Refinement for Multiagent Protocols

Samuel H. Christie V
North Carolina State University
schrist@ncsu.edu

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

Munindar P. Singh
North Carolina State University
mpsingh@ncsu.edu

ABSTRACT

An interaction protocol specifies a decentralized multiagent system operationally by specifying constraints on messages exchanged by its member agents. Engineering with protocols requires support for a notion of *refinement*, whereby a protocol may be substituted without loss of correctness by one that refines it. We identify two desiderata for refinement. One, *generality*: refinement should not restrict enactments by limiting protocols or infrastructures under consideration. Two, *preservation*: to facilitate modular verification, refinement should preserve liveness and safety.

We contribute a novel formal notion of protocol refinement based on enactments. We demonstrate generality by tackling the declarative framework of information protocols. We demonstrate preservation by formally establishing that our notion of refinement is safety and liveness preserving. We show the practical benefits of refinement by implementing a checker. We demonstrate that it is less time-intensive to check refinement (and thereby gain safety and liveness) than to recheck safety and liveness of a composition.

ACM Reference Format:
Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. 2020. Refinement for Multiagent Protocols. In *Proc. of the 19th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2020), Auckland, New Zealand, May 9–13, 2020,* IFAAMAS, 9 pages.

1 INTRODUCTION

We are concerned with multiagent systems (MAS) where each agent represents an autonomous real-world principal. Such systems are decentralized: agents exercise independent decision making and engage in arms-length communications with each other via asynchronous messaging. An interaction protocol enables agents in a MAS to coordinate their computations by specifying constraints on messaging, including the format, content, and conditions under which an agent may send and receive a message. A protocol being the primary operational specification for a MAS makes protocols crucial in engineering MAS. Notable approaches for specifying protocols include AUML [27], trace expressions (Trace) [1, 13], state machines [5], hierarchical state machines (HAPN) [38], session types [17, 18], and information protocols (BSPL) [30].

Protocols are doubly modular [35]. Along one dimension of modularity, each role of a protocol is a module. Role-as-module raises the question of when a role specification, derived from a protocol, may be substituted by another specification. A role R' *conforms* to role R if and only if R' can interoperate with any set of roles with which R can interoperate [4, 5, 12]. However, existing works don't handle interactions between three or more parties and several impose FIFO message delivery. Works that tackle multiparty settings are restricted to protocols in which only one agent can make an autonomous (internal) choice, thereby limiting concurrency.

Along the other dimension of modularity, a protocol is a module that may be composed with other protocols [24, 36]. Protocol-as-module raises the question of when a protocol P may be substituted in a composition by a protocol Q: Q must be a specialization of P.

Refinement captures a notion of specialization that legitimizes substitution. Suppose *Purchase* is a protocol that facilitates goods for payment transactions between buyers and sellers in which either buyer or seller may initiate a transaction. A refinement of *Purchase* could be *Purchase-Escrow*, where instead of paying the seller directly, the buyer pays an escrow service, which pays the seller when appropriate. Another refinement could be *Buyer-Initiates*, where only the buyer may initiate the transaction.

We identify two desirable features of protocol refinement: One, *generality*: refinement doesn't restrict protocols or the infrastructure. Two, *correctness preservation*: substitution by a refinement preserves correctness properties, specifically, liveness and safety, of the composition. That is, if the original composition satisfies liveness (or safety), then a composition resulting from the substitution must also satisfy liveness (or safety). Practically, checking refinement once saves the effort of checking every composition post substitution. Suppose we established that a composition using *Purchase* were safe. Then, substituting *Purchase-Escrow* for *Purchase* in that composition would yield a safe composition.

Our contribution is two-fold. One, we contribute a notion of protocol refinement that satisfies both of the foregoing desiderata. We satisfy generality by formalizing refinement for *information protocols* [30, 31], which are declaratively described and whose semantics supports asynchronous messaging without ordering guarantees. We satisfy preservation by proving that substituting a constituent of a composition by the constituent's refinement ensures safety and liveness. We prove that the composition obtained from substituting a constituent by its refinement is itself a refinement of the original composition.

Two, we contribute an algorithm for refinement checking and its implementation in a tool. We show that checking refinement is no more time-intensive than checking liveness and safety, the properties preserved. The relative performance makes it worthwhile to check refinement once rather than checking the liveness and safety of each individual composition post substitution.

Section 2 introduces information protocols. Section 3 motivates refinement with examples. Section 4 formalizes our definition. Section 5 presents important theorems. Section 6 presents an empirical evaluation of our tool. Section 7 discusses related work. Section 8 concludes with a discussion.

2 PROTOCOLS

We briefly describe the main ideas of information protocols with the help of examples. Listing 1 gives a protocol named *Purchase*.

Proc. of the 19th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2020), B. An, N. Yorke-Smith, A. El Fallah Seghrouchni, G. Sukthankar (eds.), May 9–13, 2020, Auckland, New Zealand.

Listing 1: A simple purchase protocol.

```
Purchase {
  roles B, S // Buyer, Seller
  parameters out ID key, out item, out price, out payment, out
       deliver
  B ↦ S: rfq[out ID key, out item]
  S ↦ B: quote[in ID key, in item, out price]
  B ↦ S: pay[in ID key, in item, in price, out payment]
  S ↦ B: ship[in ID key, in item, in price, out deliver]
}
```

Purchase specifies roles B and S, and a list of public parameters which must be bound to complete an enactment. Parameter ID is marked *key* and uniquely identifies each enactment by its binding. Each parameter is marked ⌜in⌝ or ⌜out⌝ to indicate whether a given protocol or message depends on or binds that parameter, thereby capturing causality constraints. There are four messages, each of which may be sent at any time provided all of their ⌜in⌝ parameters are bound. When B sends *rfq* to S, it produces bindings for ID and item. Once S observes the ID and item parameters, it may send the *quote* message, binding price. Then B can respond with *pay*, binding payment, and S can send *ship*, binding deliver. Here, *pay* and *ship* can be sent in parallel, because they do not depend on each other. Once both have been sent, the protocol is complete because all of its public parameters are bound.

An enactment of a protocol is identified by bindings of its key parameters. BSPL requires information *integrity*: any parameter may be bound only once in an enactment. Therefore, no role may send two messages with the same ⌜out⌝ parameter in any enactment. That is, *accept* and *reject* in Listing 2, which both bind decision, are mutually exclusive; the buyer may send at most one of them.

Listing 2: Mutually exclusive messages.

```
B ↦ S: accept[in offer key, out accept, out decision]
B ↦ S: reject[in offer key, out reject, out decision]
```

Although other notions of safety exist in other contexts, an information protocol is defined to be *safe* if and only if for any tuple of bindings for its key parameters, any other parameter is bound exactly once. Whereas a role can ensure that it binds any parameter once based on its own local history of interactions, safety ensures that no two roles may bind the same parameter. *Purchase* in Listing 1 is safe. However, if you modify *Purchase* in Listing 1 so that in *pay*, price is adorned ⌜out⌝ instead of ⌜in⌝, the resulting protocol would be unsafe as both buyer and seller may bind payment. Let's refer to this modified protocol as *Unsafe-Purchase*.

A protocol is *live* if and only if each enactment can be extended by the emission of some messages to a complete enactment. *Purchase* in Listing 1 is live. However, *Unsafe-Purchase* is not live because in the enactment where the buyer receives *quote* before sending *pay*, *pay* is disabled, meaning that a binding for payment—required for completion—cannot be produced.

