Protocol Refinement: Formalization and Verification

Scott N. Gerard
North Carolina State University

Cross-organizational business processes are not adequately supported by traditional methods, which fail to separate the interactions and business relationships of the participants from their internal implementations. We adopt a recent approach that gives primacy to the interactions among the participants. In this approach, a (business) protocol describes an interaction among two or more participants in high-level terms, specifically, via the creation and manipulation of the commitments among the participants. In this manner, a protocol offers increased flexibility to the participants while supporting a notion of correctness: all commitments that are activated are discharged.

This paper addresses a key challenge in engineering using business protocols. Specifically, it lifts the notion of refinement from objects (as in traditional software engineering) to protocols. Intuitively, it is clear that some interactions are refinements of others. For example, in intuitive terms, PayViaCheck is a kind of Pay. This paper provides a formal definition of protocols and of refinement with respect to a mapping between two protocols. It proposes an approach based on model checking using which we can compute whether refinement holds between any two protocols with respect to a mapping. We have implemented this approach over the MCMAS model checker and have evaluated it on several protocols occurring in the literature.

Categories and Subject Descriptors: F.3.1 [Specifying and Verifying and Reasoning about Programs]: Mechanical verification; I.2.11 [Artificial Intelligence]: Distributed Artificial Intelligence—Multiagent Systems

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. Business partners interoperate, but just barely. Even small changes in one partner’s components must be propagated to others, even when such changes are inconsequential 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 interagent 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 ground between rigid automation and flexible manual execution. Using protocols as a mechanism to structure communication, agent implementations can be less sophisticated. Protocol designers design and analyze protocols for desirable properties. Agents publicly declare the protocols in which they can participate making it easier to find agents with whom to interoperate.
Protocols standardize communication patterns so agents can be used in many different multiagent interactions. Consider the simple protocol *Pay* consisting of two actions where a payer commits to pay, and then 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. Intuitively, *OrderPayShip* includes a payment like the one in *Pay*. We can make the refinement relationship clear by mapping elements between the two protocols. 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 *PayViaInt* (pay by intermediary) is introduced where the payer first pays a middleman, who in turn pays the payee. Since both *Pay* and *PayViaInt* send a payment from the payer to the payee, we expect *PayViaInt* refines *Pay*. Similar arguments suggest *PayViaCheck*, *PayViaCredit*, and others should also refine *Pay*. If *PayViaInt* becomes popular, we would like to construct a new protocol *OrderPayViaIntShip*, which is just like *OrderPayShip*, except payments are made using *PayViaInt* rather than *Pay*.

This diagram shows a few of the expected refinement relationships between various protocols.

We have implemented a protocol refinement checking tool based, in part, on the MCMAS model checker [Lomuscio et al. 2009].

We do not describe interaction protocols in operational terms as simple message sequences. Rather, we take a less rigid, declarative approach that accounts for differences between protocols which “do the same thing.” Protocols between multiple agents are formalized by their meanings on commitments. We propose a method that demonstrates the refinement relationships above. Our refinement approach takes two protocols, *P* and *Q*, and a mapping function *M*, and answers whether *Q* is a subprotocol of *P*, or “*Q* refines *P*.” We develop the technical foundation of protocol refinement through a series of four intuitions.

**Contributions**

The main contributions of this paper are a definition of refinement between two protocols, the notion of commitment covering (when a commitment in one protocol means the same thing as a commitment in another protocol), and the definition of serial composition of commitments (the meaning of a chain of commitments). It also describes why commitment-based protocols are more flexible than traditional computer protocols using the idea of multiple states of completion.

**Organization**

Section 2 describes background material on protocols and commitments. Section 3 introduces our running examples. Section 4 describes our intuitions and framework for protocol refinement. Section 5 briefly describes our refinement checking tool. Section 6 demonstrates refinement on an example. Section 7 shows which properties are preserved by refinement. Section 8 describes our results. Section 9 evaluates our approach. Section 10
describes other works and future directions.

2. BACKGROUND
2.1 Protocols

We define a protocol as follows

**Definition 1 Protocol.** A protocol is

1. A set of guarded actions.
2. Multiple roles (agents) that send messages to each other.

We desire to minimally constrained agent implementations so a protocol can be implemented by the largest spectrum of agents. Any role can send any message at any time, constrained only by guard conditions. The syntax of a guarded message is

\[ \text{snd} \rightarrow \text{rcv} : [\text{guard}] \text{ message means } \{\text{basic actions}\} \]

where the sender role (\text{snd}) sends a message to a receiver (\text{rcv}). The message can be sent only if the guard is true. Looking ahead slightly, the meaning of the message is expressed by a set of well-defined, primitive basic actions.

