Tango: Declarative Semantics for Multiagent Communication Protocols

Munindar P. Singh\textsuperscript{1} and Samuel H. Christie V\textsuperscript{1,2}
\textsuperscript{1}Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA
\textsuperscript{2}School of Computing and Communications, Lancaster University, Lancaster LA1 4WA, UK
mpsingh@ncsu.edu, schrist@ncsu.edu

Abstract

A flexible communication protocol is necessary to build a decentralized multiagent system whose member agents are not coupled to each other’s decision making. Information-based protocol languages capture a protocol in terms of causality and integrity constraints based on the information exchanged by the agents. Thus, they enable highly flexible enactments in which the agents proceed asynchronously and messages may be arbitrarily reordered. However, the existing semantics for such languages can produce a large number of protocol enactments, which makes verification of a protocol property intractable.

This paper formulates a protocol semantics declaratively via inference rules that determine when a message emission or reception becomes enabled during an enactment, and its effect on the local state of an agent. The semantics enables heuristics for determining when alternative extensions of a current enactment would be equivalent, thereby helping produce parsimonious models and yielding improved protocol verification methods.

1 Introduction

We consider a decentralized multiagent system in which autonomous agents communicate through message passing. Such multiagent systems find natural application in settings such as commerce and the Internet of Things where multiple parties interoperate autonomously with minimal coupling.

A crucial challenge is to specify, verify, and implement multiagent systems in terms of high-level concepts such as norms and other organizational constructs [Günay et al., 2015; Jiang et al., 2015; El Menshawy et al., 2011]. However, in a decentralized setting, the high-level concepts can become entwined with details of message flow. In this context, a data-driven approach [Montali et al., 2014] can unite meanings with operations and processing [De Masellis et al., 2017].

We adopt BSPL (the Blindingly Simple Protocol Language) [Singh, 2011], which precisely expresses the information to be exchanged between agents. Thus, it yields enactments as flexible as possible given causal and integrity constraints that naturally encode meanings [Chopra et al., 2020]. However, the current BSPL semantics [Singh, 2012] and its variants produce a combinatorial explosion of enactments, which exacerbates the complexity of formal verification.

Let us describe the problem and solution informally. An enactment (of a protocol) is what may happen when it is instantiated—which messages are sent and received in what order by what participant. Consider a Purchase protocol. A role BUYER sends PO (a purchase order specifying item and price) to a role SELLER, who responds to PO with ship (of that item). BUYER sends pay of price in PO. Notice that ship and pay are independent but each depends on PO. Figure 1 shows some enactments of Purchase for a single transaction.

![Figure 1: Some possible equivalent enactments of Purchase.](image)

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

To capture decentralization, each enactment is modeled as a vector of histories, one history per role [Singh, 2012]. We term two enactments equivalent if they reflect the “same” information and decision making at each role. Representing multiple equivalent enactments doesn’t enhance the ability to verify protocol properties but does exacerbate the complexity.

Multiple equivalent enactments corresponding to a protocol can arise for three main reasons. One, flexibility. Figure 1’s enactments differ only with respect to the ordering of pay and ship—precisely the ordering that is irrelevant in the protocol. From BUYER’s perspective, PO must precede ship and pay but their order of occurrence is irrelevant. From SELLER’s perspective, PO must precede ship but how ship and pay are ordered is irrelevant. Two, the infrastructure. For example, under FIFO, SELLER would observe PO before pay, but either order is acceptable without FIFO. Three, the architecture. Specifically, decentralization means there is no
central controller that can force an agent to observe messages in a particular order.

To verify the correctness of a protocol, one would check properties such as liveness and safety by generating a model from the protocol and verifying the property in the model. Flexible protocols with many possible enactments may produce a large model that is consequently hard to verify.

We propose Tango, an approach for a protocol semantics that identifies how messages tangle with each other and produces a “small” model for a protocol that contains all the relevant variety exhibited by its enactments. The message entanglements help us determine when two enactments are semantically equivalent (satisfy or violate the same properties), and provide a basis for reducing redundancy in a model.

Our contributions are as follows: First, a new declarative semantics for protocols. Second, an algorithm for generating small models of protocols, which ensures correctness but not minimality of the models. We show empirically that our algorithm produces smaller models and verifies correctness of several sample protocols much faster than in previous work.

