Protocol Refinement: Formalization and Verification

Scott N. Gerard and Munindar P. Singh

Department of Computer Science
North Carolina State University
Raleigh, NC 27695 USA
{sngerard,singh}@ncsu.edu

Abstract. A proper definition of protocols and protocol refinement is crucial to designing multiagent systems. Rigidly defined protocols can require significant rework for even minor changes. Loosely defined protocols can require significant reasoning capabilities within each agent. Protocol definitions based on commitments is a middle ground.
We formalize a model of protocols consisting of agent roles, propositions, and commitments. We define protocol refinement between a superprotocol and a subprotocol by mapping superprotocol elements to corresponding subprotocol elements. Mapping protocol commitments depends on a novel operation called serial composition. We demonstrate protocol refinement.

1 Introduction

We focus our attention on service engagements between businesses and customers (B2B and B2C) over the Internet. In current practice, such engagements are defined rigidly and purely in operational terms. Consequently, the software components of the business partners are tightly coupled with each other, and depend closely on the engagement specification. Thus the business partners interoperate, but just barely. Even small changes in one partner's components must be propagated to others, even when such changes are not consequential to the business being conducted. Alternatively, in current practice, humans carry out the necessary engagements manually with concomitant loss in productivity.

In such an environment, if there were no mechanisms to structure inter-agent communication, agent implementations would need to handle a wide variety of communication making agent implementations complex with sophisticated reasoning capabilities as each interaction would be unique and customized. It would be difficult to predict *a priori* whether two agents could interoperate.

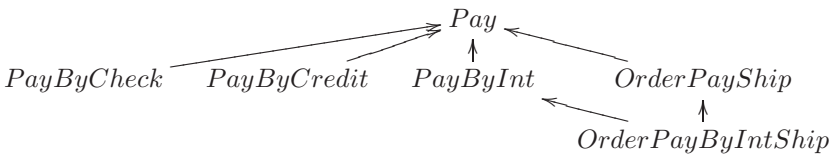
Protocols, as we understand them, provide a happy middle between rigid automation and flexible manual execution. Using protocols as a mechanism to structured communication, agent implementations can be less sophisticated. Protocol designers design and analyze protocols for desirable properties. Agents can publicly declare the protocols in which they can participate making it easier to find agents with whom to interoperate.

Protocols are a way to standardize communication patterns so agents can be used in many different multiagent interactions. Consider the simple protocol *Pay* consisting of a single action where a payer pays a payee. And consider protocol *OrderPayShip* where a

buyer and a seller agree to a price for a particular good, the buyer pays the seller and the seller ships the good to the buyer. The payer and payee roles in *Pay* should correspond to the buyer and seller roles in *OrderPayShip*. The payment in *Pay* should correspond to the payment in *OrderPayShip*. Therefore, we expect *OrderPayShip* refines *Pay*.

Suppose protocol *PayByInt* (pay by intermediary) is introduced where the payer first pays a middleman, who in turn pays the payee. Since both *Pay* and *PayByInt* send a payment from the payer to the payee, we expect *PayByInt* refines *Pay*. Similar arguments imply *PayByCheck*, *PayByCredit*, and others also refine *Pay*. If *PayByInt* becomes popular, we would like to construct a new protocol *OrderPayByIntShip*, which is just like *OrderPayShip*, except payments are made using *PayByInt* rather than *Pay*.

This diagram shows the expected refinement relationships between various protocols.



We are working to implement refinement checking via the MCMAS model checking [2] to handle complex protocols like those found in real service engagements.

Contributions

The main contributions of this paper are a definition of a refinement relation between two protocols, the notion of covering commitments, and the definition of serial composition of commitments. It describes why commitment-based protocols are more flexible than traditional computer protocols using the idea of multiple states of completion.

Organization

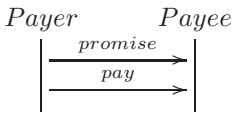
Section 2 introduces our running examples. Section 3 describes background material on commitments. Section 4 describes our intuitions and framework for protocol refinement, covering commitments, and serial composition of commitments. Section 5 briefly describes our intentions for implementing refinement checking with the MCMAS model checker. Section 6 demonstrates refinement on an example. Section 7 evaluates our approach. Section 8 describes other works and our future directions.

2 Examples

We introduce four running examples. *Pay* and *PayByInt* are basic payment protocols while *OrderPayShip* and *OrderPayByIntShip* are order protocols involving payments.

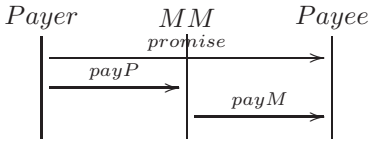
2.1 Pay

Pay is a basic payment protocol between a payer and a payee. If the payer chooses to do so, it commits to pay the payee by action promise. Then, at some later point, it sends a single payment directly to the payee. This sequence diagram describes the interaction.



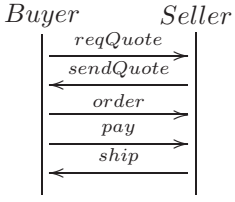
2.2 PayByInt

In protocol *PayByInt* (pay by intermediary), if the payer chooses to do so, it commits to pay the payee with promise. It then pays by sending a payment *indirectly* to the payee. The payer first pays a middleman, who in turn pays the payee. We assume the middleman commits to perform *payM* if payer performs *payP*. This sequence diagram shows a typical interaction, but sequence diagrams document only one message run. Other runs may also be valid. In this case, it is acceptable for the middleman to be generous and execute *payM* before *payP*.