In our model, propositions and commitment operations can occur exactly once. Propositions can be set true, but cannot be set false. This means we do not support looping protocols. While support for looping protocols is desirable, we simplify the initial problem and do not consider them here. Later, we hope to extend our work to cover looping protocols.

2.2 Commitments

Commitments are a formal and concise method of describing how agent roles commit to perform future actions ([Singh 1999] and [Yolum and Singh 2002]). We extend previous commitment definitions 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 the commitment operations **delegate** and **assign** with a single, new **transfer** operation.

**Definition 2.** A commitment is an object

\[ C_{\{\text{debtors}\},\{\text{creditors}\}}(\text{ant, csq}) \]  

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

1. **create**, performed only by debtors, creates a new commitment and makes it active.
2. When the antecedent becomes true, the commitment is implicitly converted to an unconditional commitment.
3. 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.
(4) **transfer.** performed by either debtors or creditors, ends the current commitment and marks it as transferred to another commitment and no longer active.

(5) **release.** performed only by creditors, releases the debtors from their commitment. The commitment is released and no longer active.

(6) **cancel.** performed only by debtors, cancels the debtors’ commitment. The commitment is violated and no longer active.

A commitment object is always in one of these states.

1. **inact:** the initial state. In this state, no commitment exists;
2. **cond:** after create with ant false, csq false, and no other operations;
3. **uncond:** after create with ant true, csq false, and no other operations;
4. **sat:** after create and csq true;
5. **xfer:** after create and transfer;
6. **rel:** after create and release; and
7. **can:** after create and cancel.

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 individually responsible for the entire commitment), or joint and several (the creditors hold one debtor fully responsible, who then pursues other debtors).

Below we will combine commitments and debtors together. We use several responsibility so each debtor’s responsibilities in the combination are limited to its input responsibilities.

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

Unconditional commitments must eventually resolve to state sat, xfer, rel, or can. While unusual, debtors may act before they are required to do so. It is possible for the consequent to become true before the antecedent. Debtors are discouraged from cancel, but in real life, circumstances may require it, with consequences handled outside the current mechanisms. 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. Fairness constraints typically constraint infinite loops, but we do not support looping protocols. Our fairness constraint ensures progress to a final state. This is a constraint on agent implementations, not a constraint on the protocol itself.

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

\[ C_{\text{Payer,Payee}}(\text{promise}, \text{pay}) \]

In **PayViaInt**, we represent the payer’s and middleman’s combined commitment as

\[ C_{\{\text{Payer,MM}\},\text{Payee}}(\text{promise}, \text{pay} P \land \text{pay} M) \]

Previous commitment descriptions allowed 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 both of these operations with a transfer operation which terminates the existing commitment plus a separate create of a new commitment.

\[
\text{delegate}(C_i, \text{debt'}) = \text{transfer}(C_i) \land \text{create}(C_{i'}')
\]

\[
\text{assign}(C_i, \text{cred'}) = \text{transfer}(C_i) \land \text{create}(C_{i'})
\]
where $C_i = C_{\text{debt,cred}}(\text{ant, csq})$, $C_i' = C_{\text{debt,cred}}'(\text{ant, csq})$ and $C_i'' = C_{\text{debt,cred}}''(\text{ant, csq})$. Since delegate and assign have essentially the same effect, transfer captures the essence of both and somewhat simplifies the definition of commitments.

### 2.3 Multiple Stages of Completion

Commitments evolve through four stages; proposition evolve through only two. The occurrence of a prop basic action (which means $\text{prop} = \text{true}$) 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.

\[ \text{inactive} \rightarrow \text{create} \rightarrow \text{ant} \rightarrow \text{uncond} \rightarrow \text{resolved} \]

Rather than waiting for final resolution, a protocol can make progress sooner if an action’s guard specifies one of the first three stages. Commitments increase protocol flexibility, because guards can specify intermediate, partially performed actions.

Example: Suppose an agent in OrderPayShip decides whether to pay based on the state of ship. Since proposition ship has only two stages, the agent’s decision can only be “all” (ship complete) or “nothing” (ship not complete). The “all” choice is represented by the guarded message

\[ [\text{ship}] \text{pay} \]

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.

\[ [\text{create}(C_{\text{Seller,Buyer}}(\text{pay, ship}))] \text{pay} \]

A protocol framework that includes commitments is inherently more flexible than traditional computer protocol frameworks. Where traditional, low-level protocols use an all-or-nothing basis with just two stages of completion, commitment-based frameworks can operate on four stages of completion. Basing a decision on a completed action provides essentially no risk to a creditor. But commitments facilitate more flexible enactments because creditors can also act based on debtor promises or commitments. While creditors assume more risk in doing so, they assume less risk than acting with no debtor commitments.

### 3. EXAMPLES

We introduce four running examples. Pay and PayViaInt are basic payment protocols while OrderPayShip and OrderPayViaIntShip are order protocols involving payments.

