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:
Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. 2018. Compositional Correctness in Multiagent Interactions. In *Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), Stockholm, Sweden, July 10–15, 2018,* IFAAMAS, 9 pages.

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 the 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

Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), M. Dastani, G. Sukthankar, E. André, S. Koenig (eds.), July 10–15, 2018, Stockholm, Sweden. © 2018 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

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 programs 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

```
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 *enactable* if and only if a valid complete enactment exists. *Request Quote* is enactable, 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.

```
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 ($\vec{t}$), public roles ($\vec{x}$), private roles ($\vec{y}$), public parameters ($\vec{p}$), key parameters ($\vec{k} \subseteq \vec{p}$), ⌜in⌝ parameters ($\vec{p_I} \subseteq \vec{p}$), ⌜out⌝ parameters ($\vec{p_O} \subseteq \vec{p}$), ⌜nil⌝ parameters ($\vec{p_N} \subseteq \vec{p}$), private parameters ($\vec{q}$), and parameter bindings ($\vec{v}, \vec{w}$). Here, $\vec{p} = \vec{p_I} \cup \vec{p_O} \cup \vec{p_N}$, $\vec{p_I} \cap \vec{p_O} = \emptyset$, $\vec{p_I} \cap \vec{p_N} = \emptyset$, and $\vec{p_N} \cap \vec{p_O} = \emptyset$. And, t and p refer to an individual role and parameter, respectively. To reduce notation, we rename private 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 $\downarrow_x$ 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* $\mathcal{P}$ is a tuple $\langle n, \vec{x}, \vec{y}, \vec{p}, \vec{k}, \vec{q}, F \rangle$, where n is a name; $\vec{x}$, $\vec{y}$, $\vec{p}$, $\vec{k}$, and $\vec{q}$ are as above; and F is a finite set of f references, $\{F_1, \ldots, F_f\}$. $(\forall i : 1 \leq i \leq f \Rightarrow F_i = \langle n_i, \vec{x_i}, \vec{p_i}, \vec{k_i} \rangle$, where $\vec{x_i} \subseteq \vec{x} \cup \vec{y}$, $\vec{p_i} \subseteq \vec{p} \cup \vec{q})$, $\vec{k_i} = \vec{p_i} \cap \vec{k}$, and $\langle n_i, \vec{x_i}, \vec{p_i}, \vec{k_i} \rangle$ is the public projection of a protocol $\mathcal{P}_i$ (with roles and parameters renamed).

Definition 2: The *public projection* of a protocol $\mathcal{P} = \langle n, \vec{x}, \vec{y}, \vec{p}, \vec{k}, \vec{q}, F \rangle$ is given by the tuple $\langle n, \vec{x}, \vec{p}, \vec{k} \rangle$.

We treat a message schema ⌜$s \mapsto r : m\ \vec{p}(\vec{k})$⌝ as an atomic protocol with exactly two roles (sender and receiver) and no references: $\langle m, \{s, r\}, \emptyset, \vec{p}, \vec{k}, \emptyset, \emptyset \rangle$. Here $\vec{k}$ is the set of key parameters of the message schema. Usually, $\vec{k}$ is understood from the protocol in which the schema is referenced: $\vec{k}$ equals the intersection of $\vec{p}$ with the key parameters of the protocol declaration.

Below, let roles($\mathcal{P}$) = $\vec{x} \cup \vec{y} \cup \bigcup_i$ roles(F_i); params($\mathcal{P}$) = $\vec{p} \cup \vec{q} \cup \bigcup_i$ params(F_i); msgs($\mathcal{P}$) = $\bigcup_i$ msgs(F_i) and msgs($s \mapsto r : m\ \vec{p}$) = $\{m\}$. Definition 3 assumes that the message instances are unique up to the key specified in their schema.

