Compositional Correctness in Multiagent Interactions

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

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

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

ABSTRACT
An interaction protocol specifies the constraints on communication between agents in a multiagent system. Ideally, we would like to be able to treat protocols as modules and compose them in a declarative manner to systematically build more complex protocols. Supporting composition correctly requires taking into account information-based causality relationships between protocols. One important problem that may arise from inadequate consideration of such relationships is that the enactment of a composite protocol may violate atomicity; that is, some components may be initiated but prevented from completing. We use the well-known all or nothing principle as the basis for formalizing atomicity as a novel correctness property for protocols.

Our contributions are the following. One, we motivate and formalize atomicity and highlight its distinctiveness from related correctness notions. Two, we give a decision procedure for verifying atomicity and report results from an implementation. For concreteness of exposition and technical development, we adopt BSPL as an exemplar of information causality approaches.

KEYWORDS
Interaction protocols; causality; atomicity; multiagent systems

ACM Reference Format:

1 INTRODUCTION
An interaction protocol specifies the rules of encounter between autonomous agents in a multiagent system. Two kinds of protocols have been studied extensively in the multiagent systems literature: protocols that specify constraints on message ordering and occurrence, and protocols that specify the meanings of interactions. We refer to the former as operational and the latter as meaning-based. RASA [18], HAPN [30], and UML interaction diagrams and its variants such as AUML [20] are among several languages developed to specify operational protocols. Meaning-based protocols are exemplified by work on commitment protocols [4, 9, 28, 32].

We are primarily concerned here with the composition of operational protocols. Ideally, we would like to be able to treat a protocol as a module, analogous to the way a program may be treated as a module, and compose it with other protocols to obtain more complex protocols. Further, we would like to be able to compose protocols declaratively on the basis of information-based causality constraints [25, 27]. Specifically, such constraints would capture what an agent must know (or not know) to produce a new piece of information. For example, in a typical sales transaction, one constraint would capture that a seller cannot fill an order until it knows what the order is. For concreteness, say there were a message schema FillOrder with a parameter order whose value reflected the details of the items ordered in the transaction. An instance of FillOrder could not be sent by the seller until it knew the details of the items ordered through a prior interaction in the transaction, say, through the receipt of an instance of the message schema PlaceOrder (sent by the buyer) that produces the value for order. Ignoring other details, the two specifications PlaceOrder and FillOrder may be composed into a protocol named, say Purchase, on the basis that PlaceOrder, by producing a binding for order, satisfies the causal constraint on FillOrder that the binding of order be known. Notice how the dependency induces a message ordering: in any instance of Purchase, PlaceOrder must be received by seller before it can send FillOrder.

Although the foregoing example composes message specifications, note that a message specification is an elementary protocol and the idea of causality-based composition applies to protocols in general. The Blindingly Simple Protocol Language (BSPL) [24] and its extensions such as Splee [6] are exemplars of the information-based causality approaches.

Compositions based on causality yield declarative specifications and support flexible enactments in fully decentralized asynchronous settings [25]. Here, “decentralized” means shared nothing and “asynchronous” means not relying on ordered channels (in contrast to much of the literature that assumes ordered channels). However, not all compositions would be desirable, as the following example demonstrates. Say the aforementioned protocol Purchase were extended with Transfer (pay by wire) and Credit (pay by credit card) protocols intended to offer a mutually exclusive choice between two payment methods. The specification of Purchase would clearly be incorrect if both Transfer and Credit could be enacted in the same instance of Purchase. Purchase would also be incorrect, specifically, nonatomic, if there were enactments wherein Transfer had been initiated but was blocked from completion even though Credit had completed. Interestingly, it may be the completion of Credit itself that blocks the completion of Transfer (as a way of ensuring that only one payment method goes through). Such cases, where the enactment of Transfer is initiated but blocked, in effect, left dangling, may be indicative of semantic errors. For instance, the errors may take the form of active commitments that cannot possibly be discharged.

To capture undesirable compositions that could result in dangling enactments, we propose atomicity as a correctness criterion for protocols. The basis for atomicity is the observation that a message...
schema is *de facto* atomic: either its enactment occurs completely or not at all—there is nothing in between. For a composed protocol, the situation is not as straightforward, as we saw above for *Purchase*. Some constituents may complete, others may block. Lifting the idea of atomicity to protocols generally, we say that a composed protocol is atomic if and only if each constituent is atomic and completion of a constituent implies that the composed protocol can complete. Informally, our notion of atomic protocols is analogous to the notion of atomic programs [11] and transactions [10] in that it captures a notion of *all or nothing*. However, atomicity for protocols and transactions is typically formulated in shared memory settings whereas we address decentralized settings.

Clearly, the problem in the foregoing example could be avoided if the buyer prudently enacts either *Transfer* or *Credit* but not both. Further, we would like to guarantee atomicity from the protocol specification alone, without resort to agent specifications. This motivation reflects the essential doctrine for protocols: capturing the interaction logic and presenting it in a reusable form [3, 19, 22].