#### 3.1 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 sending the promise message. Then, at some later point, it sends a single payment message directly to the payee. Figure 1(a) shows the sequence diagram for the interaction, and protocol 1 on page 16 below shows the proposed protocol definition for Pay.

#### 3.2 PayViaInt

In protocol PayViaInt (pay via intermediary), if the payer chooses to do so, it commits to pay the payee by sending the promise message. It then sends a payment message indirectly
to the payee. The payer first pays the middleman, who in turn pays the payee. We assume
the middleman commits to perform payM message if payer performs payP message. The
sequence diagram in Figure 1(b) shows a typical interaction, but sequence diagrams doc-
ument only one message run. In this case, other valid runs exist; the middleman can be
generous and execute payM message before payP message. Protocol 2 on 17 below shows
the proposed protocol definition for PayViaInt.

3.3 OrderPayShip

In OrderPayShip in Figure 1(c) 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.

3.4 OrderPayViaIntShip

Protocol OrderPayViaIntShip in Figure 1(d) is similar to OrderPayShip except the payer
uses PayViaInt for payment. This introduces a new middleman role.

4. FRAMEWORK

This section builds our definition of protocol refinement through a series of intuitions.
A note on terminology must be made. We use the terms superprotocol and subprotocol
to refer to generalization and specialization of protocols. This is the same sense as in
object-oriented software engineering. We use the terms super-x and sub-x in a different
sense. super-x means “an element x in the superprotocol,” and sub-x means “an element
x in the subprotocol.” For example, a super-role is a role in the superprotocol and a sub-
commitment is a commitment in the subprotocol.
4.1 Intuition: Every Sub-run is a Super-run

The Liskov substitution principle [1994] states, 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. Subprotocols must satisfy every superprotocol property $\phi(x)$, and 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.

Comparing runs is the fundamental intuition in our approach, and is also the basic intuition in [Mallya and Singh 2007].

**Definition 3 Protocol Refinement.** Every sub-run must be a super-run.

Or, every run of the subprotocol must be a run of the superprotocol. The subprotocol must not allow a run prohibited by the superprotocol. With only this intuition, we compare runs of messages.

The comparison is performed by the model checker. If the subprotocol can send a sub-message then the superprotocol must also be able to send its corresponding super-message. When a sub-guard is true, super-guard must also be true. The model checker verifies this with the CTL formula

\[ AG(sub.g \rightarrow super.g) \] (3)

We represent the run comparison operation as a comparison function between the set of sub-runs with the set of super-runs. Figure 2(a) diagrams the refinement check.

**RunComp**(sub, super)

Consider the superprotocol Pay and the subprotocol OrderPayShip. We describe the mapping of super-messages (left-hand side) to sub-messages (right-hand side) with

- promiseMsg $\mapsto$ orderMsg
- payMsg $\mapsto$ payMsg

Pay has two super-runs. The leading number is an identifier for the run. No messages are sent in run one. In run two, the Payer promises and then pays. Runs are shown between ⟨ and ⟩ brackets.

1 : ⟨⟩
2 : ⟨promiseMsg, payMsg⟩

OrderPayShip has five sub-runs. The identifiers match those in Pay’s super-runs.

1 : ⟨⟩
1 : ⟨reqQuoteMsg⟩
1 : ⟨reqQuoteMsg, sendQuoteMsg⟩
2 : ⟨reqQuoteMsg, sendQuoteMsg, orderMsg, payMsg, shipMsg⟩
2 : ⟨reqQuoteMsg, sendQuoteMsg, orderMsg, shipMsg, payMsg⟩

Since reqQuoteMsg, sendQuoteMsg, and shipMsg are not part of the superprotocol Pay (they do not appear on the right-hand side of any mapping), they are ignored when comparing runs.
While run comparison is fundamental, it requires messages to have the same meaning in both super- and subprotocols. Different protocols can not organize or package functions in different messages that accomplish the same thing. To address this limitation, we introduce intuition 2: Decomposition.

4.2 Intuition: Decomposition

A message may have multiple functions. To better understand and characterize a message, we decompose each message into its meaning as an unordered set of primitive, well-defined basic actions. Each basic action has well-defined semantics useful for analyzing and understanding the meanings of message.

A basic action is either a Boolean proposition or a commitment operation. A propositional basic action sets the value of the proposition to true. The commitment basic actions are the operations create, transfer, release, and cancel.

Decomposition converts from a “protocol of guarded messages” to a “protocol of guarded basic actions.”

\[
\text{Means}([\text{guard}_i \text{msg}_i] \Rightarrow [\text{guard}_i \text{msg}_i \mapsto \bigwedge_j \text{act}_{i,j}] \bigwedge \text{act}_{i,j})
\]

where each message msg_i is decomposed into a conjunction of its basic actions act_{i,j}, and exp[x \mapsto y] is the result of replacing all occurrences of x with y in exp.