In an information protocol, polymorphism is expressed via messages that have the same name and schema but different adornments for some of the parameters. Because each parameter can be bound only once, a role must make a choice between multiple messages that bind the same parameter.

Listing 3: Polymorphic RFQ protocol.

```
Polymorphic-RFQ {
  roles B, S
  parameters out ID key, out item, out price
  B ↦ S: greet[out ID key]
  B ↦ S: rfq[out ID key, out item]
  S ↦ B: offer[in ID key, out item, out price]
  S ↦ B: offer[in ID key, in item, out price]
}
```

The *Polymorphic-RFQ* protocol in Listing 3 demonstrates simple polymorphism and choice: B has two ways to bind ID, and S has one way to bind price in response to each, both named *offer* to make the polymorphism clear. This example shows how multiple mutually exclusive choices can be specified, and the power to bind a parameter can be given to different roles based on those choices.

3 MOTIVATING PROTOCOL REFINEMENT

Intuitively, we consider protocol Q to refine protocol P if its enactments are at least as elaborate with at most the same flexibility.

An enactment q of Q is *elaborates on* enactment p of P if q binds the parameters in p in the same order. By elaborating on p, q supports any actions or extensions that p does. A protocol is less flexible if there are fewer decisions or *alternative paths* available during an enactment. For example, in a flexible protocol either the buyer or seller may name the price, whereas a less flexible refinement may specify that only the seller may do so.

3.1 Enactment Paths

To frame the motivations of refinement using examples and later formalize the concepts, we first describe our intuitions and a notation for illustrating them.

A crucial intuition is that if Q refines P, then every enactment of Q *maps* to an enactment of P [22]. Q may have fewer enactments and its enactments may have more details, but each enactment binds the same parameters as some enactment of P.

Under BSPL's semantics, an enactment of a protocol is a vector of role histories, which can be denoted by $[r_1{:}h_1, r_2{:}h_2, \ldots, r_n{:}h_n]$, where n is the number of roles in the protocol, r_i is the name of the ith role, and h_i is the history for role r_i. Each history is a sequence of message instances. Each message instance is a tuple of name, sender, receiver, and parameter-value pairs. A complete enactment is one in which all of the protocol's public parameters appear.

To more explicitly and concisely capture the choices made during an enactment, we project an enactment to a sequential view of events called a *path*. A path represents a possible perspective on the relative ordering of events: if two events A and B are independent then both [A, B] and [B, A] are valid paths. A path is a sequence of message instances, each with sender s, recipient r, key parameters $\vec{k}$, parameter list $\vec{p}$, and optional transmission offset o, written as $\langle s{\mapsto}r, \vec{k}|\vec{p}\rangle_o$. We omit the offset parameter when it is 0.

The only use of paths is to represent asynchronous enactments (history vectors) compactly. Suppose a path is $[m_0, m_1, \ldots]$. For a role R, order its emissions in the same order as in the path. If m_j with offset k is directed at R, the m_j arrives any time after all of R's emissions from m_0 to m_{j+k} have been made and before any of R's emissions from m_{j+k+1} onward is made.

To be a valid path, each message instance must be enabled by the information known to the sender from the subpath preceding it. A complete path is a path that represents a complete enactment; not all valid paths are complete. The set of all paths extending a path is its set of *branches*; that is, paths with that path as a prefix.

For brevity, we do not duplicate the keys in the parameter list, and show only the parameters introduced by (that is, observed by the sender or recipient for the first time in) a message.

Listing 4 shows an example path for *Purchase*. Three of the messages have a offset of 0, meaning they are received before the next message is sent. The third message, introducing a binding for the parameter payment, has a offset of 1 to indicate that it is not received until *after* the following message emission occurs, capturing the case where *pay* and *ship* are being transmitted concurrently, since they are received at the same point in the path.

Listing 4: An example path in Purchase.

```
[⟨b↦s, ID | item⟩0 ,
 ⟨s↦b, ID | price⟩0 ,
 ⟨b↦s, ID | payment⟩1 ,
 ⟨s↦b, ID | deliver⟩0 ]
```

The following sections give examples of how a refinement may be less flexible or more elaborate than the protocol it refines.

3.2 Running Example

We illustrate several kinds of refinement via variants of the following protocol, the overall composition of which is given in Listing 5.

Listing 5: Composition for developing further examples.

```
Commerce {
  roles B, S, C // Buyer, Seller, Catalog
  parameters out ID key, out item, out shipped
  private price, payment

  Either-Starts(B, S, out ID key, out item)
  Lookup-Prices(S, C, in ID key, out query key, out price)
  S ↦ B: quote[in ID key, in item, in price]
  Flexible-Payment(B, S, in ID key, in item, in price, out
      payment, out shipped)
}
```

This composition *references*, i.e., includes as constituents, three protocols. *Either-Starts* specifies the beginning of the enactment, where either the buyer or the seller can request a quote or recommend an item, respectively. Then, the seller queries the catalog for the price information regarding the item being sold in *Lookup-Prices*. The *quote* message forwards the price information to the buyer, so it can proceed with payment. And, *Flexible-Payment* describes the conclusion of the transaction, in which the buyer can pay before or after the seller ships the product.

3.3 Polymorphism Reduction

The basic RFQ protocol in Listing 6 is a refinement of the polymorphic RFQ protocol given in Listing 3.

Listing 6: Simple RFQ protocol.

```
RFQ {
  roles B, S
  parameters out ID key, out item, out price
  B ↦ S: rfq[out ID key, out item]
  S ↦ B: offer[in ID key, in item, out price]
}
```

The complete paths of *Polymorphic-RFQ* are:

```
[⟨B↦S, ID |⟩ , ⟨S↦B, ID | item, price⟩ ] ,
[⟨B↦S, ID | item⟩ , ⟨S↦B, ID | price⟩ ]
```

The only complete path of *RFQ* is:

```
[⟨B↦S, ID | item⟩ , ⟨S↦B, ID | price⟩ ]
```

Thus the paths of *RFQ* are a subset of the paths of *Polymorphic-RFQ*, so *RFQ* is a refinement.

3.4 Initiation Reduction

Initiation reduction removes alternative initiating messages that may be sent by different roles. Consequently, the resulting protocol has a subset of otherwise identical paths.

Listing 7: Either-Starts protocol.

```
Either-Starts {
  roles B, S
  parameters out ID key, out item
  B ↦ S: rfq(out ID key, out item)
  S ↦ B: recommend(out ID key, out item)
}
// complete paths: [⟨B↦S, ID | item⟩], [⟨S↦B, ID | item⟩]
```

In *Either-Starts* in Listing 7, both B and S have the option to send the initiating message, since both *rfq* and *recommend* produce a binding for the key parameter ID.

Listing 8: Buyer-Starts refinement of Either-Starts.

```
Buyer-Starts {
  roles B, S
  parameters out ID key, out item
  B ↦ S: rfq(out ID key, out item)
}
// complete path: [⟨B↦S, ID | item⟩]
```

Buyer-Starts in Listing 8 is a refinement of *Either-Starts*, because it selects only one of the two possible enactments.