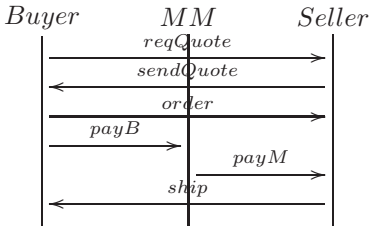
2.3 OrderPayShip

In *OrderPayShip* a buyer orders goods from a seller. The buyer requests a price quote for a good from the seller. The seller sends the price quote along with its commitment to ship the good if the buyer orders. The buyer can accept the offer by ordering and making its commitment to pay for the good if it orders. The seller can ship first, or the buyer can pay first.



2.4 OrderPayByIntShip

Protocol *OrderPayByIntShip* is similar to *OrderPayShip* except the payer uses *PayByInt* for payment. This introduces a new middleman role.



3 Background

3.1 Commitments

Commitments are a formal and concise method of describing how agent roles commit to perform future actions. We extend previous commitment definitions [5] in two ways. First, we allow both debtors and creditors to be sets of roles. This handles situations where a chain of debtors and intermediaries must all act to fulfill a commitment, and where a chain of creditors and intermediaries all need to know whether a commitment is satisfied. Second, we implement prior uses of **delegate** and **assign** with a single, new **transfer** operation.

Definition 1. *A commitment is an object*

$$\mathsf{C}_{\{debtors\},\{creditors\}}(\text{ant}, \text{csq}) \tag{1}$$

where debtors and creditors are sets of roles, ant is the antecedent, and csq is the consequent. When a commitment is active, the debtors are conditionally *committed to the creditors. Once ant becomes true, the debtors are* unconditionally *committed to make csq true at some point in the future.*

The valid operations on commitments are

- ***create**, performed only by debtors, creates a new commitment and makes it active.*
- *When the antecedent becomes true, the commitment is implicitly converted to an unconditional commitment.*
- *When the consequent becomes true, the commitment is implicitly satisfied and no longer active. Typically the consequent become true only after the antecedent becomes true, but this is not required.*
- ***transfer**, performed by either debtors or creditors, ends the current commitment and marks it as transferred to another commitment and no longer active.*
- ***release**, performed only by creditors, releases the debtors from their commitment. The commitment is released and no longer active.*
- ***cancel**, performed only by debtors, cancels the debtors' commitment. The commitment is violated and no longer active.*

A commitment is always in one of these states.
- *inact: the initial state;*
- *cond: after **create** with ant false, csq false, and no other operations;*
- *uncond: after **create** with ant true, csq false, and no other operations;*
- *sat: after **create** and csq true;*
- *xfer: after **create** and **transfer**;*
- *rel: after **create** and **release**; and*
- *can: after **create** and **cancel**.*

A commitment in state *sat*, *xfer*, *rel*, or *can* is said to be *resolved*.

For unconditional commitments, the debtors are committed to eventually make *csq* true. If the debtors fail, responsibility can be *several* (each debtor is responsible for just its portion), *joint* (each debtor is fully responsible for the entire commitment), or *joint and several* (the creditors hold one debtor fully responsible, who then pursues other debtors). We use *several* responsibility.

We note that contracts are built from multiple commitments; each party commits to perform the actions for which it is responsible. So while contracts are created by both debtors and creditors, commitments are created only by debtors.

Before a commitment's **create** (state *inact*), the commitment has no force. A created commitment (state *cond*) is conditionally committed. Unconditional commitments must eventually resolve to state *sat*, *xfer*, *rel*, or *can*. It is possible for the consequent to become true before the antecedent. While unusual, debtors have the option to act before being required to do so. Debtors are discouraged from **cancel**, but circumstances may require it, with consequences handled outside the current mechanisms.

In *Pay*, we represent the payer's commitment to paying the payee as

$$\mathsf{C}_{\text{Payer,Payee}}(\mathit{promise}, \mathit{pay})$$

In *PayByInt*, we represent the payer's and middleman's combined commitment as

$$\mathsf{C}_{\{\text{Payer,MM}\},\text{Payee}}(\mathit{promise}, pay_P \wedge pay_M)$$

Previous commitment descriptions allow debtors to **delegate**, or creditors to **assign**, a commitment to another role. Both terminate the existing commitment and create a new commitment with modified roles. We model these operations as a **transfer** operation which terminates the existing commitment plus a separate **create** of a new commitment.

$$\begin{aligned}\textbf{delegate}(\mathsf{C}_i, debt') &= \textbf{transfer}(\mathsf{C}_i) \wedge \textbf{create}(\mathsf{C}_{d'})\\ \textbf{assign}(\mathsf{C}_i, cred') &= \textbf{transfer}(\mathsf{C}_i) \wedge \textbf{create}(\mathsf{C}_{c'})\end{aligned}$$

where $\mathsf{C}_i = \mathsf{C}_{\text{debt,cred}}(\mathit{ant}, \mathit{csq})$, $\mathsf{C}_{d'} = \mathsf{C}_{\text{debt',cred}}(\mathit{ant}, \mathit{csq})$ and $\mathsf{C}_{c'} = \mathsf{C}_{\text{debt,cred'}}(\mathit{ant}, \mathit{csq})$. Since **delegate** and **assign** have essentially the same effect, **transfer** captures the essence of both and somewhat simplifies the definition of commitments.

3.2 Unconditional Commitments Must Resolve

We require debtors must eventually resolve all their unconditional commitments, even if that is **cancel**.