The result of decomposition is a set of guarded basic action expressions, but the model checker can not execute action expressions. We must convert any guarded expression into an equivalent set of guarded actions. In separation, we use the message’s guard as the guard for each of its basic actions. No basic action can fire early. We will relax this condition in the last step: Diffusion.

\[
\text{Sep}([\text{guard} \text{exp}] \Rightarrow \{[\text{guard} \text{act}] \forall \text{act} \in \text{exp}\}
\]
Separation may contain multiple statements with the same action. Collection creates a single statement for each action whose guard is the conjunction of the input guards. Figure 2(b) diagrams the modified refinement check.

\[
\text{Col}([\text{guard}_i \land \text{act}_i]) \Rightarrow \{ (\bigwedge_j \text{guard}_j, \land \text{act}_i \land \forall \text{act}_i = \text{act}_j) \}
\]

For example, the Pay protocol has the two guarded messages

\[
\begin{align*}
\text{Payer} \rightarrow \text{Payee} : [T] & \text{promiseMsg means } \{ \text{promise, create}(\text{C}_{\text{pay}}) \} \\
\text{Payer} \rightarrow \text{Payee} : [\text{promiseMsg}] & \text{payMsg means } \{ \text{pay} \}
\end{align*}
\]

which are decomposed into three guarded basic actions

\[
\begin{align*}
\text{Payer} \rightarrow \text{Payee} : [T] \text{ promise} \\
\text{Payer} \rightarrow \text{Payee} : [T] \text{ create}(\text{C}_{\text{pay}}) \\
\text{Payer} \rightarrow \text{Payee} : [\text{promise} \land \text{create}(\text{C}_{\text{pay}})] \text{ pay}
\end{align*}
\]

PromiseMsg generates guarded basic actions for both promise and create(\text{C}_{\text{pay}}). PayMsg generates one guarded basic action for pay.

Decomposition does not require that super-messages exactly match sub-messages, but it does require super-basic actions exactly match sub-basic actions. This is a serious limitation for protocols written at different levels of abstraction using different propositions and different commitments.

### 4.3 Intuition: Mapping

Since superprotocols represent higher-level abstractions than subprotocols, comparing protocols must address differences in abstraction level. There is often no one-to-one correspondence between super-elements and sub-elements. Protocol elements (roles, propositions and commitments) must be mapped between the two protocols to even compare them.

A particularly interesting type of difference in abstraction is the introduction of intermediaries in lower-levels of abstraction. Whereas two super-roles may communicate directly with each other using a single message in a high-level protocol, there is a natural tendency for message communication to pass through multiple, intermediary sub-roles as that protocol is refined to lower-levels of abstraction. Protocol refinement must allow super-elements to span intermediaries. One super-proposition could map to an expression of multiple sub-propositions, each controlled by different sub-roles (intermediaries). One super-commitment could be fulfilled through multiple sub-commitments and their intermediate sub-debtors.

We map every super-element to an expression of sub-elements, and figure 2(c) diagrams the refinement check.

\[
\text{Map}([\text{guard}] \exp, x_i \mapsto e_i) \Rightarrow [\text{guard}[x_i \mapsto e_i]] \exp[x_i \mapsto e_i]
\]

Subprotocols may contain sub-elements that do not correspond with any super-element. Each super-role is mapped to a set of sub-roles.

\[
\text{role}_{\text{super}} \mapsto \{ \text{role}_{\text{sub}} \}
\]
Example: The super-role Payer in Pay is mapped to the coalition of the two sub-roles Payer and MiddleMan in PayViaInt.

\[ \text{Payer} \mapsto \{\text{Payer, MM}\} \]

Each super-proposition is mapped to a Boolean expression of sub-propositions.

\[ \text{prop}_{\text{super}} \mapsto \text{BooleanExp(prop}_{\text{sub}}) \]

Example: the pay super-proposition in Pay becomes the two sub-propositions payP and payM in PayViaInt. These two basic actions are sent in different messages in PayViaInt, because they are performed by different roles.

\[ \text{pay} \mapsto \text{payP} \land \text{payM} \]

Each super-commitment is mapped to a commitment expression of sub-commitments. We introduce commitment expressions in two steps: commitment covering and serial composition.

First, we map each super-commitment to a single sub-commitment only if the sub-commitment covers the super-commitment, where covers is written as \( \leq \). The super- and sub-commitments can be different.

\[ \text{C}_{\text{super}} \mapsto \text{C}_{\text{sub}} \quad \text{if} \quad \text{C}_{\text{super}} \leq \text{C}_{\text{sub}} \]

In the second step, we combine chains of multiple sub-commitments into a single, implied, new sub-commitment.

\[ \text{C}_{\text{super}} \mapsto \text{C}_{\text{sub}_1} \oplus \text{C}_{\text{sub}_2} \quad \text{if} \quad \text{C}_{\text{super}} \leq \text{C}_{\text{sub}_1} \oplus \text{C}_{\text{sub}_2} \]

Every super-commitment must be covered by some expression of sub-commitments. This guarantees that the super-commitment is discharged whenever the sub-commitment is discharged.