2 Background: Information-Based Protocols

We now introduce the core ideas of information-based protocols using BSPL [Singh, 2011]. Listing 1 shows an example protocol. Each protocol involves roles (here, B and S); these roles are adopted by agents when the protocol is enacted. The parameters specify the information to be computed: think of these as forming a relational schema, each instance of which corresponds to a complete enactment of the protocol.

Listing 1: Protocol for purchase with cancellation.

```plaintext
protocol PO Pay Cancel Ship {
roles B, S
parameters out ID key, out item, out price, out outcome
private pDone, gDone, rescind

B → S: PO [out ID key, out item, out price]  
B → S: cancel [in ID key, nil pDone, nil gDone, out rescind]  
B → S: pay [in ID key, in price, in item, out pDone]  
S → B: ship [in ID key, in item, nil rescind, out gDone]  
S → B: cancelAck [in ID key, in rescind, nil gDone, out outcome]  
S → B: payAck [in ID key, in pDone, out outcome]  
}
```

BSPL specifies interactions in terms of causality and integrity constraints. Message emissions are causally constrained by dependencies between their parameters. Each message schema lists parameters adorned with “in”, “out”, or “nil”, and at least one “key”. The sender must already know the binding for each “in” parameter, not know the binding for any “out” or “nil” parameter, and create a binding for each “out” but not for any “nil” parameter.

To send PO, B introduces new bindings for ID, item and price. These new bindings enable B to send cancel and pay. However, sending pay disables cancel, because cancel has pDone adorned with “nil”. Similarly, receiving ship prevents B from sending cancel, and receiving cancel prevents S from sending ship; it is reception that disables those messages, not emission, because each agent acts only on local knowledge. Thus, there is a race condition when B sends cancel: the PO is canceled only if S receives cancel before sending ship.

Integrity requires that each parameter have at most one binding in an enactment, as identified by the key parameters. In PO Pay Cancel Ship, each binding of ID identifies a unique enactment, in which there can only be one binding for item, price, and so on. This constraint makes “out” parameters mutually exclusive; S can’t send both cancelAck and payAck in the same enactment, because both bind outcome. If all enactments of a protocol satisfy integrity, we say that the protocol is safe. PO Pay Cancel Ship is safe because none of its parameters can be bound by more than one role in an enactment.

A protocol is live if every enactment can produce bindings for all public parameters (assuming no message loss). PO Pay Cancel Ship is live, because PO binds ID, item, and price. Both cancelAck and payAck bind outcome; in every enactment, at least one them is enabled, completing the enactment.

BSPL semantics [Singh, 2012] maps each protocol to a set of history vectors comprising one sequential history for each of its roles. A history vector $H$ is $[H_1; H_2; \ldots ]$, where each $H_i$ is a sequence of observations $[o_1^i, o_2^i, \ldots ]$ made by role $r_i$. There is no ordering across histories. One history vector progresses to another as any of the roles appends an enabled observation to its observation sequence.

Below $x!m$, e.g., $B!pay$, means the emission of $m$ by $x$; and $x?m$, e.g., $S?pay$, means the reception of $m$ by $x$. For a message of the form $x \mapsto y$, if $y?m$ appears in $y$’s sequence in a vector, $x!m$ also appears in $x$’s sequence in a prior vector.

3 Semantic Tableaux for Protocols

A protocol configuration corresponds to a state of the world as the protocol is enacted. In general, a configuration comprises the local knowledge of each role in the protocol as well as the contents of every communication channel (i.e., between each pair of roles). We include the message emissions and receptions at each role. A message that has been sent but not (yet) received is in the corresponding channel.

In an initial configuration, no observations have occurred, thus no messages are in transit, and each role knows exactly the “in” parameters of the protocol. Our example protocols have no “in” parameters and can be enacted standalone—without other protocols to provide the “in” parameters.

Each protocol enactment goes through a series of configurations where the transition from each configuration to the next respects causality and no role emits a message that would locally cause integrity violation.

Semantic tableaux [D’Agostino et al., 1999; Fitting, 1999] are a way to organize a proof as a tree where inference rules determine how one node leads to another node. Here, tableaux capture the notion of transition for generating a model of a protocol, as needed for our protocol semantics.

