Requirement Patterns for Engineering Multiagent Interaction Protocols

Amit K. Chopra[1] , **Samuel H. Christie V**[2] and **Munindar P. Singh**[2]
[1]School of Computing and Communications, Lancaster University, Lancaster LA1 4WA, UK
[2]Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA
amit.chopra@lancaster.ac.uk, schrist@ncsu.edu, mpsingh@ncsu.edu

Abstract

An interaction protocol specifies how the member agents of a decentralized multiagent system may communicate to satisfy their respective stakeholders' requirements. We focus on information protocols, which are fully declarative specifications of interaction and support asynchronous communication. We offer Mambo, an approach for protocol design. Mambo identifies common patterns of requirements, provides a notation to express them, and a verification procedure. Mambo incorporates heuristics to generate small internal representations for efficiency. Experimental results demonstrate Mambo's effectiveness on practical protocols.

1 Introduction

Interaction has been recognized as a crucial construct in multiagent systems since the field's inception [Boissier *et al.*, 2023]. Practical approaches address various aspects of interaction [Casadei *et al.*, 2021]. A *protocol* captures the interactions between the agents in a multiagent system (MAS). Specifically, it guides the development of a system whose member agents communicate via messaging based on protocols. Thus, the quality of a multiaent system and its effectiveness in meeting stakeholder requirements depend on the quality of the protocol on which it is based. It is crucial the protocol be flexible: ideally, it should constrain the agents just enough (so they coordinate properly) but no more than necessary (so they can exercise their autonomy), both aspects essential to achieving what stakeholders require.

As an example, consider a protocol *Sale* (formalized shortly) that involves the roles of buyer, seller, bank, and shipper and supports exchanging payment for goods. What makes *Sale*—or any protocol—correct? *Safety* means that no enactment of *Sale* violates integrity. *Liveness* means that any enactment can complete, i.e., if the agents want it to. In addition, *Sale* should satisfy stakeholder requirements. For example, the stakeholders may require that if payment happens but items are not delivered, then a refund is issued. Or, if payment happens, it precedes delivery.

The kind of multiagent system we have in mind is *decentralized*, meaning there is **no** reliance on a central decision maker or store of global state. Its member agents may be instantiated on the same or different machines—we don't require or forbid either—and its correctness should not be affected by such variations. Ideally, an agent waits for another only if it relies on information to be obtained from the other or if it is beneficial for it to wait. Thus, to achieve decentralization, we cannot rely on synchronous or order-preserving communication channels.

Information protocols avoid the limitations of traditional protocol representations [Chopra *et al.*, 2020] by not demanding shared state or order-preserving communication. Combining a flexible semantics over asynchrony leads to a protocol having a large number of enactments: potentially, exponential in its size. For example, *Sale* has 639 enactments, which are too many to verify manually.

We introduce *Mambo*, an approach for verifying information protocols against requirements specified as path queries. We (i) introduce a language to encode requirements; although the language is simple, we highlight its effectiveness in capturing interesting requirements patterns for protocols; (ii) show how to verify these requirements in a semantic tableau generated from a protocol, and (iii) for purposes of efficiency, show how to generate a small model (tableau) that maintains coverage for verifying a given requirement.

Mambo's novelty lies in that it is the first general-purpose verification approach for information protocols. Its significance lies in that information protocols is a paradigm of growing importance [Chopra and Christie, 2023] in engineering multiagent systems using programming models geared to different approaches for implementing agents [Christie *et al.*, 2023; Christie *et al.*, 2022; Baldoni *et al.*, 2025; Chopra *et al.*, 2025]. More fundamentally, Mambo's significance lies in the fact that it helps modularize the engineering of multiagent systems by simplifying the engineering of member agents given a correct protocol.

Organization. Section 2 describes the related work on protocols. Section 3 introduces information protocols and our language for expressing protocol requirements. Section 4 presents a variety of requirement patterns. Section 5 shows how to construct a tableau for a protocol and how to verify requirements with it. Section 6 shows how to construct an equivalent smaller tableau by exploiting the structure of the requirement being verified. Section 7 describes our implementation and presents performance results. Section 8 makes some methodological observations and discusses future work.

2 Related Work: Interaction Protocols

Commitments capture the meaning of an interaction [Yolum, 2007] and thus express multiagent requirements [Chopra *et al.*, 2014]. Fornara and Colombetti [2003] define interaction protocols via UML sequence diagrams but give soundness conditions that depend on the meanings of speech acts; Yolum [2007] formalizes properties related to liveness, consistency, and robustness of commitment protocols; Günay *et al.* [2015] dynamically generate protocols given commitments and goals; Kalia and Singh [2015] generate commitment protocols from interaction scenarios; Chopra *et al.* [2014] give rules for refining requirements into commitments. These works address requirements, but not operational challenges, such as those due to decentralization (distributed knowledge and asynchronous communication). Winikoff [2007] describes some difficulties in implementing commitment-based interactions in asynchronous settings, especially those involving more than two agents.

Tooling for operational protocols is valuable. Rooney *et al.* [2004] propose a graphical editor for protocols expressed as sequence diagrams in Agent UML (AUML) [Huget and Odell, 2004]. Winikoff *et al.* [2018] defines a hierarchical state-machine notation and model for information protocols that captures state transitions graphically. These works offer no support for verification. MAS engineering methodologies, e.g., Prometheus [Padgham and Winikoff, 2005] and Bliss [Singh, 2014], offer informal guidance for engineering protocols. Ferrando *et al.* [2019] provide a valuable analysis of protocol enactability in decentralized settings under increasingly strong message ordering assumptions. Information protocols are enactable under the weakest possible assumption, no message ordering, thus making verification nontrivial.

