Langshaw: Declarative Interaction Protocols Based on Sayso and Conflict

Munindar P. Singh[1] , **Samuel H. Christie V**[1] and **Amit K. Chopra**[2]

[1]Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA
[2]School of Computing and Communications, Lancaster University, Lancaster LA1 4WA, UK
mpsingh@ncsu.edu, shcv@sdf.org, amit.chopra@lancaster.ac.uk

Abstract

Current languages for specifying multiagent protocols either over-constrain protocol enactments or complicate capturing their meanings. We propose Langshaw, a declarative protocol language based on (1) *sayso*, a new construct that captures who has priority over setting each attribute, and (2) nono and nogo, two constructs to capture conflicts between actions. Langshaw combines flexibility with an information model to express meaning. We give a formal semantics for Langshaw, procedures for determining the safety and liveness of a protocol, and a method to generate a message-oriented protocol (embedding needed coordination) suitable for flexible asynchronous enactment.

1 Introduction

Our setting of interest is a *decentralized* multiagent system (MAS) [Boissier *et al.*, 2023] in which agents, each reflecting its stakeholder's autonomy, interact in a loosely coupled manner. To achieve interoperation between agents independently of their construction or reasoning presupposes that we specify their interactions at a high level [Poslad, 2007].

Classically, agents interact by sending *messages*, each a bundle of information, to one another. Achieving interoperation requires (see Section 2) specifying (1) the information each message conveys, (2) constraints on message ordering and occurrence, and (3) the meaning of each message in mental [Sadek *et al.*, 1997] or social [Singh, 2000] constructs.

However, modeling interaction via messages is problematic. First, a message is a low-level construct—the smallest operational unit of interaction. Models based on messages must contend with an explosion in possible messages and orders, and the complexity of coordination. Second, there is not always a natural mapping between messages and meanings.

In contrast to existing MAS approaches, we model communicative actions directly inspired by Austin's (1962) speech act theory, separately both from messages and meanings.

Approach. We propose *Langshaw* (after Austin's middle name), a declarative language for specifying protocols in terms of communicative actions. Langshaw enables the precise and succinct specification of protocols via novel primitives for expressing the information content and the social arrangements needed for coordination. Respecting the fundamental Austinian doctrine of *saying makes it so*, Langshaw introduces *sayso* to capture who can declare what information (i.e., "make it so"). To support flexible interactions, the saysos of agents over the same information are prioritized. Langshaw also introduces primitives to directly capture *conflict* between actions. Moreover, to support flexible interactions, Langshaw supports agents concurrently *attempting* actions. The semantics ensures the consistency of the system state without unduly compromising flexibility.

Contributions. One, we introduce the Langshaw language and give its semantics and efficient decision procedures for checking the safety and liveness of Langshaw specifications.

Two, we bridge the gap between synchronous and asynchronous communication. Assuming synchronous actions simplifies specification and reasoning. We give Langshaw a synchronous semantics. However, synchrony is an unrealistic model for MAS intended to be deployed over asynchronous infrastructures such as the Internet. We show how to compile a Langshaw protocol into a protocol that can be enacted via asynchronous messaging. Our compilation highlights Langshaw's simplicity and high-level nature: A compiled protocol may involve subtle coordination that would be difficult to write by hand. We prove the correctness of the compiler.

2 Related Work and Novelty

Protocols are crucial to MAS engineering methodologies [Cernuzzi and Zambonelli, 2004; Desai *et al.*, 2005; Rooney *et al.*, 2004; Padgham and Winikoff, 2005], but are traditionally expressed in semiformal notations such as AUML. Thus, they fall short of the goals for engineering MAS [Winikoff, 2007]. Through its precision and flexibility, Langshaw offers an opportunity to rethink MAS methodologies by capturing stakeholder intuitions and helping realize implemented MAS.

Ferrando *et al.* [2019] specify protocols as trace expressions over messages. They study diverse communication models, ranging from fully synchronous to fully asynchronous, but do not give a method to translate from one model to another, as we do. Moreover, as is common, Ferrando *et al.* express the information content implicitly and coordination explicitly, which limits flexibility. Ferrando *et al.* [2017] devise techniques for the runtime verification of protocols; here, we focus on static verification.

BSPL (Blindingly Simple Protocol Language) [Singh, 2011a,b, 2012] and HAPN (Hierarchical Agent Protocol Notation) [Winikoff *et al.*, 2018] mix information content with coordination, gaining flexibility and concurrency at the cost of unwieldiness. HAPN's state-machine model is compatible with our synchronous semantics. BSPL supports flexible asynchronous enactments, and we show how to compile Langshaw into BSPL to benefit from asynchrony.

Baldoni *et al.*'s (2014) 2CL specifies social meaning via commitments and temporal and relational constraints. 2CL does not model information or support asynchrony. Langshaw supports encoding social meaning and thus helps support accountability [Baldoni *et al.*, 2023]. Its model enhances those of Custard [Chopra and Singh, 2016] and Clouseau [Singh and Chopra, 2020] via primitives to capture coordination.

CArtAgO [Ricci *et al.*, 2009] undergirds multiagent programming frameworks such as JaCaMo [Boissier *et al.*, 2013]. CArtAgO agents coordinate via tuple spaces [Carriero and Gelernter, 1989]. MAS programming models such as JaCaMo, Jason [Vieira *et al.*, 2007], and Jade [Bellifemine *et al.*, 2007; Bergenti *et al.*, 2020] lack support for protocols. Ricci *et al.* [2019] argue for decentralization and scalability by combining MAS with the Web, but their interaction model is limited to HTTP (as on the Web). Langshaw give an abstract model for richer interactions, supporting asynchrony while based on a conceptually central social state (Section 3).

Traditional approaches rely on a "synchronizer" in the infrastructure to decide which actions occurred in what order. For example, in Weyns and Holvoet's (2004) model, when agents perform conflicting actions simultaneously, the environment picks the successful ones. Such a synchronizer is not a social entity. It may avoid an integrity violation by forcing an ordering, but only arbitrarily. In Langshaw, such decisions are handled socially—a good protocol would have the right saysos to avoid the mishaps of concurrency.