Model checkers have fairness constraints which eliminate unfair paths from consideration. Fairness constraints are typically used to eliminate unfair scheduler paths. We use fairness constraints to eliminate paths where agents never resolve their unconditional commitments. This is a constraint on agent implementations, not a constraint on the protocol itself.

4 Framework

4.1 Protocol Basics

A protocol specification language needs to describe all participating agent roles, the actions they can perform, and any constraints (guards) on their actions. Agents send *message actions* to each other. Message actions are an agent-level concept. Below we decompose message actions into an unordered set of *basic actions*.

We model protocols using CTL. We consider runs of basic actions which generate state runs as is traditional for model checking. An action run is modeled as

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$$

When we compare points in time and actions, "$s_i < a_j < s_k$" means $i \leq j < k$. When we compare two actions, "$a_i < a_j$" means $i < j$.

While support for looping protocols is desirable, we simplify the initial problem and do not consider them here. We hope to extend our work to cover looping protocols later.

4.2 Messages and Guards

Both message actions and basic actions can have guard conditions. An action is enabled for execution only when its guard is true.

Some protocols must constrain message orders. For example, (1) when one message provides a value required by another message, or (2) due to regulatory requirements (you must show a valid ID before boarding an airplane). An action's guard is written

$$guard < action$$

which means *action* can occur in a state only if *guard* is true in that state. The action is not required to execute when the guard is true. Guards $g_1 < action$ and $g_2 < action$ can be combined into $g_1 \wedge g_2 < action$.

Guards for message actions have been used elsewhere including MCMAS. We also introduce guards for basic actions since the protocol designer may need to constrain the ordering of basic actions in subprotocols. The designer can specify multiple guards for an action. The complete guard condition for a basic action includes any guards for the basic action as well as any guards for its containing message action. We expect tooling to combine all designer-specified guards into a single guard expression.

Example: A protocol designer might require

$$reqQuote \wedge sendQuote < order$$

4.3 Multiple Stages of Completion

Commitments evolve through four stages; proposition evolve through only two. The occurrence of a $prop = value$ basic action divides time into two stages: before and after the basic action.

A commitment evolves through four stages: (1) before creation (inactive), (2) conditionally committed (cond), (3) unconditionally committed (uncond), and (4) resolved.

$$\xrightarrow{inact} \textbf{create} \xrightarrow{cond} ant \xrightarrow{uncond} csq \xrightarrow{resolved}$$

Commitments increase protocol flexibility, because guards can specify "partially performed" actions. A protocol can make progress sooner if an action's guard specifies one of the first three stages.

Example: Using proposition *ship*, *OrderPayShip* can guard *pay* based only on the two stages of *ship*. The decision is "all" (*ship*ped) or "nothing" (not *ship*ped).

$$ship < pay \tag{2}$$

Using commitments, the protocol can guard *pay* based on any of the four commitment stages. A guard can enable *pay* as soon as the debtor has committed to make *ship* true.

$$\textbf{create}(\mathsf{C}_{\text{Seller,Buyer}}(pay, ship)) < pay \tag{3}$$

A protocol framework that includes commitments is inherently more flexible than traditional computer protocol frameworks. Where traditional protocol frameworks operate on an all-or-nothing basis with just two stages of completion, agent-based frameworks can operate based on four stages of completion (commitments have four stages of completion). Basing a decision on a completed action provides essentially no risk to a creditor. But commitments allow more flexible enactments because creditors can also base their decisions on the promises of the debtors which are partial or intermediate stages of completion. While creditors assume more risk in doing so, they assume less risk than acting without debtor promises.

4.4 Every Sub-run is a Super-run

According to the Liskov substitution principle [1], if $\phi(p)$ is a property provable about objects p of type P, then $\phi(q)$ should be true for objects q of type Q when Q is a subtype of P. We apply this principle to protocols. If $\phi(x)$ holds for a superprotocol, then it must also hold for its subprotocols. For example, let $\phi(x)$ be the property that action $order$ precedes action *pay*. $\phi(p)$ will be true of some runs, but not of other runs.

Subprotocols must satisfy every superprotocol property $\phi(x)$, but they can satisfy additional properties $\psi(x)$. Every subprotocol run must satisfy both $\phi(x)$ and $\psi(x)$, so there are fewer subprotocol runs than superprotocol runs because $\psi(x)$ eliminates some runs $\phi(x)$ does not.

Definition 2 (Protocol Refinement). *Every subprotocol basic action run must be a superprotocol basic action run.*

Our definition of protocol refinement does not imply that agents that can participate in a superprotocol can necessarily participate in a subprotocol unchanged. Agents work with message actions, but our definition for run comparison is based on basic actions. In our model, agents may need to be modified to participate in subprotocols. For example, an agent capable of participating in a basic payment protocol may need to change to handle payments via check or credit card.

4.5 Time Expansion

Basic actions which are concurrent at a high-level of abstraction (superprotocol) may not be concurrent at a lower-level of abstraction (subprotocol). When a superprotocol's message action is refined in the subprotocol, its basic actions can spread out over time (no longer concurrent).

Decomposing a message action into multiple basic actions requires splitting action intervals and creating additional time points in the run (expanding time).

4.6 Decomposition

A message action may have multiple effects. To better understand and characterize a *message action*, we decompose each message action into an unordered set of one or more *basic actions*. Each basic action has well-defined semantics and is useful for analyzing and understanding the meanings of message actions.