3.5 Key Parameter Reduction

Demoting a key parameter, i.e., turning it into a non-key parameter, strengthens the original key constraints. Thus demoting a key parameter reduces flexibility, and is a valid refinement. (A valid protocol must have at least one key.)

Consider Listing 9, which has two key parameters.

Listing 9: Multiple lookups via two key parameters.

```
Lookup-Prices {
  roles S, C
  parameters in ID key, out query key, out price
  S -> C: lookup[in ID key, out query key]
  C -> S: result[in ID key, in query key, out price]
}
// possible paths
[..., ⟨S↦C, [ID, query] | query⟩, ⟨C↦S, [ID, query] | price⟩]
[..., ⟨S↦C, [ID, query] | query⟩, ⟨S↦C, [ID, query] | query⟩,
 ⟨C↦S, [ID, query] | price⟩, ⟨C↦S, [ID, query] | price⟩]
...
```

The paths are shown with an elided prefix, because there are ⌜in⌝ parameters that need to be bound by other messages to enable *lookup*. Because *Lookup-Prices* has a composite key, each parameter need only have a unique binding for each pair of bindings of ID and query. Thus, we explicitly include some of the normally elided key bindings to show that the second path contains multiple bindings for query, and can have multiple corresponding prices.

Listing 10: Protocol with reduced keys.

```
Single-Lookup {
  roles S, C
  parameters in ID key, in item, out price
  S -> C: lookup[in ID key, in item]
  C -> S: result[in ID key, in item, out price]
}
// only one complete path
```

```
[ ... , ⟨S↦C, ID | item⟩ , ⟨C↦S, ID | price⟩ ]
```

Single-Lookup in Listing 10 is a refinement of *Lookup-Prices*, because the one path of *Single-Lookup* corresponds to a path of *Lookup-Prices*, and its keys are a subset of the keys of *Lookup-Prices*.

3.6 Concurrency Elimination

Listing 11 specifies *Flexible-Purchase*, in which the order that the buyer and seller respectively pay and deliver is flexible.

Listing 11: Purchase with concurrent payment and delivery.

```
Flexible-Purchase {
 roles B, S
 parameters in ID key, in item, in price, out payment, out shipped

 B ↦ S: pay[in ID key, in item, in price, out payment]
 S ↦ B: ship[in ID key, in item, in price, out shipped]
}
```

Flexible-Purchase does not specify a dependency relationship between *pay* and *ship*, so they are not ordered and any enactment may complete in one of three ways.

(1) B sends *pay* after receiving *ship*, yielding the path:

```
[ ... , ⟨S↦B, ID | shipped⟩0 , ⟨B↦S, ID | payment⟩0 ]
```

(2) S sends *ship* after receiving *pay*. This enactment has the path:

```
[ ... , ⟨B↦S, ID | payment⟩0 , ⟨B↦S, ID | shipped⟩0 ]
```

(3) B and S concurrently send *pay* and *ship*, respectively, yielding either of the following two paths:

```
[ ... , ⟨B↦S, ID | payment⟩1 , ⟨S↦B, ID | shipped⟩0 ]
[ ... , ⟨S↦B, ID | shipped⟩1 , ⟨B↦S, ID | payment⟩0 ]
```

Having a nonzero offset captures asynchrony, by allowing another message to be sent before the previous message is received. In the first path, for example, *pay* is sent before *ship* but has a offset of 1, so they are both received at the same point in the path. That an enactment involving concurrency can be represented in two ways shows its flexibility, and distinguishes a protocol that enables concurrency from one that enables only a single sequence.

The *Pay-First* protocol, given in Listing 12 entertains exactly one complete enactment, where S sends *ship* after receiving *payment*.

Listing 12: Eliminating concurrency leads to refinement.

```
Pay-First {
  roles B, S
  parameters in ID key, in item, in price, out payment, out shipped

  B ↦ S: pay[in ID key, in item, in price, out payment]
  S ↦ B: ship[in ID key, in item, in payment, out shipped]
}
```

The only complete path for this protocol is:

```
[ ... , ⟨S↦B, ID | payment⟩ , ⟨B↦S, ID | shipped⟩ ]
```

This path is identical to the second path of *Flexible-Purchase* in Listing 11. Further, all other paths of *Pay-First* are prefixes of this complete path, which means that they are paths of *Flexible-Purchase* as well. Therefore, we claim that *Pay-First* refines *Flexible-Purchase*.

3.7 Adding an Intermediary

Adding roles to a protocol yields a valid refinement if care is taken to ensure that all of the original roles observe the same information in the same sequences. Parameters observed together in the original protocol must be observed together in a refinement.

Listing 13 gives *Direct-Purchase*, in which the buyer and seller interact directly, while Listing 14 gives *Indirect-Purchase*, where the purchase is made through an intermediary.

Listing 13: Direct purchase protocol.

```
Direct-Purchase {
  roles B, S // Buyer, Seller
  parameters out ID key, out item, out deliver
  private payment
  S ↦ B: greet[out ID key]
  B ↦ S: order[in ID key, out item]
  S ↦ B: ship[in ID key, in item, out deliver]
}
// one complete path
[⟨S↦B, ID |⟩, ⟨B↦S, ID | item⟩, ⟨S↦B, ID | deliver⟩]
```

Listing 14: Indirect purchase protocol.

```
Indirect-Purchase {
  roles B, S, I // Buyer, Seller, Intermediary
  parameters out ID key, out item, out deliver
  private payment
  S ↦ B: greet[out ID key]
  B ↦ I: order[in ID key, out item]
  I ↦ S: confirm[in ID key, in item]
  S ↦ B: ship[in ID key, in item, out deliver]
}
// one complete path
[⟨S↦B, ID |⟩, ⟨B↦I, ID | item⟩, ⟨I↦S, ID |⟩, ⟨S↦B, ID | deliver⟩]
```

In *Indirect-Purchase*, B does not send *order* directly to S, but through an intermediary I instead. Communicating through an intermediary does not affect the observations of the original roles, because it merely decouples the information emission and reception in the same way as asynchrony.

3.8 Private Parameters

A refinement may introduce new private parameters, though care must be taken to avoid private safety conflicts that are not caught by refinement. Listing 15 shows how using private parameters enables splitting information across multiple messages.

Listing 15: Purchase with Escrow.

```
Escrowed-Purchase {
  roles B, S, I // Buyer, Seller, Intermediary
  parameters out ID key, out item, out deliver
  private payment, transfer // new private parameters

  S ↦ B: greet[out ID key]
  B ↦ S: order[in ID key, out item]
  B ↦ I: pay[in ID key, out payment]
  I ↦ S: transfer[in ID key, in payment, out transfer]
  S ↦ B: ship[in ID key, in item, in payment, out deliver]
}
// one complete path, projected to public parameters only
[⟨S↦B, ID |⟩, ⟨B↦S, ID | item⟩, ⟨B↦I, ID |⟩,
 ⟨I↦S, ID |⟩, ⟨S↦B, ID | deliver⟩]
```

Listing 15 shows an extended refinement of *Direct-Purchase*, in which the buyer sends the order directly to the seller, but sends additional payment information through an intermediary that acts as an escrow. Since the payment parameter is private, this change does not violate refinement. And, *transfer* does not have any ⌜out⌝ parameters, so it does not introduce any new bindings, but is necessary to communicate the binding of payment to S. The fact that *Escrowed-Purchase* is a refinement of *Direct-Purchase* can be seen from the paths, because the B and S roles observe the same information in the same sequence. The additional messages are ignored.