3 Langshaw Syntax and Informal Semantics

We first conceive a Langshaw-based multiagent system as operating *one* social artifact, the locus of its social state. Synchronous (and concurrent) social actions by (agents playing) roles in the system update this artifact. These updates respect local consistency (only different roles make take conflicting actions) and causality (an action may occur in a state only if the the information it relies on is present there). The social state is nothing but the set of such (performed) actions. The purpose of a Langshaw protocol is to specify such a system. Enacting the protocol means updating the social state.

Listing 1 illustrates Langshaw's syntax (Table 1); we use it to explain how a protocol is enacted. As shown, the *Purchase* protocol specifies roles SELLER, BUYER, and SHIPPER.

Lines 4:–10: specify social actions, each to be performed by a role; each action specifies one or more attributes (to capture its meaning), including one or more key attributes to uniquify instances. For example, BUYER may attempt the action *RFQ* by providing bindings (values) for ID and item and SELLER may attempt *Quote* by providing bindings for ID, item, and price. Each *Action* is reified in an implicit *action* attribute, which is bound if and only if that action has been instantiated for the specified key bindings. When an action includes such an attribute, it indicates a reliance on the first action. For example, *Reject* applies to an instance of *Quote*. *Accept* could include *Quote* but item and price make its meaning clearer.

Attributes are defined only in reference to their keys: it makes no sense to talk of an item without its ID. Therefore, we assume that if an attribute occurs in two actions, their keys overlap and their intersection uniquely determines that attribute. For a key attribute, a role may either generate a fresh binding or reuse a previous binding from the social state. For any other attribute, if it is bound in the social state relative to a key binding, a role must use the same binding relative to that key. For example, in a social state with *RFQ* with some ID and item, a *Quote* or an *Accept* with the same ID must

Protocol	$\longrightarrow$	Name Roles Attrs Dos Saysos $\lfloor$Nogos$\rfloor$ $\lfloor$Nonos$\rfloor$
Roles	$\longrightarrow$	who $\mathscr{R}^+$
Attrs	$\longrightarrow$	what Info
Info	$\longrightarrow$	$[\mathscr{A}$ key$]^+$ [Expr]*
Expr	$\longrightarrow$	$\mathscr{A}$ | Expr and Expr | Expr or Expr
Dos	$\longrightarrow$	do Action$^+$
Action	$\longrightarrow$	$\mathscr{R}$: $[\mathscr{B}$(Info)$]^+$
Saysos	$\longrightarrow$	sayso [Ranking : $\mathscr{A}]^+$
Ranking	$\longrightarrow$	$\mathscr{R} \mid \mathscr{R}$ > Ranking
Nogos	$\longrightarrow$	nogo [Action $\not\rightarrow$ Action]$^+$
Nonos	$\longrightarrow$	nono [Action Action]$^+$

Table 1: Langshaw syntax. $\longrightarrow$, |, $*$, $+$ indicate production, choice, zero or more repetitions, and one or more repetitions, respectively. [] and $\lfloor\ \rfloor$ indicate grouping and optionality, respectively. $\mathscr{A}$, $\mathscr{B}$, and $\mathscr{R}$ are, respectively, sets of (terminal) attributes, actions, and roles.

Listing 1 Purchase in Langshaw.

```
1: Purchase   // Name of the protocol
2:  who  Buyer, Seller, Shipper // roles
3:  what  ID key, Reject or Deliver //
        Completion
4:  do  // How to change the social state
5:    Buyer: RFQ(ID, item)
6:    Seller: Quote(ID, item, price)
7:    Buyer: Accept(ID, item, price,
        address)
8:    Buyer: Reject(ID, Quote)
9:    Seller: Instruct(ID, Accept, item,
        address, fee)
10:   Shipper: Deliver(ID, Instruct,
        item, address)
11: sayso  // Social authority over what
12:   Buyer > Seller: item
13:   Seller > Buyer: price
14:   Buyer: address
15:   Seller: fee
16: nono  // What pairs are incompatible
17:     Accept Reject
18:     Reject Deliver
19: nogo  // What actions prevent another
20:     Reject -/> Instruct
```

contain the same item. If no such binding exists, the role may generate a fresh binding for that attribute only if it has *sayso* over that attribute, as Line 11: and those following illustrate. For example, BUYER and SELLER can both generate item.

Attempts may be concurrent. When an agent attempts multiple actions, their bindings must be consistent with each other and the social state. For example, BUYER may concurrently attempt several *RFQ*s, each with a distinct binding for ID. When multiple agents attempt actions concurrently, the agents need not be consistent with each other, just within each agent's actions and between the agent and the social state. For example, BUYER and SELLER may concurrently attempt an *RFQ* and *Quote* by generating the same binding for ID but different bindings for item (since they both have sayso over item). In Langshaw, the saysos over the same attribute must be prioritized over agents (indicated by $>$): here, by Line 12:, BUYER's sayso (being ranked higher) *dominates*. Consequently, its attempt succeeds (updates the social state) whereas SELLER's attempt fails (is a noop).

If only one agent attempts any actions, then all its attempts succeed and become part of the social state. If two or more agents attempt actions, exactly those of their collected attempts succeed where for each attribute not already bound in the social state, the attempting role has the highest sayso of all the roles concurrently attempting to produce a binding for that attribute. When attempts dominate each other (on different attributes), neither affects the social state. For example, if BUYER and SELLER concurrently attempt *Accept* and *Quote* by each generating item and price for the same ID, both attempts fail because BUYER's sayso dominates for item whereas SELLER's sayso dominates for price.

Line 16: and those following specify pairwise conflicts between actions. *Accept* and *Reject* conflict, meaning that they are mutually exclusive for the same bindings of ID. But since they are both actions of BUYER, it can choose either one.

Line 19: and those following specify an asymmetric conflict. Once *Reject* is in the social state, *Instruct* may not be performed. Importantly, *Instruct* may precede or occur simultaneously with *Reject*. This constraint is not needed in *Purchase* but we insert it to explain the construct.

