-C (Castagna et al., 2012, p. 14), we eliminate ∧ from the protocol to obtain the equivalent protocol in Listing 10. Trace-C determines the protocol in Listing 10 as unrealizable.

Listing 10: A Trace-C protocol equivalent to the protocol in Listing 9.

```
  Buyer $\xrightarrow{Request}$ Seller;
  ((Buyer $\xrightarrow{Payment}$ Seller ; Seller $\xrightarrow{Shipment}$ Buyer) ∨ (Seller $\xrightarrow{Shipment}$ Buyer;
      Buyer $\xrightarrow{Payment}$ Seller))
```

Let us see why. Listing 11 gives the projections that Trace-C yields for Listing 10. The choice (denoted by ∨) in the protocol must be interpreted as external choice (denoted +) for one agent and internal choice (denoted ⊕) for the other. Recall that an agent with an internal choice can choose autonomously. An agent with an external choice cannot; its choice is determined by the internal choice of another agent. In Listing 11, BUYER has the internal choice and SELLER the external choice (it would not matter to our analysis if it were the other way around since the situation is symmetric).

Listing 11: Trace-C projections of the protocol in Listing 10.

```
Buyer:  Seller!Request.
         ((Seller?Shipment.Seller!Payment) ⊕
            (Seller!Payment.Seller?Shipment))

Seller:  Buyer?Request.
         ((Buyer!Shipment.Buyer?Payment) +
            (Buyer?Payment.Buyer!Shipment))
```

Given the projections in Listing 11, if buyer chooses to send *Payment*, when *Payment* reaches seller, it effectively determines the choice to receive *Payment* by seller. Such an enactment realizes the protocol trace where *Payment* happens before *Shipment*, so no problem here. However, if buyer chooses to receive *Shipment*, seller must send it. The seller could send *Shipment* if it knew of buyer's choice or it could act autonomously. However neither is a possibility. Constraint 5 rules out covert communication and synchronization and therefore rules out the possibility of the seller learning of buyer's choice. As seller's choice is internal, it cannot send *Shipment* autonomously. This means the system is deadlocked, which leads Trace-C to conclude that the protocol is unrealizable. The situation where agents must make mutually compatible choices to ensure correctness is known as nonlocal choice (Ladkin & Leue, 1995).

Listing 12 shows the projections in Trace-F for the protocol in Listing 9. Under both unordered and FIFO asynchrony, the protocol is determined unrealizable by Trace-F, no matter what interpretation is chosen for the sequence operator. The reason behind the rejection is the same reason the Trace-C protocol above is rejected: a nonlocal choice that cannot always be made in a mutually compatible manner by the agents.

Listing 12: Trace-F projections of the protocol in Listing 10.

```
Buyer:  Seller!Request.  ((Seller?Shipment.Seller!Payment) ∨
(Seller!Payment.Seller?Shipment))

Seller:  Buyer?Request.
   ((Buyer!Shipment.Buyer?Payment) ∨ (Buyer?Payment.Buyer!Shipment))
```

Listing 13 shows how we might model Use Case 3 in Scribble. For the same reasons as for Trace-C, the protocol in the listing is determined unrealizable by Scribble.

Listing 13: Flexible purchase (Use Case 3) in Scribble.

```
global protocol FlexiblePurchase (role Buyer, role Seller) {
 Request() from Buyer to Seller;
 choice at Buyer {
   Payment() from Buyer to Seller;
   Shipment() from Seller to Buyer;
 } or {
   Shipment() from Seller to Buyer; // not valid
   Payment() from Buyer to Seller;
 }
}
```

Some research branches of Scribble (Demangeon, Honda, Hu, Neykova, & Yoshida, 2015) have included a *parallel* operator, which however is absent from the main Scribble language and implementation. We hypothesize that a parallel operator would manifest as a problematic nonlocal choice. Our hypothesis is based on the fact that Trace-C's $\wedge$ operator is in essence a parallel operator and as we showed in the analysis of flexible purchase in Trace-C, $\wedge$ manifests as a problematic nonlocal choice in the projections.

Figure 6's HAPN protocol captures only the first two enactments of Figure 5, not the concurrent one because HAPN requires synchrony.

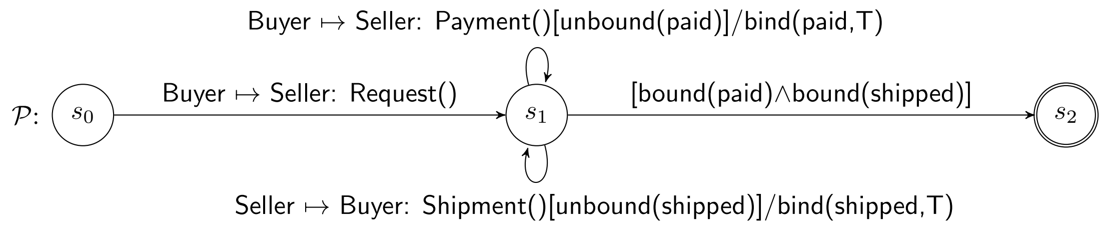

Buyer ↦ Seller: Payment()[unbound(paid)]/bind(paid,T)
𝒫:
s_0
Buyer ↦ Seller: Request()
s_1
[bound(paid)∧bound(shipped)]
s_2
Seller ↦ Buyer: Shipment()[unbound(shipped)]/bind(shipped,T)

Figure 6: Flexible Purchase (Use Case 3) in HAPN.

Listing 14 gives a BSPL protocol. It supports the enactment in Figure 5c because after BUYER sends *Request*, it has the information needed to send *Payment* and, upon receiving *Request*, SELLER has the information needed to send *Shipment*. The protocol also supports the enactments in Figures 5a and 5b.

Listing 14: Flexible purchase (Use Case 3) in BSPL.

```
Flexible Purchase {
 role Buyer, Seller
 parameter out ID key, out item, out shipped, out paid

 Buyer ↦ Seller: Request[out ID, out item]
 Seller ↦ Buyer: Shipment[in ID, in item, out shipped]
 Buyer ↦ Seller: Payment[in ID, in item, out paid]
}
```

4.2 Extensibility

Is the protocol language such that an agent may participate in multiple, potentially unrelated protocols specified in it? If the answer is yes, we refer to the language as *extensible.* Technically, extensibility means that an agent may interleave observations of messages from several protocols and yet be compliant with each of them. If an agent may participate in only one protocol, that is, observe messages from only one protocol, then the language is nonextensible.

Extensibility is a practical necessity. For example, an organization as an agent may interact with other organizations using one protocol but interact with its own members using another protocol. Further, nonextensibility would be an undue restriction on an agent's design and therefore its principal's autonomy.

As basic a requirement as extensibility appears to be, we show below that several languages are in fact not extensible.

Use Case 4 (Pricing and Catalog) *SELLER and BUYER engage in the Pricing protocol, by which a BUYER may obtain offers for requested items from SELLER. In addition, SELLER engages with PROVIDER via the Catalog protocol to obtain information about the newest products. Further, BUYER is unaware of Catalog and PROVIDER is unaware of Pricing, indicating that these protocols are not composed into a single protocol.*

Figure 7 shows an enactment of Use Case 4 in which the messages of *Pricing* and *Catalog* are interleaved. In particular, notice that SELLER observes messages from both protocols. There is nothing in the enactment that tells us it should be deemed incorrect. Such enactments would in fact be indicative of flexibility. If a protocol language were not extensible, then enactments such as the one in Figure 7 would be deemed incorrect.

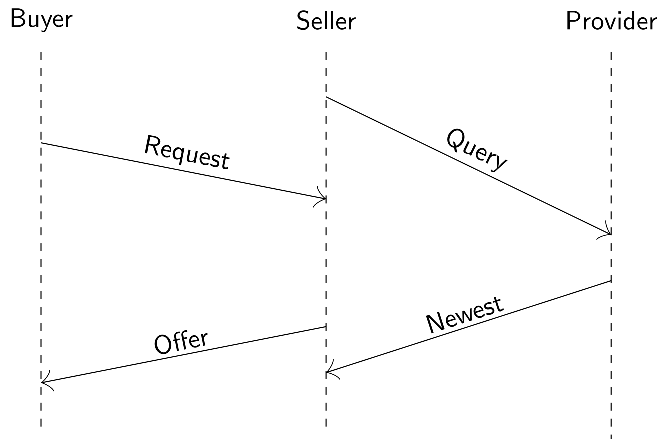


Figure 7: Extensibility means supporting enactments that interleave messages from two protocols, as shown.

Listing 15: Pricing and Catalog (Use Case 4) in Trace-C and Trace-F.

```
//Pricing
Buyer $\xrightarrow{\text{Request}}$ Seller ; Seller $\xrightarrow{\text{Offer}}$ Buyer

//Catalog
Seller $\xrightarrow{\text{Query}}$ Provider; Provider $\xrightarrow{\text{Newest}}$ Seller
```

Listings 15 gives the specifications of *Pricing* and *Catalog* in Trace-C. Castagna et al. (2012, p. 3) promote the correctness criterion of *fitness* with respect to a protocol, which says that an agent correctly implements (*fits*) a protocol only if the agent does not observe any message that is not in the protocol. Castagna et al. give the example of a protocol in which SELLER may send BUYER messages corresponding to *price* and *descr* (of an item) and say that a seller that sent any message other than *price* and *descr* would violate the protocol. Naturally, it follows that if an agent observes messages from two or more Trace-C protocols, then it correctly implements none of those protocols. In essence, fitness limits agents to implementing a single protocol.