4 FORMALIZATION

We use $\downarrow_x$ to project a list to those of its elements that belong to x.

The basic element of a protocol, and thus of a path, is a message.

Definition 1: A *message schema* $\ulcorner s \mapsto r : m\,\vec{p}(\vec{k})\urcorner$ associates sender s, recipient r, message name m, parameter list $\vec{p}$, and keys $\vec{k}$.

When message schemas are enacted, they produce message instances that contain additional details unique to the enactment, namely parameter values and a reception offset.

Definition 2: A *message instance* associates a schema with a list of values $\vec{v}$ and an integer reception offset o. An instance has the form $m[s, r, \vec{p}(\vec{k}, \vec{v}), o]$. Below, s_i, p_i, etc. refer to the fields of a message instance m_i. And, $\mathsf{ins}(m)$, $\mathsf{outs}(m)$, and $\mathsf{nils}(m)$ refer to parameters of m that are adorned $\ulcorner\mathsf{in}\urcorner$, $\ulcorner\mathsf{out}\urcorner$, and $\ulcorner\mathsf{nil}\urcorner$, respectively.

Definition 3: The *offset* of a message instance (o_i for message m_i) is the number of events that occur before the message is received. Thus, a message with offset 0 is effectively received immediately, before any other events occur, whereas a message of offset 1 is received after the next event, and so on.

A *protocol* is either a message schema or a bag of protocols (references). Like messages, protocols also have roles, parameters, and keys. Parameters that are not public are renamed uniquely, to provide encapsulation when used in composition.

Definition 4: A *protocol* P is a tuple $\langle n, \vec{x}, \vec{y}, \vec{p}, \vec{k}, \vec{q}, F\rangle$, where n is a name; $\vec{x}$ and $\vec{y}$ are the public and private roles, respectively; $\vec{p}$, $\vec{k}$, and $\vec{q}$ are the public, key, and private parameters, respectively; and F is a finite set of f references to other protocols, $\{F_1, \ldots, F_f\}$. $(\forall i : 1 \le i \le f \Rightarrow F_i = \langle n_i, \vec{x_i}, \vec{p_i}, \vec{k_i}\rangle$, 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 $\langle n_i, \vec{x_i}, \vec{p_i}, \vec{k_i}\rangle$ projects protocol P_i to its public components (with roles and parameters renamed for uniqueness).

Below, C_P is a *composition* (i.e., a composite protocol) that references P—that is, P is a *constituent* of C_P. $C_{P/Q}$ is a composition where the reference to P is replaced by a reference to Q.

Definition 5: If C_P has references $F = F_1, \ldots, F_f$, then Q *substitutes* P in $C_{P/Q}$ if and only if $C_{P/Q}$ has references $F' = F'_1, \ldots, F'_f$ such that for every $i \le f$, $F_i = P$ and $F'_i = Q$ or $F_i = F'_i$.

A *universe of discourse* (UoD) consists of a set of roles and a set of messages they can enact. Generally a UoD is taken from a protocol specification, but multiple protocols can be composed together, or enacted in a context that includes other protocols.

Definition 6: A *UoD* is a pair $\mathcal{U} = \langle \mathcal{R}, \mathcal{M}\rangle$, 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 $\mathcal{R}$.

The universe of discourse for the roles and messages of protocol P is denoted $\mathcal{U}_P$, and the union of UoDs $\mathcal{U}_1 \cup \mathcal{U}_2$ is $\langle \mathcal{R}_1 \cup \mathcal{R}_2, \mathcal{M}_1 \cup \mathcal{M}_2\rangle$.

A *path* is a sequence of events, namely message instances, corresponding to an enactment. Each path induces a history vector, which can be derived by appending each message instance to the sender's history in the order the instances occur, and then to the recipient's after the corresponding offset. Conversely, multiple paths may induce a history vector, since role histories are independent.

Definition 7: A *path* is a list of instances, $\tau = (m_1, m_2, \ldots, m_n)$, with length $\|\tau\|$.

An *extension* of a path appends one or more messages to it.

Definition 8: (*extension*) If $\tau = (m_1, m_2, \ldots m_n)$, then
$\tau \circ m' = (m_1, m_2, \ldots m_n, m')$
and $\tau \circ [m_{n+1}, m_{n+2}, \ldots, m_{n+j}] = (m_1, m_2, \ldots m_{n+j})$

A message instance m_i is *received* on a path if its offset is less than or equal to the number of events after m_i, else it is in transit. That is, the position of m_i describes when it was sent, and the offset describes how many events occur before it is received.

Definition 9: $\mathsf{received}(\tau) = \{m_i \in \mathsf{P} \mid i + o_i \le \|\tau\|\}$

Definition 10 captures the information that a role observes after a sequence of messages has been sent. Each parameter in the message is known by role R if the parameter values match the bindings in the enactment, and either R is the sender, or R is the recipient and the message has been received.

Definition 10: $\mathsf{known}(\tau, \vec{k}, \vec{v}, R) = \{p \mid \exists m_i \in \tau : p \in \vec{p_i} \Rightarrow \vec{v_i}\downarrow_{\vec{k}} = \vec{v}\downarrow_{\vec{k}}$ and $(s_i = R$ or $r_i = R$ and $m_i \in \mathsf{received}(\tau))\}$

A message instance is *viable* on a path if the sender knows the $\ulcorner\mathsf{in}\urcorner$ parameters, but not the $\ulcorner\mathsf{out}\urcorner$ or $\ulcorner\mathsf{nil}\urcorner$ parameters.

Definition 11: Message instance $m[s, r, \vec{p}(\vec{k}), \vec{v}]$ is *viable* on path τ if and only if $p \in \mathsf{ins}(m)$ implies $p \in \mathsf{known}(\tau, \vec{k}, \vec{v}, s)$, and $p \in \mathsf{known}(\tau, \vec{k}, \vec{v}, s)$ implies $p \notin \mathsf{outs}(m)$ and $p \notin \mathsf{nils}(m)$

A path is *valid* if and only if it is either empty or the extension of a valid path by a viable message. A path is *extensible* in a universe of discourse if there are viable messages that can be appended to it. The set of all valid paths in $\mathcal{U}$ is denoted $\mathsf{paths}(\mathcal{U})$. The set of paths produced by a protocol must cover all possible orderings of independent events. That is, if a and b are independent events, then both paths $(\ldots, a, \ldots, b, \ldots)$ and $(\ldots, b, \ldots, a, \ldots)$ are possible.

Definition 12: $\tau \in \mathsf{paths}(\mathcal{U})$ if and only if $\tau = \emptyset$, or there is path $\tau' \in \mathsf{paths}(\mathcal{U})$, and message $m \in \mathsf{viable}(\tau')$ such that $\tau = \tau' \circ m$.

The *branches* at path τ in universe of discourse $\mathcal{U}$ is the set of all paths in $\mathcal{U}$ with prefix τ.

Definition 13: $\mathit{branches}(\mathcal{U}, \tau) = \{\tau' \mid \tau \sqsubseteq \tau'$ and $\tau' \in \mathsf{paths}(\mathcal{U})\}$, where $\tau \sqsubseteq \tau'$ denotes that τ is a prefix of τ'.