Consider *safety*. Concurrent conflicting attempts by two agents can succeed, resulting in an inconsistent social state. A *safe* protocol prevents such attempts. *Purchase* is safe. *Accept* and *Reject* are both actions of BUYER, which means that if BUYER does one, it cannot do the other. *Reject* and *Deliver* are actions of BUYER and SHIPPER, respectively, but there is no social state in which both can be attempted concurrently. To do *Deliver*, SHIPPER needs to know the binding address, which can only happen if BUYER does *Accept*, which rules out *Reject* because it would be a local conflict at BUYER.

Let protocol *Unsafe Purchase* be obtained from Listing 1 by omitting Line 17:: thus, both *Accept* and *Reject* may occur. Suppose *Accept* has occurred, followed by *Instruct*. Now the social state is such that *Reject* and *Deliver* may both be attempted concurrently by BUYER and SHIPPER, respectively. Both BUYER and SHIPPER will succeed, thus violating the specified nono between *Reject* and *Deliver*.

Consider *liveness*. Line 3: gives a *completion* criterion for *Purchase*'s enactments as a list of attributes or disjunctions of attributes; one or more of which are designated ⌜key⌝ and distinguish enactments. Specifically, it says that an enactment of *Purchase*, as identified by a binding for ID, is complete when a binding exists for either *Reject* or *Deliver*.

A protocol violates liveness if at least one of its enactments fails to complete. *Purchase* has two alternative branches for an enactment, one ending with *Reject* and the other ending with *Deliver*. Thus, on each branch, *Purchase*'s completion criterion is satisfied, which means *Purchase* is live. Nonliveness could result from too little sayso or where the saysos induce a cyclic information dependency between the actions.

4 Formal Semantics

Semantic tableaux [Fitting, 1999] are a computational representation for proofs. Each node is a *social* state of the system, and the transitions are the successful concurrent actions attempted. Each tableau captures all possible enactments of a protocol: one per *branch* beginning from the root. Our semantics (Figure 1) is framed as inference rules to derive all possible transitions. Section 5 gives heuristics to produce a small tableau, i.e., a tractable model of a protocol.

Figure 1 lists the inference rules for our semantics, based on propositional logic. Below, m is an instance $x\colon m[\vec{a}]$ where $\vec{a}$ is its entire set of attributes. We write $x\colon m[\vec{k},\vec{p}]$ or $m[\vec{k},\vec{p}]$ to mean role x performs m with attributes $\vec{p}$ and key attributes $\vec{k} \subseteq \vec{p}$. We omit the role where it is clear. S captures the social state. $\mathsf{Y}_{x,y}\ a$ means x has higher sayso over a than y (both x and y have sayso). $\mathsf{Y}_x\ a$ means x has sayso over a of whatever priority. Braces $\{\ \}$ capture sets. Subscripts on sets and operators capture the obvious indices and ranges. An instance $m_i[\vec{k_i},\vec{p_i}]$ fits $m[\vec{k},\vec{p}]$ if and only if their common key attributes have the same bindings, i.e., $m_i[\vec{k_i}\cap\vec{k}] = m[\vec{k_i}\cap\vec{k}]$.

In Z-SOCIAL, Z captures what bindings can be inferred from the social state relative to a (potential or actual) message instance. For an instance $m[\vec{k},\vec{p}]$, the bindings of its attributes occurring in a fitting instance in the present social state can be inferred (i.e., are known). Z-SUBSET and Z-UNION state that Z bindings are closed under subset and union.

In ATTEMPT, an instance is *attemptable* by x if (1) it includes any bindings already established (for the key) in the social state and (2) x has sayso over the remaining attributes.

In ABIDE, an instance *abides* by another if: if the bindings of their common key attributes agree, then the bindings of all their common attributes also agree. Any nono and nogo constraints are ignored here and captured in FEASIBLE.

UNSOCIAL-M states that an instance is *unsocial* (i.e., socially inconsistent) if there is an instance in the social state with which it does not abide. UNSOCIAL-N states that an instance is *unsocial* if an instance in the social state has a nono or a nogo (asymmetric) constraint toward that instance.

FEASIBLE-1 states that an instance is *feasible* if (1) it is attemptable and (2) its attribute bindings are not inconsistent with the social state. In FEASIBLE, a set of instances is *feasible* if each instance is feasible and for each pair of instances performed by the same role, they are not related by a nono and each instance in a pair abides by the other. This is a crucial design decision in Langshaw: Feasibility avoids only local