Returning to our example, Trace-C says that to correctly implement *Pricing*, SELLER must not observe any message from *Catalog* and to correctly implement *Catalog*, SELLER must not observe any message from *Pricing*. An agent that engages in both *Pricing* and *Catalog* violates both. Considering the SELLER's projections for *Pricing* and *Catalog*, given in Listing 16, helps see the violations clearly. If you consider the projection from *Pricing*, the only trace that satisfies the projection is *Request* followed by *Offer*. In particular, no trace with *Query* or *Newest* satisfies the projection from *Pricing*. By an analogous argument, no trace with *Request* or *Offer* would satisfy the projection from *Catalog*. Based on the foregoing analysis, we conclude that Trace-C does not support extensibility.

Listing 16: Projections of *Pricing* and *Catalog* in Trace-C (and Trace-F) for Seller.

```
//Seller's projection from Pricing
Seller: Buyer?Request ; Buyer!Offer

//Seller's projection from Catalog
Seller: Provider!Query ; Provider?Newest
```

Following an analogous line of reasoning, we can see that Trace-F and Scribble do not support extensibility either: *Pricing* and *Catalog* yield projections for SELLER, none of which entertains traces with events from the other (in fact, the projections in Listing 16 double as projections in Trace-F as well). HAPN's state-machine semantics rules out interleavings with other protocols. Therefore, HAPN too does not support extensibility.

The underlying reason Trace-C, Trace-F, Scribble, and HAPN come up short on extensibility is they all implicitly identify the universe of discourse (the messages that agents may observe) with the messages specified in the protocol.

BSPL supports extensibility. In fact, BSPL refers to a universe of discourse, which for an agent would include any messages it observes, regardless of the protocols they feature in. Any message schema is an elementary protocol is BSPL and determining the correctness of an observation of the message depends on the causality and integrity constraints specified in the schema. Correctness does not depend upon the protocol in which a message schema occurs. Listing 17 and Listing 18 specify *Pricing* and *Catalog*, respectively, in BSPL. The role SELLER features in both protocols and interleaves observations of messages from both protocols.

Listing 17: Pricing in BSPL.

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

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

Listing 18: *Catalog* in BSPL.

```
Catalog {
 role Seller, Provider
 parameter out qID key, out req, out products

  Seller ↦ Provider: Query[out qID, out req]
  Provider ↦ Seller: Newest[in qID, in req, out products]
}
```

One may argue that the problem of extensibility is rendered moot if both Pricing and Catalog were composed into a single protocol. Composing the protocols would enable an agent to interleave interactions from both protocols without compromising fitness. Listing 19 gives *Pricing+Catalog*, a composition of *Pricing* and *Catalog*, in Trace-C. The fitness of SELLER with respect to the composed protocol is thus a possibility. However, there are significant drawbacks to requiring explicit composition just for the sake of fitness. First, it would create large unwieldy protocols with potentially unrelated communications and introduce otherwise unrelated agents into the MAS just because they are reachable from each other based on their interactions with common agents. Second, when protocols are large and unwieldy, projections for agents and their implementations would become correspondingly complicated. Third, it would prevent any organizational abstraction. For example, the protocol by which an organization trades with others will have to be composed with all the protocols pertaining to interactions internal to the organization, which would be undesirable. Fourth, the composite protocol may turn out to be unrealizable anyway. *Pricing+Catalog* in Listing 19 is in fact unrealizable. The reason is that the protocol sets up a problematic nonlocal choice (SELLER sends *Query* or BUYER sends *Request*).

Listing 19: *Pricing+Catalog* in Trace-C.

// Pricing+Catalog

(Buyer $\xrightarrow{\text{Request}}$ Seller; Seller $\xrightarrow{\text{Offer}}$ Buyer) $\wedge$ (Seller $\xrightarrow{\text{Query}}$ Provider;
    Provider $\xrightarrow{\text{Newest}}$ Seller)

5. Information Modeling

Does a language enable capturing the information model underlying the interactions in a MAS? Further, does the information model captured in a protocol specification enable computing social meaning correctly?

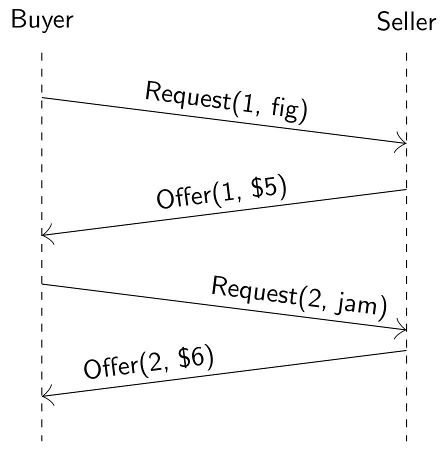
It is the information conveyed via messaging that enables coordination in a MAS. Such information has a model, starting with constraints such as would be captured in a message schema, e.g., to capture message instances, and extending to constraints between message schemas in a protocol specification, e.g., to capture correlation and integrity in protocol enactments, which may be viewed as groups of message instances. Further, due to Constraint 6, since a meaning-level event is computed as a view over one or more protocol events, the soundness of information at the meaning-level relies on the soundness of information generated in protocol-level events.

We elaborate below on protocol instances and the integrity of information as a lead up to social meaning.

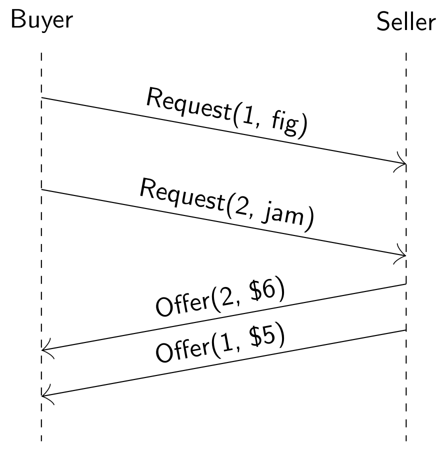
5.1 Protocol Instances

A practical requirement is that a protocol, being a pattern of communication, may be instantiated several times. We refer to each instantiation as a protocol instance that would be comprised of some appropriately correlated messages. How well does a protocol language support—via its constructs—the modeling of protocol instances? Consider Use Case 5.

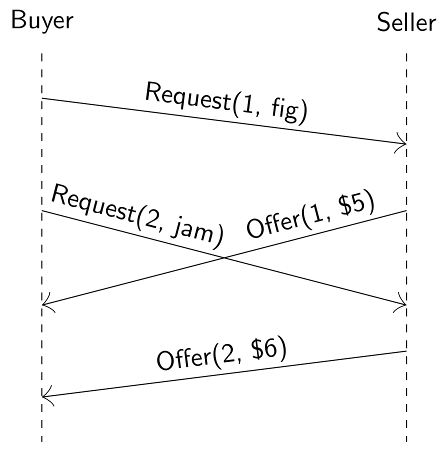
Use Case 5 (Concurrent Pricing) *A buyer and seller may engage in several, possibly concurrent engagements, in each of which a buyer sends a request for some item and the seller responds with an offer.*



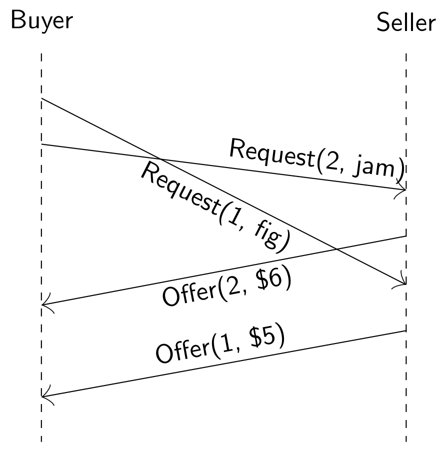
(a) Serial: BUYER sends the second *Request* after receiving an *Offer* for the first.



(b) Second first: BUYER sends two *Request*s. They arrive at the SELLER in the same order as they were sent. SELLER responds to each *Request* with an *Offer* but in the reverse order.



(c) Concurrent: BUYER sends a second *Request*; concurrently, SELLER responds to the first with an *Offer*. The messages cross in transit.



(d) Out of order: BUYER sends two *Request*s, which cross in transit. SELLER responds to the *Request*s with *Offer*s in the order they arrive.

Figure 8: Four possible enactments of Use Case 5, in each of which buyer and seller engage in two instances of *Pricing*. Messages with identifier `1` belong to one instance and messages with identifier `2` to the other instance.

Figure 8 illustrates some enactments involving two instances of the pattern in Use Case 5. The messages contain identifiers (1 and 2) to help distinguish the instances from each other and to correlate messages within an instance.

Listing 20 gives a candidate Scribble protocol capturing Use Case 5. Here, the use of the language feature of recursion enables BUYER and SELLER to engage repeatedly in the pricing pattern of Use Case 5. However, each pricing engagement must happen in its entirety before another can start; that is, the protocol does not allow interleaving of multiple pricing engagements. Thus, although the protocol supports the enactment of Figure 8a, it excludes the enactments of Figures 8b and 8c. The enactment of Figure 8d is also excluded but for a different reason: it violates FIFO, which is a requirement for Scribble.

Listing 20: Concurrent Pricing (Use Case 5) in Scribble.

```
global protocol Pricing(role Buyer, role Seller) {
  Request(ID : String, item : String) from Buyer to Seller;
  Offer(ID, price : String) from Seller to Buyer;
  do Pricing(Buyer, Seller);
}
```

Listing 21 gives a Trace-C protocol for Use Case 5. Trace-C supports iteration via *, meaning that the enclosed pattern may be repeated zero or more times. As for Scribble, each iteration must complete before another can begin, thereby excluding the enactments of Figures 8b and 8c due to the semantics of the language and excluding Figure 8d due to the FIFO requirement.