Each branch or sequence of nodes beginning from the root
of a tableau has a natural semantic interpretation as a possible protocol enactment. A tableau is thus a model of a protocol, i.e., the set of its enactments. The tableaux we define lend themselves to natural reasoning to eliminate redundant branches, thereby compressing the model of a protocol.

We adopt propositional logic and the usual rules along with modalities \( K_x \), \( U_x \), \( L_x \) to capture that role \( x \) knows, doesn’t know, and observes the emission or reception of a message.

<table>
<thead>
<tr>
<th>Logical Form</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_x m )</td>
<td>Message ( m ) has been observed by ( x )</td>
</tr>
<tr>
<td>( K_x p )</td>
<td>Binding for parameter ( p ) is known to ( x )</td>
</tr>
<tr>
<td>( U_x p )</td>
<td>Binding for parameter ( p ) is still unknown to ( x )</td>
</tr>
</tbody>
</table>

Knowledge and observations Initially, each role knows bindings for exactly the set of \( \lnot \)in\( p \) parameters of the protocol (the empty set for self-contained protocols). What is known to each role grows monotonically because parameter bindings are immutable, and the emission and reception of each message adds 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 \( \lnot \)out\( p \) and \( \lnot \)in\( p \) 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.

Emitting a message Let \( m \) be a message schema of the form (overloading \( m \) as the name of the schema for convenience) \( x \rightarrow y \): \( m[\vec{p}_1, \vec{p}_O, \vec{p}_N] \), where \( \vec{p}_1, \vec{p}_O, \vec{p}_N \) are sets of its parameters respectively adorned \( \lnot \)in\( p \), \( \lnot \)out\( p \), and \( \lnot \)nil\( p \). Then, an instance of \( m \) is enabled for emission by \( x \) if and only if \( x \) knows \( \vec{p}_1 \) but doesn’t know \( \vec{p}_O \) or \( \vec{p}_N \). As a consequence of the observation, the sender produces, and comes to know, the bindings for \( \vec{p}_O \).

\[
\frac{K_x \vec{p}_1 \quad U_x \vec{p}_O \quad U_x \vec{p}_N}{L_x(x \rightarrow y \colon m[\vec{p}_1, \vec{p}_O, \vec{p}_N])}
\]

Upon the sender observing a message, the message enters the communication channel from its sender to its receiver.

Receiving a message Let \( m \) be a message of the form \( x \rightarrow y \colon m[\vec{p}_1, \vec{p}_O, \vec{p}_N] \). Then, \( m \) is enabled for reception by \( y \) if and only if \( x \) has observed sending that message, i.e., it is present in the incoming channel. Concomitantly, \( y \) comes to know the bindings for \( \vec{p}_1 \) and \( \vec{p}_O \).

\[
\frac{L_x(r = x \rightarrow y \colon m[\vec{p}_1, \vec{p}_O, \vec{p}_N])}{L_y r \quad K_y \vec{p}_1 \quad K_y \vec{p}_O \quad U_y \vec{p}_1 \quad U_y \vec{p}_O}
\]

Generating branches Whether a message is enabled for emission may depend upon its sender not knowing bindings of some parameters. Therefore, enablement is transient; it is computed implicitly within this inference rule but not stored as an assertion in the tableau. For completeness, we must ensure that each enabled message is observed on some tableau branch growing from a configuration.

Our desired tableaux would have a branch for each enactment. To this end, there is a choice between all allowed observations by any of the roles. For completeness, we must consider all choices. In tableaux for traditional logics, there is branching for any disjunction. Likewise, here we need branching for all available choices. Let \( C \) be a configuration comprising all assertions of the form \( K_x p, U_x p, \) and \( L_x m \), and define the following.

- \( E_x = \{(x, e) \mid e = x \rightarrow y \colon m[\vec{p}_1, \vec{p}_O, \vec{p}_N]\} \) and \( K_x \vec{p}_1 \) and \( U_x \vec{p}_O \) and \( U_x \vec{p}_N \).
- \( R_y = \{(y, r) \mid r = x \rightarrow y \colon m[\ldots]\} \) and \( L_x r \).
- \( L = \bigcup_x E_x \cup \bigcup_y R_y \), the enabled observations in \( C \).