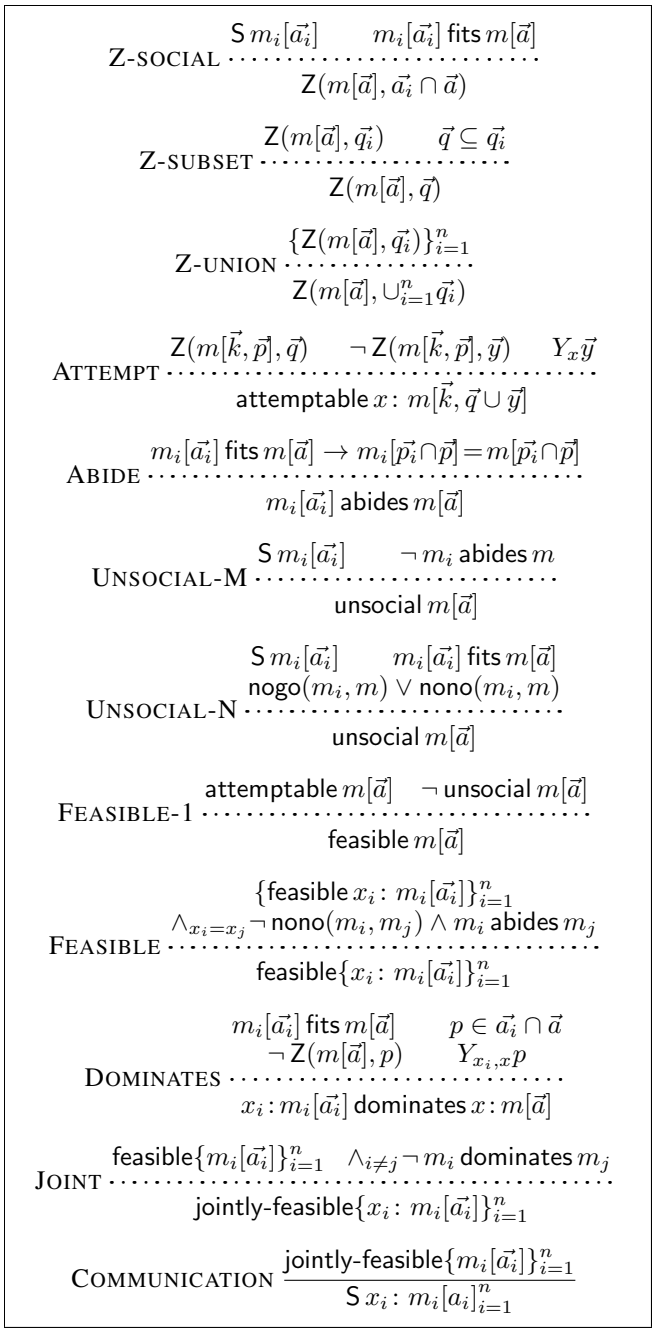

Figure 1: Synchronous semantics for Langshaw.

incompatibilities. A protocol may be incorrect (e.g., unsafe) because of nono conflicts across roles. Since such errors cannot be avoided in a decentralized architecture, proper saysos are essential in a protocol that remains correct in asynchrony.

DOMINATES captures that m_i *dominates* m if they have the same bindings for their common key attributes, a common *unbound* attribute p, and x_i has sayso priority on p over x. (Thus, m_i and m can dominate each other.) Dominance goes away once the conflicting sayso attribute (y above) is bound.

JOINT states that a feasible set of instances whose members do not dominate another is *jointly feasible*. If m_i and m dominate each other, at most one can be part of a jointly feasible set. Note that joint feasibility is closed under subsets.

4.1 Generating Enactments from a Tableau

The preceding rules (dotted lines) show inference within a single social state: no S assertions are inferred, hence no state change. COMMUNICATION (solid line) shows the progress of time. A set of jointly feasible actions may be concurrently performed and the resulting social state records each of the actions as having occurred via S assertions.

A tableau begins from a root node—no S assertion. Any action there relies entirely on its performer's sayso. Under COMMUNICATION, each jointly feasible set forks a tableau branch. Where a branch terminates (i.e., at a state with no more feasible actions) forms a unique history.

4.2 Verifying Correctness Properties

Properties of interest, e.g., liveness and safety, are concerned with the reachability or otherwise of a state through an enactment. These properties are expressed via propositional combinations of the actions (and attributes). We assert a property at the root. A consistent branch that ends provides an example of the property at the root. A branch that hits a contradiction is *closed*; a tableau closes if all its branches close, which indicates the property is inconsistent and its negation is proved.

For liveness, we derive a formula from its what line—for the simple case, this means each protocol attribute becomes bound. To this end, we expand the social state to include bound attributes. For example, from S *Deliver* we can infer S *ID* and S *item*. A liveness violation occurs precisely when at least one of the attributes remains unbound. For example, in Listing 1, $\neg$ S *ID* $\vee$ ($\neg$ S *Reject* $\wedge \neg$ S *Deliver*). A consistent branch is a counterexample since our formula is negated. But if every branch closes, liveness is *established*.

For safety, an integrity violation is when two actions of different performers with a nono constraint occur. In Listing 1, *Reject* and *Deliver* are such actions. A safety violation occurs for any such pair of actions. The negation of this property means that safety is preserved. We place the negated property, $\neg$ S *Reject* $\vee \neg$ S *Deliver*, at the root of the tableau. If any branch of the tableau closes, i.e., runs into a contradiction, we determine that safety is *violated*.

Theorem 1. *Liveness and safety properties based as above on the* what *and* nono *parts of a protocol are verified by negating and checking for contradiction.*

Argument. Follows from the construction of the tableau.

5 Reducing the Tableau

A tableau produced naively would include a branch for every jointly feasible set of actions at a state (node), leading to an exponential explosion of mostly redundant information. If we take the branches as individual actions, we will end up unrolling every feasible interleaving of the actions, also leading to an exponential explosion. How can we effectively reduce that redundancy? Our idea is to consolidate branches where possible and retain enough (heuristically, typically a few) branches to cover all semantically distinct possibilities.

5.1 How Actions Relate in a Protocol

Three cases are relevant. First, mutually *unrelated* pairs of actions may occur in any combination or order. We can discard all but one branch in which they are performed: a single-shot concurrent performance would lead to a shallower tableau, but any arbitrary interleaving would produce the same social state. Second, where one action *enables* another, the tableau would simply unfold such that an action would not become attemptable unless the right S assertions were present.

Third, where two actions may interfere with each other, we must include enough branches in the tableau so that each alternative is included. We capture potential interference as *hampers* below. If an action m_1 may disable m_2 (as based on a nono or a nogo) or prevent its concurrent performance (as based on sayso priority), we must include both possibilities with m_1 or m_2 going first and the other action being disabled or allowed later. We must recognize such cases even if one of the actions is not presently enabled and make sure we do not prematurely eliminate it. Figure 2 formalizes this reasoning.

In ENABLE, action $x_i\colon m_i$ *enables* action $x\colon m$ if m_i is a potential precursor to m. That is, x lacks sayso on some attribute p of m that is not established from the social state (i.e., $\neg\,\mathsf{Z}(m[\vec{a}], p)$) but x_i has sayso on p and p appears in m_i. Action m_i *enables* action m if there is a chain of one or more direct enablements from m_i to m. CHAIN expresses that enablement is transitively closed.

Action m_i may *hamper* action m in two ways. The first case is if there is a nono or a nogo assertion involving m_i and m. *Accept* and *Reject* and *Reject* and *Deliver* are examples. BLOCK captures this case. The second case is when x_i has priority over x for an attribute p that is not yet established in the social state, exactly as specified by dominates. *Quote* and *Accept* is an example pair. DELAY captures this case.

FUTURE extends *hampers* to accommodate future (potential) interference. Action m_i *hampers* action m if m_i hampers action m_j and m enables m_j.

5.2 Computing Jointly Feasible Sets of Actions

Identify all feasible actions at a state. Next, group these actions into jointly feasible sets, reflecting two intuitions.

