Compositional Correctness in Multiagent Interactions

274

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 the causal dependencies between protocols. One particular problem that may arise from inadequate consideration of causal dependencies is that the enactment of a composite protocol may violate *atomicity*; that is, some components may be initiated but prevented from completing. We use this *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-based approaches.

KEYWORDS

Communication protocols; Engineering multiagent systems

1 INTRODUCTION

An interaction protocol specifies the rules of encounter between autonomous agents in a multiagent system. In this paper, we are primarily concerned operational protocols, that is, protocols where the purpose of the rules of encounter is to capture the constraints on the enactment of messages and, more generally, protocols. UML interaction diagrams and its variants for multiagent systems such as AUML [18] are among several languages developed to specify operational protocols. Meaning-based protocols, as exemplified by work on commitment protocols [5, 8, 28], are outside the present scope.

A protocol encapsulates a group of related interactions. 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 in order to obtain more complex protocols. Further, we would like to be able to compose protocols *declaratively* on the basis of *causality* as reflected in the relevant causal dependencies [23, 25]. Such dependencies capture what an agent must know (or not know) to produce a new piece of information.

For example, we may imagine a message specification *FillOrder* that cannot be sent by the seller until it has received an order to be filled from the buyer. Because of this dependency, a seller enacting a *Purchase* protocol composed of *FillOrder* and a message specification *PlaceOrder* that binds order must receive *PlaceOrder* before it can send *FillOrder*.

Although the foregoing example composes message specifications, note that a message specification is an elementary protocol and the ideas of causality-based composition applies to protocols in general. Thus we can describe enactments in terms of causal dependencies between information parameters. We refer to this approach as being *information-based*. The Blindindly Simple Protocol Language (BSPL) [22] and its extensions such Splee [7] are early exemplars of the information-based approach.

Compositions based on causality yield declarative specifications and support flexible enactments in completely decentralized settings over a fully asynchronous infrastructure [23]. However, because some compositions may undesirable, we propose *atomicity* as a correctness criterion for protocols. The basis is the observation that a message schema is *de facto* atomic: either its enactment occurs completely or not at all. A composed protocol is atomic if and only if the completion of one of its components implies that the composition can complete, and all of its constituent components are atomic in the context of the composition. Informally, our notion of atomic protocols is analogous to the notion of atomic programs [10] and transactions [9] in that it captures a notion of *all or nothing*, the difference being that the above are shared memory approaches whereas we address decentralized settings with no shared memory.

An example helps for concreteness. Take the above *Purchase* protocol, extended with *Transfer* and *Credit* protocols intended to offer a choice between two payment methods. The specification would be incorrect if both *Transfer* and *Credit* could be enacted in the same enactment of *Purchase*. More interestingly, *Purchase* would be nonatomic if there were enactments wherein *Transfer* had been partially enacted but could not be completed because *Credit* had completed. In such cases the enactment of *Transfer* is left *dangling*, indicating unfinished business. The enactment of *Transfer* may have set up commitments that cannot be discharged because it cannot complete. A violation of atomicity is thus potentially indicative of *semantic* errors.

Clearly, the problem in the foregoing example could be avoided if the buyer prudently enacts either *Transfer* or *Credit* but not both. Ideally, we would like to guarantee atomicity from the protocol specification alone, without resort to agent specifications. This reflects the essential motivation for protocols: capturing the *interaction logic* and presenting it in a reusable form [4, 17, 20].

Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), M. Dastani, G. Sukthankar, E. Andre, S. Koenig (eds.), July 2018, Stockholm, Sweden

https://doi.org/doi

For concreteness, we adopt BSPL as our information-based protocol language (Section 2). 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

BSPL [22] specifies protocol constraints in terms of causality, as described above, and *integrity* based on *key constraints* [9] 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 S (seller) and B (buyer); and two public parameters order and product. Parameter order is annotated key, meaning that order functionally determines the other parameters. Both parameters are adorned ⌜out⌝, meaning that their bindings are generated by enacting the protocol, i.e., 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 order 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 message *PlaceOrder* has order as a parameters annotated ⌜out⌝, meaning that a buyer can generate bindings for it when sending an instance of *PlaceOrder*. In *FillOrder*, order, and payment are ⌜in⌝, meaning that a seller may send an instance of *FillOrder* with some bindings for order and payment only if it has received an instance of *Transfer* or *Credit* with those bindings. Bindings for order identify enactments of *Purchase*. That is, distinct tuples of bindings as allowed by the key constraints correspond to distinct enactments. A *complete* enactment of *Purchase* corresponds to a tuple of bindings for all its public parameters.

Singh [24] formalizes BSPL and properties and gives verification techniques. Informally, protocol is *enactable* iff there is a complete enactment; i.e. a set of message instances that yields bindings for all of its public parameters. *Request Quote* is enactable, because the enactment *PlaceOrder* → *Transfer* → *FillOrder* is valid and binds all public parameters of *Purchase*.

A protocol is *live* iff 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. Imagine an alternative specification of *Purchase*, say *Purchase Alt*, without either *Transfer* or *Credit*. This would mean that no enactment of *Purchase Alt* would bind payment, preventing *FillOrder* from being sent and completing the enactment. *Purchase Alt* therefore would not be live.

Informally, a protocol is *safe* iff 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. 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 can concurrently produce bindings for payment; that is, a nonlocal conflict exists.

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 *RefinedPurchase* (Listing 3) replaces *Transfer* and *Credit* with composite protocols serving the same purpose.

Listing 3: Refined purchase protocol

```
RefinedPurchase {
  roles Buyer, S // Buyer, Seller
  parameters out order key, out payment, out product
  B ↦ S: PlaceOrder[out order]
  Transfer(B, S, in order, out payment)
  Credit(B, S, in order, out payment)
  S ↦ B: FillOrder[in order, in payment, out product]
}
```

2.1 Formalization

We adopt Singh's [24] formal framework. Definitions 1–15 are taken from there.

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 $\vec{p_N} \cap \vec{p_O} = \emptyset$. Also, 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.

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 $\ulcorner s \mapsto r : m\ \vec{p}(\vec{k}) \urcorner$ 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 $\text{roles}(\mathcal{P}) = \vec{x} \cup \vec{y} \cup \bigcup_i \text{roles}(F_i)$; $\text{params}(\mathcal{P}) = \vec{p} \cup \vec{q} \cup \bigcup_i \text{params}(F_i)$; $\text{msgs}(\mathcal{P}) = \bigcup_i \text{msgs}(F_i)$ and $\text{msgs}(s \mapsto r : m\ \vec{p}) = \{m\}$. 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 $\ulcorner s \mapsto r : m\ \vec{p}(\vec{k}) \urcorner$ with a list of values, where $|\vec{v}| = |\vec{p}|$, where $\vec{v} \downarrow_p = \ulcorner \text{nil} \urcorner$ iff $p \in \vec{p_N}$.

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. A history may be infinite in general but we assume each enactment in which a tuple of parameter bindings is generated is finite.

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 iff (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) $(\forall p \in \vec{p} : p \in \vec{p_I}$ iff $(\exists m_i[s_i, r_i, \vec{p_i}, \vec{v_i}] \in H^\rho$ and $p \in \vec{p_i}$ and $\vec{k} \subseteq \vec{p_i}))$.

Definition 7 captures that a *history vector* for a protocol is a vector of histories of role that together are causally sound: a message is received only if it has been emitted [14].

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. Further, we make no FIFO assumption about message delivery. The viability of the messages emitted by any role ensures that the progression is epistemically correct with respect to each role.