To visualize the following explanations, we diagram a commitment as a labeled arrow.

The name of the commitment is written in the top-center of the arrow. Debtors and creditors are on the top-ends of the arrow. The antecedent and consequent are on the bottom-ends of the arrow. When multiple terms appear in a position, they are implicitly combined using the operator in parentheses. In general, antecedents and consequents can also include logical-or (\( \lor \)) operators, but we do not need that for the examples in this paper.

4.4 Commitment Covering

We must compare commitments. In particular, we compare a (stronger) sub-commitment with a (weaker) super-commitment.

**Definition 4 Commitment Covering.** A stronger commitment \( C_S \) covers (is stronger
than) a weaker commitment \( C_W \), written \( C_W \leq C_S \), iff

\[
\begin{align*}
\text{map}(d) \cap C_S\text{.debt} &\neq \emptyset \quad \forall d \in C_W\text{.debt} \\
\text{map}(c) \cap C_S\text{.cred} &\neq \emptyset \quad \forall c \in C_W\text{.cred} \\
\text{map}(C_W\text{.ant}) &\vdash C_S\text{.ant} \\
\text{map}(C_W\text{.ant}) \land C_S\text{.csq} &\vdash \text{map}(C_W\text{.csq})
\end{align*}
\]

where \( \cap \) is set intersection, \( \emptyset \) is the empty set, \( \vdash \) is derives, and \( \text{map}(x) \) is the mapping of super-element \( x \) to an expression of sub-elements.

Every super-debtor is partially (severally) responsible for discharging the super-commitment. Each super-role is mapped to (implemented by) a set of sub-roles. So each super-debtor’s responsibilities must be covered by one or more sub-debtors. Equation (4) captures the requirement that at least one of the super-debtor’s sub-roles must be a debtor of the sub-commitment. Similarly, each super-creditor is a partial beneficiary of the super-commitment. Equation (5) captures the requirement that at least one of the super-creditor’s sub-roles must be a beneficiary of the sub-commitment.

Whenever the super-commitment becomes unconditionally committed (\( C_W\text{.ant} = \text{true} \)), equation (6) ensures the sub-commitment also becomes unconditionally committed. Whenever the sub-commitment is discharged (\( C_S\text{.csq} = \text{true} \)), equation (7) ensures the super-commitment is also discharged. Equations (6) and (7) are based on the following diagram.

\[
\begin{array}{c}
C_W\text{.ant} \quad \text{map}(C_W\text{.ant}) \quad C_W\text{.csq} \\
\downarrow \quad \downarrow \quad \downarrow \\
C_S\text{.ant} \quad C_S\text{.csq}
\end{array}
\]

If \( C_S \) covers \( C_W \), then both side implications are true by equations (6) and (7). If \( C_S \) is discharged, then the bottom implication is true. If those three implications are true, then \( C_W \), the top implication, will necessarily be discharged.

In software engineering, the Liskov Substitution Principle (LSP) and behavioral subtyping are common approaches to characterizing the behavioral relationships between pieces of code [Liskov and Wing 1994]. Many different proposals have been made for checking whether a subclass provides the same behavior as a superclass [Chen and Cheung 2000] and [Toth 2005]. These papers describe a spectrum of check conditions. Two of interest here are the plug-in condition and relaxed plug-in (also called satisfies) condition. Equations (6) and (7) implement the relaxed plug-in condition. The plug-in condition is similar except the \( \text{map}(C_W\text{.ant}) \) term would not be included in (7).

These papers show the plug-in condition implies the relaxed plug-in condition. In earlier versions, we used the plug-in condition between super-commitments and sub-commitments, but now use the relaxed plug-in condition since it is weaker and allows sub-commitments to cover more super-commitments.

We note those papers use method pre- and post-conditions to compare two complete classes, where we use commitment antecedents (pre-conditions) and consequents (post-conditions) only on commitments.
Example: The $C_{pay}$ sub-commitment in $OrderPayShip$ covers the $C_{pay}$ super-commitment in $Pay$. The mapping from $Pay$ to $OrderPayShip$ is

$$
\begin{align*}
\text{Payer} & \mapsto \text{Buyer} \\
\text{Payee} & \mapsto \text{Seller} \\
\text{promise} & \mapsto \text{order} \\
\text{pay} & \mapsto \text{pay} \\
C_{pay} & \mapsto C_{pay}
\end{align*}
$$