We adopt BSPL (Section 2) to convey our ideas concretely and formally. Our contributions are the following.

- We motivate atomicity for information-based protocols and provide examples and patterns of atomicity violations and their corrections (Section 3).
- We formalize atomicity and distinguish it from liveness and safety of information-based protocols (Section 4).
- We give a decision procedure for verifying atomicity and describe an implementation of the decision procedures (Section 5). We report results from running the implementation on examples in the present paper.

Finally, Section 6 discusses related work and future directions.

## 2 BSPL OVERVIEW AND FORMALIZATION

BSPL [24] specifies protocol constraints in terms of causality, as motivated above, and integrity based on key constraints [10] on the information model. The key constraints capture the idea that in any protocol enactment a role may not send or receive conflicting information. Listing 1 illustrates BSPL’s main concepts via a simple protocol.

### Listing 1: Purchase with payment options

```plaintext
Purchase {
    roles B, S // Buyer, Seller
    parameters out order key, out product
    B → S: PlaceOrder [out order]
    B → S: Transfer [in order, out payment]
    B → S: Credit [in order, out payment]
    S → B: FillOrder [in order, in payment, out product]
}
```

The listing declares *Purchase* as the name of the protocol, two public roles *B* (buyer) and *S* (seller), and two public parameters order and product. Parameter order is adorned key, meaning that bindings of order uniquely identify enactments of *Purchase* and functionally determine the other parameters. Both parameters are adorned "out", meaning that their bindings are produced by enacting the protocol, that is, enacting the messages declared in it.

The public parameters of a protocol serve as its interface and facilitate composition. *Purchase* declares four message schemas (the sequence of their listing is irrelevant). By convention, any key parameter of the protocol is a key parameter for any message in which it appears, though a message may have additional key parameters. Thus, *PlaceOrder* is from the buyer to the seller and its key is order. The "out" adornment means that sending *PlaceOrder* produces a binding for order. In *FillOrder*, order and payment are adorned "in", meaning that a seller may send an instance of *FillOrder* only if it has observed bindings for both. Parameters may in addition be adorned with "nil", indicating that the message cannot be sent if that parameter is bound.

A set of message instances (an enactment) completes a BSPL protocol if the messages cover all of its public "out" parameters; that is, each public "out" parameter is included in at least one message instance. For *Purchase*, *PlaceOrder* and *FillOrder* cover order and product. An enactment of *Purchase* is complete when an order has been both placed and filled. However, *FillOrder* has parameter payment adorned "in". Thus the buyer must produce payment by sending either *Transfer* (representing a bank wire transfer) or *Credit* (representing payment via credit card) before the order can be filled.

Singh [26] formalizes BSPL and properties and gives verification techniques. Informally, a protocol is *executable* if and only if a valid complete enactment exists. *Request Quote* is executable, because the enactment consisting of *PlaceOrder* followed by *Transfer* and then *FillOrder* is valid and covers all public parameters of *Purchase*.

A protocol is *live* if and only if any enactment can progress to completion. *Purchase* is live: for any value of order, *PlaceOrder* may be sent followed by *Transfer* or *Credit* and then *FillOrder*, which would complete the *Purchase* enactment. An alternative specification without *Transfer* and *Credit* would not be live because without any way to produce a binding for payment, *FillOrder* cannot be sent and the enactment cannot be completed.

Informally, a protocol is *safe* if and only if it is impossible to produce conflicting bindings for a parameter in any enactment. A potential safety violation would be if a buyer sent two instances of *Transfer* for the same order, one with a payment of $10 and one with a payment of $20. Such a violation can be easily avoided by the buyer based solely on its local knowledge, and so is not a flaw in the specification. A real safety violation occurs when two agents may produce conflicting bindings in an enactment. *Purchase* is safe. If the message *Gift* in Listing 2 were added to *Purchase*, it would become unsafe. Both seller and buyer could concurrently produce bindings for payment; that is, a nonlocal conflict would exist.

### Listing 2: An unsafe extension to Purchase.

```plaintext
S → B: Gift [in order, out payment, out product]
```

BSPL supports composition in a natural manner. A single message is an elementary protocol in BSPL. Thus, *PlaceOrder*, *Transfer*, and *Credit*, and *FillOrder* are all elementary protocols. *Purchase* composes these messages by referring to them. A BSPL protocol may have references to one or more protocols. Protocol *Refined-Purchase* (Listing 3) replaces *Transfer* and *Credit* with composite protocols *RefinedTransfer* and *RefinedCredit*, respectively.