Then, this metarule captures all branches in the tableau:

\[
C \quad L_x m \mid \ldots \mid, \text{where each } (z, m) \in L
\]

Generating Enactments from Tableaux Each branch of a tableau, generated from the inference rules above, respects the causal structure of BSPL as well as local consistency for the emissions made by a role.

Each branch up to a configuration corresponds to a unique history vector in the BSPL semantics. Specifically, we find a branch from the root to a given configuration. For each role \( x \), we record all the \( L_x \) statements in the order in which they occur as the observation sequence of \( x \). The history (of some role changes) any time there is an \( L_x m \) assertion. The rest of the tableau, i.e., between consecutive \( L_x m \) and \( L_y m \) assertions is for reasoning about the same state.

Verifying Correctness Properties We assert a property at the root. A branch ends when no more observations are enabled. A consistent branch that ends provides an example for the property at the root. A branch that hits a contradiction is closed; a tableau closes if all its branches close, which indicates the property is inconsistent.

The properties of interest, e.g., liveness and safety, are concerned with the reachability or otherwise of a configuration through an enactment. These properties are expressed via propositional combinations of local knowledge states of the roles and the observations made by the roles.

For liveness, we require each public parameter of the protocol to be known to at least one role. A liveliness violation occurs precisely when at least one of the parameters remains unknown to each role that emits a message with an \( \lnot \)out\( p \) on that parameter. For example, in Listing 1, \( \lnot K_8 \text{ID} \lor \lnot K_8 \text{item} \lor \lnot K_8 \text{price} \lor \lnot K_8 \text{outcome} \). If every tableau branch closes, liveness is established.

For safety, let \( m_1 \) and \( m_2 \) be two messages that conflict by having the same \( \lnot \)out\( p \) parameter. If both messages were to be emitted, we would have conflicting bindings for the parameter, reflecting an integrity violation. A safety violation
may be reduced to a disjunction of conjunctions of expressions such as \( L_x m_1 \) and \( L_y m_2 \), where \( x \neq y \). 

Listing 1 has no such case since both cancelAck and payAck, which conflict on outcome, are sent by \( S \). For illustration, let’s augment Listing 1 with

\[
\text{S} \rightarrow \text{B: getLost [in ID key, out rescind]}
\]

Then, the resulting protocol’s safety violation would occur if and only if \( L_x \text{cancel} \land L_y \text{getLost} \) can hold. Therefore, we would place its negation \( \neg L_x \text{cancel} \lor \neg L_y \text{getLost} \) at the root of the tableau. If any branch of the tableau closes, i.e., runs into a contradiction, safety is violated.

4 Reducing the Tableau

Avoiding combinatorial branching for true concurrency

Recall that in BSPL, each role is serial but the various roles can proceed concurrently. Our interleaved expansion of a tableau is adequate for representing concurrent enactments because BSPL decouples observations by different roles. Therefore, any set of concurrent observations can be realized by interleaving them without any intervening observations from outside of that set.

Theorem 1. Each history vector corresponds to at least one possible branch in a tableau for that protocol.

Proof Sketch. Each step on a tableau branch involves one inference rule being applied. A history vector is built inductively starting from a vector of empty histories. When the inductive step involves exactly one role sending or receiving a message, the corresponding inference rule generates the node that corresponds to the new vector.

When roles act concurrently, the effect equals that of interleavings of individual steps, which are captured by the tableau construction. Consider a history vector \([H_x; H_y]\) for a protocol with two roles, \( x \) and \( y \). Suppose \( x \) emits \( m_1 \) and \( y \) emits \( m_2 \). Then, under true concurrency, first both emit: \([H_x, x!m_1; H_y, y!m_2]\). And next both receive: \([H_x, x?m_1; H_y, y?m_2]\).

We can show that interleaveings would produce configurations equivalent to those produced by concurrency, e.g., \([H_x, x!m_1; H_y]\) and \([H_x, x!m_1; H_y, x?m_1]\), which are legitimate BSPL history vectors. That is, a tableau produced without concurrency captures all history vectors in the BSPL semantics.