Mazouzi *et al.*'s [2002] approach to protocol verification brings forth key themes, including the use of formal methods and evaluation of various correctness properties. Though this work is strong for its era, Mambo differs from it in important respects: (1) Mambo provides a formal specification language for the requirements to be verified. (2) Mambo works on fully declarative information protocols, whose semantics is natively asynchronous, whereas they assume the restrictive sequence diagram notation of AUML. They support asynchronous messages but not order-free channels, as we do. (3) Mambo incorporates heuristics based on the structure underlying an information protocol to carry out heavy partial order reduction whereas they do not support such reductions, which would make their work intractable.

Singh [2012] and Singh and Christie [2021] are formal approaches for verifying information protocols based on satisfiability (SAT) solvers and semantic tableaux, respectively. The latter, Tango, is much faster than the former on account of its partial order reduction. Both approaches are limited to Boolean properties of a protocol, especially safety and liveness. Mambo is closer in spirit to Tango in adopting semantic tableaux. However, it goes beyond previous approaches by supporting a temporal specification language and heuristics to enable incremental tableau construction while progressing a query representation.

3 Background

We now introduce information protocols and our language for expressing stakeholder requirements.

3.1 Information Protocols

The Blindingly Simple Protocol Language (BSPL) [Singh, 2011a] is a language for specifying *information protocols*, i.e., protocols based on causality and integrity constraints based on the information exchanged. BSPL captures asynchrony and flexibility better than traditional approaches [Chopra *et al.*, 2020]. An information protocol is a bag of messages, each of which specifies a sender, a receiver, a message name, and parameters. Listing 1 gives a protocol *Sale* in BSPL. Message *offer*'s sender and receiver are SELLER and BUYER, respectively, and its parameters are ID, item, and price. A protocol enactment (a set of correlated messages) is identified by the bindings of the specified ⌜key⌝ parameters. Parameter ID constitutes *Sale*'s key.

The parameter adornments ⌜in⌝, ⌜out⌝, and ⌜nil⌝ capture information causality (dependencies) and determine the messages that are *enabled* for emission by an agent relative to an enactment. Specifically, a message is enabled if each ⌜in⌝ parameter (its binding) is *known* from (exists in) the agent's *local state* (set of sent and received messages), and no ⌜out⌝ or ⌜nil⌝ parameter is known. Because all of *offer*'s parameters are ⌜out⌝, it is enabled for SELLER at the start of the enactment. Message *rescind* (pulling back the *offer*) is enabled for SELLER provided it has already sent or received messages that make the parameters ID, item, and price known and leave bail, payinfo, and instruction unknown.

To send an enabled message, the agent fleshes out the message by generating bindings for only its ⌜out⌝ parameters via its internal logic. For example, to send *offer*, SELLER must generate bindings for all its parameters; to send *rescind*, it must generate a binding for bail but not for payinfo or instruction. The internal logic may differ across agents and not be revealed to others. The emission of a message records it in the local state, making the bindings of its ⌜out⌝ parameters known. An agent may receive a message whenever it arrives. The received message is recorded in the local state, making the bindings of its parameters known to the agent.

Information protocols support multiple agents sending or receiving messages concurrently. Given the causal semantics, any concurrent enactment is equivalent to a sequential one in which the send ('!') and receive ('?') events at different agents are interleaved, the only restriction being that the reception of a message follows its emission. A possible enactment of *Sale* is ⟨Seller!offer, Buyer?offer, Buyer!reject, Seller?reject⟩, ignoring message parameters.

3.2 Query Language

We express protocol requirements in *Precedence*, a propositional logic with one temporal operator indicating *occurs (sometime) before*. We adopt Singh's [2012] formalization.

The atoms of Precedence are events. Below, e and f are events. If e is an event, its complement $\overline{e}$ is also an event. Precedence treats e and $\overline{e}$ on par. The term $e \cdot f$ means that e occurs prior to f. The Boolean operators: '$\vee$' and '$\wedge$' have

Listing 1 A BSPL protocol for conducting ebusiness.

```
1: Sale {
2: roles Buyer, Seller, Bank, Shipper
3: parameters out ID key, out item, out
       price, out outcome
4: private address, decision, payinfo,
       instruction, authcode, choice,
       acc, bail
5:
6: Seller ↦ Buyer: offer[out ID key, out
       item, out price]
7: Buyer ↦ Seller: accept[in ID key, in
       item, in price, out address, out
       acc, out decision]
8: Buyer ↦ Seller: reject[in ID key, in
       item, in price, out decision, out
        outcome]
9:
10: Seller ↦ Buyer: rescind[in ID key, in
        item, in price, out bail, nil
       payinfo, nil instruction]
11:
12: Buyer ↦ Seller: rescindAck[in ID key,
        in item, in price, in bail, nil
       authcode, out outcome]
13:
14: //Buyer authorizes Bank if not
       rescinded
15: Buyer ↦ Bank: pay[in ID key, in price
       , nil bail, in acc, out authcode]
16: Bank ↦ Seller: transfer[in ID key,
       in price, in authcode, out
       payinfo]
17:
18: //Seller either instructs Shipper to
       ship or bank to refund
19: Seller ↦ Shipper: ship[in ID key, in
       item, in address, in payinfo, out
        instruction, out choice]
20:
21: Seller ↦ Bank: refund[in ID key, in
       item, in payinfo, out choice, out
        outcome]
22: Shipper ↦ Buyer: deliver[in ID key, in
        item, in address, out outcome]
23: }
```

the usual meanings. The syntax (Figure 1) follows the conjunctive normal form.

$$\begin{array}{rcl} I & \longrightarrow & \mathit{clause} \mid \mathit{clause} \wedge I \\ \mathit{clause} & \longrightarrow & \mathit{term} \mid \mathit{term} \vee \mathit{clause} \\ \mathit{term} & \longrightarrow & \mathit{event} \mid \mathit{event} \cdot \mathit{event} \end{array}$$

Figure 1: Verification query language syntax.