We reproduce here the formal semantics of BSPL (from Singh [26]).
Listing 3: Refined purchase protocol

RefinedPurchase |
roles B, S // Buyer, Seller
parameters out order key, out payment, out product
B → S: PlaceOrder(out order)
RefinedTransfer(B, S, in order, out payment)
RefinedCredit(B, S, in order, out payment)
S → B: FillOrder(in order, in payment, out product)

For convenience, we fix the symbols by which we refer to finite lists of roles (î), public roles (ê), private roles (û), public parameters (î), key parameters (k ∈ ï), "in" parameters (î1 ⊆ ï), "out" parameters (î2 ⊆ ï), "nil" parameters (î3 ⊆ ï), private parameters (û), and parameter bindings (û, v). Here, ð = î1 ∪ î2 ∪ î3, private parameters (û), and parameter bindings (û, v). Here, ð = î1 ∪ î2 ∪ î3, = 0, and P ∈ ï. And, t and r refer to an individual role and parameter, respectively. To reduce notation, we rename public roles and parameters to be distinct in each protocol, and the public roles and parameters of a reference to match the declaration in which they occur. Throughout, we use ↓x to project a list to those of its elements that belong to x.

Definition 1 captures BSPL protocols. A protocol may reference another protocol. The references bottom out at message schemas. Above, Purchase references Transfer. And, if a protocol were to reference Purchase, it would be able to reference (from its public or private parameters) only the public parameters of Purchase, not payment, which is a private parameter.

Definition 1: A protocol P is a tuple (n, ð, ù, k, õ, F), where n is a name; ð, ù, k, and õ are as above; and F is a finite set of f references, (F1, . . . , Ff). (Vi : 1 ≤ i ≤ f ⇒ Fi = (ni, ði, ùi, ki)), where ni ⊆ ð ∪ ùi, ùi ⊆ ði ∪ õ, ki = ði ∩ k, and (ni, ði, ùi, ki) is the public projection of a protocol P1 (with roles and parameters renamed).

Definition 2: The public projection of a protocol P = (n, ð, ù, k, õ, F) is given by the tuple (n, ð, ù, k).

We treat a message schema "s → r : m ð(k)n as an atomic protocol with exactly two roles (sender and receiver) and no references: (m, (s, r), Ø, ð, k, Ø, Ø). Here, k is the set of key parameters of the message schema. Usually, k is understood from the protocol in which the schema is referenced: k equals the intersection of ð with the key parameters of the protocol declaration.

Below, let roles(P) = ð ∪ ù ∪ ∪j roles(Fj); params(P) = ð ∪ ù ∪ ∪j params(Fj); msgs(P) = ∪j msgs(Fj) and msgs(s → r : m ð) = {m}. Definition 3 assumes that the message instances are unique up to the key specified in their schema.

Definition 3: A message instance m[s, r, ð, v] associates a message schema "s → r : m ð(k)n with a list of values, where [v] = [ð], where "v = nil and if only if p ∈ ð.

Definition 4 introduces a universe of discourse (UoD). Definition 5 captures the idea of a history of a role as a sequence (equivalent to a set in our approach) of all and only the messages the role either emits or receives. Thus H contains the local view of an agent who might adopt role r during the enactment of a protocol.

Definition 4: A UoD is a pair (R, M), where R is a set of roles, M is a set of message names; each message specifies its parameters along with its sender and receiver from R.

Definition 5: A history of a role r, Hr, is given by a sequence of zero or more message instances m1 o m2 o . . . Each m is of the form m[s, r, ð, v] where ð = s or r = r, and means sequencing.

Definition 6 captures the idea that what a role knows at a history is exactly given by what the role has seen so far in terms of incoming and outgoing messages. Here, 2(i) ensures that m[s, r, ð(k), v], the message under consideration, does not violate the uniqueness of the bindings. And, 2(ii) ensures that r knows the binding for each "r in" parameter and not for any "out" or "nil" parameter.

Definition 6: A message instance m[s, r, ð(k), v] is viable at role r’s history Hr if and only if (1) r = r (reception) or (2) s = ð (emission) and (i) (V m[s, r, ð, v] ∈ H) if k ⊆ k, and (ii) (V p ∈ ð : p ∈ ð) and if only if (V m[s, r, ð, v] ∈ H) and p ∈ ð, and k ⊆ k).

Definition 7 captures that a history vector for a protocol contains a history for each role, and that all of the histories together are causally sound: a message is received only if it has been previously emitted [16].

Definition 7: Let (R, M) be a UoD. We define a history vector for (R, M) as a vector [H1, . . . , H|R|], such that (Vs, r : 1 ≤ s, r ≤ |R| : Hs is a history and (Vm[s, r, ð, v] ∈ H : m ∈ M and m[s, r, ð, v] ∈ H)).

The progression of a history vector records the progression of an enactment of a multiagent system. Under the above causality restriction, a vector that includes a reception must have progressed from a vector that includes the corresponding emission.