Relationships between messages based on a protocol

The following intuitions underlie our approach of reducing the set of interleaveings of messages to be considered in a tableau. We begin from identifying relationships between messages to construct a graph of potential incompatibilities.

Unrelated messages may occur in any order in reality and any arbitrary order is equivalent to all of them, so we can discard all but one of the branches in the tableau.

Necessary enabler If one message must occur before another, we need consider them only in that order. However, the tableau would unfold in the correct order and we need do nothing special to accommodate it.

Order matters If one message may disable another, we must include both possibilities in the tableau. We need to recognize such orders even if one of the messages is not presently enabled and make sure we don’t prematurely eliminate viable branches from the tableau.

We consider message transmissions realized as separate observations by the sender and recipient, respectively.

Observation \( a \) directly endows observation \( b \), \( a \vdash b \), if and only if \( a \) is a necessary precursor to \( b \). In BSPL, \( b \) has a parameter adorned \( "m" \) and \( a \) is an observation of the unique message that has the same parameter adorned \( "o" \). Emis-

sions directly endow their reception. For example, \( b!\text{pay} \) directly endows \( s?\text{pay} \), and \( s?\text{pay} \) directly endows \( s!\text{payAck} \) because it is the only observation for \( S \) that binds \( \text{pDone} \).

Observation \( a \) endows \( c, a \vdash c \), if and only if there is a chain of one or more direct endowments from \( a \) to \( c \).

Observation \( a \) directly disables observation \( b \), \( b \vdash a \), if and only if the occurrence of \( a \) disables \( b \). In BSPL, \( b \) must have an \( "o" \) parameter adorned \( "m" \) and \( a \) is observed by the unique message that has the same parameter adorned \( "o" \). For example, \( s!\text{cancel} \) directly disables \( s!\text{ship} \). An emission can be disabled by other observations by the sender, but a reception cannot be disabled. Additionally, \( a \) must not endow \( b \); otherwise, \( b!\text{pay} \) would be considered to disable \( b!\text{PO} \) since \( \text{pay} \) has \( \text{ID} \) \( "m" \) and \( \text{PO} \) has \( \text{ID} \) \( "o" \); but clearly \( \text{pay} \) depends on \( \text{PO} \) and cannot disable it.

Observation \( a \) directly enables observation \( b \), \( a \vdash b \), if and only if \( a \) is a potential precursor to \( b \). In BSPL, \( b \) is the reception for an emission \( a \), or an emission that has a parameter adorned \( "m" \) and \( a \) is observed by the sender and has the same parameter adorned \( "o" \). Emisions directly enable the corresponding reception observation at the recipi-

ent, and both emissions and receptions can enable emissions at the sender. For example, \( b!\text{pay} \) directly enables \( s?\text{pay} \), and \( s?\text{pay} \) directly enables \( s!\text{payAck} \). An emission can also enable another emission, as when \( b!\text{PO} \) enables \( b!\text{pay} \).

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

<table>
<thead>
<tr>
<th>( \text{in}^m ) p ∈ a</th>
<th>( \text{out}^m ) p ∈ b</th>
<th>( \text{nil}^m ) p ∈ b</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a \vdash !b )</td>
<td>( a \vdash !b )</td>
<td>( a \vdash !b )</td>
</tr>
<tr>
<td>( a \vdash !b )</td>
<td>( a \vdash !b )</td>
<td>( a \vdash !b )</td>
</tr>
<tr>
<td>( a \vdash !b )</td>
<td>( a \vdash !b )</td>
<td>( a \vdash !b )</td>
</tr>
</tbody>
</table>

Observation \( a \) enables observation \( c, a \vdash c \), if and only if there is a chain of one or more direct enablements from \( a \) to \( c \). This notion of enablement is an upper bound in that \( a \) may have multiple enabling messages, not all of which would be needed as precursors of \( c \), and some of which may be mutually disabling.

Observation \( a \) tangles with observation \( c, a \vdash c \), if and only if (1) \( a \) doesn’t endow \( c \); and (2) \( a \) directly disables \( c \) or \( a \) directly disables a message \( b \) where \( b \) enables \( c \). That is, \( a \) tangles with the causal precursors of any message it disables. Considering causal precursors helps us identify entanglement even when a message doesn’t directly disable another.
Two observations are *incompatible* if and only if at least one is an emission and one tangles with the other.