The semantics of Precedence is given by runs of events (instances). Let Γ be a set of events where $e \in \Gamma$ if and only if $\overline{e} \in \Gamma$. A run is a function from the natural numbers to the power set of Γ, i.e., $\tau : \mathbb{N} \mapsto 2^{\Gamma}$. The i^{th} index of τ, $\tau_i = \tau(i)$. The length of τ is the first index i at which $\tau(i) = \emptyset$ (after which all indices are empty sets). We say τ is empty if $|\tau| = 0$. The subrun from i to j of τ is notated $\tau_{[i,j]}$. Its first $j - i + 1$ values are extracted from τ and the rest are empty, i.e., $\tau_{[i,j]} = \langle \tau_i, \tau_{i+1} \ldots \tau_{j-i+1} \ldots \emptyset \ldots \rangle$. On any run, e or $\overline{e}$ may not both occur. Events are nonrepeating.

Here, $\tau \models_i E$ means that τ satisfies E at i or later and τ is a *model* of query E if and only if $\tau \models_0 E$. E is *satisfiable* if and only if it has a model. Below, r and u are formulas.

S_1 $\tau \models_i e$ if and only if $(\exists j \geq i : e \in \tau_j)$
S_2 $\tau \models_i r \vee u$ if and only if $\tau \models_i r$ or $\tau \models_i u$
S_3 $\tau \models_i r \wedge u$ if and only if $\tau \models_i r$ and $\tau \models_i u$
S_4 $\tau \models_i e \cdot f$ if and only if $(\exists j \geq i : \tau_{[i,j]} \models_0 e$ and $\tau_{[j+1,|\tau|]} \models_0 f)$

The semantics justifies applying De Morgan's laws and treating complementation as negation. That is, we can complement a query since it can be reduced to one where only atoms are complemented: (1) $\overline{\overline{e}} \equiv e$; (2) $\overline{r \vee u} \equiv \overline{r} \wedge \overline{u}$; (3) $\overline{r \wedge u} \equiv \overline{r} \vee \overline{u}$; and (4) $\overline{e \cdot f} \equiv \overline{e} \vee \overline{f} \vee (f \cdot e)$.

In surface syntax, we accept 'and' for '$\wedge$'; 'or' for '$\vee$'; 'before' for '$\cdot$'; and '-' and 'no' for overline (complementation). (The binding order is complementation $> \cdot > \wedge > \vee$.)

4 Requirement Patterns

Sale, the protocol in Listing 1, is complex; the purpose of the present paper is to facilitate verifying whether an information protocol meets stakeholder requirements. We distinguish two kinds of requirements. Structural requirements apply to any protocol. Stakeholder requirements are protocol-specific.

The requirements are expressed using events such as the occurrence of a message, e.g., offer; a parameter being bound, e.g., item, and the observation of a message or parameter binding by an agent, e.g., Seller:offer and Bank:authcode.

4.1 Structural Properties

These protocol-agnostic requirements establish the well-formedness of a protocol. Failing these patterns often indicates other problems in the conception of a protocol.

Pattern 1 (Liveness). Liveness captures the idea that all enactments can progress to completion, i.e., all parameters in a protocol (except those declared private) (Line 3: in *Sale*) must be known. A protocol is *live* if and only if in every enactment, all its completion parameters are known. *Desired:* ID $\wedge$ item $\wedge$ price $\wedge$ outcome is true for all enactments.

Pattern 2 (Safety). Safety captures the lack of *integrity* violations in a protocol's enactments. A protocol is *safe* if and only if in every enactment, a parameter is bound at most once. To determine safety, for every pair of messages in which a common parameter is $\ulcorner$out$\urcorner$, we check that at most one of the messages in the pair occurs in any enactment. *Desired:* (refund $\wedge$ deliver) $\vee$ (rescindAck $\wedge$ refund) $\vee$ (rescindAck $\wedge$ deliver) $\vee$ (reject $\wedge$ refund) $\vee$ (reject $\wedge$ deliver) is false on all enactments.

Mambo automatically generates queries for Liveness and Safety from a protocol.

4.2 Stakeholder Requirements

In general, even if a protocol were structurally acceptable, it could fail to serve its stakeholders' needs.

Pattern 3 (Desired End). Any *Sale* enactment must end in one of the following states: (i) *offer* is rescinded by SELLER; (ii) *offer* is rejected by BUYER; (iii) BUYER and SELLER exchange payment for goods; and (iv) BUYER pays but SELLER refunds. *Desired:* rescindAck ∨ reject ∨ transfer ∧ (refund ∨ deliver) is true for all enactments.

Pattern 4 (Complementary). Some pairs of semantically "opposite" messages, e.g., *deliver* and *reject*, must be mutually exclusive. *Desired:* deliver ∧ reject is false on all enactments.

Pattern 5 (Concede). *Sale* illustrates an interesting design involving semantically "opposite" messages: SELLER's *rescind* (end) and BUYER's *pay* (proceed). Since they can occur concurrently, a conflict between them would be unsafe. Therefore, they are expressed such that if BUYER sends *pay*, the transaction proceeds (*pay* is on a path to completion), but BUYER cannot send *pay* once it has received *rescind*. To ensure liveness, BUYER can send *rescindAck* once it has received *rescind* to allow completion on the non-*pay* path. *Desired:* pay · Buyer:rescind ∨ (no rescind ∧ no rescindAck) ∨ rescind · rescindAck is true on all enactments.