The *sources* of a parameter in a path are the roles that send messages binding that parameter in the path. Safe protocols have at most one source for each parameter in any path.

Definition 14: $\mathit{sources}(\tau, p) = \{s_i \mid p \in \mathsf{outs}(m_i \in \tau)\}$

The *keys* of a path are the combinations of key parameters that appear in message instances on the path.

Definition 15: $\mathit{keys}(\tau) = \{\vec{k_i} \mid m_i \in \tau\}$, where k_i is the list of key parameters of message m_i.

Subsumption captures the idea that one path induces the same knowledge as another, such that the same combinations of relevant parameters must be bound in the same order and by the same roles.

Definition 16: Path τ' *subsumes* path τ for parameter list $\vec{p}$, denoted $\tau' \triangleright_{\vec{p}} \tau$, if and only if

(1) $\forall p \in \vec{p} : \mathsf{sources}(\tau', \mathrm{p}) = \mathsf{sources}(\tau, \mathrm{p})$
(2) $\forall R \in \mathcal{R}, \forall \vec{k} \in \mathsf{keys}(\tau), \forall \vec{v}$:
$\mathsf{known}(\tau', \vec{k}, \vec{v}, R) \cap \vec{p} = \mathsf{known}(\tau, \vec{k}, \vec{v}, R) \cap \vec{p}$

(3) $\tau_2 \sqsubseteq \tau \Rightarrow \exists \tau_2'$ such that $\tau_2' \sqsubseteq \tau'$ and $\tau_2' \triangleright_{\vec{p}} \tau_2$
where $\mathcal{R}$ is the set of roles that appear in τ.

Definition 17 captures refinement in terms of paths. Informally, Q refines P if and only if every path in Q subsumes some path in P, and every path in Q that subsumes some extensible path in P is extensible or has unreceived messages. That is, if P has branches, then Q must also have branches or unreceived messages.

Definition 17: Protocol Q *refines* protocol P for parameters $\vec{p}$, denoted $Q \preceq_{\vec{p}} P$, if and only if for every $\mathcal{U}$ and path $\tau_Q \in \mathsf{paths}(\mathcal{U} \cup \mathcal{U}_Q)$, there is a path $\tau_P \in \mathsf{paths}(\mathcal{U} \cup \mathcal{U}_P)$ such that (1) $\tau_Q \triangleright_{\vec{p}} \tau_P$, and (2) $\mathsf{branches}(\mathcal{U} \cup \mathcal{U}_P, \tau_P) \neq \emptyset$ implies ($\mathsf{branches}(\mathcal{U} \cup \mathcal{U}_Q, \tau_Q) \neq \emptyset$ or $\|\mathsf{received}(\tau_Q)\| < \|\tau_Q\|$).

$Q \preceq P$ denotes the common case of refinement with respect to the public parameters of P.

Safety is the correctness property that no parameter takes on more than one value for a given set of keys in any path.

Definition 18: Protocol P is *safe* if every path in $\mathcal{U}_P$ is safe. A path τ is safe if and only if, for every $m_i \in \tau$, there is no $m_{j \neq i}$ such that $\vec{k}_i = \vec{k}_j \wedge \vec{v}_i \downarrow_{\vec{k}_i} \neq \vec{v}_j \downarrow_{\vec{k}_j}$

Liveness is a correctness property requiring that every enactment of a protocol be extensible by a finite sequence of emissions to an enactment in which all public ⌜out⌝ parameters are bound.

Definition 19: A protocol P with public ⌜out⌝ parameters $\vec{p}$ is *live* in $\mathcal{U}_P$ if and only if every path in $\mathcal{U}_P$ is a prefix of some path in $\mathcal{U}_P$ such that $\vec{p} \subseteq \bigcup_{\vec{k},\vec{v},R} \mathsf{known}(\tau, \vec{k}, \vec{v}, R)$

5 THEORETICAL RESULTS

We say that the protocols are safe or live *up to* $\vec{p}$, meaning that there are no violations involving those parameters. In simpler terms, introducing unused public parameters can artificially violate liveness and introducing conflicts on private parameters can artificially violate safety. Such conditions are readily checked.

Theorem 5.1 establishes that refinement preserves liveness. Hence, if a protocol is live, it is sufficient to check that a substitute protocol is a refinement—and we verify its liveness for free.

Theorem 5.1. *If P is live and $Q \preceq_{\vec{p}} P$, then Q is live up to $\vec{p}$.*

Proof Sketch. Since P is live, for every path τ in $\mathcal{U}_P$, one of the following holds: (1) $\mathsf{branches}(\mathcal{U}_P, \tau) \neq \emptyset$; (2) $\|\mathsf{received}(\tau)\| < \|\tau\|$; or (3) $\vec{p} \subseteq \bigcup_{\vec{k},\vec{v},R} \mathsf{known}(\tau, \vec{k}, \vec{v}, R)$. Since $Q \preceq P$, for every path $\tau_Q \in \mathcal{U}_Q$ there is a path $\tau_P \in \mathsf{paths}(\mathcal{U}_P)$: $\tau_Q \triangleright_{\vec{p}} \tau_P$, and $\mathsf{branches}(\mathcal{U}_P, \tau_P) \neq \emptyset \Rightarrow \mathsf{branches}(\mathcal{U}_Q, \tau_Q) \neq \emptyset$ or $\|\mathsf{received}(\tau_Q)\| < \|\tau_Q\|$. From subsumption, $\bigcup_{\vec{k},\vec{v},R} \mathsf{known}(\tau_Q, \vec{k}, \vec{v}, R) = \bigcup_{\vec{k},\vec{v},R} \mathsf{known}(\tau_P, \vec{k}, \vec{v}, R)$. Thus we have one of: (1) $\mathsf{branches}(\mathcal{U}_Q, \tau) \neq \emptyset$; (2) $\|\mathsf{received}(\tau_Q)\| < \|\tau_Q\|$; or (3) $\vec{p} \subseteq \bigcup_{\vec{k},\vec{v},R} \mathsf{known}(\tau_Q, \vec{k}, \vec{v}, R)$. So Q is live up to $\vec{p}$. □

Theorem 5.2 establishes that refinement preserves safety.

Theorem 5.2. *If P is safe and $Q \preceq_{\vec{p}} P$ then Q is safe up to $\vec{p}$.*

Proof Sketch. Suppose P is safe, $Q \preceq_{\vec{p}} P$ and $\vec{p}_i \subseteq \vec{p}$ for all messages in Q. Because $Q \preceq_{\vec{p}} P$, the sources of any p in $\vec{p}$ on any path in Q are the same as the sources of p on some path in P. Thus, for every parameter in $\vec{p}$, there is at most one source for each parameter of Q in any path in Q, and Q is safe up to $\vec{p}$. □

Theorem 5.3 establishes that refinement applies to substitution of a constituent in a composition. That is, if C_P is a composition in which P is a constituent, and Q is a refinement of P, then a composition $C_{P/Q}$ with Q substituted for P is a refinement of C_P.

Theorem 5.3. *If $Q \preceq P$, then $C_{P/Q} \preceq C_P$.*

Proof Sketch. Suppose $Q \preceq P$, with P having public parameters $\vec{p}$. Let $\mathcal{U}_C$ be the universe of discourse of C_P without the messages in P, such that $\mathcal{U}_{C_P} = \mathcal{U}_C \cup \mathcal{U}_P$, and $\mathcal{U}_{C_{P/Q}} = \mathcal{U}_C \cup \mathcal{U}_Q$.