Listing 21: Concurrent Pricing (Use Case 5) in Trace-C.

$$(\text{Buyer} \xrightarrow{\text{Request(ID, item)}} \text{Seller} \ ; \ \text{Seller} \xrightarrow{\text{Offer(ID, price)}} \text{Buyer})^*$$

Like Scribble, Trace-F too supports recursion. Listing 22 gives a recursive Trace-F protocol for Use Case 5. As for Scribble and Trace-C, each iteration must complete before another can begin, thereby excluding the enactments of Figures 8b and 8c due to the semantics of the language. Under unordered asynchrony, the protocol is unrealizable because the enactment in Figure 8d may occur but the protocol cannot handle it. Under FIFO asynchrony, Figure 8d is excluded, just as for Scribble and Trace-C.

Listing 22: Concurrent Pricing (Use Case 5) in Trace-F.

$$\text{P} = \text{Buyer} \xrightarrow{\text{Request(ID, item)}} \text{Seller} \ ; \ \text{Seller} \xrightarrow{\text{Offer(ID, price)}} \text{Buyer} \ ; \ \text{P}$$

With the aim of supporting the enactment in Figure 8c, Listing 23 gives an alternative Trace-F protocol. This protocol has two problems. One, it would support at most two instances. Two, and in any case, it is unrealizable. This is because the protocol would manifest in a problematic nonlocal choice in the projections for BUYER and SELLER: after sending the first *Request*, BUYER can either send the second *Request* or receive *Offer*, and after receiving the first *Request*, SELLER can either send *Offer* or receive the second *Request*.

Listing 23: Another attempt at Concurrent Pricing (Use Case 5) in Trace-F.

$$(\text{Buyer} \xrightarrow{\text{Request(ID, item)}} \text{Seller} \ ; \ \text{Seller} \xrightarrow{\text{Offer(ID, price)}} \text{Buyer}) \wedge (\text{Buyer} \xrightarrow{\text{Request(ID, item)}} \text{Seller} \ ; \ \text{Seller} \xrightarrow{\text{Offer(ID, price)}} \text{Buyer})$$

Listing 24 (unclear if it is a legal Trace-F specification) improves upon Listing 23 by allowing an unbounded number of instances; however, it remains unrealizable for the same reason as Listing 23.

Listing 24: Concurrent Recursive Pricing (Use Case 5) in Trace-F.

$$P = \text{Buyer} \xrightarrow{\text{Request(ID, item)}} \text{Seller} \; ; \; \text{Seller} \xrightarrow{\text{Offer(ID, price)}} \text{Buyer} \wedge P$$

Figure 9 specifies the protocol in HAPN, which supports iteration by introducing cycles in the state machine. As for the Scribble and Trace-C protocols, and the Trace-F protocol in Listing 22, each iteration must complete before another can begin, thereby excluding the enactments of Figures 8b and 8c. Notice that the requirement of synchrony eliminates Figures 8c and 8d. That is, there are two strikes against Figure 8c.

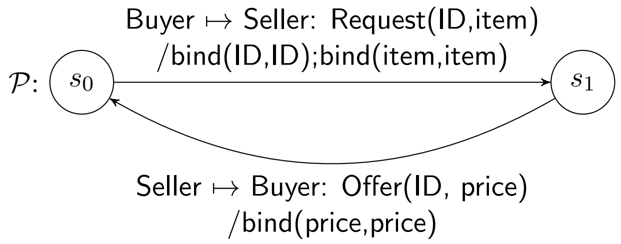

Buyer ↦ Seller: Request(ID,item)
/bind(ID,ID);bind(item,item)
$\mathcal{P}$:
s_0
s_1
Seller ↦ Buyer: Offer(ID, price)
/bind(price,price)

Figure 9: Concurrent Pricing in HAPN.

Because Scribble, Trace-C, and Trace-F support one of the four enactments in Figure 8, we conclude that they partially support instances. By contrast, BSPL fully supports the specification of instances via *key* parameters of protocols. Listing 17 in fact gives the BSPL protocol that supports all the enactments in Figure 8.

To help understand why the protocol in Listing 17 supports each of the enactments in Figure 8, the following observations suffice. First, BUYER can send a *Request* at any point because it can generate bindings for both parameters of *Request*, namely, ID and item. However, as *ID* is key, no two *Request*s may contain the same binding for ID. Second, SELLER may send *Offer*s only for those IDs whose bindings it knows from prior communications—here, from *Request* messages it has received. Further, once SELLER knows a binding for ID, it can send (depending on whether its reasoner determines that it should send) an *Offer* with that binding of ID at any point since the protocol allows it to generate a binding for the only other parameter in *Offer*, namely, price.

As an alternative to using recursion or iteration to capture instances, one could use a *conversation identifier* to capture protocol instances. In this approach, each message would be tagged with an externally generated identifier. Specifically, let m be an instance of a protocol message and c be a conversation identifier; then instead of merely transmitting m, a tuple $\langle c, m \rangle$ would be transmitted. Messages with the same conversation identifier may be considered to belong to the same instance. FIPA ACL (implemented in the JADE platform (Bellifemine, Caire, & Greenwood, 2007)), adopts such an approach, supporting the tagging of messages with conversation identifiers.

Consider the Scribble specification in Listing 25 (analogous specifications could be written in Trace-C, Trace-F, and HAPN). One could argue that when used with conversation identifiers, the specification would capture the essence of Use Case 5, and specifically the enactments in Figures 8a, 8b, and 8c.

Listing 25: Another attempt to model Use Case 5 in Scribble

```
global protocol Pricing(role Buyer, role Seller) {
   RFQ(item:string) from Buyer to Seller;
   Offer(price:string) from Seller to Buyer;
 }
```

However, in being external to protocol languages, the construct of a conversation identifier constitutes an ad hoc mechanism for identifying instances. In particular, using a conversation identifier would go against Constraint 5 by introducing additional coupling between agents (via the particulars of the conversation identifier). Further, no language-based support, e.g., in a protocol adapter, could be provided for managing conversations. The burden of conversation management would fall upon the agent developer. For example, the developer would have to ensure that no two *Request*s are sent with the same conversation identifier; and when a $\langle c, m \rangle$ tuple is sent or received, it advances the state of the correct instance, that is, of the instance identified by c.

Finally, consider that a single conversation identifier will be insufficient for many scenarios. For example, in an instance of an insurance policy subscription, there could be several claims, each a distinct subinstance. Modeling the scenario would therefore require two identifiers, one for the policy subscription (say, pID) and a composite one that identifies a claim in the context of a subscription (say, $\langle$pID, cID$\rangle$). In general, the specification of identifiers would depend on the scenario being modeled, which illustrates really the need of supporting identifiers—and through them, instances—generally and systematically in protocol languages, e.g., as BSPL does.

5.2 Integrity

Building upon instances, *integrity* means that information in a protocol instance must not be inconsistent. For example, in any instance of *Purchase*, item must have a unique binding; it cannot be *fig* in *Request* and *jam* in *Offer*.

Scribble does not support integrity. In Scribble, the message names and the data types of the information carried in a message matter; however, the information carried in the message does not matter. The motivating example of a Scribble travel booking protocol (Yoshida et al., 2013, p. 8) is illuminating in this regard. Listing 26 reproduces the relevant parts of that protocol.

Listing 26: Relevant parts of a Scribble travel booking protocol (Yoshida et al., 2013).

```
global protocol BookJourney(role Customer as C, role Agency as A, role
   Service as S) {
  ...
 query(journey: String) from C to A;
 price(Int) from A to C;
  ...
}
```

Figure 10 partially reproduces the customer's finite state machine (FSM) that is extracted from the above protocol and that serves as the basis for compliance checking in the agent: a deviation from the FSM is a violation of the protocol (Yoshida et al., 2013, p. 10). Notice that the parameter journey is absent; all that matters is that customer sends a *String* to agency. To drive home the point about the lack of information modeling in Scribble, we refer the reader to the implementation of the customer agent (Yoshida et al., 2013, p. 11), which does not mention journey.

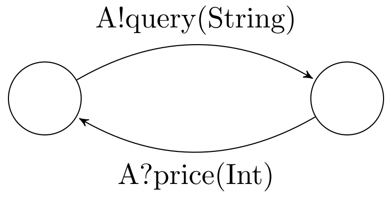


Figure 10: Part of customer's FSM, derived from Listing 26. Here, A is the agency role with which customer interacts.

Returning to the domain of our running examples, consider the Scribble protocol in Listing 27, a variant of Listing 20. Specifically, in the *Alt-Pricing* protocol, *Offer* additionally includes the parameter item. Figure 11 gives the seller's FSM. Notice again that the parameter names are absent; what matters are the data types. This machine would determine to be legal even those protocol enactments that violate integrity, e.g., where *Request* contains (the item binding) `fig` but *Offer* contains (the item binding) `jam`.