Pattern 6 (Late Action). Here, *rescind* disables *accept*; that is, *accept* cannot occur after *rescind*. *Desired:* no accept ∨ no rescind ∨ accept · Buyer:rescind is true on all enactments.

Pattern 7 (Flexibility). Suppose *accept* means a commitment from BUYER to SELLER that if *deliver* occurs, then *transfer* will occur. In conjunction with Pattern 9, this pattern ensures a flexible exchange of payment and goods. *Desired:* accept · deliver ∧ deliver · transfer is true on some enactments.

Pattern 8 (Compensation). There may be a compensation commitment to *refund* BUYER in case SELLER violates its *offer* commitment. Informally, if *transfer* has occurred, then either *deliver* or *refund* occurs; *refund* implies *transfer*; and *refund* and *deliver* are mutually exclusive. *Desired:* (no transfer ∨ deliver ∨ refund) ∧ (no refund ∨ transfer) ∧ (no refund ∨ no deliver) is true on all enactments.

Pattern 9 (Create-Detach-Discharge). Suppose *offer* creates a commitment from SELLER to BUYER that if *accept* and *transfer* both occur (the detach condition), then *deliver* will occur (the discharge condition). Given this commitment, at the operational level, a simple happy path is that the commitment is created, detached, and discharged in that order. *Desired:* offer · accept ∧ offer · transfer ∧ accept · deliver ∧ transfer · deliver is true for some enactment.

Pattern 10 (Delegation Guarantee). BUYER relies upon BANK to pay SELLER via a *transfer*, i.e., a delegation [Singh, 1999]. Stakeholders may require that *pay* ensures *transfer* and must precede it. *Desired:* no pay ∧ no transfer ∨ pay · transfer is true on all enactments.

4.3 Toward a Catalog of Requirement Patterns

Table 1 shows how our patterns capture various design needs. It shows how our patterns could form the basis for a catalog of patterns that a protocol designer could draw upon in eliciting requirements from stakeholders.

Criterion	Relevant Patterns
Progress	Liveness means the protocol doesn't prevent the agents from completing the interaction
Integrity	Safety means the protocol doesn't allow integrity violations
Goal state	Desired End captures a stakeholder's goals
Operation	Complementary, Concede, and Late Action avoid or deal with conflicting actions
Contract	Flexibility gives options and Compensation captures semantic recovery from a violation
Trust	Create-Detach-Discharge and Delegation Guarantee indicate caution in interacting

Table 1: Our patterns address important protocol design criteria.

5 Formalization Background

We adopt Singh and Christie's [2021] tableau framework. A *configuration* corresponds to a state during the enactment of a protocol. Each protocol enactment goes through a series of configurations where the transition respects causality and no role emits a message that would locally cause integrity violation. Initially, no observations have occurred and each role knows all and only the ⌜in⌝ parameters of the protocol.

A semantic tableau [D'Agostino *et al.*, 1999; Fitting, 1999] is a proof tree. Here, the nodes of the tree are configurations. The inference rules determine how one node leads to another node. Thus, each *path*—sequence of nodes beginning from the root—of a tableau is a protocol enactment. We adopt propositional logic with its usual rules. Below, m is 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 respectively adorned ⌜in⌝, ⌜out⌝, and ⌜nil⌝. The modalities K_x and U_x capture that role x knows or does not know parameter bindings, and L_x that x observes the emission or reception of a message.

5.1 Knowledge and Observations

What is known to each role grows monotonically because parameter bindings are immutable, and the emission and reception of each message add to a role's knowledge.

$L_x m$ means role x has observed message m. In BSPL, the only observations are local, i.e., emissions or receptions. Chaining back a role's observations to the root gives us its history in reverse. This is crucial in relating nodes to history vectors and tableaux to sets of history vectors [Singh, 2012]. The $K_x p$ assertions in a tableau node together characterize a protocol configuration. U_x expresses which bindings are (so far) unknown to x. Each transition involves an observation and the associated increase in the knowledge of the observing role. The BSPL treatment of ⌜out⌝ and ⌜nil⌝ parameters at emission relies upon their bindings being unknown. To capture this increase in knowledge, we delete $U_x p$ assertions simultaneously with when we infer $K_x p$ assertions.

Figure 2 summarizes the core BSPL semantics [Singh and Christie, 2021]. SND says that 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, the sender produces and comes to know the bindings for $\vec{p}_O$, and the message enters the communication channel from its

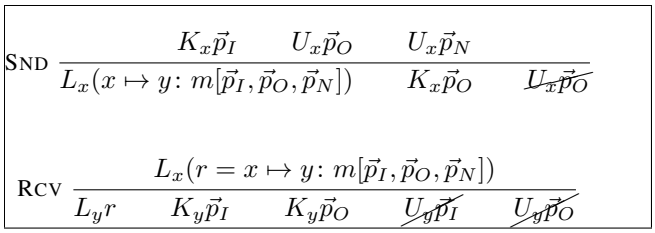

Figure 2: Message emission and reception in BSPL [Singh and Christie, 2021].

sender to its receiver. RCV states that a message from x to y is enabled for reception by y if and only if x has observed sending that message. Concomitantly, y comes to know the bindings for $\vec{p}_I$ and $\vec{p}_O$.

5.2 Verifying Correctness Properties

The properties of interest concern paths unfolding as specified and reaching a specified configuration. We assert a property at the root of the tableau. A path ends when no more observations are enabled. A consistent path that ends provides an example of the property at the root. A path that hits a contradiction is *closed*; a tableau closes if all its paths close, which indicates the property is inconsistent and thus its negation is proved. The foregoing argument works because each path of a tableau, as generated from the SEND and RECEIVE rules above, respects the causal structure of BSPL and local consistency when a role sends a message.