(0) Suppose it is not true that $C_{P/Q} \preceq C_P$. By the definition of refinement, (1) there must be some path τ_Q in $paths(\mathcal{U}_{C_{P/Q}})$ that does not subsume any path τ_P in $\mathcal{U}_{C_P}$, or (2) τ_P has branches but τ_Q does not and τ_Q does not have any unreceived messages.

(1) Suppose there is some path $\tau_Q \in \mathsf{paths}(\mathcal{U}_{C_{P/Q}})$ that subsumes a path $\tau_P \in \mathsf{paths}(\mathcal{U}_{C_P})$, but for some message m in $\mathcal{U}_{C_{P/Q}}$, $\tau_Q \circ m$ does not subsume any path in $\mathsf{paths}(\mathcal{U}_{C_P})$. Since τ_Q subsumes τ_P, they must induce the same knowledge at each role, and thus would enable the same messages. Thus for $\tau_Q \circ m$ to not subsume any path, m must not exist in $\mathcal{U}_{C_P}$, which means it must be in Q. But by the definition of refinement, for any $\mathcal{U}$, $\forall \tau_Q \in \mathsf{paths}(\mathcal{U} \cup \mathcal{U}_Q), \exists \tau_P \in \mathsf{paths}(\mathcal{U} \cup \mathcal{U}_P) : \tau_Q \triangleright_{\vec{p}} \tau_P$, contradicting (1).

(2) Suppose a path $\tau_Q \in \mathsf{paths}(\mathcal{U}_{C_{P/Q}})$ does not have branches or any unreceived messages, but subsumes a path $\tau_P \in \mathsf{paths}(\mathcal{U}_{C_P})$ that has branches. Since τ_Q subsumes τ_P, they must induce the same knowledge at each role, and thus would enable the same messages; thus any messages enabled on τ_P but not τ_Q must be in P, since all the messages in $\mathcal{U}_C$ are in both $\mathsf{paths}(\mathcal{U}_{C_{P/Q}})$ and $\mathsf{paths}(\mathcal{U}_{C_P})$. But $Q \preceq P$ and by the definition of refinement $\forall \mathcal{U} \mathsf{branches}(\mathcal{U} \cup \mathcal{U}_P, \tau_P) \neq \emptyset \Rightarrow (\mathsf{branches}(\mathcal{U} \cup \mathcal{U}_Q, \tau_Q) \neq \emptyset \vee \|\mathsf{received}(\tau_Q)\| < \|\tau_Q\|)$. Thus τ_Q has branches in contradiction of supposition (2).

Thus, (1) and (2) are false, contradicting (0), so $C_{P/Q} \preceq C_P$. □

Theorems 5.4 and 5.5 establish that the safety and liveness of a composition are preserved when substituting a constituent by a refinement. This result yields assurance of the liveness and safety of a composition without full reverification; only the replacement protocol needs to be checked for safety and refinement.

Theorem 5.4. *If C_P is live and $Q \preceq P$, then $C_{P/Q}$ is live.*

Proof Sketch. Suppose C_P is live and $Q \preceq P$. Assume that Q does not pathologically have public parameters not in P that block messages in C. By Theorem 5.3, $C_{P/Q} \preceq C_P$. Since C_P is live, every path in $paths(\mathcal{U}_{C_P})$ either has branches or binds every ⌜out⌝ parameter of C_P. By refinement, every path in $\mathcal{U}_{C_{P/Q}}$ subsumes some path in $\mathcal{U}_{C_P}$. Thus, every path in $\mathcal{U}_{C_{P/Q}}$ either has branches, or binds every ⌜out⌝ parameter of C_P. Since C_P and $C_{P/Q}$ have the same public parameters, this means $C_{P/Q}$ is live. □

Theorem 5.5. *If C_P is safe and $Q \preceq_{\vec{p}} P$, then $C_{P/Q}$ is safe up to $\vec{p}$.*

Proof Sketch. Let C_P be safe, and $Q \preceq_{\vec{p}} P$. By Theorem 5.3, $C_{P/Q} \preceq C_P$ and Q enables only what P did. C_P is safe, so any safety violations are internal to Q. By Theorem 5.2 Q is safe up to $\vec{p}$, so $C_{P/Q}$ is safe up to $\vec{p}$. □

6 EVALUATION

Algorithm 1 verifies refinement based on Definition 17. It enumerates all possible paths in both P and Q, and checks that each path q in Q subsumes some path p in P such that if p has branches then q does as well, otherwise it is not a refinement.

Algorithm 1: Refinement algorithm

```
foreach τ_Q in paths(U_Q) do
    match ← ∅ ;
    for τ_P in paths(U_P) do
        match ← τ_P ;
        if There are no branches of τ_P, or τ_Q has branches or
          unreceived messages then
            break;          // found match; stop looking
    if no match then
        return False;           // failed Definition 17.1
    if match has branches but τ_Q does not have branches or
      unreceived messages then
        return False;           // failed Definition 17.2
    return True;
```

We have implemented the refinement checking algorithm in Python 3.6.5 enhancing our toolchain for BSPL. Testing was done via pytest, executing the relevant functions in a for loop, skipping the first iteration so that any loading or caching occurs before our measurements. All tests were performed on a laptop running Gentoo Linux, kernel version 4.19.27, with an Intel i7-6600U cpu, 16GB of DDR3 memory, and 1TB SSD. All timings are in milliseconds; and the minimum, mean, maximum, and sample standard deviation (σ) are computed over 10 runs. The code and results are available at https://gitlab.com/masr/protocheck.

Table 1 shows the time required to verify three of the example refinements, indicating that the approach is tractable.

Protocol	Min	Mean	Max	σ
Buyer-Starts $\preceq$ Either-Starts	1	1	2	0.6
Single-Lookup $\preceq$ Lookup-Prices	103	138	169	21.3
Pay-First $\preceq$ Flexible-Purchase	53	70	87	9.6

Table 1: Time to verify refinement for example protocols.

For comparison, Table 2 lists the time it takes to verify liveness and safety for the *Commerce* protocol resulting from the substitution of a single protocol by its refinement (e.g., *Either-Starts* with *Buyer-Starts*). The last row gives the time for verifying liveness and safety after all three constituents are substituted; this protocol is the simplest since all three constituents have been simplified, and so takes the least time to verify. As these measurements demonstrate, the time to check refinement is less than the time to check liveness for the *Commerce* protocol post substitution, thus establishing the practical benefit of checking refinement.

The time to check safety post substitution is also higher than the time to check refinement except when *Buyer-Starts* is substituted for *Either-Starts*, as in the first and last rows, where the safety checking tool statically detects that there are no messages with the same ⌜out⌝ parameter but different senders, ruling out the violation of safety in time linear in the size of the protocol specification.

Substitution	Property	Min	Mean	Max	σ
sub. *Buyer-Starts*	Liveness	14 419	14 745	15 099	222
	Safety	0	0	0	0
sub. *Single-Lookup*	Liveness	18 483	18 568	18 714	68
	Safety	18 296	18 401	18 541	74
sub. *Pay-First*	Liveness	18 408	18 512	18 601	78
	Safety	18 246	18 347	18 517	78
substitute all	Liveness	11 624	11 868	12 279	197
	Safety	0	0	0	0.03

Table 2: Times to verify properties of *Commerce* (Listing 5) with select constituent replaced by its refinement, as given in Table 1.