Definition 8: A history vector over (R, M), [H1, . . . , H|R|], is viable if and only if either (1) each of its element histories is empty or (2) it arises from the progression of a viable history vector through the emission or the reception of a viable message by one of the roles, that is, (Vs, m : Hs = Hs o m, and [H1, . . . , H|R|] is viable).

The heart of our formal semantics is the intension of a protocol, defined relative to a UoD, and given by the set of viable history vectors, each corresponding to its successful enactment. Given a UoD, Definition 9 specifies a universe of enactments, based on which we express the intension of a protocol. We restrict attention to viable vectors because those are the only ones that can be realized. We include private roles and parameters in the intension so that compositionality works out. In the last stage of the semantics, we project the intension to the public roles and parameters.

Definition 9: Given a UoD (R, M), the universe of enactments for that UoD, U(R, M), is the set of viable history vectors, each of which has exactly |R| dimensions and each of whose messages instantiates a schema in M.

Definition 10 states that the intension of a message schema is given by the set of viable history vectors on which that schema is
We now motivate atomicity informally with the help of protocol (1) a protocol is atomic if each of its constituent protocols is atomic virtue of being message schemas, thus satisfying (1) above. Further, any enactment where any of them occurs can be completed, thus satisfying (2) above. For example, consider an enactment where Transfer has occurred for some values of order and payment; order must already have been bound by an instance of PlaceOrder. Now FillOrder can occur in the enactment, which produces a binding for product. Thus, all of Purchase’s parameters are bound and the enactment is complete.

However, some conflicts violate atomicity if they occur between protocols after they have both been initiated and prevent one of them from completing. Consider RefinedPurchase in Listing 3, in conjunction with the following definitions for RefinedTransfer and RefinedCredit.

### Listing 4: Refined payment protocols

```plaintext
RefinedTransfer {
    roles B, S
    parameters in order key, out payment
    B → S: OfferTransfer[ in order, out transferOffer ]
    S → B: AcceptTransfer[ in order, in transferOffer, out transferAccepted ]
    B → S: InitiateTransfer[ in order, in transferAccepted, out payment ]
}
RefinedCredit {
    roles B, S
    parameters in order key, out payment
    B → S: OfferCredit[ in order, out creditOffer ]
    S → B: AcceptCredit[ in order, in creditOffer, out accept ]
    B → S: PayCredit[ in order, in accept, out payment ]
}
```

In Listing 4, the OfferTransfer and OfferCredit messages do not conflict with each other, as neither binds any parameters that would prevent the other from being sent. This lack of conflict means that both RefinedTransfer and RefinedCredit can be initiated in the same enactment. The seller may additionally send AcceptTransfer and AcceptCredit without conflict. However, the buyer is prevented from sending both InitiateTransfer and PayCredit because both produce bindings for payment. Thus both protocols can be initiated but only one can be completed, violating atomicity.

This violation of atomicity indicates that the specification is flawed. Perhaps the RefinedTransfer and RefinedCredit protocols should be mutually exclusive as in the original Purchase protocol, requiring the buyer to initiate only one of them. Alternatively, some mechanism for canceling one of the two should be added, or the protocols should be modified so they do not conflict with each other, as would be the case if the buyer is allowed to split the payment across multiple methods.

For example, the FixedPurchase variant of Purchase adds the offer parameter to OfferTransfer and OfferCredit messages as in Listing 5, so that the buyer may initiate only one of the two payment protocols. Since only one of the protocols can be initiated the other cannot prevent it from completing, so atomicity is restored.

### Listing 5: FixedPurchase with explicit choice

```plaintext
B → S: OfferTransfer[ in order, out offer, out transferOffer ]
B → S: OfferCredit[ in order, out offer, out creditOffer ]
```

Not all atomicity violations require mutual exclusion. Some can be resolved by an alternative path to completion that avoids the conflict. For example, consider ShareHealthData in Listing 6.
We define \( \text{ref}(\mathcal{P}) \) as the set of references of \( \mathcal{P} \).

### 4 Atomicity Formalization

We define \( \text{ref}(\mathcal{P}) \) as the set of references of \( \mathcal{P} \).

**Listing 6: Sharing private health records**

<table>
<thead>
<tr>
<th>ShareHealthData</th>
<th>roles P, R, C // patient, researcher, clinic</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameters out ID key, out granted, out revoked</td>
<td></td>
</tr>
<tr>
<td>P \rightarrow C: Authorize [out ID, out granted]</td>
<td></td>
</tr>
<tr>
<td>AccessData(R, C, in ID, in granted, nil revoked, out data)</td>
<td></td>
</tr>
<tr>
<td>P \rightarrow C: Revoke [in ID, in granted, out revoked]</td>
<td></td>
</tr>
</tbody>
</table>

In *ShareHealthData*, a patient can *Authorize* a clinic to share their data with researchers, until they subsequently *Revoke* access. While access is granted a researcher may *Request* the data, which the clinic sends them via *Provide*. However, because of the "nil" adornment, *Provide* cannot be sent after revoke is bound. Since data must be bound to complete *AccessData*, atomicity is violated in enactments where the patient sends *Revoke* after *Request* occurs but before *Provide*.

