Peach: Program Each Agent and Communicate Howsoever

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

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

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

ABSTRACT

We contribute Peach, an approach to program an agent that decouples reasoning about the meanings of communications from the underlying communication infrastructure. Peach's conceptual component is a programming model based on Langshaw, a protocol language based on communicative actions, whose abstractions promote meaning and hide coordination details. Peach's computational component is an adapter that can be configured for different infrastructures. We provide an operational semantics for this adapter that maps our programming model to two main kinds of infrastructure: decentralized (via messages) or shared-memory.

CCS CONCEPTS

• **Computing methodologies** → **Multi-agent systems**.

KEYWORDS

Declarative Interaction Protocols; Programming models

ACM Reference Format:
Amit K. Chopra, Samuel H. Christie V, and Munindar P. Singh. 2026. Peach: Program Each Agent and Communicate Howsoever. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), Paphos, Cyprus, May 25 – 29, 2026*, IFAAMAS, 9 pages. https://doi.org/10.65109/SAXY3841

1 INTRODUCTION

A multiagent system comprises intelligent agents (representing autonomous real-world principals) who interact with each other via some communication infrastructure. An agent's reasoning determines its actions in any situation. In this paper, the actions of interest are *communicative acts* [1].

To interact, agents perform operations on a communication infrastructure that decouples an agent's reasoning from how its actions are realized. An important concern in multiagent systems is the engineering of general-purpose communication infrastructures.

The choice of a communication infrastructure for a multiagent system depends on factors such as (1) the installed IT systems; (2) need to facilitate auditing by regulators; and (3) scalability and fault tolerance requirements. The diverse communication infrastructures for multiagent systems fall into two main categories. In the *synchronous* or *shared memory* approaches, the social state is maintained in a single entity that synchronizes the actions of the various agents and thus determines which actions happen when (by delaying or denying them as needed). Examples include blackboards [17], tuple spaces [6, 19], artifacts [5, 21], and Web services.

In *asynchronous* or *decentralized* approaches, the social state is not stored in one place. Instead, each agent's *local state* is a projection of the social state and comprises the actions observed by the agent [23]. The asynchronous approach is best illustrated by the work on declarative interaction protocols, as exemplified by BSPL [22, 24, 25]. Several infrastructure-supported programming models for engineering BSPL agents are now available [3, 10–12]. These infrastructures help agents maintain their local states; take actions that accord with the relevant interaction protocols; and handle communication failures.

This variety of infrastructures poses a challenge for engineering multiagent systems. Introducing an abstraction layer is the classic computing response to diversity. Accordingly, we develop a protocol-based abstraction layer called *Peach*.

Peach decouples the choice of infrastructure from reasoning by an agent to meet its stakeholders' needs. Peach adopts the Langshaw [26] language for declaratively specifying interaction protocols in terms of communicative acts. Thus, Peach enables engineering multiagent systems that can be deployed over distinct infrastructures without having to rework the business logic.

Peach also lowers the barrier for developers unfamiliar with multiagent systems by providing a programming model at the level of communicative actions. A programmer specifies the reasoning for each action modularly and without regard to infrastructure—indeed, they may simply imagine a shared memory model. The Peach layer realizes concurrent actions according to the capacities of the specific infrastructure, handling details such as mapping multiple actions onto a single physical message.

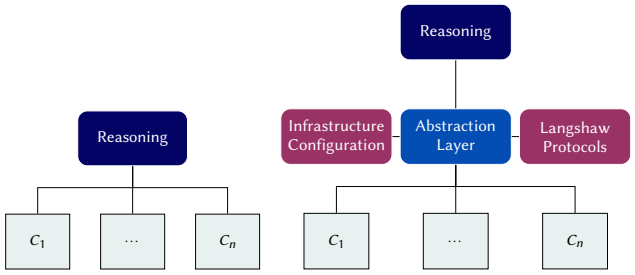


Figure 1: Peach introduces a protocol-based abstraction layer that enables switching communication infrastructures (C_i).

In principle, we could switch the communication infrastructure by changing only configuration information, without modifying

Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), C. Amato, L. Dennis, V. Mascardi, J. Thangarajah (eds.), May 25 – 29, 2026, Paphos, Cyprus. © 2026 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). https://doi.org/10.65109/SAXY3841

agent reasoning. The same agent may even participate in multiple multiagent systems built on different infrastructures. Figure 1 illustrates how Peach differs from conventional approaches.

Our contributions are the Peach programming model for engineering agents that participate in Langshaw protocols and an associated adapter that realizes Figure 1b's abstraction layer. We formalize three concrete adapters via inference rules. Two of them enable realizing a synchronous (shared memory) communication infrastructure, and one enables realizing an asynchronous (decentralized) communication infrastructure. The asynchronous adapter relies on BSPL protocols compiled from Langshaw protocols. All three adapters support the same programming model, meaning that they can be switched without affecting agent reasoning.

The rest of the paper is organized as follows. Section 2 describes the relevant literature. Section 3 gives the necessary Langshaw background. Section 4 describes the Peach programming model based on the notion of an adapter that interprets Langshaw protocols. Sections 5 and 6 give the designs for the synchronous and asynchronous adapters. Section 6 introduces BSPL and includes a compiler from Langshaw to BSPL. Section 7 summarizes the paper and gives some future directions.

2 RELATED WORK

Interaction is the essence of a multiagent system. The diverse approaches for modeling interaction in multiagent systems may be divided into two broad categories. In the category of *interaction protocols* are approaches such as AUML [18], trace expressions [14], HAPN [29], BSPL, and Langshaw. These approaches seek to capture the operational constraints on interaction, typically by prescribing when an agent may send or receive a message, relative to other messages. Of the approaches mentioned above, AUML is informal; the others are formal. Formal approaches are valuable because they can support automated verification, as BSPL and Langshaw do. In the category of *meaning* are approaches that model how messages change the normative state of the system modeled, for instance, in terms of commitments [2, 28, 30].

BSPL and Langshaw, both applied in this paper and described in more detail later, are declarative approaches. As they are motivated from meaning, they capture stakeholder intuitions more naturally and support flexible interactions. However, for purposes of modularity, they leave the normative aspects to a higher layer, as illustrated concretely in recent work [8]. Langshaw protocols may be enacted synchronously or via loosely-coupled, asynchronous messaging, as dictated by the nonfunctional requirements. In contrast to BSPL and Langshaw, which are formal, general-purpose approaches for modeling interactions, the FIPA protocols [15] capture only a handful of rigid interaction patterns informally (in AUML).