We also tested preexisting protocol specifications, checking that the NetBill protocol [9] is a refinement of the Bliss variant [32], and that a variant of HL7's CreateLaboratoryOrder flow [19] is a refinement of the original. Table 3 shows that checking refinement is faster than re-verifying liveness in both cases.

Protocol	Property	Min	Mean	Max	σ
NetBill	Refinement	1 256	1 294	1 323	21
	Liveness	13 615	13 926	14 141	174
CreateLabOrder	Refinement	5 879	6 003	6 162	83
	Liveness	45 257	46 469	50 878	1 648

Table 3: Time to verify liveness and refinement for the *NetBill* and *CreateLabOrder* protocols.

7 RELATED WORK

Refinement, a preorder relation on protocols, is somewhat related to *simulation* [28], which is a preorder relation on processes. A process P simulates a process Q if P can match all of Q's moves; that is, if a transition is available at a state in Q, the same transition is available at a matching state in P. A simulating process can do everything the simulated process can do, and therefore substitute for (impersonate) it. Refinement is concerned with substitution in compositions where additional states may introduce conflicts, such as two roles binding the same parameter, and so is more like specialization than extension. Each path in a refinement must have the same observations in the same order as the paths it subsumes, but may have fewer branches.

Process testing [10] defines notions of refinement for communicating processes, in which one process (e.g., a server) refines another if it passes the same tests (e..g., supports the same clients). Bernardi and Hennessy [6] extend testing to mutual testing such as peer-to-peer relationships. However, testing applies to processes, not interactions. Even more recent work on multiparty testing [11] addresses only multiple clients interacting with the same server, relaxing the preorder to accommodate flexible ordering in the server's responses to uncoordinated clients.

The Liskov substitution principle [21] states that subtypes of a type satisfy properties of the type. Refinement is a kind of subtyping and preserves safety and liveness. However, Liskov's behavioral substitution applies not to protocols, but to roles, which describe behavior. Gay [14] discusses channel-oriented and process-oriented subtyping for session types [17]. Session types work that tackles multiparty asynchronous protocols [18] requires FIFO channels.

We distinguish conformance from *compliance*, the problem of determining at runtime whether an agent satisfies a role [16, 29, 33]. In this terminology, Ancona et al. [2] deals with compliance rather than conformance.

Mazouzi et al. [23] describe a design methodology for communication protocols involving translation of abstract protocol specification diagrams in AUML to recursive colored Petri nets (RCPNs) for formal verified at various levels of abstraction. However, although RCPNs support concurrency, they are synchronous across the various roles, and do not account for the possible risks to information integrity introduced by asynchrony.

Refinement relates to prior work on atomicity of protocols [26], which requires that if any constituent is partially enacted it must be possible to complete, in that both address correctness of protocols used together in a composition. Although atomicity entails liveness for the overall composition, it is primarily concerned with detecting conflicts that would prevent a constituent from completing and leave it "dangling." Atomicity can be verified only by using the composition as a whole; it cannot be determined from the specification of a constituent protocol in isolation. By contrast, refinement verifies that particular substitutes for a constituent protocol do not affect the liveness of the composition, without needing to consider the other constituents. Thus the two concepts complement each other, with atomicity proving that a constituent will be live in a composition, and refinement verifying that a substitute protocol will behave the same way.

Information protocols enable layering meaning protocols, e.g., based on commitment and norms [3, 20, 25, 34, 37, 39]. Refinement has been studied for commitment protocols [7, 15, 22]. Our intuitions regarding refinement—that Q refines P if every run of Q embeds some run of P—agree with Gerard and Singh's [15]. We additionally require that every enactment in Q be able to progress if the corresponding enactment in P can progress. The present framework accommodates asynchronous communications, which the above works do not address.

In ongoing work [8, p. 18], we show that *Flexible-Payment* (Listing 11), which supports payment and shipment in any order or concurrently, cannot be expressed in protocol languages based on trace expressions or session types. These languages can express an analog of *Pay-First* (Listing 12)—a refinement of *Flexible-Payment*.

8 CONCLUSIONS AND FUTURE WORK

We have motivated and formalized a notion of refinement of information protocols. We established important results about this notion of refinement. Specifically, we showed that (1) a refinement of a safe and live protocol is, respectively, safe and live; (2) safety and liveness of a composition are preserved when substituting a constituent protocol with its refinement; and (3) a composition with a refinement substituted for a constituent protocol is itself a refinement of the original composition.

We described an algorithm for checking refinement and provide an empirical demonstration of its performance that establishes that checking refinement once is worthwhile compared to checking the liveness and safety of compositions post substitution. Future work will consider optimizations—such as taking advantage of path symmetry, memoization, and lazy generation—that should make the refinement-checking algorithm even more efficient.

Note that although refinement preserves the safety and liveness of a protocol, sometimes a refined protocol may be safe or live when the protocol it refines is not. Indeed, that would be a common occurrence during engineering. Engineers may specify a protocol that meets their application requirements and upon determining (through a checker tool) that the protocol is not safe or live, may proceed to refine it to obtain one that is safe and live—and therefore appropriate for developing agents to play roles in it.

We could support richer kinds of refinement if we exploited semantic mappings between the parameters of protocols. For example, suppose a purchase protocol supports separate delivery addresses for each item. It should be reasonable to refine this protocol to one in which the same address is used for all of the items. Such a protocol should be a refinement, since it reduces the number of choices and does not produce any less information: the customer could provide the same address multiple times in the unrefined protocol. However, such a refinement goes beyond this paper, as this paper does not support mappings between parameters.

Our notion of protocol refinement shares with works on conformance the intuition that refinement means a reduction of emission choices relative to the original protocol. A future direction is to investigate the relationship between refinement and conformance. We would expect that if protocol Q refines protocol P and if a role ρ features in both, Q projected to ρ would conform with P projected to ρ.

An interesting line of work would be to consider the refinement of commitment specifications along with the refinement of information protocols. For example, introducing an intermediary for purposes of escrow (Listing 15) is a refinement of the direct purchase protocol (Listing 6) in the current work; however, this does not consider that the relevant commitment specifications may have to be refined as well, e.g., to set up commitments between escrow and the other agents. It would be necessary to ensure that protocols support enactments that enable agents to comply with their commitments, along the lines of the Clouseau approach [34].

9 ACKNOWLEDGEMENTS

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 National Science Foundation under grant IIS-1908374.

REFERENCES

[1] 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.

[2] Davide Ancona, Angelo Ferrando, and Viviana Mascardi. 2018. Agents Interoperability via Conformance Modulo Mapping. In *Proceedings of the 19th Workshop on From Objects to Agents (WOA) (CEUR)*. CEUR-WS.org, Palermo, Italy, 109–115.

[3] Matteo Baldoni, Cristina Baroglio, and Federico Capuzzimati. 2014. A Commitment-Based Infrastructure for Programming Socio-Technical Systems. *ACM Transactions on Internet Technologies* 14, 4 (Dec. 2014), 23:1–23:23.