We consider two different kinds of basic actions. Setting the value of a proposition to true or false is a propositional basic action. Each of the commitment operations **create**, **transfer**, **release**, and **cancel** are commitment basic actions.

Example: The seller's message action *sendQuote* in *OrderPayShip* decomposes into two basic actions. It sets the propositional *sendQuote* fluent to true to record the fact that it responded to the buyer. And it creates a commitment that the seller will ship if the goods are ordered.

See the discussion for diffusion below for the exact guard conditions.

4.7 Mapping

Since superprotocols represent a higher-level abstractions than subprotocols, the differences in levels must be addressed. There is often no one-to-one correspondence between superprotocol and subprotocol elements. Protocol elements must be mapped between the two protocols to compare them. Since subprotocols typically contain more detail than superprotocols, we map every single superprotocol element (role, proposition or commitment) to an expression of one or more corresponding subprotocol elements. Subprotocols may contains sub-elements that do not correspond with any super-element.

Example: an *openAccount* basic action in a high-level, banking superprotocol, might decompose to multiple basic actions in a low-level subprotocol: *checkIdentity*, *checkCredit*, *createAccount*, and *notifyCustomer*.

Example: the *pay* basic action in *Pay* maps to the *payP* and *payM* expression in *PayByInt*.

While it would be desirable for the mapping between superprotocols and subprotocols to be automatically determined, there can be multiple, distinct mappings between some protocol pairs. In the mapping between *Pay* and *PayByInt*, one mapping groups the Middleman with the Payer, making the Middleman work on behalf of the Payer. Another mapping groups the Middleman with the Payee, making the Middleman work on behalf of the Payee. Both interpretations are valid.

Every super-basic-action is mapped to an expression of sub-basic-actions. There is one mapping function for each super-basic-action. The mapping function for basic propositions is a boolean expression of sub-basic-propositions. The mapping function for super-basic-commitments is a serial composition of sub-basic-commitments.

$$super \mapsto map_{super}(sub_1, sub_2, \dots)$$

4.8 Projection

Let SubOnly be the set of basic actions that occur only in the subprotocol (basic actions not in the superprotocol and not in a mapping expression). Ignore all SubOnly basic actions in the subprotocol during run comparison.

This means we compare all basic actions in the superprotocol with just the matching basic actions in the subprotocol. The subprotocol is free to include additional basic actions that are unknown to the superprotocol.

4.9 Diffusion

Consider the case where a super-basic-action p maps to two sub-basic-actions $p \mapsto q_1 \wedge q_2$. For p to occur at the same time point in both sub-run and super-run, p must occur at the same time as the *last* of q_1 and q_2. For conjunction, the other sub-actions can occur at any *earlier* and possibly non-adjacent points in the run. In the case where $p \mapsto q_1 \vee q_2$, then p must occur at the same time point as the *first* of q_1 or q_2. For disjunction, the other sub-actions can occur at any *later* and possibly non-adjacent points in the run.

Example: in *OrderPayShip*, the buyer's order action is composed of the two basic actions of setting an *order* proposition and creating a commitment. In refinements of *OrderPayShip* these two basic actions can occur at different points in time.

When a sub-action guard is true, we require the corresponding super-action guard to also be true. Otherwise, the subprotocol could perform the sub-action while the superprotocol can not perform its corresponding super-action. The model checker tests this using formula

$$\mathbf{AG}(sub.guard \rightarrow super.guard) \tag{4}$$

We represent the diffusion condition as a guard condition. Since runs are serialized basic actions, we do not need to consider multiple basic actions occurring simultaneously.

If $super \mapsto sub_1 \wedge \dots \wedge sub_i$, the super-basic-action's guard must be true when the last of the sub-basic-action's guard is true.

$$sub_i.effguard = sub_i.baguard \wedge [sub_i.maguard \vee \neg(\bigwedge_{j \neq i} sub_i.occurred)]$$

The sub-basic-action's effective guard is built from two pieces. Any guard specifically applied to the sub-basic-action ($sub_i.baguard$) must always be true for the basic action to fire. Also, the message action guard must be true, or this must not be the last sub-basic-action. This effective guard is checked with equation 4.

If $super \mapsto sub_1 \vee \cdots \wedge sub_i$, the super-basic-action's guard must be true when the first of the sub-basic-action's guards is true.

$$sub_i.effguard = sub_i.baguard \wedge [sub_i.maguard \wedge (\bigwedge_i \neg sub_i.occurred)]$$

The super-basic-action's effective guard is again built from two pieces. Any guard specifically applied to the sub-basic-action ($sub_i.baguard$) must always be true for the basic action to fire. Also, the message action guard must be true when no sub-basic-actions have fired. This effective guard can be checked with equation 4.

Example: $pay \mapsto pay_P \wedge pay_M$ generates the effective guards

$$pay_P.effguard = true \wedge [(\textbf{create}(\mathsf{C}_{payM}) \wedge promised) \vee \neg payM]$$
$$pay_M.effguard = true \wedge [(\textbf{create}(\mathsf{C}_{payM}) \wedge promised) \vee \neg payP]$$

Since neither pay_P nor pay_M have specific basic action guards ($baguard = true$), the message action guard is **create** and $promised$, and they each require the other basic action must not have fired.

4.10 Run Comparison

Figure 1 illustrates the steps required for refinement. Compare one super message action run and one sub message action run, as follows.

