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:

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 AUMIL [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.
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: quote[ in ID key, in item, in price, out payment ]
S ← B: offer[ in ID key, in item, in price, out payment ]
S ← B: ship[ in ID key, in item, 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 `quote` 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 `payment`, binding payment, and S can send `deliver`, binding deliver. Here, `payment` and `deliver` 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 payment`, 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 `payment`, `payment` 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: quote[ out ID key, out item ]
B ← S: offer[ out ID key, out item ]
S ← B: quote[ in ID key, in 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 \( \langle r_1|h_1, r_2|h_2, \ldots, r_n|h_n \rangle \), where \( n \) is the number of roles in the protocol, \( r_i \) is the name of the \( i \)th 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 \( k \), parameter list \( \bar{p} \), and optional transmission offset \( o \), written as \( \langle s=r, k, \bar{p}, o \rangle \). 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 \( m_i \) to \( m_{i+k} \) have been made, and before any of \( R \)’s emissions of \( m_{i+k+1} \) 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.

```plaintext
[⟨B→s, ID | item⟩₀ ,
 ⟨s→b, ID | price⟩₀ ,
 ⟨b→s, ID | payment⟩₀ ,
 ⟨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 RFQ protocol given in Listing 3.

#### 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.

```plaintext
RFQ |
roles B, S
parameters out ID key, out item, out price
B ⟷ S: rfq(out ID key, out item)
S ⟷ B: query(out query key, out price)
```

The complete paths of Polymorphic-RFQ are:

- “⟨⟨B→S, ID | item⟩₀, ⟨S→B, ID | item, price⟩₀⟩”
- “⟨⟨B→S, ID | item⟩₀, ⟨S→B, ID | ID⟩₀⟩”

The only complete path of RFQ is:

“⟨⟨B→S, ID | item⟩₀, ⟨S→B, ID | price⟩₀⟩”

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.

```plaintext
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)
```

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.

```plaintext
Buyer-Starts |
roles B, S
parameters out ID key, out item
B ⟷ S: rfq(out ID key, out 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.

```plaintext
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)
```

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.

```plaintext
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"
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 | payment)\}, (B~S, ID | shipped)\}]

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

   [ . . . (B~S, ID | payment)\}, (B~S, ID | shipped)\}]

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

   [ . . . (B~S, ID | payment)\}, (B~S, ID | shipped)\}]

   [ . . . (S~B, ID | payment)\}, (S~B, ID | shipped)\}]

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 an 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 price, 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 $\downarrow_{a}$ 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 $s \mapsto r : m \{ \tilde{p}(k) \}^n$ associates sender $s$, recipient $r$, message name $m$, parameter list $\tilde{p}$, and keys $\tilde{k}$.

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 $(m[s, r, \tilde{p}(k, \vec{v}), o])$. Below, $s_i, p_i, \tau_i$ etc. refer to the fields of a message instance $m_i$. And, $\text{ins}(m)$, $\text{outs}(m)$, and $\text{nils}(m)$ refer to parameters of that are adored $\tau_{\text{in}}^n$, $\tau_{\text{out}}^n$, and $\tau_{\text{nil}}^n$, 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 $(n, \tilde{x}, \tilde{y}, \tilde{\tilde{p}}, \tilde{\tilde{k}}, \tilde{\tilde{q}})$, where $n$ is a name; $\tilde{x}$ and $\tilde{y}$ are the public and private roles, respectively; $\tilde{\tilde{p}}$, $\tilde{\tilde{k}}$, and $\tilde{\tilde{q}}$ are the public, key, and private parameters, respectively; and $F$ is a finite set of $f$ references to other protocols, $(F_1, \ldots, F_f)$.

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 $U = (\mathcal{R}, \mathcal{M})$, 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 $U_1 \cup U_2$ is $(\mathcal{R}_1 \cup \mathcal{R}_2, \mathcal{M}_1 \cup \mathcal{M}_2)$.

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:** received$(\tau) = \{ m_i \in P \mid i + o_i \leq \|\tau\| \}$

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:** known$(\tau, R, \vec{v}, R_i) = \{ p \mid \exists m_i \in \tau \exists p_i \Rightarrow \vec{v}_i \downarrow_{\vec{v}} \downarrow_{\vec{v}}$ and $(s_i = R \text{ or } r_i = R \text{ and } m_i \in \text{received}(\tau)) \}$

A message instance is viable on a path if the sender knows the $\tau_{\text{in}}^n$ parameters, but not the $\tau_{\text{out}}^n$ or $\tau_{\text{nil}}^n$ parameters.

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

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 paths$(\mathcal{U})$.

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

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

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

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

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:** sources$(\tau, R) = \{ s_i \mid p \in \text{outs}(m_i \in \tau) \}$

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

**Definition 15:** keys$(\tau) = \{ k_i \mid m_i \in \tau \}$, 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 $\tau'$ subsumes path $\tau$ for parameter list $\tilde{p}$, denoted $\tau' \triangleright \tilde{p} \tau$, if and only if

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


We say that the protocols are safe or live if and only if, for every \( Q \), there is a path \( \tau \) such that \( \tau \subseteq \tau' \) and \( \tau' \triangleright \tilde{p} \tau_2 \), where \( \tilde{R} \) is the set of roles that appear in \( \tau \).

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 \( \tilde{p} \), denoted \( Q \leq_p P \), if and only if for every \( \tau \) and path \( \tau_0 \) in paths of \( U \cup \tilde{U} \), there is a path \( \tau \) in paths of \( U \cup \tilde{U} \) such that (1) \( \tau \triangleright \tilde{p} \tau_0 \), and (2) branches(\( \tilde{U} \cup \tilde{U}, \tau \)) \( \neq \emptyset \) implies branches(\( \tilde{U} \cup \tilde{U}, \tau_0 \)) \( \neq \emptyset \) or \( \| \text{received}(\tau_0) \| < \| \text{\textit{tr}}(\tau) \| \).

Proof. Suppose \( Q \leq_p P \), with \( \tilde{p} \) having public parameters \( \tilde{p} \). Let \( \tilde{U} = \{ \tilde{U} \cup \tilde{U}, \tau \} \) that does not subsume any path \( \tau_0 \) in \( \tilde{U} \cup \tilde{U} \), or (2) \( \tau_0 \) has branches but \( \tau_0 \) does not and \( \tau_0 \) does not have any unreceived messages.

(1) Suppose there is some path \( \tau_0 \) in paths of \( U \cup \tilde{U} \) that subsumes some path \( \tau_0 \) in paths of \( U \cup \tilde{U} \), but for some message \( m \) in \( U \cup \tilde{U} \), \( \tau_0 \circ m \) does not subsume any path in paths of \( U \cup \tilde{U} \). Since \( \tau_0 \) subsumes \( \tau \), they must induce the same knowledge at each role, and thus would enable the same messages. Thus for \( \tau_0 \circ m \) to not subsume any path, \( m \) must not exist in \( U \cup \tilde{U} \), which means it must be in \( Q \). But by the definition of refinement, for any \( \tau \), \( \tau_0 \in P \) in paths of \( \tilde{U} \cup \tilde{U} \), \( \| \text{received}(\tau) \| < \| \tau_0 \| \). Thus \( \tau_0 \) has branches in contradiction of supposition (2).

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

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 \( Q \leq P \), then \( C_{P/Q} \leq C_p \).

Proof Sketch. Suppose \( Q \leq P \), with \( P \) having public parameters \( \tilde{p} \). Let \( \tilde{U} = \{ \tilde{U} \cup \tilde{U}, \tau \} \) that does not subsume any path \( \tau_0 \) in \( \tilde{U} \cup \tilde{U} \), or (2) \( \tau_0 \) has branches but \( \tau_0 \) does not and \( \tau_0 \) does not have any unreceived messages.

(1) Suppose there is some path \( \tau_0 \) in paths of \( U \cup \tilde{U} \) that subsumes some path \( \tau_0 \) in paths of \( U \cup \tilde{U} \), but for some message \( m \) in \( U \cup \tilde{U} \), \( \tau_0 \circ m \) does not subsume any path in paths of \( U \cup \tilde{U} \). Since \( \tau_0 \) subsumes \( \tau \), they must induce the same knowledge at each role, and thus would enable the same messages. Thus for \( \tau_0 \circ m \) to not subsume any path, \( m \) must not exist in \( U \cup \tilde{U} \), which means it must be in \( Q \). But by the definition of refinement, for any \( \tau \), \( \tau_0 \in P \) in paths of \( \tilde{U} \cup \tilde{U} \), \( \| \text{received}(\tau) \| < \| \tau_0 \| \). Thus \( \tau_0 \) has branches in contradiction of supposition (2).

Thus, (1) and (2) are false, contradicting (0), so \( C_{P/Q} \leq C_p \).
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