Definition 3: A *message instance* $m[s, r, \vec{p}, \vec{v}]$ associates a message schema ⌜$s \mapsto r : m\ \vec{p}(\vec{k})$⌝ with a list of values, where $|\vec{v}| = |\vec{p}|$, where $\vec{v} \downarrow_p =$ ⌜nil⌝ if and only if $p \in \vec{p_N}$.

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^ρ captures the local view of an agent who might adopt role ρ during the enactment of a protocol.

Definition 4: A *UoD* is a pair $\langle \mathcal{R}, \mathcal{M} \rangle$, where $\mathcal{R}$ is a set of roles, $\mathcal{M}$ is a set of message names; each message specifies its parameters along with its sender and receiver from $\mathcal{R}$.

Definition 5: A *history* of a role ρ, H^ρ, is given by a sequence of zero or more message instances $m_1 \circ m_2 \circ \ldots$. Each m_i is of the form $m[s, r, \vec{p}, \vec{v}]$ where $\rho = s$ or $\rho = r$, and $\circ$ 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 $m[s, r, \vec{p}(\vec{k}), \vec{v}]$, the message under consideration, does not violate the uniqueness of the bindings. And, 2(ii) ensures that ρ knows the binding for each ⌜in⌝ parameter and not for any ⌜out⌝ or ⌜nil⌝ parameter.

Definition 6: A message instance $m[s, r, \vec{p}(\vec{k}), \vec{v}]$ is *viable* at role ρ's history H^ρ if and only if (1) $r = \rho$ (reception) or (2) $s = \rho$ (emission) and (i) $(\forall m_i[s_i, r_i, \vec{p_i}, \vec{v_i}] \in H^\rho$ if $\vec{k} \subseteq \vec{p_i}$ and $\vec{v_i} \downarrow_{\vec{k}} = \vec{v} \downarrow_{\vec{k}}$ then $\vec{v_i} \downarrow_{\vec{p} \cap \vec{p_i}} = \vec{v} \downarrow_{\vec{p} \cap \vec{p_i}})$ and (ii) $(\forall p \in \vec{p} : p \in \vec{p_I}$ if and only if $(\exists m_i[s_i, r_i, \vec{p_i}, \vec{v_i}] \in H^\rho$ and $p \in \vec{p_i}$ and $\vec{k} \subseteq \vec{p_i}))$.

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 $\langle \mathcal{R}, \mathcal{M} \rangle$ be a UoD. We define a *history vector* for $\langle \mathcal{R}, \mathcal{M} \rangle$ as a vector $[H^1, \ldots, H^{|\mathcal{R}|}]$, such that $(\forall s, r : 1 \leq s, r \leq |\mathcal{R}| : H^s$ is a history and $(\forall m[s, r, \vec{p}, \vec{v}] \in H^r : m \in \mathcal{M}$ and $m[s, r, \vec{p}, \vec{v}] \in H^s))$.

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 $\langle \mathcal{R}, \mathcal{M} \rangle$, $[H^1, \ldots, H^{|\mathcal{R}|}]$, 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, $(\exists i, m_j : H^i = H'^i \circ m_j$ and $[H^1, \ldots, H'^i, H^{|\mathcal{R}|}]$ 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 $\langle \mathcal{R}, \mathcal{M} \rangle$, the *universe of enactments* for that UoD, $\mathcal{U}_{\mathcal{R}, \mathcal{M}}$, is the set of viable history vectors, each of which has exactly $|\mathcal{R}|$ dimensions and each of whose messages instantiates a schema in $\mathcal{M}$.

Definition 10 states that the intension of a message schema is given by the set of viable history vectors on which that schema is instantiated, that is, an appropriate message instance occurs in the histories of both its sender and its receiver.

Definition 10*:* The intension of a message schema is given by: $[\![m(s,r,\vec{p})]\!]_{\mathcal{R},\mathcal{M}} = \{H | H \in \mathcal{U}_{\mathcal{R},\mathcal{M}}$ and $(\exists \vec{v}, i, j : H_i^s = m[s,r,\vec{p},\vec{v}]$ and $H_j^r = m[s,r,\vec{p},\vec{v}])\}$.