5.3 How Messages Relate in a Protocol

Table 2 summarizes direct enablement and disablement relationships between observations based on what parameters occur in them [Singh and Christie, 2021]. It is possible syntactically to have a protocol with two messages that both enable and disable each other.

	$\ulcorner\mathsf{in}\urcorner\,\mathbf{p} \in b$	$\ulcorner\mathsf{out}\urcorner\,\mathbf{p} \in b$	$\ulcorner\mathsf{nil}\urcorner\,\mathbf{p} \in b$
$\ulcorner\mathsf{in}\urcorner\,\mathbf{p} \in a$	$x?a \vdash y!b$	$x?a \dashv y!b$	$x?a \dashv y!b$
$\ulcorner\mathsf{out}\urcorner\,\mathbf{p} \in a$	$x\,?a \vdash y!b$	$x\,?a \dashv y!b$	$x\,?a \dashv y!b$
$\ulcorner\mathsf{nil}\urcorner\,\mathbf{p} \in a$	–	–	–

Table 2: Direct enablement or disablement by a of b. The interrobang, as in $x\,?a$, indicates that x may either emit or receive a; b is always an emission. Here, $p \in m$ indicates parameter p occurs in message schema m; x and y may be the same or different roles.

6 Mambo Formalization

A naively generated tableau faces a combinatorial explosion because it encodes each possible interleaving of events separately. We generate a small tableau that covers all possibilities implicitly via heuristics that reflect the causality and integrity constraints of a protocol and our query language.

Specifically, we combine interleavings that are equivalent in that they exhibit the same causal relationships and provide the same options for integrity preservation and violation. (1) If one message is a necessary enabler of another, the tableau would unfold in the correct order and the wrong interleaving is impossible. (2) If two messages may interfere (e.g., one message may disable another), we must include both possibilities in the tableau. We need to recognize such interleavings even if one of the messages is not presently enabled and make sure we do not prematurely eliminate viable paths from the tableau. (3) If the truth of a query depends on the interleavings, we must include a representative for each interleaving in the tableau. (4) If two messages are causally independent and don't interfere, we can retain one of the interleavings and generate a path for it.

We refine Singh and Christie's [2021] *tangles with* construct. Observation a *tangles with* observation c, if and only if (1) a directly disables c or (2) a directly disables a message b where b enables c. Two observations are *incompatible* if and only if at least one is an emission and tangles with the other. Like Tango, the rest of the construction builds an undirected graph of incompatible observations and computes an approximate vertex cover to find sets of compatible observations. Each such set corresponds to a logically distinct path that effectively stands in for each permutation of these observations. A key difference from Tango is that this graph is computed at each tableau node in light of the current reduced query, which is also reduced at each step as events occur.

7 Implementation and Results

Our implementation is an extension of Tango, and proceeds as follows: (1) Create a tree representation of the requirement query. (2) Use the parameters of each clause in the query to construct a conflict graph of parameters whose order of occurrence in an enactment matters. (3) Construct an incompatibility graph from the protocol, reflecting conflicts due to parameter adornments as well as the query. (4) Expand a tableau, with incompatible events placed on different paths. (5) Filter the paths based on the query: paths where the query resolves to true or false terminate and the remaining paths are retained for further expansion (in the previous step).

Tango doesn't support flexible queries. And, for the properties that it handles—safety and liveness—Tango doesn't close paths early if the property is satisfied. By contrast, Mambo incorporates the query in each step of the tableau expansion, especially in the incompatibility graph generation (significant for queries that contain 'before' relationships). And, it applies query-specific optimizations and returns paths as soon as a query resolves to true or false—termed "short-circuiting" below.

7.1 Incremental Query Reduction

We model a property based on when it first becomes true on a path, as follows: (1) `None`: the property is indeterminate; (2) `False`: the property is violated; (3) ∞: the event occurs eventually; and (4) $n \in \mathbb{N}^+$: the property is true upon the nth event. We evaluate a query tree as described next.

Occurs The leaf nodes of a query, checking when an event occurs. These nodes return the event's index in the enactment if it has occurred, otherwise None.

Or Returns the minimum of two integers, or otherwise prefers ∞ over `None` over `False`.

And Prefers `False` over `None` over the maximum of its arguments (including ∞).

Not Returns ∞ if its argument is `False` or `None`, and `False` if its argument is ∞ or an `int`.

Before Returns the first argument that is not an `int`, the second argument if they are ordered, else `False`.

7.2 Optimizations and Heuristics

We implement three important optimization heuristics:

Pruning Stop extending a path once the query resolves to a definite `int` or `False`, since further simulation would not change the result.

Residuation Save the results of a query subtree as soon as they resolve to a definite `int` or `False`, avoiding reevaluation on that path of the tableau. Other paths are not affected.

Incremental `Occurs` nodes don't scan the entire path, but look at the newest event for relevant information and save the results using residuation.

7.3 Experimental Setup

We evaluated Mambo's performance using a framework that generates every combination of the setup parameters, including protocol, query, method, and iteration ID (so the experiment runs a specific number of times).

We use Tango as a baseline since it is the best prior method for verifying information protocols. We used the latest version (taken from its public repository) of the Tango implementation and protocol specifications, so that all of the methods are run on the same hardware over the same protocols.

We created a second baseline, dubbed Tango+, as a hybrid of Tango and Mambo. Tango+ generates all maximal enactments, minus partial order reduction, using the traditional Tango method, then filters them using the Mambo query system without any of the optimizations enabled. Thus, Tango+ is expected to have a nearly worst-case performance, since it does not do any short-circuiting. The exceptions are cases where the protocol is trivially safe. Mambo constructs a query for potential conflicts, and a lack thereof proves safety before checking any enactments. Since Tango+ uses Mambo's queries, it also short-circuits on trivially safe protocols.