$$
\begin{array}{c|c|c}
\text{Payer} & C_{pay} & \text{Payee} \\
\hline
\text{promise} & C_{pay} & \text{order} \\
\text{pay} & C_{pay} & \text{pay}
\end{array}
$$

$$
\text{map(Payer)} \cap \{\text{Buyer}\} = \{\text{Buyer}\} \cap \{\text{Buyer}\} \neq \emptyset. \text{ verifies (4).}
$$

$$
\text{map(Payee)} \cap \{\text{Seller}\} = \{\text{Seller}\} \cap \{\text{Seller}\} \neq \emptyset. \text{ verifies (5).}
$$

$$
\text{map(promise)} = \text{order} \vdash \text{order}. \text{ verifies (6).}
$$

$$
\text{map(promise)} \land \text{pay} = \text{order} \land \text{pay} \vdash \text{pay} \text{ verifies (7).}
$$

\text{THEOREM 5.} Commitment strength is reflexive.

\text{PROOF.} Consider a commitment C. The map for the reflexive case is the identity function. Equation (4) verifies for each $d \in C.debt$, that $d \cap C.debt \neq \emptyset$. A parallel argument for creditors verifies (5). Equation (6) becomes $C.ant \vdash C.ant$. Equation (7) becomes $C.ant \land C_{S.csq} \vdash C.csq$. $\square$

\text{THEOREM 6.} Commitment strength is transitive.

\text{PROOF.} Given $C_A \leq C_B$, $C_B \leq C_C$ and the corresponding versions of equations (4,7) show $C_A \leq C_C$ with $\text{map}_{AC}(x) = \text{map}_{BC}(\text{map}_{AB}(x))$.

By (4), every debtor of $C_A$ passes its responsibility to at least one debtor of $C_B$, and every debtor of $C_B$ passes its responsibility to at least one debtor of $C_C$. Therefore, every debtor of $C_A$ passes its responsibility to at least one debtor in $C_C$, and (4) is transitive. A parallel proof for creditors demonstrates (5) is transitive.

Function composition shows (4) is transitive.

By the deduction theorem and exportation, (7) can be written $\text{map}(C_{W.ant}) \rightarrow (C_{S.csq} \rightarrow \text{map}(C_{W.csq}))$. By (6), $C_{A.ant}$ implies all other antecedents, and also implies $C_{C.csq} \rightarrow \text{map}_{BC}(C_{B.csq}) \rightarrow \text{map}_{BC}(\text{map}_{AB}(C_{A.csq}))$. Therefore $\text{map}_{AC}(C_{A.csq}) \rightarrow (C_{C.csq} \rightarrow \text{map}_{AC}(C_{A.csq}))$, and (7) is transitive. $\square$

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

Example: $C(\text{order, ship}) \leq C(\text{order, ship} \land \text{expressDelivery})$ since the stronger commitment commits to the additional consequent $\text{expressDelivery}$.

The single super-commitment in $Pay$, from the payer to the payee, is not covered by any single commitment in $PayViaInt$. But $PayViaInt$ has two sub-commitments that form a chain passing through the middleman. This commitment chain does commit the payer to pay the payee. Serial composition formalizes this idea.
4.5 Serial Composition

We need a way for commitments to span intermediaries like the middleman in PayViaInt. Previous commitment formulations [Singh 2008] included the idea of commitment chaining, but we formalize this in a new way as serial composition of commitments. Serial composition computes an implied, result commitment from a chain of commitments.

**Definition 7 Serial Composition.** Two commitments \( C_A \) and \( C_B \) are combined into a resultant commitment \( C_\oplus = C_A \oplus C_B \) iff the operation is well-defined

\[
C_A.\text{ant} \land C_A.\text{csq} \vdash C_B.\text{ant}
\]

Then \( C_\oplus \) is defined as

\[
\begin{align*}
C_\oplus.\text{debt} & := C_A.\text{debt} \cup C_B.\text{debt} \\
C_\oplus.\text{cred} & := C_A.\text{cred} \cup C_B.\text{cred} \\
C_\oplus.\text{ant} & := C_A.\text{ant} \\
C_\oplus.\text{csq} & := C_A.\text{csq} \land C_B.\text{ant} \land C_B.\text{csq}
\end{align*}
\]

\[
\begin{align*}
\text{create}(C_\oplus) & := \text{create}(C_A) \land \text{create}(C_B) \\
\text{transfer}(C_\oplus) & := \text{transfer}(C_A) \lor \text{transfer}(C_B) \\
\text{release}(C_\oplus) & := \text{release}(C_A) \lor \text{release}(C_B) \\
\text{cancel}(C_\oplus) & := \text{cancel}(C_A) \lor \text{cancel}(C_B)
\end{align*}
\]

where \( := \) defines the left-hand side as the right-hand side.