Several agent programming approaches, e.g., Jason [27], Gwendolen [13], and GOAL [16] provide cognitive abstractions based on the BDI (Belief-Desire-Intention) model of agent reasoning. Although these approaches nominally support asynchronous communication between agents, they do not support protocol-based programming. Moreover, their claim to decentralization is weakened by their reliance on KQML-based communication primitives [7, 12]. JADE [4] enables developing Java agents that communicate asynchronously; however, agent behaviors are modeled in terms of low-level and inflexible finite state machines.

In existing agent programming approaches, agents are coded to a particular communication infrastructure, which cannot readily be replaced by another. For example, in JaCaMo, the agent code refers to operations on the coordinating artifact [5, pp. 755]. Moreover, although agents may simultaneously use different communication channels (e.g., JaCaMo agents may simultaneously use artifacts and direct messaging), such a mixture is ad hoc, with no programming model support for interactions.

3 BACKGROUND: LANGSHAW

We now describe Langshaw following Singh et al. [26]. Langshaw is a language for modeling multiagent interaction protocols. The base Langshaw semantics assumes a shared store of *social state* that agents update synchronously—step-by-step—by performing communicative acts (actions). The social state is a set of actions, i.e., those already performed. In each update step, each agent may attempt a set of actions; the successful actions are added to the social state. Langshaw thus supports concurrent actions by agents.

A Langshaw protocol specifies the sets of actions that an agent may attempt in a social state and the constraints that must be satisfied for attempted actions to be successful. We explain how one may specify constraints in Langshaw with the help of *Purchase*, the Langshaw protocol specified in Listing 1. The protocol specifies roles SELLER, BUYER, and SHIPPER.

Listing 1: Purchase in Langshaw [26].

```
1  Purchase   // Name of the protocol
2   who  Buyer, Seller, Shipper // Roles
3   what  ID key, Reject or Deliver // Completion (
       for liveness)
4   do  // Actions: ways 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: Shipment(ID, Instruct, item, address)
11   sayso  // Who has social authority over the
       information (to generate bindings)
12   Buyer > Seller: item   // Buyer has higher
       authority than the Seller on item
13   Seller > Buyer: price
14   Buyer: address // Only Buyer has authority
15   Seller: fee
16   nono  // What action pairs are incompatible
17    Accept Reject
18    Reject Shipment
19   nogo  // What action prevents another
20    Reject ↛ Instruct
```

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 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 with distinct bindings for ID. When multiple agents attempt actions concurrently, the agents need not be consistent with each other. For example, assume that *RFQ* has occurred; thus, ID and item are bound. BUYER and SELLER may then concurrently attempt an *Accept* and *Quote* for that ID, meaning that both generate bindings for price (they both have sayso over price). In Langshaw, the saysos over the same attribute must be prioritized over agents (indicated by >): here, by Line 13, SELLER's sayso (being ranked higher) *dominates*. Consequently, its *Quote* attempt succeeds and updates the social state, whereas BUYER's *Accept* attempt fails (is a noop). Later, BUYER may attempt *Accept* again with the price in the *Quote*.

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 agent has the highest sayso of all the agents concurrently attempting to produce a binding for that attribute. When attempts dominate each other (on different attributes), each is a noop and neither affects the social state.

Lines 16–18 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.

Lines 19–20 specify an asymmetric conflict (not really needed in *Purchase*, but we insert it to explain the construct). Once *Reject* is in the social state, *Instruct* may not be performed. Importantly, *Instruct* may precede or occur simultaneously with *Reject*.

Concurrent conflicting attempts by two agents can succeed, resulting in an inconsistent social state. A *safe* protocol prevents such attempts. Every protocol specifies a criterion in terms of the information necessary for an enactment to be deemed *complete* (e.g., Line 3). A protocol is live if every enactment completes. *Purchase* is safe and live. Before a protocol is made available as a basis for implementing a multiagent system, it would be prudent to verify safety and liveness (Singh et al. [26] describes how).

4 PROGRAMMING MODEL AND ADAPTER

Each Peach agent has two main parts. The *reasoner* represents the agent's decision logic, including internal state. It sits atop the *adapter* that is configured with a Langshaw protocol and information about the infrastructure. For example, if the infrastructure is decentralized, it may include the network addresses of the agents that constitute the multiagent system so they may communicate directly with each other. If it is a shared database, the configuration would include the network address of the database.

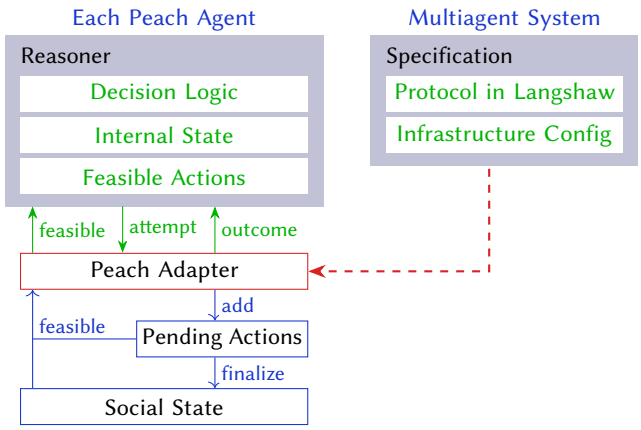


Figure 2: Peach architecture and programming model.

The idea of *pending actions* is a Peach innovation. Langshaw's semantics is organized around abstract steps: agents attempt actions, which are evaluated together to determine which succeed. In a concrete infrastructure, a step would have some duration. Any action attempted in a step becomes pending until the end of the step when *finalization* occurs and the action's success can be determined based on the other actions that have occurred in the step. Exactly how and when success is determined depends on the type of adapter, as we shall see later in Sections 5 and 6.

The reasoner uses the adapter to *attempt* sets of actions. To make an attempt, the reasoner queries the adapter for *feasible* actions, i.e., those that may be performed in the adapter's current state. The state of the adapter consists of the *social state* and the agent's pending actions. Actions that depend on information from pending actions and actions that are nono with them are deemed infeasible since a pending action may resolve as either successful or failed. Deeming them infeasible avoids complexity in the programming model. For each feasible action, the adapter populates the known attribute bindings—this helps avert integrity violations. To attempt a feasible action, the reasoner supplies bindings for the remaining attributes. If the agent's state has changed between querying for feasible actions and attempting a set of them, so that performing some action in the set would violate the protocol, the adapter deems the attempt illegal. The adapter also deems illegal any attempt that is not internally consistent; that is, the attempt contains actions

with different reasoner-supplied bindings for the same attribute. If an attempt is not rejected as illegal, all its actions become pending.