Definition 8*:* A history vector over $\langle \mathcal{R}, \mathcal{M} \rangle$, $[H^1, \ldots, H^{|\mathcal{R}|}]$, is *viable* iff 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, i.e., $(\exists i, m_j : H^i = H'^i \circ m_j$ 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, i.e., 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, i.e., 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 $cover(\mathcal{P}, G) \equiv \ G \subseteq F \,|\, (\forall p \in \vec{p} : (\exists G_i \in G : G_i = \langle n_i, x_i, p_i \rangle$ 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 $U = \langle \{\text{B}, \text{S}\}, \{$*PlaceOrder*, *Transfer*, *Credit*, *FillOrder*$\}\rangle$.

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

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

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

Definition 15*:* A protocol $\mathcal{P}$ is *live* iff each history vector in the universe of enactments $\text{UoD}(\mathcal{P})$ can be extended through a finite number of message emissions and receptions to a history vector in $\text{UoD}(\mathcal{P})$ 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 in the context of the composition and (2) the completion of a component implies the composition can complete.

Let's apply this concept to *Purchase*. In *Purchase*, *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 the refinement of *Purchase* 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 have any prerequisites or conflicts with each other. This means the buyer may send both and so initiate both *RefinedTransfer* and *RefinedCredit* at the same time. The seller may also accept both offers without conflict. Yet a conflict between the two protocols arises when the buyer makes the payments—both protocols cannot be completed without producing conflicting bindings for payment, leaving one of the two dangling.

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

For example, Listing **??** adds the offer parameter to *OfferTransfer* and *OfferCredit* messages, so that the buyer may initiate only one of the two payment protocols.

Listing 5: 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 *AccessData* in Listing 6.

Listing 6: Sharing private health records

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

In the *AccessData* protocol in Listing 6 the patient must authorize the sharing of their health records before a researcher can access the data. Until the access is revoked, the researcher may request the data at any time. An atomicity violation occurs if a patient revokes access to the data after the researcher has made a request, but before the data is provided. In that scenario, the clinic is no longer allowed to *Provide* the data, so *AccessData* will be left dangling.

Messages containing a ⌜nil⌝ parameter cannot be sent after it is bound, but may be sent before. The conflict between ⌜out⌝ and ⌜nil⌝ parameters causes the protocols to be partially ordered rather than mutually exclusive: the *ShareHealthRecords* 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.