\( C_\oplus \) is a new commitment object whose properties are defined in terms of the properties of \( C_A \) and \( C_B \). \( C_\oplus \) does not provide any information beyond that given in \( C_A \) and \( C_B \), but expresses it in the form of a new commitment.

The well-defined condition (8) generalizes the chain rule in [Singh 2008]. \( C_\oplus \) states the debtor group is committed to the creditor group to bring about consequent \( C_A.\text{csq} \land C_B.\text{ant} \land C_B.\text{csq} \) when antecedent \( C_A.\text{ant} \) is true. Debtors are severally responsible for \( C_\oplus \). Debtors do not become more responsible than in their input commitments. Every debtor in \( C_A.\text{debt} \) is partially responsible for discharging \( C_A \), and thus is partially responsible for discharging \( C_\oplus \). Also, every debtor in \( C_B.\text{debt} \) is partially responsible for \( C_\oplus \). (9) is the union of all partially responsible parties. A parallel argument holds for the creditors who are beneficiaries (10).

A chain of the two sub-commitments in PayViaInt cover Pay’s super-commitment.

\[
C_{\text{pay}} \xrightarrow{} C_{\text{payP}} \oplus C_{\text{payM}} \quad \text{if } C_{\text{pay}} \leq C_{\text{payP}} \oplus C_{\text{payM}}
\]

The bottom-left arrow states, if promise becomes true, then Payer commits to making payP true. The bottom right arrow states, if payP becomes true, than MiddleMan commits to making payM true. In the serial composition in the top row, if promise becomes true, Payer and MiddleMan (severally) commit to making both payP and payM true. The
well-defined condition ensures the discharge of the bottom-left commitment implies the antecedent of the bottom-right commitment which makes it unconditionally committed. As Payee is the creditor of the bottom left commitment, and Payer is the creditor of the bottom right commitment, both are creditors of the serial composition.

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

$$C_{12...n} = ((C_1 \oplus C_2) \oplus \ldots) \oplus C_n$$

**Theorem 8.** Serial composition is not commutative and not associative.

Serial composition creates commitments that are at least as strong, and typically stronger than their inputs. $C_A \oplus C_B$ is typically stronger than $C_A$ alone because, even though both have the same antecedent ($C_{A,ant}$), in general, $C_A \oplus C_B$ has a stronger consequent ($C_{B,csq}$ vs. $C_{A,csq} \land C_{B,ant} \land C_{B,csq}$). However, the next theorem shows this is not always the case. Operator $\oplus$ obeys the following idempotent-like property.

**Theorem 9.** Extending a serial composition with a commitment already part of the chain, does not create a stronger commitment.

$$C_1 \oplus \ldots \oplus C_k \oplus \ldots \oplus C_n \oplus C_k$$

(17)

$$= C_1 \oplus \ldots \oplus C_k \oplus \ldots \oplus C_n$$

(18)

**Proof.** If $C_1 \oplus \ldots \oplus C_n$ is well-defined, then $(C_1 \oplus \ldots \oplus C_n) \oplus C_k$ is well-defined because $C_{k,ant}$ is already part of the left-hand side of equation (8). By simple inspection, conditions (9-16) are the same for both sides. Expression (17) is identical to, not stronger than, expression (18).

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 $n!$.

While it would be desirable to automatically compute the mapping between superprotocols and subprotocols, there can be multiple, different mappings between some protocol pairs. In the mapping between Pay and PayViaInt, one mapping groups the Middleman into a coalition with the Payer. A different mapping groups the Middleman into a coalition with the Payee. Both mappings are valid.

Mapping each super-element to an expression of sub-elements gives us nearly everything we want, but it still violates one further intuition. OrderPayViaIntShip still does not refine OrderPayShip. The problem is the order of the basic actions is still too rigid.

Example: The top line shows the first portion of a run from OrderPayShip. The bottom line shows the first portion of a run from OrderPayViaIntShip.

$$\langle rQ \; sQ \; create(C_{ship}) \; create(C_{payM}) \; ord \; create(C_{payP}) \; \ldots \rangle$$

$$\langle create(C_{payM}) \; rQ \; sQ \; create(C_{ship}) \; ord \; create(C_{payP}) \; \ldots \rangle$$

where reqQuote, sendQuote and order have been abbreviated to rQ, sQ, and ord respectively.

The basic actions in both runs align except create($C_{payM}$) which occurs at different times. The runs are not equal because of this misalignment, so OrderPayViaIntShip fails to refine OrderPayShip. To fix this, we loosen the ordering of basic actions using 4:Diffusion.
4.6 Intuition: Diffusion

If super-proposition $p$ maps to a conjunction of sub-basic-actions $p \rightarrow q_1 \land q_2 \land \cdots \land q_n$, then $p$ becomes true when the last of the $q_i$s becomes true. If super-proposition $p$ maps to a disjunction of sub-basic-actions $p \rightarrow q_1 \lor q_2 \lor \cdots \lor q_n$, then $p$ becomes true when the first of the $q_i$s becomes true.