```plaintext
foreach $\tau_Q$ in paths($\mathcal{U}_Q$) do
match $\leftarrow \emptyset$
for $\tau_P$ in paths($\mathcal{U}_P$) do
  match $\leftarrow \tau_P$
  if There are no branches of $\tau_P$, or $\tau_Q$ 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 $\tau_Q$ 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 ($\sigma$) 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.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Min</th>
<th>Mean</th>
<th>Max</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buyer-Starts $\leq$ Either-Starts</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0.6</td>
</tr>
<tr>
<td>Single-Lookup $\leq$ Lookup-Prices</td>
<td>103</td>
<td>138</td>
<td>169</td>
<td>21.3</td>
</tr>
<tr>
<td>Pay-First $\leq$ Flexible-Purchase</td>
<td>53</td>
<td>70</td>
<td>87</td>
<td>9.6</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Substitution Property</th>
<th>Min</th>
<th>Mean</th>
<th>Max</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub. Buyer-Starts</td>
<td>1419</td>
<td>14745</td>
<td>15099</td>
<td>222</td>
</tr>
<tr>
<td>sub. Single-Lookup</td>
<td>18483</td>
<td>18568</td>
<td>18714</td>
<td>68</td>
</tr>
<tr>
<td>sub. Pay-First</td>
<td>18408</td>
<td>18512</td>
<td>18601</td>
<td>78</td>
</tr>
<tr>
<td>substitute all</td>
<td>11624</td>
<td>11868</td>
<td>12279</td>
<td>197</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Property</th>
<th>Min</th>
<th>Mean</th>
<th>Max</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetBill</td>
<td>Refinement</td>
<td>1256</td>
<td>1294</td>
<td>1323</td>
<td>21</td>
</tr>
<tr>
<td>NetBill</td>
<td>Liveness</td>
<td>13615</td>
<td>13926</td>
<td>14141</td>
<td>174</td>
</tr>
<tr>
<td>CreateLabOrder</td>
<td>Refinement</td>
<td>5879</td>
<td>6003</td>
<td>6162</td>
<td>83</td>
</tr>
<tr>
<td>CreateLabOrder</td>
<td>Liveness</td>
<td>45257</td>
<td>46469</td>
<td>50878</td>
<td>1648</td>
</tr>
</tbody>
</table>

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.
We have motivated and formalized a notion of refinement of information protocols. We established important results about this notion of refinement—that 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.

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. One could consider protocol refinement to meet its 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 $\rho$ features in both, $Q$ projected to $\rho$ would conform with $P$ projected to $\rho$.

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].

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

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 verification 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.

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.