The conflict between "out" and "nil" parameters causes the protocols to be partially ordered rather than mutually exclusive: the *AccessData* component can be enacted completely as long as actions are performed in the correct sequence.

The patient should be able to revoke access even if there is a pending request. To enable this possibility without violating atomicity, the clinic should be able to complete *AccessData* without sending *Provide*, such as by rejecting the request. Listing 7 adds a reject message to *AccessData* to restore atomicity. (Data in this case would be empty or a rejection message)

**Listing 7: Alternative path to complete AccessData**

| C \rightarrow R: Reject [in ID, in req, out data, nil revoked] |

Based on the kinds of conflicts that are possible with simple causal relationships as expressed in BSPL, we have identified several kinds of atomicity violations, illustrated by the above examples and summarized in Table 1. Exclusion conflicts involving either an "in" or "out" conflicting with an "out", such as that in *RefinedPurchase*, can be resolved by only enabling one of the conflicting protocols. Ordering conflicts involving an "in" or "out" conflicting with a "nil", such as that in *ShareHealthData*, can additionally be resolved by specifying an alternative path to completion. Indirect conflicts involving an "in" parameter and an "out" or "nil" parameter are equivalent to direct conflicts involving an "out" and either "out" or "nil" respectively; the information simply passes through at least one intermediary before the conflict becomes apparent. Because these are the only combinations of BSPL parameter types that produce conflicts, this list is exhaustive.

### 4.1 Distinction From Existing Properties

We demonstrate the orthogonality of atomicity to BSPL’s notions of safety and liveness by exemplifying every combination of atomicity and safety or liveness, as shown in Table 2.
Table 2: Protocols demonstrating orthogonality of properties.

<table>
<thead>
<tr>
<th>Atomic</th>
<th>Nonatomic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safe Purchase</td>
<td>ShareHealthData</td>
</tr>
<tr>
<td>Unsafe Purchase+Gift RefinedPurchase</td>
<td>Transfer Stuck</td>
</tr>
</tbody>
</table>

For completeness, a trivially nonlive, nonatomic protocol `Stuck` is provided in Listing 8.

**Listing 8: Trivially nonlive and nonatomic protocol**

```plaintext
Stuck {
    roles A, B
    parameters out begin key, out end
    A \iff B: Start [out begin]
}
```

### 4.2 Relationships with Existing Properties

Although atomicity is orthogonal to the other properties, there are some connections. The following theorem shows that a protocol is atomic if it is not only nonlive, but none of its messages is enactable.

**Theorem 1**: A protocol is atomic if its universe of enactments is empty.

**Proof**: If the universe of enactments of protocol \( \mathcal{P} \) is empty, then \( \forall m \in \text{msgs}(\mathcal{P}), [m]_{U\circ D}(\mathcal{P}) = \emptyset \). Similarly, \( \mathcal{P} \) is atomic.

Let a protocol of height 0 be a message schema which is an elementary protocol and therefore atomic. Let a protocol of height 1 be a protocol which references protocols of height 0 or less.

Suppose protocols of height \( n \) or less with an empty universe of enactments are atomic. Let \( \mathcal{Q} \) be a protocol of height \( n + 1 \) with an empty universe of enactments. Then all references \( \mathcal{R} \in \text{ref}(\mathcal{Q}) \) are of height \( n \) or less, and \( \mathcal{U}_{U\circ D}(\mathcal{R}) \subseteq \mathcal{U}_{U\circ D}(\mathcal{Q}) = \emptyset \), and so have empty universes of enactments. Thus \( \forall \mathcal{R} \in \text{ref}(\mathcal{Q}), \mathcal{R} \) is atomic by assumption and \( \mathcal{Q} = \emptyset \), so \( \mathcal{Q} \) is atomic. By induction, any protocol with an empty universe of enactments is atomic.

Conversely, if a protocol is both enactable and atomic then it must be live. Liveness concerns the protocol as a whole and guarantees it can always complete, while atomicity concerns compositions of distinct component protocols, recursively ensuring that each will complete if initiated.

**Theorem 2**: Any protocol that is enactable and atomic is live.

**Proof**: A protocol \( \mathcal{P} \) is live if for each \( \tau \in \mathcal{U}_{U\circ D}(\mathcal{P}), \exists \tau' \in [\mathcal{P}]_{U\circ D}(\mathcal{P}) \) such that \( \tau \leq \tau' \).

Suppose protocol \( \mathcal{P} \) is enactable. Then \( \exists m \in [\mathcal{P}]_{U\circ D}(\mathcal{P}) \).

Suppose \( m \) is a history vector in \( \mathcal{U}_{U\circ D}(\mathcal{P}) \), then either \( m \) is empty, or \( m \in [\mathcal{P}]_{U\circ D}(\mathcal{P}) \), for some message \( m \in \text{msgs}(\mathcal{P}) \).