Listing 7: Alternative path to complete *AccessData*

```
C ↦ R: Reject[in ID, in record, 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. Exclusive conflicts, such as that in *RefinedPurchase*, can be resolved by only enabling one of the conflicting protocols. Ordering conflicts, such as that in *AccessData*, can be resolved by adding an alternate path to completion. Indirect conflicts between an ⌜in⌝ parameter and an ⌜out⌝ or ⌜nil⌝ parameter are resolved in the same way; they simply go through more steps from when the parameter is initially bound before the conflict occurs.

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

Table 1: Atomicity violations and their resolutions.

4 ATOMICITY FORMALIZATION

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

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

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

Definition 16*:* A protocol $\mathcal{Q}$ is atomic in the context of $UoD(\mathcal{P})$ iff $\forall \mathcal{R} \in ref(\mathcal{Q})$,

(1) $\mathcal{R}$ is atomic in the context of $UoD(\mathcal{P})$, and
(2) $[\![\mathcal{R}]\!]_{UoD(\mathcal{P})} \sqsubseteq [\![\mathcal{Q}]\!]_{UoD(\mathcal{P})}$

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

Although the definition considers only direct references, its recursive nature means that if any message is sent, every composition containing 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 leaf messages to the highest level composition.

The intension $[\![\mathcal{Q}]\!]$ of protocol $\mathcal{Q}$ is the set of enactments that complete $\mathcal{Q}$ 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 roles and messages of the root composition $\mathcal{P}$ for the universe of discourse means that conflicts can occur between messages anywhere in the composition, rather than just within the one component protocol.

For example, $[\![\mathit{Transfer}]\!]_{UoD(\mathrm{Purchase})}$ projected to just the history vector of B is:

{[*PlaceOrder*, *Transfer*], [*PlaceOrder*, *Transfer*, *FillOrder*]}

For this intension, each history vector in $[\![\mathit{Transfer}]\!]_{UoD(\mathrm{Purchase})}$ 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 of *Purchase*, so it is atomic.

Conversely, $[\![\mathit{OfferTransfer}]\!]_{UoD(\mathrm{RefinedPurchase})}$ 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 $\mathcal{Q}'$, it is not in the intension of $\mathcal{Q}$ 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*.

Theorem 1*:* Any protocol containing no enactable messages is atomic.

Proof 1 (Proof)*:* Let $\mathcal{P}$ be a protocol such that none of its messages is enactable. Then the intension of each message, $[\![\mathcal{M}]\!]_{UoD(\mathcal{P})}$ is empty.

Now consider some reference $\mathcal{R} \in ref(\mathcal{P})$. Either $\mathcal{R}$ is a message, and so its intension is empty, or it is a composite protocol. If it is a composite protocol, then it also has references $ref(\mathcal{R})$ and $[\![\mathcal{R}]\!]_{UoD(\mathcal{P})} \subseteq \bigcup_{\mathcal{R}' \in ref(\mathcal{R})} [\![\mathcal{R}']\!]_{UoD(P)}$ So the intension of each $\mathcal{R}$ from $ref(\mathcal{P})$ is either empty, or a subset of a union of other intensions. By induction, $[\![\mathcal{R}]\!]_{UoD(\mathcal{P})}$ is empty. The definition of atomicity requires that $[\![\mathcal{R}]\!]_{UoD(\mathcal{P})} \sqsubseteq [\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$ but $[\![\mathcal{R}]\!]_{UoD(\mathcal{P})}$ is empty, so the statement is vacuously true and the protocol is atomic.

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

Proof 2 (Proof)*:* A protocol $\mathcal{P}$ is live if for each $\tau \in UoD(\mathcal{P})$, $\exists \tau' \in [\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$ such that $\tau \preceq \tau'$.

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

Suppose υ is a history vector in $UoD(\mathcal{P})$. Then either υ is empty, or $\upsilon \in [\![\mathcal{M}]\!]_{UoD(\mathcal{P})}$ for some message $\mathcal{M} \in \mathcal{P}$.

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

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

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

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

4.1 Distinction from existing properties

In some sense all protocol faults can be interpreted as violations of either liveness or safety [2], but atomicity is a useful property not trivially related to either as defined for protocols in BSPL.

Because there are protocols that exemplify every combination of atomicity and safety or liveness, as shown in Table 2, atomicity is orthogonal to both.

	Atomic	**Non-atomic**
Safe	*Purchase*	*AccessData*
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]
}
```

Also, though the techniques used are similar, the procedure for verifying atomicity also differs from that used for verifying liveness by Singh [24]. Liveness concerns the protocol as a whole and guarantees it can always complete, while atomicity concerns compositions of self-contained components, recursively checking that each will complete if initiated.

5 VERIFICATION

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

This implementation was written from scratch for this project, but the verification process is similar to the method used by Singh [24]. 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. The expressions are then evaluated using the *boolexpr* SAT solving library for Python.

5.1 Logic Semantics

The temporal logic language we adopt, Precedence, was used by Singh to verify the BSPL correctness properties of enactability, safety, and liveness [21, 24].