Listing 27: Alternative Pricing in Scribble.

```
global protocol Alt-Pricing(role Buyer, role Seller) {
  Request(ID:String, item:String) from Buyer to Seller;
  Offer(ID, item, price:String) from Seller to Buyer;
  do Pricing(Buyer, Seller);
}
```

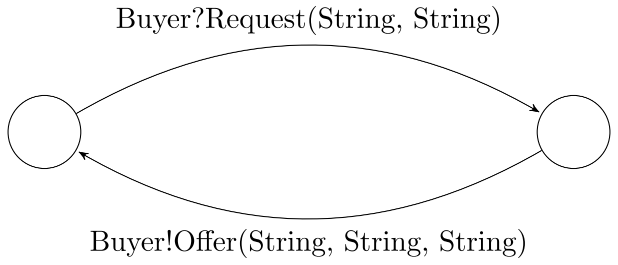

Buyer?Request(String, String)
Buyer!Offer(String, String, String)

Figure 11: The seller's FSM, derived from Listing 27

Like Scribble, Trace-C does not support integrity either. Like Scribble, messages are opaque in Trace-C: the message names matter but their contents do not, as Listing 3 illustrates.

HAPN partially guarantees information integrity. Once a variable is bound in an enactment, as item is at state s_1 in Figure 9, any message attempting to use that variable with a different value is illegal. HAPN's "unbind" operation does not compromise integrity under its assumptions of synchronicity and shared state; all agents would simultaneously see any updates, and would never have inconsistent bindings for any variable. However, HAPN does not ensure integrity when implementing the protocol in a decentralized asynchronous environment, expecting the designer to check for realizability and to make corrections to any problems arising.

BSPL supports integrity, as described in Section 3.5.

Some authors of Trace-F have investigated a language closely related to Trace-F that supports parameters (Ancona, Ferrando, & Mascardi, 2017). This language (which we dub Trace-Parameters), like HAPN, partially captures integrity by supporting parameter bindings. Listing 28 (reproduced from (Mascardi & Ferrando, 2019)) gives a protocol in Trace-Parameters. The expression on the right-hand side of the = may be read as a binding for a parameter (here, ID) followed by the scope over which that binding holds. That is, ID must have the same fresh binding throughout the specified scope. The fresh binding mechanism is not sufficiently expressive to capture integrity, since integrity needs identifiers (as BSPL supports via key parameters). Specifically, the protocol in Listing 28 allows two *Request* messages to be sent with the same binding for ID. Not only that, it allows two *Request* messages with the same binding of ID to have different bindings for item.

Listing 28: A protocol in Trace-Parameters that requires the same binding for ID throughout in every recursion instance.

$$T = ID \;.\; \mathsf{Buyer} \xrightarrow{\text{Request(ID, item)}} \mathsf{Seller} \;;\; \mathsf{Seller} \xrightarrow{\text{Offer(ID, price)}} \mathsf{Buyer} \;;\; T$$

5.3 Social Meaning

In general, a specification of social meaning specifies the lifecycle of meaning instances, as exemplified by recent work on commitments (Chopra & Singh, 2015). For example, the Deliver-Payment commitment specification in Use Case 2 specifies the lifecycle of Deliver-Payment commitment instances. Where each of the instances differ would be in their information content, as Use Case 6 illustrates.

Use Case 6 (Commitment instances) BUYER *and* SELLER *repeatedly engage in purchase (Use Case 1). A distinct instance of the Deliver-Payment commitment (Use Case 2) is created for every Accept. Several commitment instances may exist at any moment.*

The information content of a commitment instance would need to obey certain soundness constraints. Each commitment instance referred to in Use Case 6 would involve a specific item binding and a specific price binding. And in fact, those bindings must be *immutable* over the entire lifecycle of the instance. In particular, it cannot be that the commitment instance is *created* with item binding `fig` and *discharged* with item binding `jam`; such an enactment would clearly be unsound. Notably, reasoning about the state of a commitment instance involves reasoning about several correlated messages that satisfy integrity—e.g., a message to create the instance and a message to discharge it.

Given that meaning-level information is derived from (is a *view* over) underlying protocol events (Constraint 6), identifiers for commitment instances and the necessary correlation and integrity must already be present in protocol enactments. For each instance of the

Deliver-Payment commitment, an instance of *Purchase* must generate the relevant information in a sound manner: an identifier to identify the commitment instance and the bindings for item and price. This means that a commitment instance created by a particular *Accept* (belonging to a particular protocol instance) may be discharged only by a properly correlated instance of *Payment* (belonging to that same protocol instance). And integrity of the protocol instance would ensure that the bindings are immutable over the lifecycle of the commitment instance.

It was common in early work on commitments to introduce arbitrary (syntactic) identifiers for them disconnected from the underlying information. As Chopra and Singh (2009) show, such schemes are incompatible with commitment reasoning.

In a nutshell, the soundness of computations at the commitment level and more generally meaning level imposes requirements relating to protocol instances and their integrity on the protocol language. Scribble, Trace-C, Trace-F, and HAPN at best partially support instances and integrity and therefore fall short on this criterion. BSPL supports both instances and integrity and therefore better supports social meaning. Indeed, Singh (2011a, p. 498) discusses social meaning and other work has applied BSPL toward commitment consistency in decentralized settings (King, Günay, Chopra, & Singh, 2017). Listing 29 specifies the Deliver-Payment commitment in Cupid (Chopra & Singh, 2015). It says that each instance of this specification is from BUYER to SELLER; it is created upon an *Accept*; detached upon a *Deliver* that happens within three days of *Accept*; and discharged upon a *Payment* that happens within three days of *Payment*. Notably, the events in the commitment specification refer to observations of messages of BSPL *Purchase* protocol given in Listing 8. Since BSPL supports protocol instances and their integrity, as a protocol instance progresses, the commitment instance too progresses soundly.

Listing 29: Deliver-Payment commitment in Cupid.

```
commitment Deliver−Payment Buyer to Seller
 create Accept
 detach Deliver [, Accept + 3]
 discharge Payment [, Deliver + 3]
```

6. Operational Environment

How strong are the assumptions that a language makes of the operational environment of a multiagent system? The properties a language requires of a communication infrastructure are assumptions about the agent's operational environment. The stronger the assumptions a language makes, the more restrictive and less practical the language.

We split this criterion into two: whether a language can work over asynchronous infrastructures and if so, whether it can work with unordered message delivery.

6.1 Asynchronous Communication

Assuming synchronous communication would be so strong an assumption as to make the language impractical for MAS. Asynchronous communication, on the other hand, promotes loose coupling between agents and is also practical in that it matches real-world constraints.

Scribble, Trace, and BSPL accommodate asynchrony but HAPN (Winikoff et al., 2018, p. 61) does not.

6.2 Unordered Communication

Asynchronous communication though comes in many flavors. Asynchronous communication with some kind of ordered delivery, for example, *pairwise FIFO* (simply *FIFO*, from here on) would be a weaker assumption (and therefore better) than synchrony. FIFO in fact is supported in practical, widely-used infrastructures such as TCP and the Advanced Message Queuing Protocol, better known as AMQP (AMQP, 2014). An even weaker assumption than ordered delivery would be asynchronous communication without any kind of ordered delivery. Then, protocols in the language could be enacted directly over the Internet and in highly resource-constrained settings such the IoT.

Relying on any kind of ordered delivery guarantees from the communication infrastructure naturally limits the kinds of infrastructure upon which a protocol may be used to implement a multiagent system. More importantly, as we show below, ordering guarantees are inadequate for ensuring the correct enactment of even simple protocols.

Prima facie, there appears to be a simple motivation for using FIFO channels between agents, as illustrated by the enactments of Use Case 7.

Use Case 7 (Want+WillPay) buyer *sends Want (some item) and then WillPay (some amount) to* seller*.*

Figure 12 shows two possible enactments of Use Case 7. Notice that in Figure 12b, the messages are reordered in transit so that *WillPay* is received by seller before *Want*.

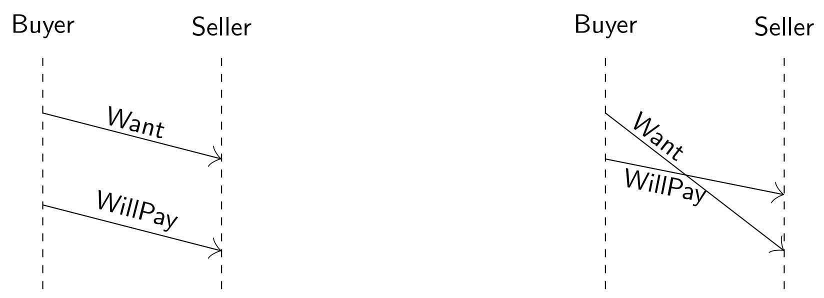
Listing 30 gives the protocol in Trace-C and Trace-F and its projections.

Listing 30: Want+WillPay in Trace-C and Trace-F along with projections.

```
//Protocol
Buyer $\xrightarrow{\text{Want}}$ Seller ; Buyer $\xrightarrow{\text{WillPay}}$ Seller

//Projections
Buyer: Seller!Want ; Seller!WillPay
Seller: Buyer?Want ; Buyer?WillPay
```

The protocol of Listing 30 cannot handle the enactment of Figure 12b because the seller's projection expects to receive *Want* before *WillPay*. The enactment of Figure 12b though is ruled out if the infrastructure guarantees FIFO delivery; only the enactment of Figure 12a is possible under FIFO. That is, in essence, there are two mutually exclusive alternatives: (1) either declare the enactment with the reordered messages to be valid and the protocol to be unrealizable; or, (2) assume FIFO channels between agents and declare the protocol to be realizable. Scribble and Trace-C take the latter alternative. Trace-F also effectively takes the latter alternative: the protocol is unrealizable under unordered asynchrony but is realizable under FIFO (with the SR, SS, and RR interpretations).

Listing 31 gives a BSPL specification for Use Case 7. The protocol ensures that buyer can send *WillPay* with some binding for ID only after *Want*; however, it does not constrain when the messages should be received by seller. Thus, it supports both enactments of Figure 12.