If \( m \) is empty, then \( m \leq \tau \in [\mathcal{P}]_{U\circ D}(\mathcal{P}) \).

If \( m \in \text{ref}(\mathcal{P}) \), then by definition of atomicity \( \exists \tau' \in [\mathcal{P}]_{U\circ D}(\mathcal{P}) \) such that \( \tau \leq \tau' \).

If \( m \notin \text{ref}(\mathcal{P}) \), then \( \exists \tau \in [\mathcal{P}]_{U\circ D}(\mathcal{P}) \) such that \( \tau \leq \tau' \).

5. **VERIFICATION**

We have built a tool named Protocheck for automatically checking whether or not a protocol specification is atomic, to demonstrate that protocol atomicity is not just a theoretical property of protocols.

The verification process is similar to the method used by Singh [26]. We used a simple temporal logic to represent the definitions and constraints required for atomicity. The temporal logic itself was then implemented on top of a boolean logic solving library.

In this approach, each event is represented as a boolean variable. These events are then combined into expressions representing the integrity constraints and desired properties of the protocol. Finally, the expressions are evaluated using the `boolexpr` SAT solving library for Python.

### 5.1 Logic: Syntax and Semantics

The temporal logic language we adopt, Precedence, was used by Singh to verify the BSPL correctness properties of enactability, safety, and liveness [23, 26].

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. \( \overline{e} \) is not the simple negation or non-occurrence of \( e \), but an event indicating that \( e \) can never occur in the future. The terms \( e \cdot f \) and \( e \star f \), respectively, mean that \( e \) occurs prior to \( f \) and \( e \) and \( f \) occur simultaneously.

The Boolean operators: ‘\( \vee \)’ and ‘\( \wedge \)’ have the usual meanings.

The syntax follows conjunctive normal form:

- \( L_1 \) \( e \longrightarrow \) clause \( \wedge L_2 \)
- \( L_3 \) clause \( \longrightarrow \) term \( \vee \) clause
- \( L_4 \) term \( \longrightarrow \) event \( \mid \) event \( \cdot \) event \( \star \) event

The semantics of Precedence is given by pseudolinear runs of events (instances): “pseudo” because several events may occur together though there is no branching. Let \( \Gamma \) be a set of events where \( e \in \Gamma \) if and only if \( \overline{e} \in \Gamma \). A run is a function from natural numbers to the power set of \( \Gamma \), that is, \( \tau : \mathbb{N} \rightarrow 2^\mathbb{N} \).

The temporal logic itself is provided in Listing 8.

If \( m \notin \text{ref}(\mathcal{P}) \), then \( \exists Q \in \text{ref}(\mathcal{P}) \) such that \( m \in \text{msgs}(Q) \) and by atomicity \( [Q]_{U\circ D}(\mathcal{P}) \subseteq [\mathcal{P}]_{U\circ D}(\mathcal{P}) \). By induction \( [m]_{U\circ D}(\mathcal{P}) \) is a subset of \( [\mathcal{P}]_{U\circ D}(\mathcal{P}) \). Then by the definition of atomicity \( \exists \tau' \in [\mathcal{P}]_{U\circ D}(\mathcal{P}) \) such that \( \tau \leq \tau' \).

Thus in each case \( \tau \) can be extended to a history vector in \( [\mathcal{P}]_{U\circ D}(\mathcal{P}) \), so \( \mathcal{P} \) is live.

\( \square \)
5.2 Causality

We first define a set of clauses, $C_P$, which can be automatically derived from a protocol specification to represent the fundamental causality of BSPL enactments. Let $C_P$ be the conjunction of all clauses of the following types, illustrated with examples from Purchase in Listing 1.

1. Transmission: Each message must be sent to be received. (4 clauses)
   - $S$:PlaceOrder $\lor$ B:PlaceOrder
2. Emission: A message cannot be sent if its "out" or "nil" parameters have already been observed or if its "in" parameters are not observed. (4 clauses)
   - $S$:FillOrder $\lor$ (S:order $\land$ S:payment $\land$ S:product)
3. Reception: Either a message is not received or its "out" and "in" parameters are observed no later than the message. (4 clauses)
   - $S$:PlaceOrder $\lor$ S:PlaceOrder $\lor$ S:PlaceOrder
4. Minimality: For any role, if a parameter occurs, it occurs simultaneously with some message emitted or received. No role observes a parameter noncausally. (6 clauses)
   - $S$:product $\lor$ S:product $\lor$ S:FillOrder
5. Nonsimultaneity: A role cannot emit messages simultaneously with some message emitted or received. No parameters are not observed. (4 clauses)
   - B:Transfer $\lor$ B:Credit $\lor$ B:Transfer $\land$ B:Credit $\land$ B:Transfer

Based on the semantics of BSPL, an enactment of a protocol is valid if and only if it satisfies $C_P$. According to Singh [26], given a well-formed protocol $P$, for every viable history vector, there is a model of $C_P$ and vice versa.