[4] Matteo Baldoni, Cristina Baroglio, Amit K. Chopra, Nirmit Desai, Viviana Patti, and Munindar P. Singh. 2009. Choice, Interoperability, and Conformance in Interaction Protocols and Service Choreographies. In *Proceedings of the 8th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Budapest, 843–850. https://doi.org/10.5555/1558109.1558129

[5] Matteo Baldoni, Cristina Baroglio, Alberto Martelli, and Viviana Patti. 2006. A Priori Conformance Verification for Guaranteeing Interoperability in Open Environments. In *Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC) (Lecture Notes in Computer Science)*, Vol. 4294. Springer, Chicago, 339–351.

[6] Giovanni Bernardi and Matthew Hennessy. 2015. Mutually Testing Processes. *Logical Methods in Computer Science* 11, 2 (2015). https://doi.org/10.2168/LMCS-11(2:1)2015

[7] Amit K. Chopra and Munindar P. Singh. 2006. Contextualizing Commitment Protocols. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*. ACM Press, Hakodate, Japan, 1345–1352. https://doi.org/10.1145/1160633.1160884

[8] Amit K. Chopra, Samuel H. Christie V, and Munindar P. Singh. 2019. An Evaluation of Communication Protocol Languages for Engineering Multiagent Systems. (Oct. 2019). arXiv:1901.08441v2 [cs.SE].

[9] Benjamin Cox, J. D. Tygar, and Marvin Sirbu. 1995. NetBill Security and Transaction Protocol. In *Proceedings of the 1st USENIX Workshop on Electronic Commerce*. USENIX, New York, 77–88.

[10] Rocco De Nicola and Matthew Hennessy. 1984. Testing Equivalences for Processes. *Theoretical Computer Science* 34 (1984), 83–133. https://doi.org/10.1016/0304-3975(84)90113-0

[11] Rocco De Nicola and Hernán C. Melgratti. 2015. Multiparty Testing Preorders. In *Trustworthy Global Computing - 10th International Symposium, TGC 2015, Madrid, Spain, August 31 - September 1, 2015 Revised Selected Papers (Lecture Notes in Computer Science)*, Pierre Ganty and Michele Loreti (Eds.), Vol. 9533. Springer, 16–31. https://doi.org/10.1007/978-3-319-28766-9_2

[12] Ulrich Endriss, Nicolas Maudet, Fariba Sadri, and Francesca Toni. 2003. Protocol Conformance for Logic-based Agents. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI, Acapulco, Mexico, 679–684.

[13] Angelo Ferrando, Michael Winikoff, Stephen Cranefield, Frank Dignum, and Viviana Mascardi. 2019. On the Enactability of Agent Interaction Protocols: Toward a Unified Approach. *CoRR* abs/1902.01131v4 (Feb. 2019), 1–13.

[14] Simon J. Gay. 2016. Subtyping Supports Safe Session Substitution. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.), Vol. 9600. Springer, 95–108. https://doi.org/10.1007/978-3-319-30936-1_5

[15] Scott N. Gerard and Munindar P. Singh. 2013. Formalizing and Verifying Protocol Refinements. *ACM Transactions on Intelligent Systems and Technology (TIST)* 42, 2 (March 2013), 21:1–21:27. https://doi.org/10.1145/2438653.2438656 Appendix pages 1–7.

[16] Laura Giordano, Alberto Martelli, and Camilla Schwind. 2007. Specifying and verifying interaction protocols in a temporal action logic. *Journal of Applied Logic* 5, 2 (2007), 214–234.

[17] 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, San Francisco, 273–284.

[18] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (March 2016), 9:1–9:67.

[19] ISO/HL7. 2013. Laboratory Order Conceptual Specification. (May 2013). https://wiki.hl7.org/index.php?title=Laboratory_Order_Conceptual_Specification.

[20] Warda El Kholy, Jamal Bentahar, Mohamed El-Menshawy, Hongyang Qu, and Rachida Dssouli. 2017. SMC4AC: A New Symbolic Model Checker for Intelligent Agent Communication. *Fundamenta Informaticae* 152, 3 (2017), 223–271. https://doi.org/10.3233/FI-2017-1519

[21] Barbara Liskov and Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems* 16, 6 (1994), 1811–1841. https://doi.org/10.1145/197320.197383

[22] Ashok U. Mallya and Munindar P. Singh. 2007. An Algebra for Commitment Protocols. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 14, 2 (April 2007), 143–163. https://doi.org/10.1007/s10458-006-7232-1

[23] Hamza Mazouzi, Amal El Fallah Seghrouchni, and Serge Haddad. 2002. Open Protocol Design for Complex Interactions in Multi-Agent Systems. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. ACM Press, Bologna, 517–526.

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

[25] Marco Montali, Diego Calvanese, and Giuseppe De Giacomo. 2014. Verification of data-aware commitment-based multiagent system. In *Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems*. IFAAMAS, Paris, 157–164.

[26] Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. 2018. Compositional Correctness in Multiagent Interactions. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Stockholm, 1159–1167. https://doi.org/10.5555/3237383.3237868

[27] 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.

[28] David Michael Ritchie Park. 1981. Concurrency and Automata on Infinite Sequences. In *Theoretical Computer Science, 5th GI-Conference*. Springer, Karlsruhe, Germany, 167–183. https://doi.org/10.1007/BFb0017309

[29] Munindar P. Singh. 1998. Agent Communication Languages: Rethinking the Principles. *IEEE Computer* 31, 12 (Dec. 1998), 40–47. https://doi.org/10.1109/2.735849

[30] 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. https://doi.org/10.5555/2031678.2031687

[31] 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. https://doi.org/10.5555/2343776.2343861

[32] 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. https://doi.org/10.1109/SCC.2014.39

[33] Munindar P. Singh and Amit K. Chopra. 2010. Correctness Properties for Multiagent Systems. In *Proceedings of the 6th AAMAS Workshop on Declarative Agent Languages and Technologies (DALT 2009) (Lecture Notes in Artificial Intelligence)*. Springer, Budapest, 192–207. https://doi.org/10.1007/978-3-642-11355-0_12

[34] Munindar P. Singh and Amit K. Chopra. 2020. Clouseau: Generating Communication Protocols from Commitments. In *Proceedings of the 34th Conference on Artificial Intelligence (AAAI)*. AAAI Press, New York, 1–9.

[35] Munindar P. Singh, Amit K. Chopra, Nirmit V. Desai, and Ashok U. Mallya. 2004. Protocols for Processes: Programming in the Large for Open Systems. In *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, Vancouver, 73–83. https://doi.org/10.1145/1052883.1052893 *SIGPLAN Notices*, volume 39, number 12, December 2004, pages 73–83.

[36] Benjamin Vitteau and Marc-Philippe Huget. 2004. Modularity in Interaction Protocols. In *Advances in Agent Communication (Lecture Notes in Computer Science)*, Frank Dignum (Ed.), Vol. 2922. Springer, Berlin, 291–309.

[37] Michael Winikoff, Wei Liu, and James Harland. 2005. Enhancing Commitment Machines. In *Proceedings of the 2nd International Workshop on Declarative Agent Languages and Technologies (DALT) (LNAI)*, Vol. 3476. Springer-Verlag, Berlin, 198–220.

[38] Michael Winikoff, Nitin Yadav, and Lin Padgham. 2018. A New Hierarchical Agent Protocol Notation. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 32, 1 (Jan. 2018), 59–133.

[39] 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. https://doi.org/10.1145/544862.544867