At the end of every step, the adapter *finalizes* the pending actions. This involves determining which of them are undominated in terms of sayso by the pending actions of other agents in that step. The undominated actions are deemed successful and added to the social state. The dominated ones are deemed to have failed. Listing 2 illustrates a Peach agent's reasoning pattern in pseudocode.

Listing 2: Use of Peach adapter in agent reasoning.

```
Adapter adapter = loadConfig()
//Get feasible actions
FeasibleActions F ← feasible(adapter)
//Reasoning to flesh out some subset A of F
Actions A ← reasoning(F,···)
try {
  //Attempt returns a vector of outcomes
  future Outcomes o⃗ ← attempt(adapter, A)
} catch(IllegalAttempt A){...}
...
∀o∈o⃗: if(o.result = true)
  print(o + "succeeded")
else
  print(o + "failed")
```

5 SYNCHRONOUS PEACH ADAPTER

In a synchronous adapter, the actions attempted by agents are processed simultaneously by a conceptually central entity.

Formally, let R be the set of agents in the multiagent system. Then, the tuple $\langle S, P\rangle$ denotes the state of the system, where S, a set of actions, is the social state of the system and $P = \langle P^1 \ldots P^{|R|}\rangle$, each P^i being the set of pending actions of agent i.

An execution of a system is a sequence of states $\langle S_1, P_1\rangle \ldots \langle S_n, P_n\rangle$, where the initial state is $S_1 = \emptyset$ and $P_1 = \langle P^1 = \emptyset \ldots P^{|R|} = \emptyset\rangle$. The transition between any two consecutive states is either an attempt by an agent or a finalization (of attempts) by the adapter. An attempt by an agent adds actions to that agent's pending component of the state; it does not affect the social state.

A finalization considers all actions in the pending state and, depending on how the exercise of saysos in the actions works out, adds the successful ones to the social state. Moreover, finalization deletes the unsuccessful pending actions—there are no pending actions in the state immediately following finalization. Thus, every finalization is a point of synchronization for the agents.

Figure 3 defines the transitions via inference rules in the usual manner [20]. A transition from state $\langle S, P\rangle$ to $\langle Q, R\rangle$ due to operation o is denoted by $\langle S, P\rangle \xrightarrow{o} \langle Q, R\rangle$. For brevity, where appropriate, we highlight only the changed components of the pending tuple and omit the others. The rules Attempt and Finalize capture the state transitions resulting from the attempt and finalize operations, respectively. The other rules capture computations within a state.

Below, m is an action 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. $Y_{x,y}\, a$ means x has higher sayso over a than y (both x and y have sayso). $Y_x\, a$ means x has sayso over a of whatever priority. Braces $\{\}$ indicate sets. Subscripts on sets and operators indicate 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 indicates what bindings can be inferred from the supplied state relative to a (potential or actual) action instance. For

$$\textsc{Z-Social}\ \frac{m_i[\vec{a_i}] \in S \qquad m_i[\vec{a_i}]\ \text{fits}\ m[\vec{a}]}{\mathsf{Z}(m[\vec{a}], \vec{a_i} \cap \vec{a}, S)}$$

$$\textsc{Z-subset}\ \frac{\mathsf{Z}(m[\vec{a}], \vec{q_i}, S) \qquad \vec{q} \subseteq \vec{q_i}}{\mathsf{Z}(m[\vec{a}], \vec{q}, S)}$$

$$\textsc{Z-union}\ \frac{\{\mathsf{Z}(m[\vec{a}], \vec{q_i}, S)\}_{i=1}^{n}}{\mathsf{Z}(m[\vec{a}], \cup_{i=1}^{n} \vec{q_i}, S)}$$

$$\textsc{Attemptable}\ \frac{\mathsf{Z}(m[\vec{k}, \vec{p}], \vec{q}, S) \quad Y_x \vec{y} \quad \neg\, \mathsf{Z}(m[\vec{k}, \vec{p}], \vec{y}, S \cup P^x)}{\text{attemptable}(x\colon m[\vec{k}, \vec{q} \cup \vec{y}], S, P^x)}$$

$$\textsc{Abide}\ \frac{\begin{array}{c} m_i[\vec{a_i}]\ \text{fits}\ m[\vec{a}] \rightarrow \\ m_i[\vec{p_i} \cap \vec{p}] = m[\vec{p_i} \cap \vec{p}] \wedge \neg\, \text{nono}(m_i, m) \end{array}}{\text{abides}(m_i[\vec{a_i}], m[\vec{a}])}$$

$$\textsc{Unsocial-M}\ \frac{m_i[\vec{a_i}] \in Q \qquad \neg\, m_i\ \text{abides}\ m}{\text{unsocial}(m[\vec{a}], Q)}$$

$$\textsc{Unsocial-N}\ \frac{m_i[\vec{a_i}] \in Q \qquad m_i[\vec{a_i}]\ \text{fits}\ m[\vec{a}] \qquad \text{nogo}(m_i, m)}{\text{unsocial}(m[\vec{a}], Q)}$$

$$\textsc{Feasible}\ \frac{\begin{array}{c} \text{attemptable}(x : m[\vec{a}], S) \\ \neg\, \text{unsocial}(m[\vec{a}], S) \quad \neg\, \text{unsocial}(m[\vec{a}], P^x) \end{array}}{\text{feasible}(x : m[\vec{a}], S, P^x)}$$

$$\textsc{Attempt}\ \frac{\begin{array}{c} \tau = \text{attempt}\ \{x\colon m_i[\vec{a_i}]\}_{i=1}^{n} \\ \{\text{feasible}(m_i[\vec{a_i}], S, P^x)\}_{i=1}^{n} \\ \wedge_{i,j=1}^{n}\ \text{abides}(m_i[\vec{a_i}], m_j[\vec{a_j}]) \end{array}}{\langle S, \langle P^x\rangle\rangle \xrightarrow{\tau} \langle S, \langle P^x \cup \{m_i[\vec{a_i}]\}_{i=1}^{n}\rangle\rangle}$$

$$\textsc{Dominates}\ \frac{\begin{array}{cc} m_i[\vec{a_i}]\ \text{fits}\ m[\vec{a}] & p \in \vec{a_i} \cap \vec{a} \\ \neg\, \mathsf{Z}(m[\vec{a}], p, S) & Y_{x_i,x} p \end{array}}{\text{dominates}(x_i : m_i[\vec{a_i}], x : m[\vec{a}], S)}$$

$$\textsc{Finalize}\ \frac{\begin{array}{c} \tau = \text{finalize} \qquad Q = P^1 \cup \ldots \cup P^{|R|} \\ T = \{m_i\colon \forall m_i, m_j \in Q\colon j \neq i \rightarrow \\ \neg\ \text{dominates}(m_j, m_i, S)\} \end{array}}{\langle S, \langle P^1 \ldots P^{|R|}\rangle\rangle \xrightarrow{\tau} \langle S \cup T, \langle U^1 = \emptyset \ldots U^{|R|} = \emptyset\rangle\rangle}$$

Figure 3: How the shared memory Peach adapter handles attempts and finalizes the pending actions. That it works for the shared memory case is seen by the S representing the unitary social state store.

an action instance $m[\vec{k}, \vec{p}]$, the bindings of its attributes occurring in a fitting instance in the state can be inferred. Z-SUBSET and Z-UNION say that Z bindings are closed under subset and union.