We consider observations to be *sensitive* if they can directly disable or be directly disabled by a potential observation. Such observations must be handled carefully to ensure that all enactments are covered.

**Compatible sets of observations.** We now compute compatible sets of observations that are enabled in a particular configuration. As explained above, the reception of a message that has been sent is always enabled. Emissions are enabled based on the knowledge state of the emitting role.

Given the set of messages enabled in a configuration, identify incompatible pairs of observations. Then, partition the messages into compatible sets such that (1) no observation is incompatible with another within the same compatible set and (2) each compatible set contains an observation that is incompatible with an observation in another compatible set.

For correctness, we consider both orders for any pair of incompatible observations. We can arbitrarily reorder observations within a compatible set with no effect on correctness—desirable for reducing the number of generated orders.

We capture the incompatibilities as an undirected graph, i.e., with observations as vertices and edges as incompatibilities. We reduce the problem of identifying a suitable partition to graph coloring. Since minimal graph coloring is NP-Hard, we apply a greedy approach that ensures the observations assigned the same color are compatible though the different colors may not necessarily reflect true incompatibilities. We slightly enhance Brélaz’s [1979] approach below.

- Sort vertices in order of decreasing degree.
- Assign a color to each vertex that isn’t assigned to its neighbors, generating a new color only if necessary.
- In variation from Brélaz, choose a color that (1) has the highest cardinality; (2) within such, the color whose vertex of highest degree has the smallest degree; (3) within such, the color whose vertex of highest degree in the full protocol incompatibility graph has the smallest degree.

This choice pushes the coloring toward an imbalanced distribution of vertices, which would tend to lower the number of branches to be added to a tableau.

Within a compatible set, observations don’t conflict so we consider them in an arbitrary order. Across compatible sets, observations may conflict so we generate branches in the tableau to capture each order of incompatible observations.

Suppose the observation incompatibilities are as Figure 2 shows. A possible partition is \{\{b, f\}, \{a, d\}, \{c, e\}\}.

**Reduction of proof orders.** In a configuration \(C\), compute \(L\) as defined above; the set of pairs of roles and their enabled observations in \(C\).

For any nonsensitive observations in \(L\), select one by an arbitrary ordering such as alphabetically by name, or maximum degree in the incompatibility graph; order does not matter for correctness, but may influence the number of branches.

If only sensitive observations are enabled at \(C\), generate a subgraph of the protocol’s incompatibility graph limited to observations in \(L\). Identify compatible sets of the subgraph, as described above. Create one branch of the tableau for each compatible set, placing an observation of the first observation of that compatible set.

Preferentially handling nonsensitive observations first is not essential to correctness, but can sometimes reduce the number of branches that are introduced by a conflict and makes the branching occur closer to the actual conflict so the tree is easier to understand.

**Correctness of the method.** Since we have already shown that all history vectors (enactments) have a corresponding tableau, we need only to show that a reduced tableau covers all of the branches of the full tableau up to equivalence.

**Theorem 2.** If \(T_P\) is a full tableau for protocol \(P\) and \(R_P\) is a reduced tableau constructed according to the above rules, then every maximal branch in \(T_P\) is equivalent to some branch in \(R_P\).

**Proof Sketch.** Suppose \(B_T\) is a maximal branch in \(T_P\), i.e., a valid sequence of observations that does not enable any additional observations. Suppose also that \(B_T\) is not equivalent to any branch \(B_R\) in \(R_P\). Then, there must be some point, \(D\) where \(B_T\) diverges from \(R_P\); that is, observations on \(B_T\) up to \(D\) form subsequences of the observations on some branch in \(R_P\). This point may be the empty branch. Let \(Q\) be the set of maximal branches in \(R_P\) such that each branch in \(B_T\) is equivalent to some branch in \(Q\).

Case 1. \(B_T\) ends prematurely, so that all branches in \(Q\) are strict supersequences and thus not equivalent. But this is impossible because \(B_T\) is maximal, so the configuration induced by its observations cannot enable any additional observations.