5.3 Maximality

To support unbounded enactments and exclude failures caused by noncompliant agent behavior or transmission errors, we assume that each enactment is maximal. That is, every message will be sent and received unless it is prevented by an unmet precondition, such as an unavailable "in", or an observed "out" or "nil". The clause generated for the Transfer message is (B:Transfer $\lor$ B:order $\lor$ B:payment); either Transfer is transmitted, the prerequisite order is not observed, or a binding for payment already exists. We label the conjunction of these clauses for each message in protocol $P$ as $M_P$. By definition, an enactment is maximal if and only if it satisfies $M_P$.

If an enactment satisfies maximality yet is still incomplete, then it truly cannot be completed; there must be something other than intransigent agents or network failure preventing completion, namely, the protocol specification. This intuition is the basis of the liveness definition and verification technique used by Singh [26].

5.4 Enactability

We additionally construct clauses representing the enactability of a protocol $P$, labeled $E_P$.

An enactment satisfies $E_P$ if and only if it completes protocol $P$. A protocol is complete when each of its "out" parameters is produced by one of its messages, as outlined in Figure 1. For Purchase, the set of messages covering order is \{PlaceOrder\}, and the cover of product is \{FillOrder\}. Thus the resulting clause $E_{Purchase}$ is (S:PlaceOrder $\land$ B:FillOrder).

An algorithm generating $E_P$ is given in Figure 1. The algorithm iterates over each "out" parameter $p$ of a protocol $Q$. The occurrence of any message in $Q$ that has $p$ as an "out" parameter produces a binding for $p$, so the disjunction of all such occurrences (here labeled $cover_p$) accounts for all ways to bind $p$. As an enactment of $Q$ is complete when all of its "out" parameters is bound, we return the conjunction of all $cover_p$ as $E_Q$.

5.5 Atomicity

Suppose $P$ is a composition that includes (or may be identical to) some protocol $Q$ with reference $R$. Atomicity of a protocol requires both that each of its references be atomic, and that if the reference completes the protocol can also complete. Using the definition for $E_P$ given above, the statement that the completion of $R$ implies the completion of $Q$ can be written as $E_R \Rightarrow E_Q$. Thus if $Q$ is atomic in $UoD(P)$, any valid enactment of $Q$ will satisfy the formula $C_P \land M_P \land (\neg E_R \lor E_Q)$ for all $R \in ref(Q)$.

To prove that a composition is atomic, we verify that there are no valid nonatomic enactments by inverting the atomicity clause. Doing so yields the following formula: $C_P \land M_P \land E_R \land \neg E_Q$. If this formula cannot be satisfied for any $R$, then there are no valid, maximal, nonatomic enactments, and $Q$ must be atomic in $UoD(P)$.

With this formula, we can recursively verify the atomicity of every component in composition $P$, as shown by the algorithm in Figure 2.

In the base case, messages are always atomic and do not have any components for further recursion. The algorithm then iterates across each reference $R$ in $Q$, recursively testing for atomicity and checking that there are no enactments in which $R$ completes but $Q$ does not.

Having designed a verification procedure, we now prove it correctly verifies that a protocol $Q$ is atomic in $UoD(P)$ assuming that the clauses $C_P$, $M_P$, and $E_P$ correctly capture correctness, maximality, and enactability.
Theorem 3: A protocol $Q$ is atomic in $UoD(P)$ if and only if, for all $R$ in $ref(Q)$, $R$ is atomic and there is no enactment $\tau$ which satisfies $C_P \land M_P \land E_R \land \neg E_Q$.

Proof 3 (Proof): Let the atomicity of $Q$ be denoted $A_Q$. We desire to prove $\forall R \in ref(Q):$ atomic($R, P$) $\Rightarrow A_Q \Leftrightarrow \neg(C_P \land M_P \land E_R \land \neg E_Q)$. First, we assume from the definition that $R$ is atomic, and consider only enactments that satisfy $C_P \land M_P$ (that is, are correct and maximal), leaving $A_Q \Leftrightarrow \neg(E_R \land \neg E_Q)$. Distributing the negation reduces the statement to $\forall R \in ref(Q): A_Q \Leftrightarrow \neg E_R \land E_Q$.

Suppose protocol $Q$ is atomic in $UoD(P)$. By the definition of atomicity we know that for all $R \in ref(Q)$, any enactment $\tau$ in intension $[R]_{UoD(P)}$ can be extended with a finite number of message transmissions to some enactment $\tau'$ in intension $[Q]_{UoD(P)}$. That is, by assuming maximality, if it completes $R$ it also completes $Q$. So enactment $\tau'$ satisfies $E_R \land E_Q$, and $A_Q \Leftrightarrow \neg E_R \lor E_Q$.