In ATTEMPTABLE, an instance is *attemptable* by x if (1) it includes any bindings already established (for the key) in the social state S, (2) x has sayso over the remaining attributes, and (3) the bindings for those attributes are not already established in x's pending actions P. We avoid relying on bindings from pending actions because they are not part of the social state (and may never be if they fail).

In ABIDE, an instance *abides* by another provided: If the bindings of their common key attributes agree, then they must not be nono and the bindings of all their common attributes must agree.

UNSOCIAL-M states that an instance is *unsocial* (i.e., socially inconsistent) if there is an instance in the supplied state with which it does not abide. UNSOCIAL-N states that an instance is *unsocial* if there is an instance in the supplied state that nogoes the instance.

FEASIBLE states that an agent x's instance is *feasible* in social state S with pending actions P^x if (1) it is attemptable, and (2) it is not unsocial with respect to S or P^x.

ATTEMPT captures x's attempt to perform a set of instances in social state S and with pending actions P_x. The attempt is successful only if (1) all of its instances are feasible, and (2) they abide with each other. Upon success, the system transitions into the new state where the instances in the attempt have been added to P_x.

DOMINATES captures that m_i *dominates* m in social state S 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 ($Y_{x_i,x}p$).

FINALIZE captures the finalization of the pending instances. From the set of all pending instances of all agents, it computes the set of undominated instances and adds them to the social state. The remaining pending instances, having failed, are deleted.

Synchronous adapters are naturally realized around a central store of social state. An agent's pending actions may be stored locally or centrally, depending on application requirements. Stores may be of diverse implementations, e.g., databases, tuple stores, Web services, artifacts, and blockchains.

We categorize synchronous deployments into two broad kinds depending on when finalization occurs relative to attempts.

5.1 Batched

The batched processing of actions is common in computing systems, e.g., both blockchain and traditional banking.

A *batched* adapter finalizes multiple attempts together when triggered by a periodic event or a threshold of accumulated work. The batching duration, ranging from milliseconds to days (e.g., Bitcoin blocks or traditional banking settlement), is a configuration parameter. Batching introduces delays but yields improved flexibility in handling concurrent saysos.

Batching provides well-defined deadlines before which higher-sayso agents may act to exercise their saysos. An implementation may make the pending actions publicly visible, which could allow the higher-sayso agents to observe other attempted actions before moving themselves. This corresponds to the common business pattern of *right of first refusal*, e.g., when the current tenant of a house has the first right to buy it before the owner can offer it to someone else. Whether such disclosure is appropriate for a multiagent system depends on its stakeholders' needs. For example, in financial markets, higher-sayso agents may use that information to front-run stock transactions and thus exploit lower-sayso agents.

Figure 3 illustrates batched operation for *Purchase*. The *Server* is the locus of social state and performs the batching. The squares indicate finalization by the Server; the time between two consecutive finalizations corresponds to a batching window. Focusing on the highlighted window, *Accept* and *Quote* both become pending until finalization occurs and determines *Quote* as successful and *Accept* as failed because of the SELLER's higher sayso on price. The BUYER learns about this failure (not shown) and later does an *Accept* with the price in *Quote*.

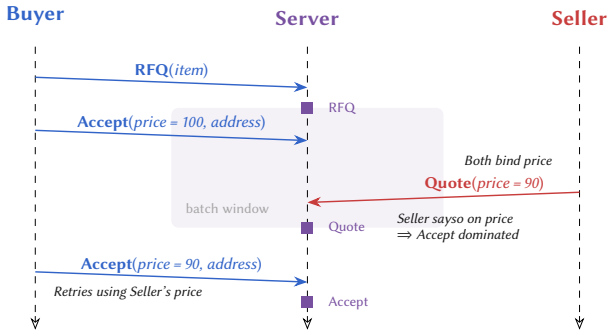


Figure 4: Synchronous enactment where Quote dominates Accept (demonstrating batching with just one window).

5.2 Instantaneous

The instantaneous adapter finalizes each attempt (recall, of a set of actions) as it occurs. It is the special case of batched in which the batching duration is zero. A database with support for atomic transactions could be used to realize such an adapter. Essentially, the adapter would wrap every attempt's processing and its finalization in a single transaction. The serialization order of the transactions would determine the order in which attempts are finalized. Alternatively, the adapter could be implemented using a persistent queue, such as Kafka. In general, the instantaneous approach relies on the consensus protocol of the underlying store to serialize the attempts.

Obviously, since the instantaneous adapter finalizes every attempt right away, it yields only executions where the attempt and finalize transitions alternate. In this sense, it is more restrictive than the semantics of Figure 3. Notice that since an attempt comprises one agent's actions, all pending actions are undominated and, therefore, succeed.

6 ASYNCHRONOUS PEACH ADAPTER

We now present an asynchronous adapter, i.e., one in which the actions attempted by agents are evaluated locally.

To realize an asynchronous infrastructure, we compile the Langshaw protocol into a BSPL protocol that the adapters enact in a decentralized manner; the agent programmer never sees this protocol. Below, we introduce BSPL, describe the compiler, and then show how the adapter implements the generated protocol in accordance with Langshaw semantics.

6.1 Background: BSPL

For concreteness, we adopt BSPL for specifying asynchronous messaging protocols. As Listing 3 shows, a BSPL protocol lists roles, a completion criterion, and one or more messages. Each message specifies information dependencies via adornments ⌜in⌝, ⌜out⌝, and ⌜nil⌝. 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 ⌜in⌝, ⌜out⌝, and ⌜nil⌝, respectively. An instance of m, also denoted $m[\vec{p}_I, \vec{p}_O, \vec{p}_N]$ for notational convenience, is a message that has bindings only for the ⌜in⌝ and ⌜out⌝ parameters; x may send an instance of m only if the bindings of the ⌜in⌝ parameters are known (already present in its local state), bindings of the ⌜out⌝ parameters are unknown (but added to the local state upon sending), and bindings of the ⌜nil⌝ parameters are unknown (and not added to the local state).