Our experiments were performed on an AMD Ryzen 7 6800U with 16GB of RAM.

7.4 Results

To compare against prior results, we performed one experiment verifying only safety and liveness for those protocols analyzed by Tango [Singh and Christie, 2021]. This evaluation is limited to liveness and safety verification since the baselines are limited to these properties.

In addition, we computed Cohen's [1988] d statistic for effect size, which is defined as the difference of means divided by the sum of standard deviations. We obtained strong effect sizes relative to Tango, with the worst case being 1.4 and the best being 29.5. The results are more varied with Tango+, which mostly performs worse than Tango, except where the protocol is trivially safe, and beats Mambo in one case. Details are in the appendix.

Protocol	Property	Mambo	Tango	Tango+
Block-Contra	liveness	9.509	13.466	13.392
–	safety	9.100	10.081	13.157
CreateOrder	liveness	51.255	55	601.023
–	safety	352.241	573.181	591.965
Independent	liveness	9.183	9.983	10.264
–	safety	7.244	10.181	7.259
NetBill	liveness	90.383	150.492	147.127
–	safety	130.463	147.179	150.924
PO Pay…Ship	liveness	15.576	27.884	27.463
–	safety	10.475	27.529	10.230
Sale	liveness	34.038	50.251	48.056
–	safety	43.971	48.369	51.718

Table 3: Mean times over 10 runs for safety and liveness checking in ms. The appendix provides standard deviations and effect sizes.

Next, we evaluate Mambo's efficiency with respect to the requirement patterns described in this paper. For practical purposes, it helps to include a semantic check for the brevity or minimality of a protocol. A *deadwood* message in a protocol is one that is not observed on any enactment. That is, message *m* is deadwood if and only if the corresponding event m is unsatisfiable. Deadwood is not implemented as a query. Instead, Mambo generates the possible enactments and subtracts their events from the set of messages in the protocol; any messages remaining are deadwood.

Table 4 shows that Mambo is able to verify each requirement in under 40 ms. As Table 4 shows, *Sale* does not meet some of the requirements. *Flexibility* is not met because the protocol entertains no enactment in which *deliver* happens before *transfer*. For *Concede*, the Mambo tooling produces a counterexample; specifically, the enactment ⟨Seller!offer, Buyer?offer, Buyer!reject, Seller?reject, Seller!rescind, Buyer?rescind⟩. We could regiment the protocol further to try to prevent the *Concede* counterexample. Specifically, we could introduce ⌜in⌝ outcome in *rescind* so that the reception of *reject* disables the emission of *rescind*. Doing so, however, will not prevent the enactment where these messages are sent concurrently. In this particular case, one could take the alternative view that the counterexample represents a harmless case because the "outcome" of the interaction (rejection of *offer*) is not in doubt. Trying to avoid such cases makes the protocol overly complex for no gain in the quality of the multiagent system produced.

If stakeholders consider *Flexibility* important and want to ensure that *transfer* can happen after *deliver* (but must happen if *deliver* does), *ship* can be modified so that it does not depend on payinfo, and instead has a ⌜nil⌝ bail to prevent shipping and rescinding in the same enactment.

8 Discussion

It is important to support the engineering of protocols to produce decentralized multiagent systems. Information protocols offer a fine-grained view of coordination between agent and enable flexible, asynchronous operationalization. This flexibility is the reason it can be hard to imagine all the operational possibilities afforded by a protocol.

Query on *Sale*	Desired	Met	Mean	Std
Live	All paths	Yes	31.785	0.286
Safe	All paths	Yes	43.623	0.540
Desired End	All paths	Yes	20.931	0.202
Complementary	No path	Yes	25.821	0.301
Concede	All paths	No	20.216	0.255
Late Action	All paths	Yes	17.083	0.147
Flexibility	Some paths	No	23.825	0.222
Compensation	All paths	Yes	35.467	0.943
Create-Detach-Discharge	Some paths	Yes	26.797	0.406
Delegation Guarantee	All paths	Yes	14.248	0.117
Deadwood	No messages	Yes	26.480	0.343

Table 4: Mean times and standard deviations over 10 runs of evaluating various requirements on *Sale*. All times are in ms. Here, *Desired* refers to whether we expect (require) the query to be satisfied on All, Some, or No enactments. *Met* states whether that desire was met. The appendix presents additional statistics.

Traditionally, protocols are presented as finished work based on their designers' intuitions. Instead, we advocate the systematic engineering of protocols as crucial software artifacts in the sense of Osterweil [2008]. That is, protocol design would take place hand-in-hand with the iterative process of acquiring and refining stakeholder requirements [Telang and Singh, 2012]. Unlike in traditional agent-oriented software engineering [Cernuzzi and Zambonelli, 2004; Bresciani *et al.*, 2004] that produce sequence diagrams, in our conception, designers would craft information protocols to achieve flexibility and rigor [Winikoff *et al.*, 2018; Singh, 2014].

Requirements in Mambo capture the constraints on the information (messages or parameters) exchanged and are thus more abstract and simpler than protocols. That is, the requirements are easier to formulate and provide a conceptual basis for evaluating protocols. Like in traditional systems, safety and liveness are prerequisites to proper implementation and have analogs in traditional approaches. Some requirements take a stakeholder's perspective. For example, a seller who doesn't trust a buyer may wish to ensure its agents will be able to delay delivery until payment or rescind an offer. Some requirements go beyond traditional systems. For example, specifying protocols that satisfy the *Concede* pattern is impossible in alternative protocol specification approaches [Chopra *et al.*, 2020]. The flexible nature of information protocols is key to supporting novel requirement patterns. A useful direction would be to build a catalog of such patterns.