A (composite) protocol completes if one or more of subsets of its references completes. For example, *Purchase* yields two such subsets, namely, {*PlaceOrder*, *Transfer*, *FillOrder*} and {*PlaceOrder*, *Credit*, *FillOrder*}. Informally, each such subset contributes all the viable interleavings of the enactments of its members, that is, the intersection of their intensions. Definition 11 captures the *cover* as an adequate subset of references of a protocol, and states that the intension of a protocol equals the union of the contributions of each of its covers.

Definition 11*:* Let $\mathcal{P} = \langle n, \vec{x}, \vec{y}, \vec{p}, \vec{k}, \vec{q}, F\rangle$ be a protocol. Let $cover(\mathcal{P}, G) \equiv G \subseteq F | (\forall p \in \vec{p_O} : (\exists G_i \in G : G_i = \langle n_i, x_i, p_i\rangle$ and $p \in \vec{p_i}))$ and $\mathcal{P}$'s intension, $[\![\mathcal{P}]\!]_{\mathcal{R},\mathcal{M}} = (\bigcup_{\text{cover}(\mathcal{P},G)}(\bigcap_{G_i \in G}[\![G_i]\!]_{\mathcal{R},\mathcal{M}}))\downarrow_{\vec{x}}$.

The UoD of protocol $\mathcal{P}$ consists of $\mathcal{P}$'s roles and messages including its references recursively. For example, *Purchase*'s UoD $U = \langle\{$B, S$\}$, {*PlaceOrder*, *Transfer*, *Credit*, *FillOrder*}⟩.

Definition 12*:* The *UoD of a protocol* $\mathcal{P}$, UoD($\mathcal{P}$) = ⟨roles($\mathcal{P}$), msgs($\mathcal{P}$)⟩.

Definition 13*:* A protocol $\mathcal{P}$ is *enactable* if and only if $[\![\mathcal{P}]\!]_{\text{UoD}(\mathcal{P})} \neq \emptyset$.

Definition 14*:* A protocol $\mathcal{P}$ is *safe* if and only if each history vector in $[\![\mathcal{P}]\!]_{\text{UoD}(\mathcal{P})}$ is safe. A history vector is *safe* if and only if all key uniqueness constraints apply across all histories in the vector.

Definition 15*:* A protocol $\mathcal{P}$ is *live* if and only if each history vector in universe of enactments $\mathcal{U}_{\text{UoD}(\mathcal{P})}$ can be extended through a finite number of message emissions and receptions to a history vector in $\mathcal{U}_{\text{UoD}(\mathcal{P})}$ that is complete.

3 ATOMICITY CONCEPTS

We now motivate atomicity informally with the help of protocol specifications in BSPL.

Consider again the protocol *Purchase* in Listing 1. In *Purchase*, a buyer places an order and then tenders payment via either wire transfer or credit. *Transfer* and *Credit* conflict because both produce a binding for payment. The conflict makes them mutually exclusive: either B can send *Transfer* or *Credit* but not both. *Purchase* is both live and safe. It is atomic too.

To see why, recall the intuition behind atomicity from Section 1: (1) a protocol is atomic if each of its constituent protocols is atomic and (2) the completion of a component implies the composition can complete. Let's apply this concept to *Purchase*. In *Purchase*, all of its references *PlaceOrder*, *Transfer*, *Credit*, and *FillOrder* are atomic by 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

```
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

```
  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.