For example, to send an *Accept*, BUYER must know ID, item, and *price*. Moreover, it must not know done already, but it must generate its binding. *Accept* and *Reject* are mutually exclusive because each includes done as ⌜out⌝.

Listing 3: A simple protocol to explain BSPL.

```
Accept-Reject //Protocol name
 roles B, S //roles Buyer and Seller
 parameters out ID key, out done //completion
 S ↦ B:Offer[out ID key, out item, out price]
 B ↦ S:Accept[in ID key, in item, in price, out
     done]
 B ↦ S:Reject[in ID key, in item, in price, out
     done]
```

Figure 5 gives an operational semantics for BSPL. Below, the notation ⌜in⌝ $\vec{p}$, ⌜out⌝ $\vec{p}$, ⌜nil⌝ $\vec{p}$ means that each attribute in $\vec{p}$ is ⌜in⌝, ⌜out⌝, and ⌜nil⌝, respectively.

$$\textsc{Send}\ \frac{s = m[\vec{p}_I, \vec{p}_O, \vec{p}_N] \qquad \mathsf{Z}(m[\vec{k}, \vec{p}_I], \vec{p}_I, L^x) \qquad \neg\,\mathsf{Z}(m[\vec{k}, \vec{p}_O], \vec{p}_O, L^x) \qquad \neg\,\mathsf{Z}(m[\vec{k}, \vec{p}_N], \vec{p}_N, L^x)}{L^x \xrightarrow{\text{send } s} L^x \cup s}$$

$$\textsc{Receive}\ \frac{(r = x \mapsto y\colon m[\vec{p}_I, \vec{p}_O]) \in L^x}{L^y \xrightarrow{\text{recv } r} L^y \cup r}$$

Figure 5: Asynchronous semantics for BSPL.

6.2 Compiling Langshaw into BSPL

Let $x\colon A(\vec{a})$ be an action in a Langshaw specification. In $x\colon A(\vec{k}, \vec{a})$, $\vec{a}$ is all the attributes of A and $\vec{k} \subseteq \vec{a}$ are the key attributes. For an attribute a, its determinant, denoted Δa, is defined as the intersection of the key attributes of all actions in which a appears. We use $Y^x\vec{p}$ to mean that x has highest sayso over the attributes $\vec{p}$ and $Y^{x_n}_{x_1}\vec{p}$ as shorthand for $Y_{x_n,x_{n-1}}\vec{p}, \ldots, Y_{x_2,x_1}\vec{p}$.

Let's associate with each a an attribute go$a$. Associate with each unordered pair of actions (A, B) related by nono an attribute nonoAB. Let $U(A)$ be the agents to whom any attribute of A is relevant. Let M_A be a BSPL message for A. It must satisfy the following constraints.

(1) The attributes of $\vec{a}$ map to either ⌜in⌝ $(\vec{a}^I)$ or ⌜out⌝ $(\vec{a}^O)$ parameters, with no ⌜nil⌝ parameters. That is, $\vec{a}^O \cup \vec{a}^I = \vec{a}$ and $\vec{a}^I \cap \vec{a}^O = \emptyset$. Let $\vec{k}^O \subseteq \vec{a}^O$ be the key attributes that map to ⌜out⌝ parameters.
(2) Let $\vec{h}^O \subseteq \vec{a}^O$ be the attributes in $\vec{a}^O$ such that, for each $h^O \in \vec{h}^O$, either x has the highest sayso over h^O (i.e., $Y^x h^O$) or h^O's determinant contains an ⌜out⌝ key (i.e., $\Delta h^O \cap \vec{k}^O \neq \emptyset$). Attributes go$\vec{h}^O$ are ⌜out⌝ as x need not request sayso approval for $\vec{h}^O$.
(3) Let $\vec{l}^O = \vec{a}^O \setminus \vec{h}^O$ be the remaining attributes of $\vec{a}^O$. Agent x must have sayso over them ($Yx\vec{l}^O$). Attributes go$\vec{l}^O$ are ⌜in⌝ as x needs sayso approval for $\vec{l}^O$.
(4) To capture the well-formedness condition that an attribute cannot have a binding independently of its key attributes, the parameters in the determinant of each a^I must be ⌜in⌝.
(5) For each a^I, goa^I must be ⌜in⌝. Our construction ensures that when x knows an attribute binding, it also has approval for it, meaning that x never has to request approval to use a binding.
(6) For each nono(A, B), attribute nonoAB must be ⌜out⌝.
(7) For each nogo(B, A), attribute B must be ⌜nil⌝.
(8) Attribute A must be ⌜out⌝.
(9) x is the sender and $U(A)$ are the receivers.

Figure 6 gives the compiler from Langshaw to BSPL via a set of rules. Rule ACTION captures the foregoing constraints.

Before an agent x may ⌜out⌝ an attribute a over which it has non-highest sayso (specifically, those in $\vec{l}^O$ above), it needs to know $\vec{\text{go}}l^O$. We introduce a coordination protocol by which the agent may request such approvals. The protocol takes advantage of the fact that saysos for an attribute are linearly organized. Before sending a message with ⌜out⌝ o, x expresses interest in exercising its sayso over o by sending a request to the next agent up in the sayso chain. However, the agent may do so only if o is not known to it and if no lower sayso agent has expressed to it an interest in binding o. If a lower sayso agent has expressed interest in o, the agent may forward the request to the next higher sayso agent. If the agent is itself the highest sayso agent, it may send an approval to the lower sayso agent, provided o is not already bound. Approval generates the corresponding go attribute. The rules REQUEST, FORWARD, and APPROVE capture this coordination. An explicit message for declining sayso to a lower agent is not needed. An agent knows its sayso request for some attribute has failed when it receives a message with a binding for the attribute. Such a message will be sent by whoever generates the binding because the attribute is relevant to the requesting agent.