Case 2. \(B_T\) is extended by some observation not in any branch in \(Q\). After divergence, \(B_T\) is extended by some observation \(o\) that is not in any branch in \(Q\). Thus, since they are maximal, and contain every observation in \(B_T\) prior to \(o\) (thus everything necessary to enable \(o\)), every branch in \(Q\) must contain some observation \(o'\) that directly disables \(o\). According to the rules of construction, sensitive observations are delayed until some sensitive event must be chosen, and then partitioned into incompatible sets which each form a new branch. Thus, at the point that \(o'\) is selected, since \(o\) is enabled, it must be placed in an incompatible set, and thus end up on some other branch, contradicting the premise that no branch in \(Q\) contains \(o\). Thus, every maximal branch in \(T_P\) is equivalent to some branch in \(R_P\).
Table 3: Performance comparison between full paths [Christie et al., 2020] and reduced (Tango) approaches on a well-known payment protocol (NetBill) and a healthcare standard (HL7) protocol as well as on conceptually challenging protocols such as Block Contra.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>No Branch Reductions</th>
<th>Tango</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nodes</td>
<td>Branches</td>
</tr>
<tr>
<td>PO Pay Cancel Ship (Listing 1)</td>
<td>1,495</td>
<td>490</td>
</tr>
<tr>
<td>Block Contra (in supplement)</td>
<td>1,802</td>
<td>612</td>
</tr>
<tr>
<td>Independent (in supplement)</td>
<td>453</td>
<td>90</td>
</tr>
<tr>
<td>NetBill [Sirbu, 1998], Bliss version [Singh, 2014]</td>
<td>4,097</td>
<td>1,246</td>
</tr>
<tr>
<td>HL7 Create Lab Order [Christie et al., 2018]</td>
<td>59,259</td>
<td>17,814</td>
</tr>
</tbody>
</table>

5 Evaluation

Figure 3 shows a graphical representation of the entire tableau computed for Listing 1, with the observations as transitions. Partitions are shown at nodes where only sensitive observations are enabled. When checking a property, we would terminate the process as soon as a counterexample is found.

Figure 3: Entire tableau structure for PO Pay Cancel Ship.

Performance evaluation We have implemented Tango in Python, and compared it with existing BSPL verification tools. All experiments were performed on the same Linux laptop, a Thinkpad T460 with an Intel i7-6600U CPU and 16GB of DDR3 RAM. In our experiments, observations are selected alphabetically by message name and nonsensitive observations are handled first.

Table 3 shows the total number of nodes (representing configurations or states), branches, and elapsed time for several sample protocols, including the examples in this paper. The columns labeled No Branch Reductions describe the prior approach for BSPL [Christie et al., 2020]. The columns labeled Tango result from our method.

In addition to the PO Order Cancel Ship protocol, we show the results for two protocols (listings in the supplement) that illustrate important cases for our evaluation. Block+Contra exhibits subtle race conditions that confound a naive method. Independent shows an ideal case for tableau reduction where there is a direct causal chain but can lead to a combinatorial explosion for a naive method.

6 Discussion

The engineering of interacting agents is an evergreen theme in multiagent systems [Mazouzi et al., 2002; Damiani et al., 2012]. The connection of interactions with high-level representations, both cognitive [Boissier et al., 2019] and social [Baldoni et al., 2015], from the standpoint of engineering multiagent systems remains crucial. A well-designed protocol would decouple its agents, helping cope with the complexity of verifying or testing them separately [Winikoff and Cranefield, 2014] and composing the results. BSPL makes it possible to specify flexible protocols that decouple the agents as much as possible given causality and integrity constraints.

BSPL is unique in its support for asynchrony and flexibility, and thus its need for reduced models. Our contribution includes fast (polynomial time) heuristics that help avoid generating a too-large tableau. Protocol languages such as trace expressions [Ferrando et al., 2019] that do not express the information exchanged (and associated causality and integrity constraints) do not enable the optimizations we described, since they do not clear which message emissions and receptions can be reordered without affecting the meaning.

Previous work on BSPL introduced the concept of refinement to reduce the cost of re-verifying a protocol after substituting a component protocol in a large composition, and used a path-based model for verification [Christie et al., 2020]. However, the model suffered from combinatorial explosion, because simulated every possible interleaving of observations as a distinct enactment.

Our source code and examples are available publicly at https://gitlab.com/masr/.

Acknowledgments

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