Listing 6: Sharing private health records

```
ShareHealthData {
  roles P, R, C // patient, researcher, clinic
  parameters out ID key, out granted, out revoked
  P ↦ C: Authorize[out ID, out granted]
  AccessData(R, C, in ID, in granted, nil revoked, out data)
  P ↦ C: Revoke[in ID, in granted, out revoked]
}
AccessData {
  roles R, C // researcher, clinic
  parameters in ID key, in granted, nil revoked, out data
  R ↦ C: Request[in ID, in granted, out req]
  C ↦ R: Provide[in ID, in req, out data, nil revoked]
}
```

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 ↦ R: Reject[in ID, in req, out data, in 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 ATOMICITY FORMALIZATION

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

Violation	Cause	Resolution
Mutual Exclusion	⌜out⌝&⌜out⌝	Enable only one
Indirect Exclusion	⌜in⌝&⌜out⌝	
Partial Ordering	⌜out⌝&⌜nil⌝	Provide other means of completion
Indirect Ordering	⌜in⌝&⌜nil⌝	

Table 1: Atomicity violations and their resolutions.

Additionally, we use $\tau \leq \tau'$ to mean that the history vector τ' is an extension of τ obtained by appending at most a finite number emissions and receptions.

$[\![\mathcal{R}]\!] \sqsubseteq [\![Q]\!]$ means $\forall \tau \in [\![\mathcal{R}]\!]$, $\exists \tau' \in [\![Q]\!]$ such that $\tau \leq \tau'$.

Definition 16*:* A protocol Q is atomic in universe of discourse U if and only if $\forall \mathcal{R} \in ref(Q)$,

(1) $\mathcal{R}$ is atomic in U, and
(2) $[\![\mathcal{R}]\!]_U \sqsubseteq [\![Q]\!]_U$

"$\mathcal{P}$ is atomic" or "the atomicity of $\mathcal{P}$" are shorthand for the atomicity of $\mathcal{P}$ in its own universe of discourse.

Although the definition considers only direct references, its recursive nature means that if any message is sent, every composition that includes it must eventually complete. This definition captures our intuition that initiating a component protocol should result in its eventual completion, all the way from the individual messages to the highest level composition.

The intension $[\![Q]\!]$ of protocol Q is the set of enactments that complete Q by the emission of at least one message from the cover of each of its ⌜out⌝ parameters. The universe of discourse specifies which roles and messages are involved in the enactments. Using the universe of discourse of a composition $\mathcal{P}$ that includes Q means that conflicts can occur between messages anywhere in the composition, rather than just within the one component protocol.

For example, $[\![Transfer]\!]_{UoD(Purchase)}$ projected to role B is: {[*PlaceOrder*, *Transfer*], [*PlaceOrder*, *Transfer*, *FillOrder*]} For this intension, each history vector in $[\![Transfer]\!]_{UoD(Purchase)}$ can be extended by a message reception to a history vector that completes *Purchase*, and the same is true for the other roles and components, so *Purchase* is atomic.

Conversely, $[\![OfferTransfer]\!]_{UoD(RefinedPurchase)}$ contains the enactment [*PlaceOrder*, *OfferCredit*, *OfferTransfer*, *AcceptCredit*, *PayCredit*] which cannot be extended to an enactment that completes *Transfer*, so *RefinedPurchase* is not atomic.

Because the cover of a protocol contains only messages within the protocol, each component protocol must be completed by its own messages to be atomic. Even if an enactment produces the same parameters via messages from another component Q', it is not in the intension of Q because its cover is not complete. Thus *Credit* is not completed by the binding of payment produced by *InitiateTransfer* because that message is not in the cover of *Credit*.

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.

	Atomic	Nonatomic
Safe	*Purchase*	*ShareHealthData*
Unsafe	*Purchase+Gift*	*RefinedPurchase*
Live	*Purchase*	*RefinedPurchase*
Non-live	*Transfer*	*Stuck*

Table 2: Protocols demonstrating orthogonality of properties.