(a) *Want* received before *WillPay*. (b) *WillPay* received before *Want*.

Figure 12: In the absence of ordering guarantees from the infrastructure, messages could become reordered. If the infrastructure provided FIFO delivery, then the enactment of Figure 12b would not be possible.

Listing 31: Want+WillPay in BSPL.

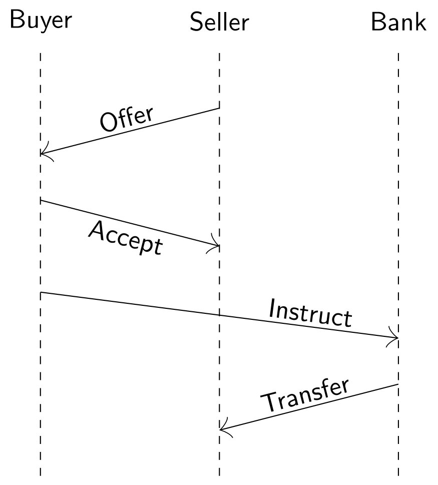
```
Want+WillPay {
 role Buyer, Seller
 parameter out ID key, out item, out price

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

Assuming FIFO to help ensure correctness seems innocuous at first glance. However, a FIFO communication infrastructure is infeasible in important application settings. For example, in the Internet of Things (IoT), many of the devices lack a capability for anything beyond packet-based communication and in particular lack the capability for buffering. Buffering is the common way to implement FIFO; another implementation approach would be to drop a message whose sequence number is not the next number to the one most recently received—but that too is not practicable since it would waste resources and exacerbate the latency of communication. In settings that demand fast interactions (e.g., for financial transactions), the additional latency due to FIFO is an avoidable overhead.

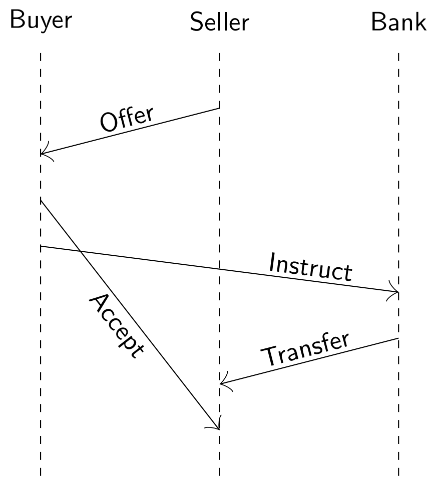
Moreover, and crucially, the FIFO assumption faces a profound semantic problem: FIFO turns out to be inadequate for correctness in settings of more than two parties, as Use Case 8 demonstrates.

Use Case 8 (Indirect payment) *In an indirect-payment purchase protocol, after receiving an Offer,* BUYER *first sends Accept to* SELLER *and then sends Instruct (a payment instruction) to* BANK. *Upon receiving Instruct,* BANK *sends a funds Transfer to* SELLER.

Figure 13 shows two enactments for Use Case 8. In Figure 13a, SELLER receives *Accept* before *Transfer* whereas in Figure 13b, SELLER receives *Accept* after *Transfer*. Both enactments satisfy FIFO since at most one message is being sent on any channel. The enactments illustrate that even with FIFO ordering, asynchrony makes ordering indeterminate for protocols involving more than two agents.



(a) In-order delivery.



(b) Out-of-order delivery.

Figure 13: FIFO communication does not guarantee consistent ordering across a multiagent system with three or more agents. Each of these enactments respects FIFO because at most one message occurs on each channel (between each pair of roles). In Figure 13b, whereas for BUYER, *Accept* occurs before *Instruct*, for SELLER, *Accept* occurs after *Transfer*, and therefore, logically, after *Instruct*.

Listing 32 is an attempt to capture Use Case 8 in Trace-C. Following the reasoning for Trace-C (Castagna et al., 2012, p. 16), this protocol is unrealizable. Specifically, the projection for SELLER expects to receive *Accept* before *Transfer* and therefore does not support the enactment in Figure 13b, which may arise despite using FIFO channels. In summary, by ruling out the protocol, Trace-C rules out realistic message orders that are simply the result of asynchrony. Listing 32 additionally serves as the specification of the protocol in Trace-F. Under FIFO, no matter what sequence interpretation is chosen, the protocol is unrealizable.

Listing 32: Indirect payment (Use Case 8) protocol and its projections in Trace-C and Trace-F.

```
//Indirect payment
Seller $\xrightarrow{\text{Offer}}$ Buyer; Buyer $\xrightarrow{\text{Accept}}$ Seller;
Buyer $\xrightarrow{\text{Instruct}}$ Bank; Bank $\xrightarrow{\text{Transfer}}$ Seller

//Projections
Buyer: Seller?Offer; Seller!Accept; Bank!Instruct
Seller: Buyer!Offer; Buyer?Accept; Bank?Transfer
Bank: Buyer?Instruct; Seller!Transfer
```

Listing 33 gives a Scribble protocol to capture Use Case 8. The Scribble projections are analogous to the Trace-C and Trace-F projections in Listing 32. In particular, SELLER cannot receive *Transfer* before *Accept*; it *blocks* on the reception of *Accept* on the channel from BUYER even when *Transfer* may have arrived earlier on the channel from BANK. The listing shows seller's projection (other agents' projections are elided). Effectively, the projection enforces a reception order for the two messages that may be different from their arrival order.

Listing 33: Indirect payment in Scribble.

```
global protocol IndirectPayment(role Buyer, role Seller, role Bank) {
  Offer() from Seller to Buyer;
  Accept() from Buyer to Seller;
  Instruct() from Buyer to Bank;
  Transfer() from Bank to Seller;
}

local protocol IndirectPayment_Seller(role Buyer, role Seller, role
    Bank) {
  Offer() to Buyer;
  Accept() from Buyer;
  Transfer() from Bank;
}
```

The BSPL specification in Listing 34 specifies a protocol that supports both of the enactments shown in Figure 13. The reason is that, in BSPL, an agent may receive a message whenever the communication infrastructure delivers a message to the agent. Information causality and integrity in BSPL constrain the emission of messages by an agent; message reception is unconstrained.

Listing 34: Indirect payment protocol in BSPL.

```
Indirect Payment {
  role Buyer, Seller, Bank
  parameter out ID key, out item, out price, out decision, out
      instruction, out OK