1. Begin with the superprotocol's and subprotocol's message action runs.
2. Decompose each run of message actions to a run of basic actions.
3. Map every super-basic-action to its expression as sub basic actions.
4. Serialize the basic actions sets in any order consistent with the basic action guards.
5. Ignore (project out) SubOnly basic actions in the sub run.
6. Diffuse basic actions.
7. Find any ordering of basic actions in both sub and super runs that satisfy all the ordering constraints.
8. If the two state runs are the same, the runs match.

4.11 Commitment Strength

We need a way to compare two commitments, in particular, to compare a commitment in the subprotocol with a commitment in the superprotocol.

Definition 3 (Commitment Strength). *A commitment C_S is stronger than a commitment C_W, written $\mathsf{C}_S \geq \mathsf{C}_W$, iff*

$$\mathsf{C}_W.debt \subseteq \mathsf{C}_S.debt \tag{5}$$
$$\mathsf{C}_W.cred \subseteq \mathsf{C}_S.cred \tag{6}$$
$$\mathsf{C}_W.ant \vdash \mathsf{C}_S.ant \tag{7}$$
$$\mathsf{C}_S.csq \vdash \mathsf{C}_W.csq \tag{8}$$

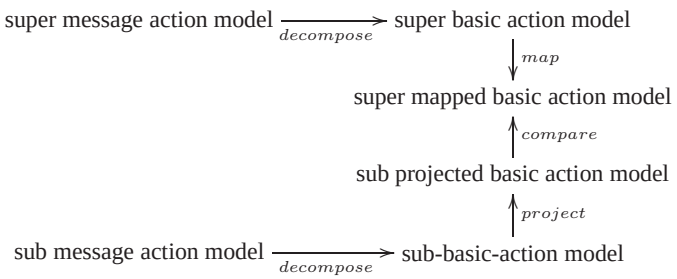
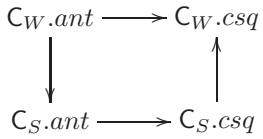


Fig. 1. Relationships between models

where $\subseteq$ is subset and $\vdash$ is derives.

Equations (5) and (6) allow additional roles in the subprotocol to be involved as both debtors and creditors. Equations (7) and (8) are based on the following diagram.



If C_S is stronger than C_W, then both side implications are true by equations (7) and (8). If C_S is satisfied, then the bottom implication is true. Then C_W, the top implication, will be true.

Theorem 1. *Commitment strength is reflexive and transitive.*

Proof. Reflexive and transitivity are immediate from the definition.

Example: $\mathsf{C}(order \vee freeCoupon, ship) \geq \mathsf{C}(order, ship)$ since the stronger commitment commits at least when $order$ is true. It also commits when $freeCoupon$ is true.

Example: $\mathsf{C}(order, ship \wedge expressDelivery) \geq \mathsf{C}(order, ship)$ since the stronger commitment commits to the additional consequent $expressDelivery$.

4.12 Covering

When a commitment C_{sub} in a subprotocol is stronger than a commitment C_{super} in a superprotocol we say the sub-commitment "covers" the super-commitment. We require every super-commitment must be covered by some sub-commitment. This guarantees that the super-commitment is satisfied whenever the sub-commitment is satisfied.

A single subprotocol commitment clearly covers an identical superprotocol commitment. A sub-commitment with more debtors, more creditors, a weaker antecedent, or a stronger consequent covers a super-commitment. However, this is not always enough. We allow serial composition of commitments as another way to cover commitments.

4.13 Intermediaries

Whereas two roles may communicate directly with each other using a single message action in a protocol at a high-level of abstraction, there is a natural tendency for message communication to pass through multiple intermediary roles as that protocol is refined to lower-levels of abstraction. Protocol refinement must properly handle intermediaries. One super-proposition could map to an expression of multiple sub-propositions, each controlled by different roles (intermediaries). One super-commitment could be fulfilled through multiple intermediaries. Super-elements must be able to span intermediaries.

Example: the *pay* action in *Pay* becomes the two distinct *payP* and *payM* actions in the subprotocol *PayByInt*. These two actions must be in different message actions in *PayByInt*, because they are performed by different roles.

Example: commitments can chain through multiple intermediaries. When *PayByInt* refines *Pay*, the single commitment from the payer to the payee in *Pay* does not appear explicitly in *PayByInt*. Rather, the subprotocol *PayByInt* has two separate commitments that form a chain passing through the middleman. That chain commits the payer to pay the payee.

4.14 Serial Composition

We also need a mechanism for commitments to span intermediaries. Previous commitment formulations [6] included the idea of commitment chaining, but we formalize this in a new way as "serial composition" of commitments. Serial composition computes a result commitment from a chain of commitments.

Definition 4 (Serial Composition). *Two commitments* C_A *and* C_B *are combined into a resultant commitment* $\mathsf{C}_\oplus = \mathsf{C}_A \oplus \mathsf{C}_B$ *if the operation is well-defined*

$$\mathsf{C}_A.ant \wedge \mathsf{C}_A.csq \vdash \mathsf{C}_B.ant \tag{9}$$

Then $\mathsf{C}_\oplus$ *is defined as*

$$\mathsf{C}_\oplus.debt := \mathsf{C}_A.debt \cup \mathsf{C}_B.debt \tag{10}$$

$$\mathsf{C}_\oplus.cred := \mathsf{C}_A.cred \cup \mathsf{C}_B.cred \tag{11}$$

$$\textbf{\textit{create}}(\mathsf{C}_\oplus) := \textbf{\textit{create}}(\mathsf{C}_A) \wedge \textbf{\textit{create}}(\mathsf{C}_B) \tag{12}$$