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

Listing 8: Trivially nonlive and nonatomic protocol

```
Stuck {
  roles A, B
  parameters out begin key, out end
  A ↦ 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 1 (Proof)*:* If the universe of enactments of protocol $\mathcal{P}$ is empty, then $\forall m \in \text{msgs}(\mathcal{P})$, $[\![m]\!]_{UoD(\mathcal{P})} = \emptyset$. Similarly, $[\![\mathcal{P}]\!]_{UoD(\mathcal{P})} = \emptyset$

Let a protocol of height 0 be a message schema which is an elementary protocol and therefore atomic.

Let a protocol of height $n+1$ be a protocol which references protocols of height n or less.

Suppose protocols of height n or less with an empty universe of enactments are atomic. Let Q be a protocol of height $n+1$ with an empty universe of enactments. Then all references $\mathcal{R} \in ref(Q)$ are of height n or less, and $\mathcal{U}_{UoD(\mathcal{R})} \subseteq \mathcal{U}_{UoD(Q)} = \emptyset$, and so have empty universes of enactments. Thus $\forall \mathcal{R} \in ref(Q), \mathcal{R}$ is atomic by assumption and $[\![\mathcal{R}]\!]_{UoD(Q)} \subseteq \mathcal{U}_{UoD(Q)} = \emptyset$, so $[\![\mathcal{R}]\!]_{UoD(Q)} \sqsubseteq [\![Q]\!]_{UoD(Q)}$, and 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 2 (Proof)*:* A protocol $\mathcal{P}$ is live if for each $\tau \in \mathcal{U}_{\text{UoD}(\mathcal{P})}$, $\exists \tau' \in [\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$ such that $\tau \preceq \tau'$.

Suppose protocol $\mathcal{P}$ is enactable. Then $\exists \tau \in [\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$.

Suppose v is a history vector in $\mathcal{U}_{\text{UoD}(\mathcal{P})}$. Then either v is empty, or $v \in [\![m]\!]_{UoD(\mathcal{P})}$ for some message $m \in \text{msgs}(\mathcal{P})$.

If v is empty, then $v \preceq \tau \in [\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$.

If $m \in ref(\mathcal{P})$, then by the definition of atomicity $\exists v' \in [\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$ such that $v \preceq v'$.

If $m \notin ref(\mathcal{P})$, then $\exists Q \in ref(\mathcal{P})$ such that $m \in \text{msgs}(Q)$ and by atomicity $[\![Q]\!]_{UoD(\mathcal{P})} \sqsubseteq [\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$. By induction $[\![m]\!]_{UoD(\mathcal{P})} \sqsubseteq [\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$. Then by the definition of atomicity $\exists v' \in [\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$ such that $v \preceq v'$.

Thus in each case v can be extended to a history vector in $[\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$, so $\mathcal{P}$ is live. □

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$_1$. $I \longrightarrow$ *clause* | *clause* $\wedge I$
L$_2$. *clause* $\longrightarrow$ *term* | *term* $\vee$ *clause*
L$_3$. *term* $\longrightarrow$ *event* | *event* $\cdot$ *event*| *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 Γ 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 Γ, that is, $\tau : \mathbb{N} \mapsto 2^{\Gamma}$. 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, that is, $\tau_{[i,j]} = \langle \tau_i, \tau_{i+1} \ldots \tau_{j-i+1} \ldots \emptyset \ldots \rangle$. On any run, e or $\overline{e}$ may not both occur. Events are nonrepeating.

$\tau \models_i E$ means that τ satisfies E at i or later. We say τ is a *model* of expression E if and only if $\tau \models_0 E$. E is *satisfiable* if and only if it has a model.

M$_1$. $\tau \models_i e$ if and only if $(\exists j \geq i : e \in \tau_j)$
M$_2$. $\tau \models_i e \star f$ if and only if $(\exists j \geq i : \{e, f\} \subseteq \tau_j)$
M$_3$. $\tau \models_i r \vee u$ if and only if $\tau \models_i r$ or $\tau \models_i u$
M$_4$. $\tau \models_i r \wedge u$ if and only if $\tau \models_i r$ and $\tau \models_i u$
M$_5$. $\tau \models_i e \cdot f$ if and only if $(\exists j \geq i : \tau_{[i,j]} \models_0 e$ and $\tau_{[j+1,|\tau|]} \models_0 f)$

5.2 Causality

We first define a set of clauses, $\mathcal{C}_P$, which can be automatically derived from a protocol specification to represent the fundamental causal semantics of BSPL enactments. Let $\mathcal{C}_P$ 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)
$\overline{\text{S:PlaceOrder}} \vee \text{B:PlaceOrder}$
(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)
$\overline{\text{S:FillOrder}} \vee (\text{S:order} \wedge \text{S:payment} \wedge \overline{\text{S:product}})$
(3) Reception: Either a message is not received or its ⌜out⌝ and ∈ parameters are observed no later than the message. (4 clauses)
$\overline{\text{S:PlaceOrder}} \vee \text{S:order} \cdot \text{S:PlaceOrder} \vee \text{S:order} \star \text{S:PlaceOrder}$
(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)
$\overline{\text{S:product}} \vee \text{S:product} \star \text{S:FillOrder}$
(5) Nonsimultaneity: A role cannot emit messages simultaneously; they are sent in some order. (4 clauses)
$\overline{\text{B:Transfer}} \vee \overline{\text{B:Credit}} \vee \text{B:Transfer} \cdot \text{B:Credit} \vee \text{B:Credit} \cdot \text{B:Transfer}$

Based on the semantics of BSPL, an enactment of a protocol is valid if and only if it satisfies $\mathcal{C}_P$. According to Singh [26], given a well-formed protocol $\mathcal{P}$, for every viable history vector, there is a model of $\mathcal{C}_P$ 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 $\vee\ \overline{\text{B:order}} \vee$ 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 $\mathcal{P}$ as $\mathcal{M}_P$. By definition, an enactment is maximal if and only if it satisfies $\mathcal{M}_P$.

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 $\mathcal{P}$, labeled $\mathcal{E}_P$.

An enactment satisfies $\mathcal{E}_P$ if and only if it completes protocol $\mathcal{P}$. 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 $\mathcal{E}_{\text{Purchase}}$ is (S:PlaceOrder ∧ B:FillOrder).

An algorithm generating $\mathcal{E}_P$ is given in Figure 1. The algorithm iterates over each ⌜out⌝ parameter p of a protocol Q. The occurrence of any message m 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 $\mathcal{E}_Q$.