First, [Colors] when an action hampers another, the tableau must include both orders and maximally group nonhampering actions to reduce its size. To this end, create a graph whose vertices are the feasible actions and an edge indicates one hampers another. A *coloring* [Brélaz, 1979] finds sets of vertices with no edge between them. Each set is jointly feasible (no action in a set hampers another), but maximality maps to minimum graph coloring, which is NP-Hard. We apply Brélaz's polynomial-time approximation to compute Colors.

Second, [Concs] when an action hampers another, the two can be performed concurrently if they have different performers, there is a nono between them (nogos are OK), and neither performer dominates the other on some unbound attribute that is common to the actions. This gives the set Concs.

We construct a *reduced tableau* by including branches whose action sets are in Colors $\cup$ Concs.

Theorem 2. *A reduced tableau is closed for a property constructed from Boolean combinations of* S *assertions if and only if the full tableau is closed for that property.*

$$\textsc{Enable}\ \frac{m_i[\vec{a_i}]\ \mathsf{fits}\ m[\vec{a}] \qquad \neg\,\mathsf{Z}(m[\vec{a}], p) \qquad \mathsf{Y}_{x_i}\,p \qquad \neg\mathsf{Y}_x\,p \qquad p \in \vec{p_i} \cap \vec{p}}{x_i\colon m_i[\vec{a_i}]\ \mathsf{enables}\ x\colon m[\vec{a}]}$$

$$\textsc{Chain}\ \frac{m_i[\vec{a_i}]\ \mathsf{enables}\ m[\vec{a}] \qquad m[\vec{a}]\ \mathsf{enables}\ m_j[\vec{a_j}]}{m_i[\vec{a_i}]\ \mathsf{enables}\ m_j[\vec{a_j}]}$$

$$\textsc{Block}\ \frac{m_i[\vec{a_i}]\ \mathsf{fits}\ m[\vec{a}] \qquad \mathsf{nono}(m_i, m) \vee \mathsf{nogo}(m_i, m)}{x_i\colon m_i[\vec{a_i}]\ \mathsf{hampers}\ x\colon m[\vec{a}]}$$

$$\textsc{Delay}\ \frac{x_i\colon m_i[\vec{a_i}]\ \mathsf{dominates}\ x\colon m[\vec{a}]}{x_i\colon m_i[\vec{a_i}]\ \mathsf{hampers}\ x\colon m[\vec{a}]}$$

$$\textsc{Future}\ \frac{m_i[\vec{a_i}]\ \mathsf{hampers}\ m_j[\vec{a_j}] \qquad m[\vec{a}]\ \mathsf{enables}\ m_j[\vec{a_j}]}{m_i[\vec{a_i}]\ \mathsf{hampers}\ m[\vec{a}]}$$

Figure 2: Reasoning about enablement and interference.

Argument. **If:** All branches in a reduced tableau are jointly feasible and occur in the full tableau. **Only if:** A branch closes based on the S assertions on it. Since the nogo and sayso constraints are incorporated into the tableau construction of Figure 1, the only closures (contradictions) observed on a branch pertain to nono constraints. Suppose the reduced tableau lacks an equivalent branch. By Concs, it must include branches for the nono constraints, let $\mathsf{S}\,m$ be the first assertion that it does not include. Therefore, it must include an action $\mathsf{S}\,m'$ that disables $\mathsf{S}\,m$. By our coloring construction, that means $\mathsf{S}\,m$ should have been in a different color from $\mathsf{S}\,m'$. Therefore, there is a branch including $\mathsf{S}\,m$ on which no $\mathsf{S}\,m'$ precedes it. Hence, we have a contradiction.

Theorem 2 is important because it means all verification of properties based on S (including safety and liveness, as seen in Section 4.2) can be carried out on a reduced tableau.

6 Compilation into Asynchronous Protocols

For concreteness, we adopt BSPL for specifying asynchronous messaging protocols. As Listing 2 shows, a BSPL protocol lists roles, a completion criterion, and one or more messages. Each message specifies information dependencies via adornments $\ulcorner\mathsf{in}\urcorner$, $\ulcorner\mathsf{out}\urcorner$, and $\ulcorner\mathsf{nil}\urcorner$. Each role has a local state and may send a message only if it is compatible with its local state. Specifically, let m be a message schema $x \mapsto y\colon m[\vec{p}_I, \vec{p}_O, \vec{p}_N]$, where $\vec{p}_I, \vec{p}_O, \vec{p}_N$ are sets of its parameters adorned $\ulcorner\mathsf{in}\urcorner$, $\ulcorner\mathsf{out}\urcorner$, and $\ulcorner\mathsf{nil}\urcorner$, respectively. An instance of m is a message that has bindings only for the $\ulcorner\mathsf{in}\urcorner$ and $\ulcorner\mathsf{out}\urcorner$ parameters; x may send an instance of m only if the bindings of the $\ulcorner\mathsf{in}\urcorner$ parameters are known (already present in its local state), bindings of the $\ulcorner\mathsf{out}\urcorner$ parameters are unknown (but added to the local state upon sending), bindings of the $\ulcorner\mathsf{nil}\urcorner$ parameters are unknown (and not added to the local state). For example, to send an *Accept*, BUYER must know ID and item and not know price. Moreover, *Accept* and *Reject*

Listing 2 A simple protocol to explain BSPL constructs.

```
1: Accept-Reject //Protocol name
2:  roles B, S //roles Buyer and Seller
3:  parameters out ID key, out done
       //completion criterion
4:  S ↦ B: Offer[out ID key, out item, out
       price] // Message
5:  B ↦ S: Accept[in ID key, in item, in
       price, out done]
6:  B ↦ S: Reject[in ID key, in item, in
       price, out done]
```

$$\textsc{Send}\ \frac{K_x\vec{p}_I \qquad \neg K_x\vec{p}_O \qquad \neg K_x\vec{p}_N}{L_x(x \mapsto y\colon m[\vec{p}_I, \vec{p}_O, \vec{p}_N]) \qquad K_x\vec{p}_O}$$

$$\textsc{Receive}\ \frac{L_x(r = x \mapsto y\colon m[\vec{p}_I, \vec{p}_O, \vec{p}_N])}{L_y r \qquad K_y\vec{p}_I \qquad K_y\vec{p}_O}$$

Figure 3: Asynchronous semantics for BSPL.

are mutually exclusive because each includes done as $\ulcorner$out$\urcorner$.