For brevity, we elide the discussion of syntactic checks, which despite being simple to understand and efficient to apply, can help avert design flaws. For example, a protocol that has an ⌜out⌝ parameter but no message with that parameter as ⌜out⌝ cannot be live. A private parameter that is ⌜in⌝ on some messages but never ⌜out⌝ indicates a redundant protocol in that those messages are clearly deadwood. And, one that is ⌜out⌝ but not ⌜nil⌝ or ⌜in⌝ can be trivially removed.

Languages for specifying workflows based on propositional temporal logics [Attie *et al.*, 1993; Montali *et al.*, 2010] are potential alternatives to Precedence. Precedence is event-oriented, which lends itself well to our setting: events naturally model message emission and reception and parameter binding. Moreover, Precedence is simpler with only one temporal operator *before*, which captures event order more naturally than the alternative approaches, which are based on *until* and produce unwieldy formulas even for simple requirements [Attie *et al.*, 1993]. Also, Precedence eschews the *next-time* operator, which is ill-defined in decentralized systems.

Mambo brings up interesting directions for future work. One, we might combine its representations with protocol verification tools for other paradigms. In particular, we posit that our query language is well-suited to efficient online evaluation and would facilitate synthesis with runtime verification [Ancona *et al.*, 2021] and monitoring [Dastani *et al.*, 2018]. Likewise, for tools for meaning-based representations [Yolum, 2007; Günay *et al.*, 2015; Dastani *et al.*, 2017], we can investigate automatically generating Mambo queries from commitments and goals [Harland *et al.*, 2017].

Two, we need techniques for verifying protocol-based agents to modularly verify a multiagent system by verifying them separately once the protocol is verified. Doing so would combat the potential high cost of system verification. An effective approach should tackle the fact that an agent would participate in multiple systems; e.g., an agent may play BUYER in *Sale* and a PATIENT in a healthcare protocol, e.g., to buy prescribed drugs.

Three, instead of verifying protocols, we could synthesize protocols from specifications in a higher-level language. Langshaw [Singh *et al.*, 2024] is such a language. Its novel idea is separating the specification of communicative acts and the coordination required between them. Coordination is specified via abstractions for conflicting actions and prioritized *saysos* of roles over parameters in the communicative acts. Langshaw specifications are more compact than the protocols synthesized from them. Mambo suggests an alternative approach for specifying coordination requirements: as formulas in Precedence. That is, it would be interesting to explore protocol synthesis given a specification of communicative acts and a set of requirements. Comparing the resulting approach with Langshaw would help develop intuitions about higher-level languages.

Four, *agentic* models combine generative AI with information and abilities to sense and act, e.g., via web services. Generative AI can help overcome the knowledge engineering bottleneck in building flexible agents. Current agentic frameworks compose agents via workflows [LangGraph, 2024; Dibia *et al.*, 2024]. However, a major and long-recognized shortcoming of workflows is their rigidity [Chrysanthis and Ramamritham, 1994; Singh and Huhns, 1994], which limits agent autonomy and responsiveness to exceptions [Singh, 2011a; Singh, 2012; Singh, 2011b]. Thus powerful agents built with generative AI can be stymied by the inability to act flexibly. Mambo enables engineering MAS based on correct, flexible protocols. An important direction would be to investigate how flexible protocol-based interaction could be applied within agentic models in lieu of workflows.

9 Reproducibility

The appendix and all software, including the protocols mentioned, are available at https://gitlab.com/masr/mambo.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. We thank the US National Science Foundation (grant IIS-1908374) for partial support.

References

[Ancona *et al.*, 2021] Davide Ancona, Luca Franceschini, Angelo Ferrando, and Viviana Mascardi. RML: Theory and practice of a domain specific language for runtime verification. *Science of Computer Programming*, 205:102610, May 2021.

[Attie *et al.*, 1993] Paul C. Attie, Munindar P. Singh, Amit P. Sheth, and Marek Rusinkiewicz. Specifying and enforcing intertask dependencies. In *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB)*, pages 134–145, Dublin, August 1993. Morgan Kaufmann.

[Baldoni *et al.*, 2025] Matteo Baldoni, Samuel H. Christie V, Munindar P. Singh, and Amit K. Chopra. Orpheus: Engineering multiagent systems via communicating agents. In *Proceedings of the 39th AAAI Conference on Artificial Intelligence (AAAI)*, pages 23135–23143, Philadelphia, February 2025. AAAI.

[Boissier *et al.*, 2023] 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.

[Bresciani *et al.*, 2004] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 8(3):203–236, May 2004.

[Casadei *et al.*, 2021] Roberto Casadei, Mirko Viroli, Alessandro Ricci, and Giorgio Audrito. Tuple-based coordination in large-scale situated systems. In *Proceedings of the 23rd IFIP WG 6.1 International Conference (COORDINATION)*, *LNCS* 12717, pages 149–167, Valletta, Malta, June 2021. Springer.

[Cernuzzi and Zambonelli, 2004] 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 & Technology Pubs.

[Chopra and Christie V, 2023] Amit K. Chopra and Samuel H. Christie V. Communication meaning: Foundations and directions for systems research. In *Proceedings of the 22nd International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 1786–1791, London, May 2023. IFAAMAS. Blue Sky Ideas Track.

[Chopra *et al.*, 2014] Amit K. Chopra, Fabiano Dalpiaz, F. Başak Aydemir, Paolo Giorgini, John Mylopoulos, and Munindar P. Singh. Protos: Foundations for engineering innovative sociotechnical systems. In *Proceedings of the 22nd IEEE International Requirements Engineering Conference (RE)*, pages 53–62, Karlskrona, Sweden, August 2014. IEEE.