Finally, COMPOSITEACTION handles nogo cycles over actions of the same agent. For concreteness, consider nogo$(x\colon A, x\colon B)$ and nogo$(x\colon B, x\colon A)$. In Langshaw, A and B may be performed concurrently by putting them in the same attempt. BSPL agents are sequential; they send messages one at a time. Following the compilation given above, if one of A or B has been sent, the other can't be sent, which means that there is no BSPL execution corresponding

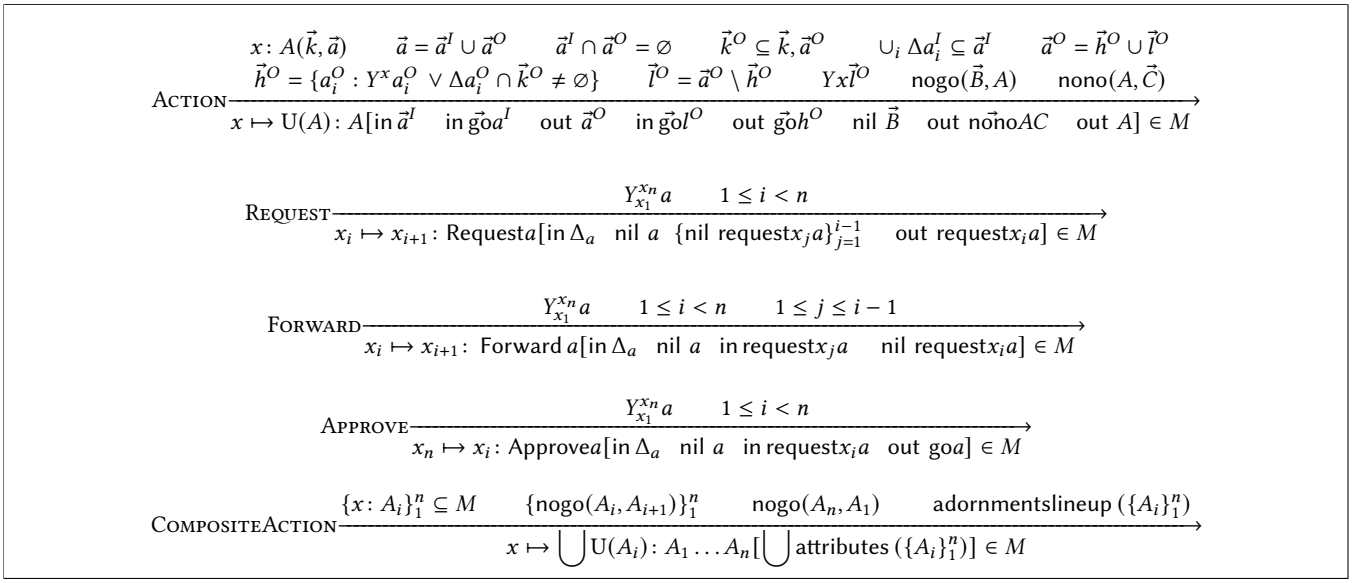


Figure 6: Rules for generating the messages M in a BSPL protocol corresponding to a Langshaw specification.

to the Langshaw execution. CompositeAction gets around this problem by creating a composite BSPL message, sending which corresponds to performing both A and B. It is essentially a union of the attributes in the messages corresponding to A and B. It considers only the messages corresponding to A and B where the adornments of the attributes line up. The completion specification of the BSPL protocol is the same as the Langshaw protocol.

6.3 Asynchronous Adapter

We now show how an adapter exercises the generated protocol. Instead of a shared social state, each agent's adapter maintains (queries and updates) a local state. The state of the system with R agents is given by a tuple of the local states of the agents and their pending actions, denoted

$$\langle\langle L^1, P^1\rangle \dots \langle L^{|R|}, P^{|R|}\rangle\rangle$$

where L^i and P^i are agent i's local state and pending actions, respectively. Figure 7 formalizes the adapter transitions.

Base says that a message is a-attemptable (async-attemptable) by x if its ⌜in⌝ attributes minus any go attributes are known from L^x and its ⌜out⌝ and ⌜nil⌝ attributes are unknown. We keep track of the go attributes because to actually send the message, their bindings must be known from the local state. Recursive captures the version of a-attemptable where some of a message's ⌜in⌝ attributes, again minus any go attributes, are outed in another fitting a-attemptable. Since a BSPL agent is sequential, we realize a Peach attempt with multiple actions that generate a binding for the attribute as BSPL messages where the attribute is ⌜out⌝ in one but ⌜in⌝ in the others.

A-Feasible says that an $x\colon m$ is a-feasible in L^x, P^x if it is a-attemptable and it is not unsocial with respect to P^x. Notice that we don't have to deal with constraints with respect to the social state as they are already encoded in the generated BSPL protocol and handled in the determination of a-attemptable.

A-Attempt says that an attempt by x transitions x to the state where all the messages in it (actions from the point of view of the reasoner) move to P^x provided that all the messages in it are a-feasible and they abide with each other. Notice that we are still carrying the vector of go attributes ($\vec{g}$).

Next, we introduce the rules that deal with the finalization of pending messages, although there isn't a single finalize rule in the asynchronous adapter.

No Delay says that if a pending message does not have attributes that require approval from others, it may be sent without further ado. The message is added to (the agent's) local state and removed from pending.

Need captures the case where some attributes require approval. For each such attribute, if the corresponding Request is attemptable, it is sent. The Request may not be attemptable because it has already been sent, but that is not a problem.

Got captures the case where the necessary gos to be able to send a message have all arrived. In this case, the message is sent, which adds it to the local state and removes it from pending.

Failure captures the case that while waiting for gos for some attributes in a pending message, the agent receives a binding for one of those attributes. This means it will never receive the necessary gos and the attempt to send it has failed. So the message is removed from pending.

Forward simply says that if a Forward for some attribute is attemptable, which means a lower sayso agent has expressed interest in it, it is sent.

MAS states how the multiagent system as a whole transitions when one of its agents transitions. The superscript $-i$ indicates all agents in the system except i.

$$\textsc{Base}\ \frac{\vec{g} = \text{go}(\vec{p}_I) \quad \vec{p}_K = \vec{p}_I \setminus \vec{g} \quad \mathsf{Z}(m[\vec{k}, \vec{p}_K], \vec{p}_K, L^x) \quad \neg\,\mathsf{Z}(m[\vec{k}, \vec{p}_O], \vec{p}_O, L^x) \quad \neg\,\mathsf{Z}(m[\vec{k}, \vec{p}_N], \vec{p}_N, L^x)}{\text{a-attemptable}(x \mapsto y : m[\vec{p}_K, \vec{p}_O, \vec{p}_N], \vec{g}, L^x)}$$

$$\textsc{Recursive}\ \frac{\vec{g} = \text{go}(\vec{I}) \quad \vec{p}_K = \vec{p}_I \setminus \vec{\text{go}} \quad \vec{p}_K = \vec{p}_S \cup \vec{p}_T \quad \mathsf{Z}(m[\vec{k}, \vec{p}_S], \vec{p}_S, L^x) \quad \neg\,\mathsf{Z}(m[\vec{k}, \vec{p}_O], \vec{p}_O, L^x) \quad \neg\,\mathsf{Z}(m[\vec{k}, \vec{p}_N], \vec{p}_N, L^x) \quad \text{a-attemptable}(m_i[\vec{k}_i, \vec{p}_{iO}], L^x) \quad m_i[\vec{a}_i]\ \text{fits}\ m[\vec{a}] \quad \vec{p}_T \subseteq \vec{p}_{iO}}{\text{a-attemptable}(x \mapsto y : m[\vec{p}_I, \vec{p}_O, \vec{p}_N], \vec{g}, L^x)}$$