  Seller ↦ Buyer: Offer[out ID, out item, out price]
  Buyer ↦ Seller: Accept[in ID, in item, in price, out decision]
  Buyer ↦ Bank: Instruct[in ID, in price, in decision, out instruction]
  Bank ↦ Seller: Transfer[in ID, in price, in instruction, out OK]
}
```

We omit HAPN from this discussion since it does not support asynchrony.

7. Mapping Back to Multiagent Systems

We now map the findings from our analysis of the protocol languages to multiagent systems.

7.1 Architecture

We identify the architectural assumptions that underlie the protocol languages discussed above.

Figure 14 shows the architecture induced by Trace-C. Communication between agents is via a FIFO-based asynchronous infrastructure. Recall that Trace-F has a pluggable communication infrastructure. Figure 14 also depicts the architecture induced by Trace-F when FIFO is assumed.

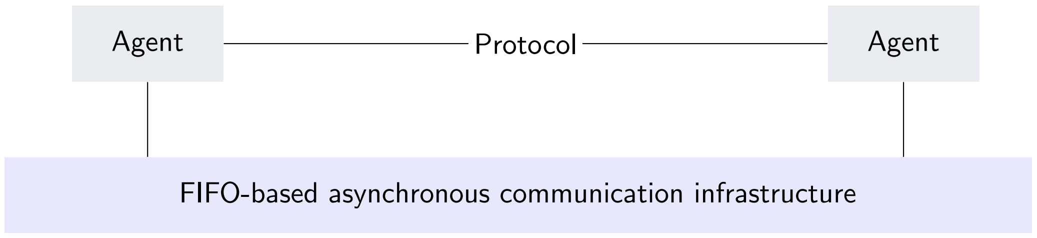


Figure 14: Architecture induced by Trace-C. Communication is asynchronous via FIFO channels. The architecture for Trace-F under FIFO-asynchrony is identical.

FIFO (delivery) though is a stronger assumption than Assumption 1, which requires only noncreativity from the infrastructure. Without FIFO, Trace-C and Trace-F would violate reception correctness (Constraint 2), which states that any reception of a message that is correctly sent is correct. The Trace-C and Trace-F *Want+WillPay* protocol (Listing 30) illustrates the violation. Even though BUYER sends *Want* before *WillPay*, as required by the protocol, SELLER receiving *WillPay* before *Want* is an incorrect enactment.

In fact, even with FIFO, both Trace-C and Trace-F violate *reception correctness* (Constraint 2). This is illustrated by the Trace-C and Trace-F *Indirect Payment* protocol (Listing 32). Asynchrony means that *Transfer* may be received by SELLER before *Accept*; however, that order of reception is incorrect according to the protocol.

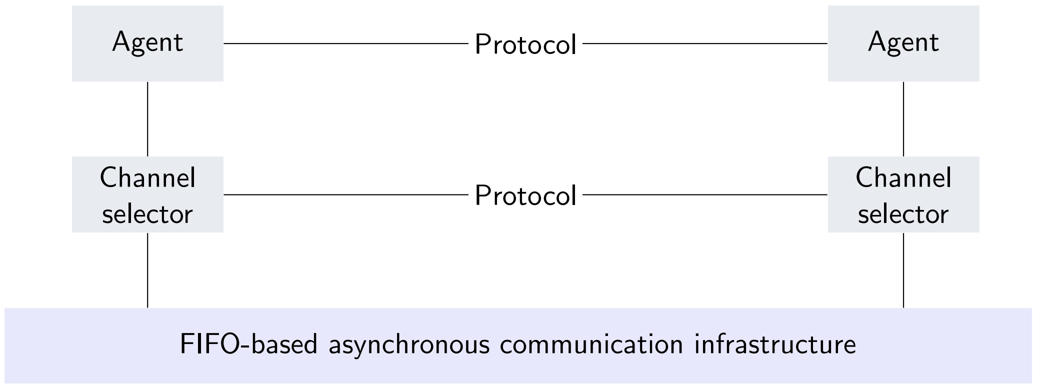


Figure 15: Architecture induced by Scribble. Communication between agents is via FIFO channels. At any time, the channel selector hides all channels from the agent except the one on which the message expected by an agent at that time will arrive.

Scribble induces the architecture in Figure 15. As with the architecture for Trace-C in Figure 14, it requires FIFO. If FIFO were to be dropped, Scribble, like Trace-C and Trace-F, would violate reception correctness (Constraint 2).

Scribble treats the reception of a message as a logically blocking operation on the channel on which the message is expected: an agent will not receive a message on any other channel until it receives the message it is blocked on. The notion of blocking reception is instrumental to Scribble's ability to handle the arrival of *Transfer* before the arrival of *Accept* in the indirect payment protocol. Specifically, SELLER blocks on the channel from BUYER, on which *Accept* is expected; in the meantime, if *Transfer* arrives on the channel from the banker, it is ignored and thereby not considered received.

In essence, Scribble reorders message arrivals to suit the agent. Such reordering is incompatible with asynchrony; specifically, it is a violation of anytime reception (Constraint 4). Scribble attempts to disguise the violation by treating the reception of a message as the result of an agent's decision to receive the message, distinct from the event of the arrival of the message. Architecturally, the reordering across channels is captured in the *channel selector* component in Figure 15 that mediates between the FIFO infrastructure and an agent and hides all channels from the agent except the one on which it is expecting to receive a message.

If the idea of reordering messages received on different channels were dropped, then even with FIFO, Scribble would violate reception correctness (Constraint 2) (just as Trace-C and Trace-F do).

By contrast, Figure 1 captures the architecture induced by BSPL. A multiagent system based on BSPL satisfies all the constraints given in Section 1. In particular, BSPL works with asynchronous communication but does not require FIFO. Given any safe BSPL protocol, any message sent according to the protocol is also received correctly regardless of when that reception occurs relative to other receptions.

We omit HAPN from this discussion because HAPN does not specify the communication assumptions under which a protocol could be correctly enacted by agents, making it difficult to imagine it architecturally.

For completeness, we comment on some recent programming and architectural frameworks for MAS. In JaCaMo (Boissier, Bordini, Hübner, Ricci, & Santi, 2013), agents coordinate their computations by shared *artifacts*—components that provide "functionalities and services" (p. 750) for agents. JaCaMo can be used to realize a MAS that satisfies the canonical architectural style of Figure 1—or any of the others for that matter. Specifically, messaging between agents could be realized using artifacts. The infrastructure can avoid centralization by, for example, using a dedicated artifact for each channel and deploying the artifacts across the system. The sender of a message would send it to the channel artifact and the receiver would pick it from that artifact. Indeed Baldoni, Baroglio, Capuzzimati, and Micalizio (2019) implement commitment-based coordination between agent by representing commitments in a shared JaCaMo artifact.

Jason (Bordini, Hübner, & Wooldridge, 2007), the language in which agents are programmed in JaCaMo, supports specifying an agent's reasoning about incoming and outgoing messages. However, Jason's programming model does not include protocols of the sort motivated here.

ReST (Vinoski, 2008) is an architectural style for Web applications that has been advocated as a basis for engineering MAS (Ciortea, Mayer, & Michahelles, 2018). An important ReST constraint is that an application's state is fully captured in the representation of the relevant Web resources (as identified by URIs) on the server. The architectural style that BSPL adopts is analogous to ReST but in a more general peer-peer setting (Singh, 2011b).

7.2 Principles for MAS

We present some broad principles that are relevant for MAS but are violated by several of the evaluated protocol languages.

Principle 1 (No unitary perspective) *A protocol should avoid specifying computations (enactments) from a unitary perspective.*

Principle 1 follows from the fact that there is no valid unitary perspective in a MAS (Gasser, 1991; Hewitt et al., 1973); the only perspectives that count are those of the agents in the MAS. Protocols that specify computations from a unitary perspective are either (1) unrealizable by agents acting based solely on their local knowledge or (2) unduly restrict concurrency.

Scribble, Trace-C, Trace-F, and HAPN violate Principle 1. In each of those languages, a protocol's computations are given from a unitary perspective in that each computation is a sequence of messaging events. Scribble, Trace-C, and Trace-F support asynchrony and each describes how to derive the local perspective, that is, the projection, for each agent. In the Scribble, Trace-C, and Trace-F approaches, extracting the projections relies on a custom (and highly complicated) theory of causality. Despite all the machinery that goes into specifying the semantics of protocols, projections, and realizability in these approaches, as we saw above, they fail to model important aspects of the simple and realistic uses cases described above.

The Scribble, Trace-C, and Trace-F protocols for *Flexible Purchase* (in Section 4.1) highlight the pitfalls of the unitary perspective. From the unitary perspective, as Listing 10 illustrates, it is a clear choice between either *Payment* first or *Shipment* first. However, when SELLER and BUYER exercise their respective choices locally, that is, on the basis of their projections, a deadlock may obtain. This leads to the protocols being ruled out as unrealizable.

Protocols in BSPL satisfies Principle 1: The computations of a BSPL protocol are given directly in terms of agent perspectives.

Any protocol is a "global" specification for a MAS in the sense that it specifies the (public) computations of the entire MAS. And every protocol conceptually yields a "local" specification, or projection, for every role in the protocol. Such a global-local distinction is perfectly reasonable and not in conflict with Principle 1. Indeed, the global-local distinction applies to BSPL as well: A BSPL protocol is a global specification in the foregoing sense and a projection for a role in the protocol is the set of all message specifications referenced by the protocol where the role is either sender or receiver. What Principle 1 rules out is giving the computations of a global protocol from a unitary perspective.

In the discussion of the following principles, Trace-F stands for Trace-F under FIFO asynchrony. We can ignore Trace-F under other communication models for two reasons. First, under one of those models, namely, unordered asynchrony, Trace-F is too weak a language—it is unable to capture a protocol as simple as Want+WillPay (Use Case 7). Second, all the remaining communication models are strictly stronger than FIFO and the conclusions we draw for Trace-F under FIFO asynchrony are valid for them as well.

Principle 2 (Noninterference) *A protocol must not prevent legitimate agent reasoning.*

An agent may want to process a message as soon as the message has arrived. The agent's motivations for doing so need not concern us since the agents are autonomous and may adopt any preferences arbitrarily in light of their autonomy. However, Scribble, Trace-C, and Trace-F, in requiring messages to be received in a certain other relative to other messages, rule out such agents and, therefore, violate Principle 2.

For example, in Use Case 7, SELLER may want to process the *WillPay* message even if *Want* has not yet been received. In Use Case 8, SELLER may want to process the *Transfer* message even if *Accept* has not been received. The Scribble, Trace-C, and Trace-F protocols rule out this possibility. BSPL, by contrast, satisfies Principle 2: the only constraints it imposes on agents have to do with information causality and information integrity. Indeed, the BSPL protocols for Use Case 7 and Use Case 8 allow SELLER to process *WillPay* before *Want* is received and *Transfer* before *Accept* is received, respectively.

Principle 3 (End-to-end principle for protocols) *Correct protocol enactment must not rely on message ordering guarantees from the communication infrastructure since the appropriate constraints are to be implemented and checked in agents.*

Scribble, Trace-C, and Trace-F violate Principle 3 because they require a FIFO communication infrastructure. BSPL satisfies Principle 3 because it requires no ordering guarantees from the infrastructure.

Principle 3 derives from the end-to-end argument for system design (Saltzer et al., 1984), which originated in the early days of computer networking in the 1960s. In its networking form, the end-to-end principle states that functionality that makes sense at a higher ("application") layer should not be replicated at a lower ("infrastructure") layer. The motivation is that if some function of a networked application can be fully and correctly implemented only at the application's end points standing above a communication infrastructure, then supporting that function partially in the infrastructure (1) is not adequate (still requires the application to work to achieve that function); (2) imposes a cost on all internal nodes; and (3) restricts the functioning of application end points that are not concerned with that function. We can think of FIFO as a "function" in this case—notably, Saltzer et al. specifically discuss the disadvantages of adopting a FIFO delivery infrastructure; such an infrastructure is also not assumed in the actor model (Agha, 1986).

A protocol can be fully and correctly implemented only by the agents who instantiate it. Relying on the infrastructure for correctness is of little benefit. For explanation, let us consider emissions and receptions, the two kinds of observations that a protocol may constrain. An agent must ensure the correctness of its emissions because emissions are driven by an agent's internal decision-making. For the correctness of receptions, an agent may rely on ordering guarantees from the infrastructure. However, such guarantees may be insufficient for correctness. For example, as we saw in the modeling of indirect payment (Use Case 8), Scribble, Trace-C, and Trace-F's reliance on FIFO turns out be insufficient for correctness.

In addition to being insufficient, infrastructure ordering guarantees may be *excessive* since they constrain even messages that are unrelated in terms of meaning but merely contingently happen to occur together. Listing 35 is illustrative of how infrastructure ordering may be excessive. The listing specifies two protocols *Just-Want* and *Hello-World*. Each protocol has a single message from BUYER to SELLER; the two messages do not have any parameters in common, signifying that there is no causal dependency between them. Even so, a FIFO infrastructure will necessarily deliver the messages to SELLER in the order in which they are sent by BUYER.

Listing 35: BSPL protocols that illustrate that ordering guarantees provided by the infrastructure may be excessive from the point of view of coordination. Although the messages in the two protocols are unrelated, if the infrastructure were FIFO, the messages would necessarily be delivered in the order sent.