Conversely, suppose for protocol $Q$ and all $R \in ref(Q)$, enactment $\tau$ satisfies $\neg E_R \lor E_Q$. By the definition of $E_P$, either $\tau$ completes $Q$ or it does not complete $R$. Thus, it is in the intension $[Q]_{UoD(P)}$ or it is not in the intension $[R]_{UoD(P)}$. Furthermore, by the assumption of maximality $\exists \tau' \in [R] \exists \tau \leq \tau'$, the definition of atomicity is vacuously satisfied, and $\neg E_R \lor E_Q \Leftrightarrow A_Q$. □

5.6 Results

<table>
<thead>
<tr>
<th>Protocol (Listing)</th>
<th>Atomic?</th>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purchase (Listing 1)</td>
<td>True</td>
<td>92</td>
</tr>
<tr>
<td>RefinedPurchase (Listing 3)</td>
<td>False</td>
<td>156</td>
</tr>
<tr>
<td>FixedPurchase (Listing 5)</td>
<td>True</td>
<td>390</td>
</tr>
<tr>
<td>ShareHealthData (Listing 6)</td>
<td>False</td>
<td>102</td>
</tr>
<tr>
<td>CreateOrder</td>
<td>False</td>
<td>690</td>
</tr>
</tbody>
</table>

Table 3: Protocheck results for example protocols.

Table 3 displays an overview of the results of running our implementation of Protocheck on each of the example protocols in this paper. The second column shows whether the protocol was verified as atomic or not. The third column shows how many clauses were generated according to the clause definitions given above. Note that the number of clauses varies because of the short-circuit nature of the recursive algorithm; FixedPurchase recurses through all of its components, while RefinedPurchase exits at the first violation.

The last entry, CreateOrder, is a specification of the Create Laboratory Order workflow defined by HL7 [13]. The workflow describes the process of a patient visiting a doctor and getting a sample collected, and a lab testing the sample and returning the results. A naive specification leaves several choices enabled when they should be mutually exclusive, as in the RefinedPurchase example. The first such choice is when the physician has the sample collected: they can collect it themselves, have a third party collect the sample before sending it to the lab, or have the lab collect the sample directly from the patient. A naive composition of these three alternatives violates atomicity, because the physician can initiate more than one of them but ultimately only one sample should be collected. When translating the workflow into a protocol, this naive approach was initially used to verify that our tool caught the violation. However, this protocol proved to be a better demonstration than first thought.

After fixing the violation by making the collection protocols mutually exclusive, the tool still reported that atomicity was violated. After looking more closely, it was discovered that a second violation had been overlooked until caught by the tool. Near the end of the workflow, the lab can either directly send the results to the physician, or simply send a notification that the results are available. The atomicity violation would occur if the physician queried the results, but was then sent them directly instead of as a response, preventing the query protocol from completing. This incident shows that atomicity is a useful property for verifying the design of composite protocols.

6 DISCUSSION

The idea of atomic action has a long history in multiagent systems; see, for example, Boissier et al. [5], Omicini and Zambonelli [21], and Alechina et al. [1]. However, unlike the present work, existing works either assume a shared memory or address a single-agent setting.

Notably, several diverse approaches for specifying protocols, including AUML [20], message sequence charts (MSCs) [14], choreography languages such as WS-CDL [31], and process calculi-inspired languages [2, 12] ignore information altogether and instead use control flow-based abstractions such as sequence, choice, and so on, to compose and constrain the enactment of protocols. However, specifying protocols in terms of control flow results in regimented enactments. Such lack of flexibility is true even of approaches such as RASA [18] and HAPN [30] that support declarative constraints on information values. Because the protocol enactments are guided by control flow constructs rather than information causality no conflicts can occur between components in a composition, just as the addition of states and transitions to a state machine does not affect the operation of other portions. For this reason we focus on the application of atomicity to the information-based approach exemplified by BSPL.

A direction for future work is to incorporate support for relative atomicity [17], so that a protocol is only required to complete after some critical subset is completed. Identifying the commitments created or discharged by each message could allow the discharging of all commitments to be used as a more precise correctness criterion. Similarly, identifying relationships between message effects should enable more sophisticated conflict resolutions such as reversion.

In contrast to the early work on commitment protocols [29, 32], newer work on commitments and more generally meanings [7, 8] leaves out altogether concerns such as message ordering and occurrence. The motivation is that meanings (of information) and causality are separate concerns. They are not disconnected however; as demonstrated in [15], the meanings are layered on top of information protocols. A rich direction is to explore further the connection between meaning-based and information protocols.

Acknowledgments

Thanks to the anonymous reviewers for helpful comments. Chopra was supported by the EPSRC grant EP/N027965/1 (Turtles). Christie and Singh were partially supported by the NCSU Laboratory for Analytic Sciences.
REFERENCES