$$\mathsf{C}_\oplus.ant := \mathsf{C}_A.ant \tag{13}$$

$$\mathsf{C}_\oplus.csq := \mathsf{C}_A.csq \wedge \mathsf{C}_B.ant \wedge \mathsf{C}_B.csq \tag{14}$$

$$\textbf{\textit{transfer}}(\mathsf{C}_\oplus) := \textbf{\textit{transfer}}(\mathsf{C}_A) \vee \textbf{\textit{transfer}}(\mathsf{C}_B) \tag{15}$$

$$\textbf{\textit{release}}(\mathsf{C}_\oplus) := \textbf{\textit{release}}(\mathsf{C}_A) \vee \textbf{\textit{release}}(\mathsf{C}_B) \tag{16}$$

$$\textbf{\textit{cancel}}(\mathsf{C}_\oplus) := \textbf{\textit{cancel}}(\mathsf{C}_A) \vee \textbf{\textit{cancel}}(\mathsf{C}_B) \tag{17}$$

$\mathsf{C}_\oplus$ is a new commitment object whose attributes are defined in terms of the attributes of C_A and C_B. $\mathsf{C}_\oplus$ does not provide any information beyond that given in C_A and C_B, but it expresses it in the form of a new commitment.

In $\mathsf{C}_\oplus$, represents that the debtor group is committed to the creditor group to bring about consequent $\mathsf{C}_A.csq \wedge \mathsf{C}_B.ant \wedge \mathsf{C}_B.csq$ when just antecedent $\mathsf{C}_A.ant$ is true.

Debtors are *severally* responsible for $\mathsf{C}_{\oplus}$, so that debtors do not become responsible for more than their input commitments.

Our well-defined condition (equation 9) generalizes the chain rule in [6].

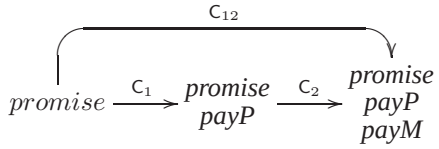
Longer chains of commitments can be composed if each operation is well-defined. We always evaluate $\oplus$ left-to-right.

$$\mathsf{C}_{12\ldots n} = ((\mathsf{C}_1 \oplus \mathsf{C}_2) \oplus \ldots) \oplus \mathsf{C}_n$$

As an example, consider the two commitments in *PayByInt*.

$$\begin{aligned}\mathsf{C}_1 &= \mathsf{C}_{\text{Payer,Payee}}(\textit{promised}, \textit{paidP})\\ \mathsf{C}_2 &= \mathsf{C}_{\text{MM,Payer}}(\textit{paidP}, \textit{paidM})\\ \mathsf{C}_{12} &= \mathsf{C}_{\{\text{Payer,MM}\},\{\text{Payer,Payee}\}}(\textit{promised}, \textit{paidP} \wedge \textit{paidM})\end{aligned}$$

which can illustrated as a chain of commitment edges connecting nodes of propositions.



Theorem 2. *Serial composition is not commutative and not associative.*

Usually, serial composition creates stronger commitments. $\mathsf{C}_A \oplus \mathsf{C}_B$ is stronger than C_A alone because, even though both have the same antecedent ($\mathsf{C}_A.ant$), in general, $\mathsf{C}_A \oplus \mathsf{C}_B$ has a stronger consequent ($\mathsf{C}_A.csq \wedge \mathsf{C}_B.ant \wedge \mathsf{C}_B.csq$) with more conjuncts. However, the next theorem shows this is not always the case.

Operator $\oplus$ obeys the following idempotent-like property.

Theorem 3. *Extending a serial composition with a commitment already part of the chain, does not create a stronger commitment.*

$$\mathsf{C}_1 \oplus \cdots \oplus \mathsf{C}_k \oplus \cdots \oplus \mathsf{C}_n \oplus \mathsf{C}_k \tag{18}$$
$$= \mathsf{C}_1 \oplus \cdots \oplus \mathsf{C}_k \oplus \cdots \oplus \mathsf{C}_n \tag{19}$$

Proof. $(\mathsf{C}_1 \oplus \cdots \oplus \mathsf{C}_n) \oplus \mathsf{C}_k$ is well-defined because $\mathsf{C}_k.ant$ is already part of the left-hand side of equation 9. By simple inspection, conditions (10-17) are the same for both sides. Expression 18 is identical to, not stronger than, expression 19.

A commitment can usefully be added to a commitment chain only once; doing so does not create a stronger serial composition. Repeating any part of a loop does not create a stronger serial composition. Given n commitments, the number of distinct serial compositions is bounded above by 2^n.

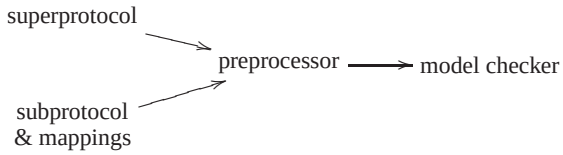


Fig. 2. Processing Steps

5 Processing

Figure 2 shows an overview of our proposed processing. The protocol specifications for the superprotocol and subprotocol are read from files. Note the subprotocol contains one or more mappings between the protocols. These files are read by a preprocessor and are used to generate an MCMAS ISPL model. The ISPL model is input to the MCMAS model checker which checks all the formulae. If all the formula are true, the subprotocol refines the superprotocol.