```
Just−Want {
 role Buyer, Seller
 parameter out ID key, out item

 Buyer ↦ Seller: Want[out ID, out item]
}

Hello−World  {
 role Buyer, Seller
 parameter out gID key, out utterance

 Buyer ↦ Seller: Greeting[out gID, out utterance]
}
```

Finally, consider that ordering guarantees from the infrastructure come with a heavy price: increased complexity and overhead in the architecture, restrictions on the settings in which a multiagent application may be deployed, and interference with higher-level agent reasoning (as we discussed in the context of Principle 2).

8. Discussion

Our contribution in this paper is an evaluation of select modern, formal, and prominent languages for specifying protocols. Our evaluation criteria have to do with representations and operational assumptions. Our evaluation is concrete and comparative, driven by the specification of protocols in the selected languages, followed by an analysis of the specifications. The Scribble, Trace-F, and BSPL protocols have been verified in their respective tooling. We understand verification tools for Trace-C and HAPN are not available.

Table 1 summarizes our findings. For reasons given above, Trace-F stands for Trace-F under FIFO-asynchrony. Our evaluation shows that BSPL is able to model the use cases considered in all their richness despite—or because of—weaker guarantees from the communication infrastructure.

We discussed how Scribble, Trace-C, and Trace-F violate the canonical MAS architectural style presented in Section 1. Scribble, Trace-C, and Trace-F reorder messages to fit an agent's perspective. BSPL works with unordered asynchrony; the MAS architecture induced by BSPL is compatible with the canonical architectural style.

Information (as captured by parameter bindings) in BSPL are immutable. For MAS where agents communicate asynchronously, the immutability of information simplifies ensuring correctness even though the participants make observations in different orders—as expressed by Chandy and Lamport (1985).

Table 1: Summary of evaluation. The table indicates for each language, whether it fully satisfies (Yes), partially satisfies (Partial), or does not satisfy (No) each criterion.

Criterion	Scribble	Trace-C	Trace-F	HAPN	BSPL
Information					
Instances	Partial	Partial	Partial	Partial	Yes
Integrity	No	No	Partial	Partial	Yes
Social meaning	Partial	Partial	Partial	Partial	Yes
Flexibility					
Concurrency	No	No	No	No	Yes
Extensibility	No	No	No	No	Yes
Operational environment					
Asynchrony	Yes	Yes	Yes	No	Yes
Unordered delivery	No	No	No	No	Yes

The immutability of information, however, calls for a different way of specifying interactions. Specifically, to achieve the effect of updates, a BSPL protocol must incorporate a composite key such that a part of the key reflects the version ID, thereby preventing the versions from interfering with each other. To capture finality, the protocol must incorporate a message that conflicts with (and thus prevents) subsequent versions.

In this way, there is a tradeoff between (1) adopting a new way of modeling interaction to accommodate realistic communication assumptions (asynchrony), and (2) making unrealistic assumptions (synchrony) to support a familiar programming style (mutability). Immutability in BSPL contrasts most clearly with HAPN, which explicitly supports both bind and unbind operations on parameters and therefore supports rebinding a parameter.

We set out to evaluate protocol languages from the standpoint of building decentralized multiagent systems. An informally but widely held attitude in the multiagent systems research community (and in related external subcommunities) is that existing approaches are adequate for tackling autonomy and flexibility as needed in decentralized settings. Accordingly, researchers have taken support for decentralization as a done deal and gone on to tackle challenges such as tool support and ease of use—which are no doubt crucial challenges. However, as we have shown in this paper, the more fundamental challenges of decentralization have largely not been overcome by languages of the established paradigms. One language from a new, information-oriented paradigm does tackle those challenges. Whether that or another, as yet unknown, paradigm prevails remains to be seen. But what we can conclude from this exercise is that a new way of thinking is needed to properly accommodate decentralization.

8.1 Other Criteria for Evaluating Protocol Languages

The criteria by which we study protocol languages in this paper are not exhaustive. Winikoff et al. (2018) compare several protocol languages, including HAPN and BSPL, for orthogonal criteria such as precision, simplicity, graphical representation, and so on. They note that whereas HAPN supports multiple agents playing a role, BSPL does not (Splee (Chopra et al., 2017), an extension of BSPL, though supports this feature). Winikoff et al. also report on their personal experience of encoding protocols in BSPL. They observe that BSPL is "more of a core calculus than a usable notation." Recent work on meaning-based languages that compile into BSPL appears to supports the idea that BSPL represents a core calculus (Singh & Chopra, 2020). Winikoff et al. also report an extensive user study that evaluated HAPN, AUML, and statecharts for ease of reading, understanding, and writing specifications.

Ancona, Ferrando, and Mascardi (2018) include BSPL, HAPN, and trace expressions in an evaluation of MAS approaches for supporting remote patient monitoring. Although their ten criteria are diverse, a predominant thrust is tooling, e.g., IDE support, static and runtime verification, testing, code generation, and so on. In their evaluation, trace expressions offer a clear advantage over both HAPN and BSPL in supporting self-adaptation. A criterion where both trace expressions and HAPN shine over BSPL is that they come with IDE support whereas BSPL does not.

8.2 Directions

With respect to future evaluations of protocol languages, two directions (with a more contingent flavor) stand out, especially since they are informed by what would be perceived as broad strengths of the languages that BSPL outperformed in the current evaluation. One, perform a comparative study of protocol languages for the kinds of properties that can be verified for both protocols and endpoints. Notably, verification (both static and runtime) has been a primary motivation in the development of Scribble and the trace-based approaches and both benefit from deep connections with programming language theory. Two, perform a comparative study of the programming models supported by the languages in terms of how they make it easier to write correct programs. Scribble, notably, has implementations in several popular languages such as Java and Python.

A more theoretical direction would be to determine the fragment of a protocol language that can be encoded in another with the purpose of formally establishing their relative expressiveness. We hope that the evaluation presented in this paper, bearing as it does solely on representational issues, will help inform such an effort.

Acknowledgments

Raymond Hu, Viviana Mascardi, and Angelo Ferrando provided extensive and helpful commentary on earlier drafts of this paper. The (anonymous) reviewers also provided extensive and helpful comments. Ulle Endriss, as the handling editor, facilitated discussion with reviewers. We are grateful to them all.

Christie and Chopra were supported by EPSRC grant EP/N027965/1 (Turtles). Christie and Singh were partially supported by the National Science Foundation under grant IIS-1908374.

References

Agha, G. A. (1986). *Actors*. Cambridge, Massachusetts: MIT Press.

Alechina, N., Halpern, J. Y., Kash, I. A., & Logan, B. (2018). Incentive-compatible mechanisms for norm monitoring in open multi-agent systems. *Journal of Artificial*

Intelligence Research, *62*, 433–458.

AMQP. (2014). *Advanced Message Queuing Protocol.* (`http://www.amqp.org`)

Ancona, D., Ferrando, A., & Mascardi, V. (2017). Parametric runtime verification of multiagent systems. In *Proceedings of the 16th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)* (pp. 1457–1459). IFAAMAS.

Ancona, D., Ferrando, A., & Mascardi, V. (2018). Improving flexibility and dependability of remote patient monitoring with agent-oriented approaches. *International Journal on Agent-Oriented Software Engineering*, *6*(3/4), 402–442.

Artikis, A., Sergot, M. J., & Pitt, J. V. (2009, January). Specifying norm-governed computational societies. *ACM Transactions on Computational Logic*, *10*(1), 1:1–1:42.

Baldoni, M., Baroglio, C., Capuzzimati, F., & Micalizio, R. (2019). Process coordination with business artifacts and multiagent technologies. *Journal on Data Semantics*, *8*(2), 99–112.

Baldoni, M., Baroglio, C., Marengo, E., & Patti, V. (2013). Constitutive and regulative specifications of commitment protocols: A decoupled approach. *ACM Transactions on Intelligent Systems and Technologies*, *4*(2), 22:1–22:25.

Baldoni, M., Baroglio, C., Martelli, A., & Patti, V. (2006). A priori conformance verification for guaranteeing interoperability in open environments. In *Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC)* (Vol. 4294, pp. 339–351). Chicago: Springer.

Banihashemi, B., De Giacomo, G., & Lespérance, Y. (2016). Online agent supervision in the situation calculus. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 922–928).

Banihashemi, B., De Giacomo, G., & Lespérance, Y. (2018). Hierarchical agent supervision. In *Proceedings of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)* (pp. 1432–1440). IFAAMAS.

Bellifemine, F. L., Caire, G., & Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE.* Wiley-Blackwell.