Figure 3 shows Singh and Christie's (2021) tableau-based asynchronous semantics of BSPL. Each tableau branch (sequence of message send and receive events) is an enactment. K_x captures that x knows specified parameter bindings. L_x means x sent or received a message, simulating a channel. Initially, each role knows no bindings. What is known to each role grows monotonically: the bindings are immutable, and each message sent and received adds to a role's knowledge.

In SEND, an instance of $x \mapsto y\colon m[\vec{p}_I, \vec{p}_O, \vec{p}_N]$ is enabled for emission by x if and only if x knows $\vec{p}_I$ but does not know $\vec{p}_O$ or $\vec{p}_N$. Concomitantly with the emission, the sender produces and comes to know the bindings for $\vec{p}_O$. In RECEIVE, a message from x to y is enabled for reception by y if and only if x has (previously) sent that message. Concomitantly with reception, y comes to know the bindings for $\vec{p}_I$ and $\vec{p}_O$.

Each tableau branch takes up one of the allowed observations (emission or reception of a message) and maps to an enactment. A tableau contains all possible observation orders.

6.1 Compiling Langshaw

We now describe how to compile Langshaw specifications into BSPL in a manner that respects the demands of asynchrony. Langshaw roles map to BSPL roles, attributes to parameters, actions to messages, saysos to data flows, and nonos to integrity constraints. Resolving the saysos and nonos requires parameters and messages not present in Langshaw.

Completion Requirements. Langshaw completion requirements are a sequence of disjunctive clauses. Each attribute (atom in a clause) yields a BSPL parameter of the same name. Each disjunction is replaced by a parameter representing the achievement of any of its terms. When generating protocol messages, we append messages that are enabled when any of the clause terms is completed to notify each role that the clause is satisfied.

Listing 3 (full version online; URL below) shows some lines generated from *Purchase*. The what line yields two

Listing 3 Purchase: Completion requirements.

```
1:  parameters out ID key, out done0
2:  ...
3:  Buyer ↦ Seller: Reject#done0[in ID key
       , in Reject, out done0]
```

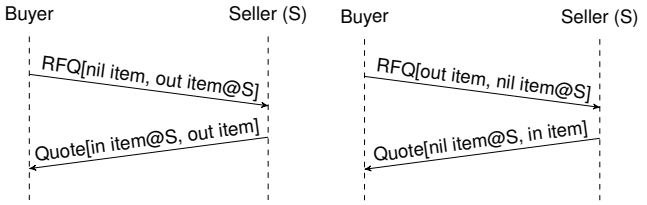

Figure 4: Sayso delegation pattern showing alternative enactments.

clauses. ID maps to BSPL parameter ID. And, Reject or Deliver maps to done0. Each role gets a message for each of its actions. We also create messages to convey important facts to other roles: e.g., *Reject* maps to messages indicating that the primary *Reject* has been sent and the what line satisfied.

Generating Messages from Actions. Each action yields a BSPL message schema, whose sender is the performer and whose receiver is a role that needs the content of the message.

- Each attribute (including the attributes reifying the actions) becomes a message parameter.
- The adornment of each parameter depends on whether the action's role (message's sender) has sayso.
- Each combination of parameter adornments yields a different *morph*, i.e., message schema variant.
- The above possibilities are constrained via a series of filters to remove incorrect and redundant combinations.

Modeling Sayso in BSPL. We model sayso in BSPL via *delegation*. That is, since BUYER has priority over SELLER to bind item, it must either bind item, or bind a newly added *delegation parameter* (item@Seller) to empower SELLER to bind it. Conversely, SELLER cannot bind item until BUYER has delegated that authority. Figure 4 illustrates the delegation pattern with message sequence diagrams. The pattern is inherently asymmetric: traditionally the priorities are arbitrary [Mattern, 1990] whereas our priorities have a social basis.

Filtering Out Improper Schemas. The above steps produce redundant or incorrect morphs. We apply a series of filters to exclude such cases. For instance, reasoning about causality is needed to ensure enactability and reasoning about priority to avoid race conditions.

Conflicts and Data Flow. To generate BSPL messages, we model a nono constraint by adding $\ulcorner$nil$\urcorner$ parameters to each of the conflicting messages. For example, in *Purchase*, the *Reject* and *Deliver* actions conflict, so we add nil `Deliver` to each generated *Reject* morph, and nil `Reject` to each generated *Deliver* morph, as shown in Listing 4. Likewise, we model a nogo constraint ($a \not\rightarrow b$) by placing a $\ulcorner$nil$\urcorner$ parameter on the message for the disabled action, i.e., $b[\ldots \ulcorner\text{nil}\urcorner\, a]$. This approach echoes the idea of disabling an action until information is received to proceed [Singh, 1996].

Listing 4 Realizing the Reject and Deliver conflict.

```
1: B ↦ S: Reject[in ID key, in Quote, out
       Reject, nil Accept, nil Deliver]
2: Sh ↦ S: Deliver[in ID key, in
       Instruct, in item, in address, out
       Deliver, nil Reject]
```

Next, we model the protocol's data flow by specifying the recipients of each message schema. A role can observe any attribute (including an action) that features in an action it can perform and any attribute present in the what line of the protocol. Information is shared at the granularity of actions, not attributes piecemeal, so we determine which roles can see what action. We generate a separate message schema for each of the desired data flows. Since BSPL does not support multicast, we emulate any multicast with separate messages. The first message schema (to one role) has the relevant parameters as ⌜out⌝; message schemas to other roles include the same parameters as ⌜in⌝. As described above, every message is given a least one ⌜out⌝ parameter.