```
1: procedure $\mathcal{E}(Q)$
2:     for all $p \in \text{out}(Q)$ do
3:         $\text{cover}_p \leftarrow \bigvee\{m \in Q \mid p \in \text{out}(m)\}$
4:     return $\bigwedge_{p \in \text{out}(Q)} \text{cover}_p$
```

Figure 1: Algorithm generating $\mathcal{E}_Q$.

5.5 Atomicity

Suppose $\mathcal{P}$ is a composition that includes (or may be identical to) some protocol Q with reference $\mathcal{R}$. 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 $\mathcal{E}_P$ given above, the statement that the completion of $\mathcal{R}$ implies the completion of Q can be written as $\mathcal{E}_R \Rightarrow \mathcal{E}_Q$. Thus if Q is atomic in $UoD(\mathcal{P})$, any valid enactment of Q will satisfy the formula $\mathcal{C}_P \wedge \mathcal{M}_P \wedge (\neg\mathcal{E}_R \vee \mathcal{E}_Q)$ for all $\mathcal{R} \in ref(Q)$.

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: $\mathcal{C}_P \wedge \mathcal{M}_P \wedge \mathcal{E}_R \wedge \neg\mathcal{E}_Q$. If this formula cannot be satisfied for any R, then there are no valid, maximal, nonatomic enactments, and Q must be atomic in $UoD(\mathcal{P})$.

With this formula, we can recursively verify the atomicity of every component in composition $\mathcal{P}$, 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 $\mathcal{R}$ in Q, recursively testing for atomicity and checking that there are no enactments in which $\mathcal{R}$ completes but Q does not.