Bentahar, J., Moulin, B., Meyer, J.-J. C., & Chaib-draa, B. (2004). A logical model for commitment and argument network for agent communication. In *Proceedings of the 3rd International Conference on Autonomous Agents and Multiagent Systems* (p. 792-799).

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

Bordini, R. H., Hübner, J. F., & Wooldridge, M. J. (2007). *Programming Multi-Agent Systems in AgentSpeak using Jason.* Chichester, United Kingdom: John Wiley & Sons.

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., & Mylopoulos, J. (2004, May). Tropos: An agent-oriented software development methodology. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, *8*(3), 203–236.

Castagna, G., Dezani-Ciancaglini, M., & Padovani, L. (2012, March). On global types and multi-party sessions. *Logical Methods in Computer Science*, *8*(1), 1–45.

Chandy, K. M., & Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, *3*(1), 63–75.

Chopra, A. K., Christie, S. H., & Singh, M. P. (2017, May). Splee: A declarative information-based language for multiagent interaction protocols. In *Proceedings of the 16th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)* (pp. 1054–1063). São Paulo: IFAAMAS. doi: 10.5555/3091125.3091274

Chopra, A. K., & Singh, M. P. (2009, May). Multiagent commitment alignment. In *Proceedings of the 8th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)* (pp. 937–944). Budapest: IFAAMAS. doi: 10.5555/1558109.1558143

Chopra, A. K., & Singh, M. P. (2015). Cupid: Commitments in relational algebra. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence* (pp. 2052–2059).

Chopra, A. K., & Singh, M. P. (2016). From social machines to social protocols: Software engineering foundations for sociotechnical systems. In *Proceedings of the 25th International World Wide Web Conference* (pp. 903–914). Montréal: ACM.

Ciortea, A., Mayer, S., & Michahelles, F. (2018, July). Repurposing manufacturing lines on the fly with multi-agent systems for the Web of Things. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)* (pp. 813–822). Stockholm: IFAAMAS.

Dastani, M., van der Torre, L. W. N., & Yorke-Smith, N. (2017, March). Commitments and interaction norms in organisations. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, *31*(2), 207–249. doi: 10.1007/s10458-015-9321-5

Demangeon, R., Honda, K., Hu, R., Neykova, R., & Yoshida, N. (2015). Practical interruptible conversations: Distributed dynamic verification with multiparty session types and Python. *Formal Methods in System Design*, *46*(3), 197–225.

Desai, N., & Singh, M. P. (2008a, July). On the enactability of business protocols. In *Proceedings of the 23rd Conference on Artificial Intelligence (AAAI)* (pp. 1126–1131). Chicago: AAAI Press.

Desai, N., & Singh, M. P. (2008b, July). On the enactability of business protocols. In *Proceedings of the 23rd Conference on Artificial Intelligence (AAAI)* (pp. 1126–1131). Menlo Park: AAAI Press.

d'Inverno, M., Luck, M., Noriega, P., Rodriguez-Aguilar, J. A., & Sierra, C. (2012, July). Communicating open systems. *Artificial Intelligence*, *186*, 38–94.

Fagin, R., Halpern, J. Y., Moses, Y., & Vardi, M. Y. (1995). *Reasoning About Knowledge.* Cambridge, Massachusetts: MIT Press.

Ferrando, A., Ancona, D., & Mascardi, V. (2017). Decentralizing MAS monitoring with DecAMon. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017* (pp. 239–248). Retrieved from `http://dl.acm.org/citation.cfm?id=3091164`

Ferrando, A., Winikoff, M., Cranefield, S., Dignum, F., & Mascardi, V. (2019). On enactability of agent interaction protocols: Towards a unified approach. In *Proceedings of the 7th International Workshop on Engineering Multi-agent Systems* (Vol. 12058, pp. 43–63). Springer.

FIPA. (2002). *FIPA Agent Communication Language Specifications.* (FIPA: The Foundation for Intelligent Physical Agents, `http://www.fipa.org/repository/aclspecs.html`)

FIPA. (2003). *FIPA Interaction Protocol Specifications.* (FIPA: The Foundation for Intelligent Physical Agents, `http://www.fipa.org/repository/ips.html`)

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

Gasser, L. (1991, January). Social conceptions of knowledge and action: DAI foundations and open systems semantics. *Artificial Intelligence*, *47*(1–3), 107–138.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, *8*, 231–274.

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

Hewitt, C. (1991). Open information systems semantics for distributed artificial intelligence. *Artificial Intelligence*, *47*, 79–106.

Hewitt, C., Bishop, P., & Steiger, R. (1973). A universal modular actor formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 235–245). Stanford: William Kaufmann.

Honda, K., Yoshida, N., & Carbone, M. (2016, March). Multiparty asynchronous session types. *Journal of the ACM*, *63*(1), 9:1–9:67.

ITU. (2004, April). *Message Sequence Chart (MSC).* (`http://www.itu.int/ITU-T/2005-2008/com17/languages/Z120.pdf`)

King, T. C., Günay, A., Chopra, A. K., & Singh, M. P. (2017). Tosca: Operationalizing commitments over information protocols. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 256–264).

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

Mascardi, V., & Ferrando, A. (2019, April). *Personal communication.*

Meneguzzi, F., Magnaguagno, M. C., Singh, M. P., Telang, P. R., & Yorke-Smith, N. (2018, July). Goco: Planning expressive commitment protocols. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, *32*(4), 459–502. doi: 10.1007/s10458-018-9385-0

OASIS. (2014, October). *MQTT 3.1.1 Specification Document.* (OASIS Standard previously known as the Message Queuing and Telemetry Transport; `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf`)

Odell, J., Parunak, H. V. D., & Bauer, B. (2001). Representing agent interaction protocols in UML. In *Proceedings of the 1st International Workshop on Agent-Oriented Software Engineering (AOSE 2000)* (Vol. 1957, pp. 121–140). Toronto: Springer.

Padget, J., Vos, M. D., & Page, C. A. (2018). Deontic sensors. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence* (pp. 475–481). Stockholm.

Padgham, L., & Winikoff, M. (2005). Prometheus: A practical agent-oriented methodology. In B. Henderson-Sellers & P. Giorgini (Eds.), *Agent-Oriented Methodologies* (pp. 107–135). Hershey, PA: Idea Group.

Saltzer, J. H., Reed, D. P., & Clark, D. D. (1984, November). End-to-end arguments in system design. *ACM Transactions on Computer Systems*, *2*(4), 277–288. doi: 10.1145/357401.357402

Scribble. (2018, January). *Scribble tools.* (`http://www.scribble.org`)

Shaw, M., & Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging*

Discipline. Upper Saddle River, NJ: Prentice-Hall.

Shelby, Z., Hartke, K., & Bormann, C. (2014, June). *The Constrained Application Protocol (CoAP)* (Tech. Rep. No. RFC 7252). Fremont, California: Internet Engineering Task Force (IETF). (Proposed standard; `https://tools.ietf.org/html/rfc7252`)

Singh, M. P. (1998, December). Agent communication languages: Rethinking the principles. *IEEE Computer*, *31*(12), 40–47. doi: 10.1109/2.735849

Singh, M. P. (2011a, May). 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)* (pp. 491–498). Taipei: IFAAMAS. doi: 10.5555/2031678.2031687

Singh, M. P. (2011b, July). 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)* (pp. 57–64). Washington, DC: IEEE Computer Society. doi: 10.1109/ICWS.2011.48

Singh, M. P. (2012, June). Semantics and verification of information-based protocols. In *Proceedings of the 11th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)* (pp. 1149–1156). Valencia, Spain: IFAAMAS. doi: 10.5555/2343776.2343861

Singh, M. P. (2013, December). Norms as a basis for governing sociotechnical systems. *ACM Transactions on Intelligent Systems and Technology (TIST)*, *5*(1), 21:1–21:23. doi: 10.1145/2542182.2542203

Singh, M. P., & Chopra, A. K. (2020). Clouseau: Generating communication protocols from commitments. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence* (pp. 7244–7252). New York: AAAI Press.

Singh, M. P., & Huhns, M. N. (2005). *Service-Oriented Computing: Semantics, Processes, Agents*. Chichester, United Kingdom: John Wiley & Sons. doi: 10.1002/0470091509

Telang, P. R., Singh, M. P., & Yorke-Smith, N. (2019, May). A coupled operational semantics for goals and commitments. *Journal of Artificial Intelligence Research (JAIR)*, *65*, 31–85. doi: 10.1613/jair.1.11494

Vieira, R., Moreira, Á. F., Wooldridge, M. J., & Bordini, R. H. (2007). On the formal semantics of speech-act based communication in an agent-oriented programming language. *Journal of Artificial Intelligence Research*, *29*, 221–267.

Vinoski, S. (2008, November). RESTful web services development checklist. *IEEE Internet Computing (IC)*, *12*(6), 95–96.

Winikoff, M., Liu, W., & Harland, J. (2005). Enhancing commitment machines. In *Proceedings of the 2nd International Workshop on Declarative Agent Languages and Technologies (DALT)* (Vol. 3476, pp. 198–220). Berlin: Springer.

Winikoff, M., Yadav, N., & Padgham, L. (2018, January). A new Hierarchical Agent Protocol Notation. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, *32*(1), 59–133.

XMPP. (2015). *XMPP Internet of Things.* XMPP Standards Foundation; `http://www.xmpp-iot.org/`.

Yolum, P., & Singh, M. P. (2002, July). 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)* (pp. 527–534). Bologna: ACM Press. doi: 10.1145/544862.544867

Yoshida, N., Hu, R., Neykova, R., & Ng, N. (2013, August). The Scribble protocol language. In *Proceedings of the 8th International Symposium on Trustworthy Global Computing (TGC), Revised Selected Papers* (pp. 22–41). Buenos Aires: Springer.

Zambonelli, F., Jennings, N. R., & Wooldridge, M. (2003, July). Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering Methodology*, *12*(3), 317–370.