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. $I \longrightarrow \ clause \mid clause \wedge I$

L_2. $clause \longrightarrow \ term \mid term \vee clause$

L_3. $term \longrightarrow \ event \mid event \cdot event \mid event \star event$

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

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

M_1. $\tau \models_i e$ iff $(\exists j \geq i : e \in \tau_j)$

M_2. $\tau \models_i e \star f$ iff $(\exists j \geq i : \{e, f\} \subseteq \tau_j)$

M_3. $\tau \models_i r \vee u$ iff $\tau \models_i r$ or $\tau \models_i u$

M_4. $\tau \models_i r \wedge u$ iff $\tau \models_i r$ and $\tau \models_i u$

M5. $\tau \models_i e \cdot f$ iff $(\exists j \geq i : \tau_{[i,j]} \models_0 e$ and $\tau_{[j+1,|\tau|]} \models_0 f)$

5.2 Causality

We first define a set of clauses, $\mathcal{C}_P$, representing 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 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 $\in$ 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 [24], 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 failure caused by non-compliant agent behavior or transmission failure, 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 $(\text{B:Transfer} \vee \overline{\text{B:order}} \vee \text{B:payment})$. 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. This is also the basis of the liveness verification technique used by Singh [24].

5.4 Enactability

We also 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 $(\text{S:PlaceOrder} \wedge \text{B:FillOrder})$.

An algorithm generating $\mathcal{E}_P$ is given in Figure 1. The algorithm iterates over each ⌜out⌝ parameter p of some protocol Q. The occurrance of any message m in Q that contains 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 E(Q)
2:     for all p ∈ out(Q) do
3:         cover_p ← ⋁{m ∈ Q | p ∈ out(m)}
4:     return ⋀_{p∈out(Q)} cover_p
```

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

5.5 Atomicity

Atomicity of a composition requires both that each of its references be atomic, and that if the reference completes the composition can also complete. Using the definition for $\mathcal{E}_P$ given above, this can be written as $\mathcal{E}_R \Rightarrow \mathcal{E}_Q$ for protocol $\mathcal{Q}$ with reference $\mathcal{R}$. Thus if $\mathcal{Q}$ is atomic in the context of $\mathcal{P}$, any valid enactment of $\mathcal{Q}$ will satisfy the formula $\mathcal{C}_P \wedge \mathcal{M}_P \wedge (\neg\mathcal{E}_R \vee \mathcal{E}_Q)$ 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. This gives 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 the protocol must be atomic.

With this formula, we can recursively verify the atomicity of every component in a protocol $\mathcal{P}$, 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 $\mathcal{Q}$, recursively testing for atomicity. If the reference is atomic in the context of $\mathcal{P}$, the formula given above is used to prove that there are no enactments in which $\mathcal{R}$ can complete but $\mathcal{Q}$ cannot. If no such enactment exists for any reference, the protocol $\mathcal{Q}$ is proven atomic in the context of $\mathcal{P}$.

```
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(C_P ∧ M_P ∧ E_R ∧ ¬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 $\mathcal{Q}$ is atomic in the context

of $\mathcal{P}$, assuming that the clauses $\mathcal{C}_P$, $\mathcal{M}_P$, and $\mathcal{E}_P$ correctly capture correctness, maximality, and enactability.

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

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

Suppose protocol $\mathcal{Q}$ is atomic in the context of $\mathcal{P}$. By the definition of atomicity we know that for all $\mathcal{R} \in ref(\mathcal{Q})$, 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 $[\![\mathcal{Q}]\!]_{UoD(\mathcal{P})}$. That is, by assuming maximality, if it completes $\mathcal{R}$ it also completes $\mathcal{Q}$. So enactment τ' satisfies $\mathcal{E}_\mathcal{R} \wedge \mathcal{E}_\mathcal{Q}$, and $A_\mathcal{Q} \Rightarrow \neg\mathcal{E}_R \vee \mathcal{E}_\mathcal{Q}$.

Conversely, suppose for protocol $\mathcal{Q}$ and all $\mathcal{R} \in ref(\mathcal{Q})$, enactment τ satisfies $\neg\mathcal{E}_\mathcal{R} \vee \mathcal{E}_\mathcal{Q}$. By the definition of $\mathcal{E}_\mathcal{P}$, either τ completes $\mathcal{Q}$ or it does not complete $\mathcal{R}$. Thus, it is in the intension $[\![\mathcal{Q}]\!]_{UoD(\mathcal{P})}$ 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 $\neg\mathcal{E}_\mathcal{R} \vee \mathcal{E}_\mathcal{Q} \Rightarrow A_\mathcal{Q}$. □

5.6 Results

After creating the above implementation, we ran it against each of the example protocols, producing the results in Table 3 below.