The preprocessor generates the following checking formulae

$$\mathbf{AG}(sub.guard \rightarrow super.guard) \tag{20}$$

$$\mathbf{AG}(\mathsf{C}_{super}.state = uncond \rightarrow \mathbf{AF}(\mathsf{C}_{super}.state \neq uncond)) \tag{21}$$

$$\mathbf{AG}(\mathsf{C}_A.ant \wedge \mathsf{C}_A.csq \rightarrow \mathsf{C}_B.ant) \tag{22}$$

$$\mathbf{AG}(\mathsf{C}_{super}.ant \rightarrow \mathsf{C}_{sub}.ant) \tag{23}$$

$$\mathbf{AG}(\mathsf{C}_{sub}.csq \rightarrow \mathsf{C}_{super.csq}) \tag{24}$$

Equation 20 ensures the super-run can perform super-basic-actions whenever the sub-run can perform corresponding sub-basic-actions. While MCMAS fairness constraints eliminate runs where unconditionally committed sub-commitments fails to resolve, equation 21 ensures unconditionally committed super-commitments must resolve. Equation 22 parallels equation 9 ensuring serial compositions are well-defined. Equations 23-24 parallel equations 13-14 ensuring sub-commitments cover super-commitments.

6 PayByInt Refines Pay

6.1 Pay

Space precludes a detailed description of our proposed protocol specification language, so we simply state the protocol specifications with a few notes. Line (1) names and defines the commitment. Lines (2)-(6) describe the Payer role. There are two message actions: *promise* and *pay*, and both are sent by Payer to Payee. The unordered set of basic actions are listed between $\{$ and $\}$. Line (4) defines the guard for *pay* as *promised* and the creation of the commitment. Payee sends no messages in this protocol.

Algorithm 1 *Pay* Protocol

1: $\mathsf{C}_{pay} = \mathsf{C}_{Payer,Payee}(promised, paid)$
2: **role** Payer {
3: $promise = Payer \rightarrow Payee\{promised, \textbf{create}(\mathsf{C}_{pay})\}$
4: $promised \wedge \textbf{create}(\mathsf{C}_{pay}) < pay$
5: $pay = Payer \rightarrow Payee\{paid\}$
6: }
7: **role** Payee {
8: }

Algorithm 2 *PayByInt* Protocol

1: $\mathsf{C}_{payP} = \mathsf{C}_{\text{Payer,Payee}}(promised, paidP)\}$
2: $\mathsf{C}_{payM} = \mathsf{C}_{\text{MM,Payer}}(paidP, paidM)$
3: **role** Payer {
4: $promise = Payer \rightarrow Payee\{promised, \textbf{create}(\mathsf{C}_{payP})\}$
5: $promised \wedge \textbf{create}(\mathsf{C}_{payM}) < payP$
6: $payP = Payer \rightarrow MM\{paidP\}$
7: }
8: **role** MM {
9: $init = \{\textbf{create}(\mathsf{C}_{payM})\}$
10: $payM = MM \rightarrow Payee\{paidM\}$
11: }
12: **role** Payee {
13: }
14: **map** M1: Pay $\mapsto$ PayByInt {
15: $Payer \mapsto \{Payer, MM\}$
16: $Payee \mapsto \{Payee\}$
17: $promised \mapsto promised$
18: $paid \mapsto paidP \wedge paidM$
19: $\mathsf{C}_{pay} \mapsto \mathsf{C}_{payP} \oplus \mathsf{C}_{payM}$
20: }
21: **map** M2: Pay $\mapsto$ PayByInt {
22: $Payer \mapsto \{Payer\}$
23: $Payee \mapsto \{MM, Payee\}$
24: $promised \mapsto promised$
25: $paid \mapsto paidP \wedge paidM$
26: $\mathsf{C}_{pay} \mapsto \mathsf{C}_{payP} \oplus \mathsf{C}_{payM}$
27: }

6.2 PayByInt

Initially (9), Middleman commits to Payer to pass along any payment it receives. This protocol does not address how this commitment came to be created. Payer can not pay Middleman until this commitment has been created (5). Since there is no guard on *payM*, Middleman is free to pay early; but the decision is part of Middleman's agent implementation.

6.3 Refinement Test

We now sketch *PayByInt* refines *Pay*. We test refinement by comparing basic action runs. This following diagram shows superprotocol *Pay* on the top half and subprotocol *PayByInt* on the bottom half. The message action runs are shown on the very top and very bottom for easy reference, but the heart of the diagram is the two basic action runs in the middle. Each position in a basic action run is a set of basic actions.

One of *PayByInt*'s message action runs its basic action run is

$$
\begin{array}{lccccc}
\textit{Pay}\ msg: & & promise & & & pay \\
\textit{Pay}\ basic: & & \left\{\begin{array}{c} promised \\ \textbf{create}(\mathsf{C}_{pay}) \end{array}\right\} & & & \{paid\} \\
\textit{PayByInt}\ basic: & \{\textbf{create}(\mathsf{C}_{payM})\} & \left\{\begin{array}{c} promised \\ \textbf{create}(\mathsf{C}_{pay}) \end{array}\right\} & \{paidP\} & \{paidM\} & \\
\textit{PayByInt}\ msg: & init & promise & payP & payM &
\end{array}
$$

Erase the message action boundaries and serialize the sets of basic actions to individual basic actions. The basic actions within a set can be serialized in any order that does not violate the basic action guards.

$$
\begin{array}{lccccc}
time: & 0 & 1 & 2 & 3 & 4 \\
\textit{Pay}: & & promised & \textbf{create}(\mathsf{C}_{pay}) & & paid \\
\textit{PayByInt}: & \textbf{create}(\mathsf{C}_{payM}) & promised & \textbf{create}(\mathsf{C}_{payP}) & paidP & paidM
\end{array}
$$

PayByInt refines *Pay* under two mappings. Here, we demonstrate only the mapping M1 in lines 14-20 where Payer and Middleman form a coalition. M1 shows the inter-protocol mappings with *Pay*'s elements are on the left and *PayByInt*'s elements are on the right. Note serial composition of the two commitments in *PayByInt* is required to cover the commitment in *Pay* (line 19).