$$\textsc{A-Feasible}\ \frac{\text{a-attemptable}(m[\vec{a}], \vec{g}, L^x) \quad \neg\,\text{unsocial}(m[\vec{a}], P^x)}{\text{a-feasible}(x \rightharpoonup y : m[\vec{a}], \vec{g}, L^x, P^x)}$$

$$\textsc{A-Attempt}\ \frac{\tau = \text{attempt}\ \{x : m_i[\vec{a}_i]\}_{i=1}^n \quad \{\text{a-feasible}(m_i[\vec{a}_i], \vec{g}_i, L^x, P^x)\}_{i=1}^n \quad \wedge_{i,j=1}^n \text{abides}(m_i[\vec{a}_i], m_j[\vec{a}_j])}{L^x, P^x \xrightarrow{\tau} L^x, P^x \cup \{m_i[\vec{a}_i]\vec{g}_i\}_{i=1}^n}$$

$$\textsc{No Delay}\ \frac{(s = x \mapsto y : m[\vec{a}]\vec{g}) \in P^x \quad \vec{g} = \varnothing}{L^x, P^x \xrightarrow{\text{send } s} L^x \cup s, P^x \setminus s}$$

$$\textsc{Need}\ \frac{x \mapsto y : m[\vec{k}, \vec{a}]\vec{g} \in P^x \quad g \in \vec{g} \quad \text{attemptable}(s = (x \mapsto z : \text{Request}g[\Delta_g, \vec{a}_g]), L_x)}{L^x, P^x \xrightarrow{\text{send } s} L^x \cup s, P^x}$$

$$\textsc{Got}\ \frac{t = (x \mapsto y : m[\vec{a}]\vec{g}) \in P^x \quad \{g_j\}_{i=1}^n \quad \wedge_{j=1}^n \text{Approve}g_j[\Delta_{g_j}, \vec{a}_j] \in L_x \quad s = (x \mapsto y : m[\vec{a} \cup \vec{g}])}{L^x, P^x \xrightarrow{\text{send } s} L^x \cup s, P^x \setminus t}$$

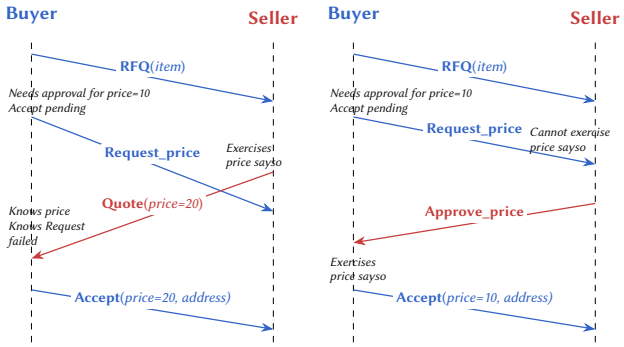
$$\textsc{Failure}\ \frac{s = (x \mapsto y : m[\vec{k}, \vec{a}]\vec{g}) \in P^x \quad \exists g \in \vec{g} : \mathsf{Z}(m[\vec{k}, g], L_x)}{L^x, P^x \xrightarrow{\text{fail } s} L^x, P^x \setminus s}$$

$$\textsc{Forward}\ \frac{\text{attemptable}(s = (x \mapsto z : \text{Forward}g[\Delta_g, \vec{a}_g]), L_x))}{L^x, P^x \xrightarrow{\text{send } s} L^x \cup s, P^x}$$

$$\textsc{MAS}\ \frac{\langle L_j^i, P_j^i \rangle \xrightarrow{o} \langle L_{j+1}^i, P_{j+1}^i \rangle}{\langle \langle L_j^{-i}, P_j^{-i} \rangle \ldots \langle L_j^i, P_j^i \rangle \rangle \xrightarrow{o} \langle \langle L_j^{-i}, P_j^{-i} \rangle \ldots \langle L_{j+1}^i, P_{j+1}^i \rangle \rangle}$$

Figure 7: The Peach asynchronous (decentralized) adapter. We know it is decentralized because there is no unitary social state; instead, there is a Local state L^i for each agent.

In Figure 8a, BUYER's request to exercise its sayso over price fails because SELLER, who has the highest sayso over price, has already



(a) Sayso request failed **(b) Sayso request succeeded**

Figure 8: Sayso coordination in decentralized settings.

exercised it. In Figure 8b, SELLER hasn't exercised it, so when it receives BUYER's request, it approves it.

7 CONCLUSIONS AND DIRECTIONS

Peach's value arises from its being a programming model that enables engineering flexible agents that operate close to the meaning level while making the communication infrastructure pluggable via mere configuration. The Peach adapter supports this programming model and may be realized either as a synchronous or asynchronous component. Peach supports our long-held intuition: Protocols are fundamental engineering abstractions for multiagent systems, regardless of how communication is operationally realized. In particular, by supporting shared memory–based communication, Peach provides an opportunity for work on declarative protocols to influence practice, where the shared memory approach dominates. Moreover, for many, embracing decentralized operation will be much easier if it is abstracted away via the programming model. We have implemented prototypes of all three adapters discussed in the paper and are making them more robust.

Our current effort enables engineering agents in Python. To make Peach's abstraction available in another framework, say Jason, one must implement a Peach adapter using Jason. Such an adapter may implement one of the operational semantics presented in the paper or a novel one altogether (e.g., that supports a mix of synchronous and asynchronous communications). With a Jason Peach adapter available, one can engineer a multiagent system whose Jason agents communicate via Langshaw protocols. The resulting Jason agents would be conceptual analogs of Listing 2.

The Agentic AI paradigm seeks to realize LLM-based AI assistants that take real-world actions on behalf of users. We believe that Langshaw is ideally-suited to this paradigm [9]. Langshaw's high-level nature suggests that LLM inferencing will not be cluttered with low-level operational details. Langshaw's ability to support flexible interaction would cohere with an LLM's ability to do flexible, contextual reasoning. Finally, Langshaw supports formal communication, which would be indispensable in any setting that demands clear interaction meaning, e.g., in healthcare, e-commerce, and finance. By using a Peach adapter as a tool, an LLM can engage in formal communication. Our ongoing efforts are geared toward approaches that synthesize LLMs and declarative protocols.