Protocol (Listing)	Atomic?	Clauses
Purchase (1)	True	92
RefinedPurchase (3)	False	156
FixedPurchase (5)	True	390
AccessData (6)	False	75
CreateOrder	False	690

Table 3: Protocheck results for example protocols.

Table 3 contains an an overview of the results of running 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 [12]. A naïve specification leaves several choices enabled when they should be mutually exclusive, as in the *RefinedPurchase* example, such as whether the physician should collect a sample themselves or send the patient to a specialist. After fixing the first obvious problem the tool still reported failure, revealing a second such point in the protocol that had been overlooked.

6 DISCUSSION

The idea of atomic action, e.g., as in [6, 19] and planning [1] has a long history in multiagent systems. However, existing works either assume a shared memory or address a single-agent setting. The idea of atomic interaction in the context of composition is however a novel one.

To discuss atomicity in the context of protocol composition requires the specifications to support meaningful interaction between components. That is, instead of simply performing the component protocols sequentially or concatenating their contents, they should remain distinct yet capable of affecting each other. If the components are not kept distinct, there is no way to apply the "all or nothing" concept to them; we are left with liveness of the composition alone. Similarly, if the components cannot meaningfully interact with each other, they will behave in composition exactly as they do in isolation—no interference capable of causing a violation is possible.

Notably, several diverse approaches for specifying protocols, including AUML [18], message sequence charts (MSCs) [13], choreography languages such as WS-CDL [27], and process calculii-inspired languages [3, 11] 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 leads to regimented enactments, limiting interaction between components. This is true even of approaches such as RASA [16] and HAPN [26] which support declarative constraints on information values. Because the protocol enactments are guided by control flow constructs rather than information parameters, there is little meaningful interaction between components, 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.

One direction for future work is to explore the value of atomicity in the context of step-wise refinements of protocols. Ideally, step-wise refinement would preserve the liveness and safety of the protocol under consideration. Another direction is to add support for *relative atomicity* [15] by identifying critical subsets of protocols. Instead of requiring an entire protocol to complete when any of its references are completed, it may only need to be completed after some critical subset has completed. Adding higher level concepts will enable more robust and sensitive guarantees of atomicity. Identifying which messages create or discharge commitments would allow the discharging of all commitments to be used as a more precise criterion for the correctness of an enactment. Similarly, identifying relationships between actions in a protocol might enable reverting them, and thus support a wider variety of ways to resolve conflicts.

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'12)*. IFAAMAS, Valencia, 1057–1064.

[2] Bowen Alpern and Fred B. Schneider. 1985. Defining liveness. *Inform. Process. Lett.* 21, 4 (October 1985), 181–185.

[3] Davide Ancona, Daniela Briola, Angelo Ferrando, and Viviana Mascardi. 2015. Global protocols as first class entities for self-adaptive agents. In *Proceedings of the Fourteenth International Conference on Autonomous Agents and Multiagent Systems*. IFAAMAS, Istanbul, 1019–1029.

[4] Farhad Arbab. 2011. Puff, The Magic Protocol. In *Formal Modeling: Actors, Open Systems, Biological Systems*. Springer, 169–206.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

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

[14] 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.

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

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

[17] 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.

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

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

[20] 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.

[21] 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.

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

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

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

[25] 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.

[26] Michael Winikoff, Nitin Yadav, and Lin Padgham. 2017. A New Hierarchical Agent Protocol Notation. *Autonomous Agents and Multi-Agent Systems* (July 2017).

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

[28] 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*. ACM Press, 527–534.