Lines 15-16 map a single super-role to one or more sub-roles. Line 17 maps a single super-proposition to a single sub-proposition and both occur at time 1 in the run. Line 18 maps the single super-proposition *paid* to a conjunction of sub-propositions. The super-proposition's occurrence must align with the latest sub-proposition's occurrence (time 4).

Finally, we must compute the serial composition in Line 19. From equation 9, we verify $promised \wedge paidP \vdash paidP$ so the composition is well-defined. Equations 10-17

define the composition

$$\mathsf{C}_{\oplus}.debt := \{Payer, MM\} \tag{25}$$
$$\mathsf{C}_{\oplus}.cred := \{Payer, Payee\} \tag{26}$$
$$\textbf{create}(\mathsf{C}_{\oplus}) := \textbf{create}(\mathsf{C}_{payP}) \wedge \textbf{create}(\mathsf{C}_{payM}) \tag{27}$$
$$\mathsf{C}_{\oplus}.ant := promised \tag{28}$$
$$\mathsf{C}_{\oplus}.csq := paidP \wedge paidM \tag{29}$$
$$\textbf{transfer}(\mathsf{C}_{\oplus}) := \textbf{transfer}(\mathsf{C}_{payP}) \vee \textbf{transfer}(\mathsf{C}_{payM}) \tag{30}$$
$$\textbf{release}(\mathsf{C}_{\oplus}) := \textbf{release}(\mathsf{C}_{payP}) \vee \textbf{release}(\mathsf{C}_{payM}) \tag{31}$$
$$\textbf{cancel}(\mathsf{C}_{\oplus}) := \textbf{cancel}(\mathsf{C}_{payP}) \vee \textbf{cancel}(\mathsf{C}_{payM}) \tag{32}$$

Then we verify the sub-commitment $\mathsf{C}_{\oplus}$ covers the super-commitment ($\mathsf{C}_{\oplus} \geq \mathsf{C}_{pay}$) using equations 5-8.

Therefore, the run is a valid sub-run and is a valid super-run, which is consistent with Definition 2. We have only demonstrated a single run here. A full refinement demonstration requires demonstrating all sub-runs. Model checking is required to check all the runs for large protocols.

7 Evaluation

We are writing a proprocessor to generate input for the MCMAS model checker from a protocol descriptions. Model checking protocols should use a fraction of the states supported by current model checking technology, so we should not experience performance or scale problems.

We don't currently support protocols with loops. While loop-free protocols are sufficient for many situations, we hope to remove this limitation in the future.

7.1 Reusable Protocol Library

Protocol design requires substantial effort. Users would like to reuse previously designed protocols rather than having to design their own protocols from scratch. We envision a library of reusable protocols.

Any reasonable definition of refinement should be reflexive and transitive. Transitivity helps to structure such a library. When searching the library, whole subtrees can be eliminated from further consideration by one refinement failure.

8 Discussion

A protocol framework that includes commitments is inherently more flexible than traditional computer protocol frameworks. Where traditional protocol frameworks operate on an all-or-nothing basis with just two stages of completion, agent-based frameworks can operate based on a commitment's four stages of completion. Commitments allow

more flexible enactments because creditors can base their decisions on the promises the debtors which are partial stages of completion.

De Silva [4] propose interaction protocols for open systems based on Petri nets. They enable actions based on past and future preconditions. While their past preconditions are similar to our guards based on propositions, our guards based on conditional or unconditional commitments are more rigorously formalized than their future preconditions. They have no notion of protocol refinement and treat each protocol independently of every other.

Singh [6] states rules similar to those those proposed here for commitment strength. Our equation 9 is slightly stronger than their chain rule, they do not directly state a rule for stronger consequents, and they do not directly state a rule similar to serial composition.

Mallya & Singh [3] propose a definition of protocol refinement (there called subsumption) that compares the order of state pairs. But, we have found if superprotocol states map to overlapping ranges of equivalent subprotocol states, their definition can return false positives. Our procedure does not allow this possibility.

References

References

1. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16:1811–1841, 1994.
2. A. Lomuscio, H. Qu, and F. Raimondi. Mcmas: A model checker for the verification of multi-agent systems. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 682–688. Springer, 2009.
3. A. U. Mallya and M. P. Singh. An algebra for commitment protocols. *Journal of Autonomous Agents and Multi-Agent Systems*, 14(2):143–163, Apr. 2007.
4. L. Priyalal, M. Winikoff, and W. Liu. Extending agents by transmitting protocols in open systems. In *In Proceedings of the Challenges in Open Agent Systems Workshop*, 2003.
5. M. P. Singh. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7:97–113, 1999.
6. M. P. Singh. Semantical considerations on dialectical and practical commitments. In *Proceedings of the 23rd Conference on Artificial Intelligence (AAAI)*, pages 176–181, Menlo Park, July 2008. AAAI Press.