```
1: procedure atomic(Q, P)
2:     if Q is a message then return True
3:     for all R ∈ $ref(Q)$ do
4:         if ¬atomic(R, P) then return False
5:         if SAT($\mathcal{C}_P \wedge \mathcal{M}_P \wedge \mathcal{E}_R \wedge \neg\mathcal{E}_Q$) then
6:             return False
7:     return True
```

Figure 2: Atomicity verification process.

Having designed a verification procedure, we now prove it correctly verifies that a protocol Q is atomic in $UoD(\mathcal{P})$ assuming that the clauses $\mathcal{C}_P$, $\mathcal{M}_P$, and $\mathcal{E}_P$ correctly capture correctness, maximality, and enactability.

Theorem 3*:* A protocol Q is *atomic* in $UoD(\mathcal{P})$ if and only if, for all $\mathcal{R}$ in $ref(Q)$, $\mathcal{R}$ is atomic and there is no enactment τ which satisfies $C_P \land \mathcal{M}_P \land \mathcal{E}_R \land \neg\mathcal{E}_Q$.

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

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

Conversely, suppose for protocol Q and all $\mathcal{R} \in ref(Q)$, enactment τ satisfies $\neg\mathcal{E}_{\mathcal{R}} \lor \mathcal{E}_Q$. By the definition of $\mathcal{E}_{\mathcal{P}}$, either τ completes Q or it does not complete $\mathcal{R}$. Thus, it is in the intension $[\![Q]\!]_{UoD(\mathcal{P})}$ or it is not in the intension $[\![\mathcal{R}]\!]_{UoD(\mathcal{P})}$. Furthermore, by the assumption of maximality $\nexists\tau' \in [\![\mathcal{R}]\!] \ni \tau \preceq \tau'$, the definition of atomicity is vacuously satisfied, and $\neg\mathcal{E}_{\mathcal{R}} \lor \mathcal{E}_Q \Rightarrow A_Q$. □

5.6 Results

Protocol (Listing)	Atomic?	Clauses
Purchase (Listing 1)	True	92
RefinedPurchase (Listing 3)	False	156
FixedPurchase (Listing 5)	True	390
ShareHealthData (Listing 6)	False	102
CreateOrder	False	690

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 naïve 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 naïve 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 naïve 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 occurence. 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

[1] Natasha Alechina, Mehdi Dastani, and Brian Logan. 2012. Programming Norm-Aware Agents. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. IFAAMAS, Valencia, 1057–1064.

[2] Davide Ancona, Daniela Briola, Angelo Ferrando, and Viviana Mascardi. 2015. Global Protocols as First Class Entities for Self-Adaptive Agents. In *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems*. IFAAMAS, Istanbul, 1019–1029.

[3] Farhad Arbab. 2011. Puff, The Magic Protocol. In *Formal Modeling: Actors, Open Systems, Biological Systems*, Gul Agha, Olivier Danvy, and José Meseguer (Eds.). Springer, 169–206.

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

[5] Olivier Boissier, Rafael H. Bordini, Jomi Fred Hübner, Alessandro Ricci, and Andrea Santi. 2013. Multi-Agent Oriented Programming with JaCaMo. *Science of Computer Programming* 78, 6 (June 2013), 747–761.

[6] Amit K. Chopra, Samuel H. Christie V, and Munindar P. Singh. 2017. Splee: A Declarative Information-Based Language for Multiagent Interaction Protocols. In *Proceedings of the 16th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, São Paulo, 1054–1063.

[7] Amit K. Chopra and Munindar P. Singh. 2015. Cupid: Commitments in Relational Algebra. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2052–2059.

[8] Amit K. Chopra and Munindar P. Singh. 2016. Custard: Computing Norm States over Information Stores. In *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. IFAAMAS, Singapore, 1096–1105.

[9] Nicoletta Fornara and Marco Colombetti. 2003. 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)*. ACM Press, Melbourne, 520–527.