[Chopra *et al.*, 2020] Amit K. Chopra, Samuel H. Christie V, and Munindar P. Singh. An evaluation of communication protocol languages for engineering multiagent systems. *Journal of Artificial Intelligence Research (JAIR)*, 69:1351–1393, December 2020.

[Chopra *et al.*, 2025] Amit K. Chopra, Matteo Baldoni, Samuel H. Christie V, and Munindar P. Singh. Azorus: Commitments over protocols for BDI agents. In *Proceedings of the 24th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, Detroit, May 2025. IFAAMAS.

[Christie V *et al.*, 2022] Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. Mandrake: Multiagent systems as a basis for programming fault-tolerant decentralized applications. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 36(1):16:1–16:30, April 2022.

[Christie V *et al.*, 2023] 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 Multi-Agent Systems (AAMAS)*, pages 1154–1163, London, May 2023. IFAAMAS.

[Chrysanthis and Ramamritham, 1994] Panos K. Chrysanthis and Krithi Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, September 1994.

[Cohen, 1988] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 2nd edition, 1988.

[D'Agostino *et al.*, 1999] Marcello D'Agostino, Dov M. Gabbay, Reiner Hähnle, and Joachim Posegga, editors. *Handbook of Tableau Methods*, Dordrecht, Netherlands, 1999. Kluwer.

[Dastani *et al.*, 2017] Mehdi Dastani, Leendert W. N. van der Torre, and Neil Yorke-Smith. Commitments and interaction norms in organisations. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 31(2):207–249, March 2017.

[Dastani *et al.*, 2018] Mehdi Dastani, Paolo Torroni, and Neil Yorke-Smith. Monitoring norms: A multi-disciplinary perspective. *Knowledge Engineering Review*, 33:e25, December 2018.

[Dibia *et al.*, 2024] Victor Dibia, Jingya Chen, Gagan Bansal, Suff Syed, Adam Fourney, Erkang Zhu, Chi Wang, and Saleema Amershi. AutoGen Studio: A no-code developer tool for building and debugging multi-agent systems. *CoRR*, abs/2408.15247, August 2024.

[Ferrando *et al.*, 2019] 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)*, *LNCS* 12058, pages 43–64, Montréal, May 2019. Springer.

[Fitting, 1999] 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, Dordrecht, Netherlands, 1999.

[Fornara and Colombetti, 2003] Nicoletta Fornara and Marco Colombetti. Defining interaction protocols using a commitment-based agent communication language. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 520–527, Melbourne, July 2003. ACM Press.

[Günay *et al.*, 2015] Akın Günay, Michael Winikoff, and Pınar Yolum. Dynamically generated commitment protocols in open systems. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 29(2):192–229, 2015.

[Harland *et al.*, 2017] James Harland, David N. Morley, John Thangarajah, and Neil Yorke-Smith. Aborting, suspending, and resuming goals and plans in BDI agents. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 31(2):288–331, March 2017.

[Huget and Odell, 2004] Marc-Philippe Huget and James Odell. Representing agent interaction protocols with agent UML. In *Proceedings of the 5th International Workshop on Agent-Oriented Software Engineering (AOSE)*, *LNCS* 3382, pages 16–30, New York, July 2004. Springer.

[Kalia and Singh, 2015] Anup K. Kalia and Munindar P. Singh. Muon: Designing multiagent communication protocols from interaction scenarios. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 29(4):621–657, July 2015.

[LangGraph, 2024] LangGraph: Building language agents as graphs, December 2024. https://langchain-ai.github.io/langgraph/. Accessed 2024-12-05.

[Mazouzi *et al.*, 2002] Hamza Mazouzi, Amal El Fallah Seghrouchni, and Serge Haddad. Open protocol design for complex interactions in multi-agent systems. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 517–526, Bologna, July 2002. ACM Press.

[Montali *et al.*, 2010] Marco Montali, Maja Pesic, Wil M. P. van der Aalst, Federico Chesani, Paola Mello, and Sergio Storari. Declarative specification and verification of service choreographies. *ACM Transactions on the Web (TWEB)*, 4(1):3:1–3:62, 2010.

[Osterweil, 2008] Leon J. Osterweil. What is software? *Automated Software Engineering*, 15(3–4):261–273, 2008.

[Padgham and Winikoff, 2005] 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.

[Rooney *et al.*, 2004] 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*, *LNCS* 2949, pages 279–293, Pisa, February 2004. Springer.

[Singh and Christie V, 2021] 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.

[Singh and Huhns, 1994] Munindar P. Singh and Michael N. Huhns. Automating workflows for service order processing: Integrating AI and database technologies. *IEEE Expert*, 9(5):19–23, October 1994.

[Singh *et al.*, 2024] Munindar P. Singh, Samuel H. Christie V, and Amit K. Chopra. Langshaw: Declarative interaction protocols based on sayso and conflict. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 202–210, Jeju, Korea, August 2024. IJCAI.

[Singh, 1999] Munindar P. Singh. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7(1):97–113, March 1999.

[Singh, 2011a] 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.

[Singh, 2011b] 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.

[Singh, 2012] 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.

[Singh, 2014] Munindar P. Singh. Bliss: Specifying declarative service protocols. In *Proceedings of the 11th IEEE International Conference on Services Computing (SCC)*, pages 235–242, Anchorage, Alaska, June 2014. IEEE.

[Telang and Singh, 2012] Pankaj R. Telang and Munindar P. Singh. Specifying and verifying cross-organizational business models: An agent-oriented approach. *IEEE Transactions on Services Computing (TSC)*, 5(3):305–318, 2012.

[Winikoff *et al.*, 2018] 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.

[Winikoff, 2007] 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.

[Yolum, 2007] Pınar Yolum. Design time analysis of multiagent protocols. *Data and Knowledge Engineering*, 63(1):137–154, October 2007.