REFERENCES

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

[2] Matteo Baldoni, Cristina Baroglio, Elisa Marengo, Viviana Patti, and Federico Capuzzimati. 2014. Engineering Commitment-Based Business Protocols with the 2CL Methodology. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 28, 4 (July 2014), 519–557. doi:10.1007/S10458-013-9233-1

[3] Matteo Baldoni, Samuel H. Christie V, Munindar P. Singh, and Amit K. Chopra. 2025. Orpheus: Engineering Multiagent Systems via Communicating Agents. In *Proceedings of the 39th AAAI Conference on Artificial Intelligence (AAAI)*. AAAI, Philadelphia, 23135–23143. doi:10.1609/aaai.v39i22.34478

[4] Fabio Bellifemine, Giovanni Caire, and Dominic Greenwood. 2007. *Developing Multi-Agent Systems with JADE.* Wiley, Chichester, UK. doi:10.1002/9780470058411

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

[6] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. 2000. MARS: A Programmable Coordination Architecture for Mobile Agents. *IEEE Internet Computing* 4, 4 (2000), 26–35. doi:10.1109/4236.865084

[7] Amit K. Chopra, Alexander Artikis, Jamal Bentahar, Marco Colombetti, Frank Dignum, Nicoletta Fornara, Andrew J. I. Jones, Munindar P. Singh, and Pınar Yolum. 2013. Research Directions in Agent Communication. *ACM Transactions on Intelligent Systems and Technology (TIST)* 42, 2, Article 20 (March 2013), 23 pages. doi:10.1145/2438653.2438655

[8] Amit K. Chopra, Matteo Baldoni, Samuel H. Christie V, and Munindar P. Singh. 2025. Azorus: Commitments over Protocols for BDI Agents. In *Proceedings of the 24th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Detroit, 490–499. doi:10.5555/3709347.3743564

[9] Amit K. Chopra, Samuel H. Christie V, and Munindar P. Singh. 2026. Tools for Implementing Multi-Agent Systems Based on Protocols. In *Agents and Multi-Agent Systems Development: Platforms, Toolkits, Technologies*, Rem Collier, Viviana Mascardi, and Alessandro Ricci (Eds.). Springer, Cham, Switzerland, 211–230. doi:10.1007/978-3-032-01082-7_8

[10] Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. 2022. Mandrake: Multiagent Systems as a Basis for Programming Fault-Tolerant Decentralized Applications. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 36, 1, Article 16 (April 2022), 30 pages. doi:10.1007/s10458-021-09540-8

[11] Samuel H. Christie V, Munindar P. Singh, and Amit K. Chopra. 2023. Kiko: Programming Agents to Enact Interaction Protocols. In *Proceedings of the 22nd International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, London, 1154–1163. doi:10.5555/3545946.3598758

[12] Samuel H. Christie V, Munindar P. Singh, and Amit K. Chopra. 2025. Argus: Programming with Communication Protocols in a Belief-Desire-Intention Architecture. *Artificial Intelligence* 348, Article 104398 (Nov. 2025), 35 pages. doi:10.1016/j.artint.2025.104398

[13] Louise A. Dennis and Berndt Farwer. 2008. Gwendolen: A BDI Language for Verifiable Agents. In *Proceedings of the AISB Symposium on Logic and the Simulation of Interaction and Reasoning*. Society for the Study of Artificial Intelligence and Simulation of Behaviour, Aberdeen, United Kingdom, 16–23. https://aisb.org.uk/wp-content/uploads/2019/12/Final-vol-09.pdf

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

[15] FIPA. 2003. FIPA Interaction Protocol Specifications. http://www.fipa.org/repository/ips.html. FIPA: The Foundation for Intelligent Physical Agents. Accessed 2024-11-24.

[16] Koen V. Hindriks. 2009. Programming Rational Agents in GOAL. In *Multi-Agent Programming: Languages, Tools and Applications*, Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni (Eds.). Springer, Dordrecht, Netherlands, Chapter 4, 119–157. doi:10.1007/978-0-387-89299-3_4

[17] H. Penny Nii. 1986. Blackboard Systems, Part One: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures. *AI Magazine* 7, 2 (Summer 1986), 38–53. http://www.aaai.org/ojs/index.php/aimagazine/article/view/537

[18] James Odell, H. Van Dyke Parunak, and Bernhard Bauer. 2001. Representing Agent Interaction Protocols in UML. In *Proceedings of the 1st International Workshop on Agent-Oriented Software Engineering (AOSE 2000) (Lecture Notes in Computer Science, Vol. 1957)*. Springer, Toronto, 121–140. doi:10.1007/3-540-44564-1_8

[19] Andrea Omicini and Enrico Denti. 2001. From tuple spaces to tuple centres. *Science of Computer Programming* 41, 3 (2001), 277–294. doi:10.1016/S0167-6423(01)00011-9

[20] Gordon D. Plotkin. 2004. A Structural Approach to Operational Semantics. *The Journal of Logic and Algebraic Programming* 60–61 (July–Dec. 2004), 17–139. doi:10.1016/j.jlap.2004.05.001

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

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

[23] Munindar P. Singh. 2011. 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)*. IEEE Computer Society, Washington, DC, 57–64. doi:10.1109/ICWS.2011.48

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

[25] Munindar P. Singh and Samuel H. Christie V. 2021. Tango: Declarative Semantics for Multiagent Communication Protocols. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI, Online, 391–397. doi:10.24963/ijcai.2021/55

[26] Munindar P. Singh, Samuel H. Christie V, and Amit K. Chopra. 2024. Langshaw: Declarative Interaction Protocols Based on Sayso and Conflict. In *Proceedings of the 33rd International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI, Jeju, Korea, 202–210. doi:10.24963/ijcai.2024/23

[27] Renata Vieira, Álvaro F. Moreira, Michael J. Wooldridge, and Rafael H. Bordini. 2007. On the Formal Semantics of Speech-Act Based Communication in an Agent-Oriented Programming Language. *Journal of Artificial Intelligence Research (JAIR)* 29 (June 2007), 221–267. doi:10.1613/jair.2221

[28] Michael Winikoff. 2007. Implementing Commitment-Based Interactions. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*. IFAAMAS, Honolulu, 868–875. doi:10.1145/1329125.1329283

[29] Michael Winikoff, Nitin Yadav, and Lin Padgham. 2018. A New Hierarchical Agent Protocol Notation. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 32, 1 (Jan. 2018), 59–133. doi:10.1007/s10458-017-9373-9

[30] Pınar Yolum and Munindar P. Singh. 2002. 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)*. ACM Press, Bologna, 527–534. doi:10.1145/544862.544867