[10] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book* (2nd ed.). Pearson.

[11] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[12] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 273–284.

[13] Health Level Seven International. 2013. Laboratory Order Conceptual Specification. (2013). http://wiki.hl7.org/index.php?title=Laboratory_Order_Conceptual_Specification

[14] ITU. 2004. Message Sequence Chart (MSC). (April 2004). http://www.itu.int/ITU-T/2005-2008/com17/languages/Z120.pdf.

[15] Thomas Christopher King, Akın Günay, Amit K. Chopra, and Munindar P. Singh. 2017. Tosca: Operationalizing Commitments over Information Protocols. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI, Melbourne, 256–264.

[16] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)* 21, 7 (July 1978), 558–565.

[17] Nancy Lynch. 1983. Multilevel Atomicity—A New Correctness Criterion for Database Concurrency Control. *ACM Transactions on Database Systems* 8, 4 (Dec. 1983), 484–502.

[18] Tim Miller and Peter McBurney. 2011. Propositional Dynamic Logic for Reasoning about First-Class Agent Interaction Protocols. *Computational Intelligence* 27, 3 (2011), 422–457.

[19] Tim Miller and Jarred McGinnis. 2008. Amongst First-Class Protocols. In *Proceedings of the 8th International Workshop on Engineering Societies in the Agents World (ESAW 2007) (Lecture Notes in Computer Science)*, Vol. 4995. Springer, Athens, 208–223.

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

[21] Andrea Omicini and Franco Zambonelli. 1999. Coordination for Internet Application Development. *Autonomous Agents and Multi-Agent Systems* 2, 3 (Sept. 1999), 251–269.

[22] Munindar P. Singh. 1996. *Toward Interaction-Oriented Programming*. TR 96-15. Department of Computer Science, North Carolina State University, Raleigh. Available at http://www4.ncsu.edu/eos/info/dblab/www/mpsingh/papers/mas/iop.ps.

[23] Munindar P. Singh. 2003. Distributed Enactment of Multiagent Workflows: Temporal Logic for Service Composition. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. ACM Press, Melbourne, 907–914.

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

[25] Munindar P. Singh. 2011. LoST: Local State Transfer—An Architectural Style for the Distributed Enactment of Business Protocols. In *Proceedings of the 9th IEEE International Conference on Web Services (ICWS)*. IEEE Computer Society, Washington, DC, 57–64.

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

[27] Munindar P. Singh. 2014. Bliss: Specifying Declarative Service Protocols. In *Proceedings of the 11th IEEE International Conference on Services Computing (SCC)*. IEEE Computer Society, Anchorage, Alaska, 235–242.

[28] Mahadevan Venkatraman and Munindar P. Singh. 1999. Verifying Compliance with Commitment Protocols: Enabling Open Web-Based Multiagent Systems. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 2, 3 (Sept. 1999), 217–236.

[29] Michael Winikoff. 2007. Implementing Commitment-based Interactions. In *Proceedings of the 6th International Conference on Autonomous Agents and Multiagent Systems*. 1–8.

[30] Michael Winikoff, Nitin Yadav, and Lin Padgham. 2017. A New Hierarchical Agent Protocol Notation. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 1, 1 (July 2017), 1–75.

[31] WS-CDL. 2005. Web Services Choreography Description Language, Version 1.0. (Nov. 2005). http://www.w3.org/TR/ws-cdl-10/.

[32] Pınar Yolum and Munindar P. Singh. 2002. Flexible Protocol Specification and Execution: Applying Event Calculus Planning using Commitments. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. ACM Press, Bologna, 527–534.