Theorem 3. *For any asynchronous messaging enactment produced by a BSPL protocol generated through our method, there is a corresponding synchronous enactment produced by the Langshaw semantics.*

Argument. We derive messages using Langshaw's semantics. Each action maps to a message and each attribute to a parameter with the constraints (nono, nogo, and sayso dominance) via crafted ⌜nil⌝ parameters. We filter the schemas according to rules derived from the delegation model of sayso, i.e., constraining the available actions according to DOMINATES. Thus, the generated BSPL protocol allows moves corresponding to any Langshaw action. Due to our filters, it cannot make any moves prevented by the sayso constraints. Although Langshaw allows concurrent actions and BSPL does not allow multicast, the effect of multicast is achieved through additional messages: such messages can be delayed, as allowed by the BSPL's asynchronous semantics, but not disabled. We capture Langshaw's disjunctive completion requirements via messages producing a designated parameter.

6.2 Empirical Results

We implemented our verifier for Langshaw protocols in Python. Table 2 shows the performance (averaged over 10 runs) of testing liveness (including the time for constructing a tableau) for several protocols from the literature. The times for safety are similar, though slightly faster in most cases.

The results show that the Langshaw verifier is effective. The node and branch numbers are sometimes fractions, due to unordered sets in the implementation randomizing the selection and thus producing different numbers of nodes and branches in different runs of the verifier.

Protocol	Nodes	Branches	Time (ms)
Redelegation	4	1	3.5
Unsafe Purchase	49.5	19.9	901.5
PO-. . . -Ship	12.3	2.4	92.6
Either-Offer	3	1	1.5
Refund	12.2	4.6	99.0
Purchase	28.2	8.1	480.1
Unsafe	5	3	3.2
Block-Contra	3	1	0.8
Nonlive	1	1	0.2
CompositeKey	3	1	2.5
RFQ-Quote	6	2	6.5
Rescind	7.5	2.5	23.7
Block-Contra v2	17	8	25.8

Table 2: Statistics of Langshaw protocol liveness verification. *Purchase*, *Unsafe Purchase*, and *Nonlive* are as specified above. The remaining protocols are in the online supplement (URL below). Time to verify is given in milliseconds. All except *Unsafe Purchase* and *Unsafe* were safe; all except *Nonlive* were live. Our experimental rig was an ASUS Zenbook S13 with an AMD Ryzen 7 6800U CPU and 16GB of LPDDR5 RAM, running Linux.

7 Discussion: Conclusion and Perspectives

Specifying coordination constraints for MAS can be nontrivial. Langshaw's abstractions force a designer to think of coordination in social terms. The synchronous semantics provides a simplified model for a MAS engineer while maintaining the realism and power of asynchrony in that conflicting action attempts by multiple agents may succeed though they violate a nono constraint. Thus, it does not obviate correctness concerns arising in asynchrony and thereby facilitates translation to an asynchronous protocol.

For now, the best approach to implement agents to participate in Langshaw protocols is to (1) verify a Langshaw protocol; (2) compile it to BSPL; and (3) apply a BSPL programming model such as Kiko [Christie *et al.*, 2023]. Research into native programming models for Langshaw is needed. Both BDI-based and the newer hypermedia-based programming models [Vachtsevanou *et al.*, 2023] are promising as direct programming models for Langshaw because they naturally complement Langshaw's information orientation.

The engineering MAS community has long debated the relative merits of synchrony and asynchrony [Singh, 1999]. Langshaw bypasses some of the debate by showing that a synchronous semantics can be a pathway to asynchrony. Popular approaches provide constructs such as artifacts [Ricci *et al.*, 2009] or the environment [Weyns *et al.*, 2007] that provide a unitary view of the state of a MAS. Though they may be accessed through low-level asynchronous means, their unitary view reflects a central point. Blockchain provides a shared state between otherwise independent entities [Mendling *et al.*, 2018]. Langshaw's synchronous semantics provides a basis for interaction governed by sayso, which is more conducive to business meaning than arbitrary ordering, thereby allowing maximal concurrency compatible with a protocol.

Theoretical studies of protocols, e.g., [Mallya and Singh, 2007; Ferrando *et al.*, 2019], may need to be revisited in light of Langshaw. Further, the problem of verifying protocol-based agents [Baldoni *et al.*, 2006] remains largely unaddressed for information-based representations.

Online supplement. The code and all examples are available online at https://gitlab.com/masr/langshaw

Acknowledgments

Thanks to the anonymous reviewers for helpful comments. Thanks to the NSF (grant IIS-1908374) and EPSRC (grant EP/N027965/1) for support.

References

John L. Austin. *How to Do Things with Words*. Clarendon Press, Oxford, 1962.

Matteo Baldoni, Cristina Baroglio, Alberto Martelli, and Viviana Patti. A priori conformance verification for guaranteeing interoperability in open environments. In *Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC)*, LNCS, pages 339–351. Springer, December 2006.

Matteo Baldoni, Cristina Baroglio, Elisa Marengo, Viviana Patti, and Federico Capuzzimati. Engineering commitment-based business protocols with the 2CL methodology. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 28(4):519–557, July 2014.

Matteo Baldoni, Cristina Baroglio, Roberto Micalizio, and Stefano Tedeschi. Accountability in multi-agent organizations: From conceptual design to agent programming. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 37(1):7, June 2023.

Fabio Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, Chichester, UK, 2007.

Federico Bergenti, Giovanni Caire, Stefania Monica, and Agostino Poggi. The first twenty years of agent-based software development with JADE. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 34(2):36, 2020.

Olivier Boissier, Rafael H. Bordini, Jomi Fred Hübner, Alessandro Ricci, and Andrea Santi. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 78(6):747–761, June 2013.

Olivier Boissier, Andrei Ciortea, Andreas Harth, Alessandro Ricci, and Danai Vachtsevanou. Agents on the Web (Dagstuhl seminar 23081). *Dagstuhl Reports*, 13(2):71–162, 2023.

Daniel Brélaz. New methods to color vertices of a graph. *Communications of the ACM (CACM)*, 22(4):251–256, April 1979.

Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM (CACM)*, 32(4):444–458, April 1989.

Luca Cernuzzi and Franco Zambonelli. Experiencing AUML in the GAIA methodology. In *Proceedings of the 6th International Conference on Enterprise Information Systems (ICEIS)*, pages 283–288, Porto, Portugal, April 2004. Science and Technology Publications, Lda.

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

Samuel H. Christie V, Munindar P. Singh, and Amit K. Chopra. Kiko: Programming agents to enact interaction protocols. In *Proceedings of the 22nd International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 1154–1163, London, May 2023. IFAAMAS.

Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. OWL-P: A methodology for business process modeling and enactment. In *Proceedings of the AAMAS Workshop on Agent Oriented Information Systems (AOIS)*, Utrecht, The Netherlands, July 2005.

Angelo Ferrando, Davide Ancona, and Viviana Mascardi. Decentralizing MAS monitoring with DecAMon. In *Proceedings of the 16th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 239–248, São Paulo, May 2017. IFAAMAS.

Angelo Ferrando, Michael Winikoff, Stephen Cranefield, Frank Dignum, and Viviana Mascardi. On enactability of agent interaction protocols: Towards a unified approach. In *Proceedings of the 7th International Workshop on Engineering Multi-Agent Systems (EMAS)*, volume 12058 of *Lecture Notes in Computer Science*, pages 43–64, Montréal, May 2019. Springer.

Melvin Fitting. Introduction. In Marcello D'Agostino, Dov M. Gabbay, Reiner Hähnle, and Joachim Posegga, editors, *Handbook of Tableau Methods*, chapter 1, pages 1–43. Kluwer, 1999.

Ashok U. Mallya and Munindar P. Singh. An algebra for commitment protocols. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 14(2):143–163, April 2007.

Friedemann Mattern. Asynchronous distributed termination—parallel and symmetric solutions with echo algorithms. *Algorithmica*, 5(3):325–340, June 1990.

Jan Mendling, Ingo Weber, Wil van der Aalst, Jan vom Brocke, Cristina Cabanillas, Florian Daniel, Søren Debois, Claudio Di Ciccio, Marlon Dumas, Schahram Dustdar, Avigdor Gal, Luciano García-Bañuelos, Guido Governatori, Richard Hull, Marcello La Rosa, Henrik Leopold, Frank Leymann, Jan Recker, Manfred Reichert, Hajo A. Reijers, Stefanie Rinderle-Ma, Andreas Solti, Michael Rosemann, Stefan Schulte, Munindar P. Singh, Tijs Slaats, Mark Staples, Barbara Weber, Matthias Weidlich, Mathias Weske, Xiwei Xu, and Liming Zhu. Blockchains for business process management – challenges and opportunities. *ACM Transactions on Management Information Systems (TMIS)*, 9(1):4:1–4:16, February 2018.

Lin Padgham and Michael Winikoff. Prometheus: A practical agent-oriented methodology. In Brian Henderson-Sellers and Paolo Giorgini, editors, *Agent-Oriented Methodologies*, chapter 5, pages 107–135. Idea Group, Hershey, Pennsylvania, 2005.

Stefan Poslad. Specifying protocols for multi-agent systems interaction. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2(4):15:1–15:24, 2007.

Alessandro Ricci, Michele Piunti, Mirko Viroli, and Andrea Omicini. Environment programming in CArtAgO. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Multi-Agent Programming, Languages, Tools and Applications*, chapter 8, pages 259–288. Springer, Dordrecht, Netherlands, 2009.

Alessandro Ricci, Andrei Ciortea, Simon Mayer, Olivier Boissier, Rafael H. Bordini, and Jomi Fred Hübner. Engineering scalable distributed environments and organizations for MAS. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 790–798, Montréal, May 2019. IFAAMAS.

Colm Rooney, Rem W. Collier, and Gregory M. P. O'Hare. VIPER: A visual protocol editor. In *Proceedings of the 6th International Conference on Coordination Models and Languages COORDINATION*, volume 2949 of *Lecture Notes in Computer Science*, pages 279–293, Pisa, February 2004. Springer.

M. David Sadek, Philippe Bretier, and Franck Panaget. ARTIMIS: Natural dialogue meets rational agency. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1030–1035, Nagoya, Japan, August 1997. Morgan Kaufmann.

Munindar P. Singh and Amit K. Chopra. Clouseau: Generating communication protocols from commitments. In *Proceedings of the 34th Conference on Artificial Intelligence (AAAI)*, pages 7244–7252, New York, February 2020. AAAI Press.

Munindar P. Singh and Samuel H. Christie V. Tango: Declarative semantics for multiagent communication protocols. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 391–397, Online, August 2021. IJCAI.

Munindar P. Singh. Synthesizing distributed constrained events from transactional workflow specifications. In *Proceedings of the 12th International Conference on Data Engineering (ICDE)*, pages 616–623, New Orleans, February 1996. IEEE.

Munindar P. Singh. Write asynchronous, run synchronous. *IEEE Internet Computing (IC)*, 3(2):4–5, March 1999. Instance of the column *Being Interactive*.

Munindar P. Singh. A social semantics for agent communication languages. In *Proceedings of the 1999 IJCAI Workshop on Agent Communication Languages*, number 1916 in Lecture Notes in Artificial Intelligence, pages 31–45, Berlin, 2000. Springer.

Munindar P. Singh. Information-driven interaction-oriented programming: BSPL, the Blindingly Simple Protocol Language. In *Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 491–498, Taipei, May 2011. IFAAMAS.

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

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

Danai Vachtsevanou, Andrei Ciortea, Simon Mayer, and Jérémy Lemée. Signifiers as a first-class abstraction in hypermedia multi-agent systems. In *Proceedings of the 22nd International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 1200–1208, London, May 2023. IFAAMAS.

Renata Vieira, Álvaro F. Moreira, Michael J. Wooldridge, and Rafael H. Bordini. On the formal semantics of speech-act based communication in an agent-oriented programming language. *Journal of Artificial Intelligence Research (JAIR)*, 29:221–267, June 2007.

Danny Weyns and Tom Holvoet. A formal model for situated multi-agent systems. *Fundamenta Informaticae*, 63(2-3):125–158, 2004.

Danny Weyns, Andrea Omicini, and James Odell. Environment as a first class abstraction in multiagent systems. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 14(1):5–30, February 2007.

Michael Winikoff, Nitin Yadav, and Lin Padgham. A new Hierarchical Agent Protocol Notation. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 32(1):59–133, January 2018.

Michael Winikoff. Implementing commitment-based interactions. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 868–875, Honolulu, May 2007. IFAAMAS.