When mapping super-propositions to sub-propositions, we only need to constrain the occurrences of the $q_i$s so $p$ becomes true at exactly the same point in both the super- and sub-runs. For conjunction, all the $q_i$s, except the last, can move to any earlier point in time. For disjunction, all the $q_i$s, except the first, can move to any later point in time. The $q_i$s can even move all the way to the run’s beginning (conjunction) or end (disjunction).

Diffusion transforms a guarded basic action expression, to a set of guarded basic actions by the following recursive transformations.

$$\text{Dif}([\text{guard}] \bigvee_i \text{exp}_i) \Rightarrow \{[\text{guard}] \text{exp}_i\}$$

$$\text{Dif}([\text{guard}] \bigwedge_i \text{exp}_i) \Rightarrow \{[\text{guard} \lor \bigvee_{j \neq i} \neg \text{exp}_j] \text{exp}_i\}$$

where guard and exp$_i$ are Boolean expressions of basic actions. Figure 2(d) diagrams the final refinement check.

Example: Consider the guarded basic action expression for payP and payM.

$$[\text{guard}] \text{payP} \land \text{payM}$$

Without diffusion (only decomposition), this is transformed to

$$[\text{guard}] \text{payP}$$

$$[\text{guard}] \text{payM}$$

Neither payP and payM can occur before the guard becomes true. With diffusion, this is transformed to

$$[\text{guard} \lor \neg \text{payM}] \text{payP}$$

$$[\text{guard} \lor \neg \text{payP}] \text{payM}$$

Both basic actions can still fire after the guard becomes true. Additionally, the first basic action to fire, can fire at anytime. That is, one basic action can slide forward in time.

4.7 Refinement Definition

This is the last of our intuitions and completes our definition of refinement. Figure 2(d) illustrates the steps required to demonstrate protocol refinement, and this is the functional form.

$$\text{RunComp}([\text{Col}([\text{Dif}([\text{Means}\{\text{sub}\}])]), \text{Col}([\text{Dif}([\text{Map}([\text{Means}\{\text{super}\}])])]))$$

5. TOOLING

We wrote a tool to check protocol refinement. Figure 3 shows the process flow. The preprocessor reads the superprotocol, subprotocol and mapping specification from files and constructs an ISPL file for input to MCMAS. The ISPL model is processed by the
Fig. 3. Process flow

MCMAS model checker which checks the generated formulae. If all formulae are true, the subprotocol refines the superprotocol.

The preprocessor generates the following checking formulae.

\[ \text{AG}(\text{sub.guard} \rightarrow \text{super.guard}) \] (19)
\[ \text{AG}(\text{C}_{\text{super.state}} = \text{uncond} \rightarrow \text{AF}(\text{C}_{\text{super.state}} \neq \text{uncond})) \] (20)

Equation (19) 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 fail to resolve, equation (20) ensures unconditionally committed super-commitments must resolve.

6. \textit{PAYVIAINT} REFINES \textit{PAY}

6.1 Pay

**Protocol 1 Pay Protocol**

1. protocol Pay {
2. role Payer, Payee;
3. prop promise; pay;
4. commitment
5. \text{scpay} = C(Payer, Payee, promise, pay);
6. action
7. Payer → Payee : promiseMsg means \{promise, create(scpay)\};
8. Payer → Payee : \{promise ∧ create(scpay)\} payMsg means \{pay\};
9. }

Space precludes a detailed description of our proposed protocol specification language, so we simply state the protocol specification in Protocol 1 with a few notes. Lines (2)–(5) declare roles Payer and Payee, propositions promise and pay, and the commitment. Both promiseMsg and payMsg messages are sent by Payer to Payee. The unordered set of basic actions are listed between \{ and \}. Line (8) includes the guard for payMsg as promise and the creation of the commitment.

6.2 PayViaInt

In Protocol 2, Middleman initially commits to Payer to pass along any payment it receives (10). Payer cannot pay Middleman without this commitment (12). Since payMsg has no
Protocol 2 PayViaInt Protocol

1: protocol PayViaInt {
2:   role Payer, MM, Payee;
3:   prop promise;
4:     payP;  \(//\) payment from Payer
5:     payM;  \(//\) payment from MM
6:   commitment
7:     scpayP = C(Payer, Payee, promise, payP);
8:     scpayM = C(MM, Payer, payP, payM);
9:   action
10:  MM : init means \{create(scpayM)\};
11:  Payer \rightarrow Payee : promiseMsg means \{promise, create(scpayP)\};
12:  Payer \rightarrow MM : [promise \& create(scpayP) \& create(scpayM)]
13:     payPM msg means \{payP\};
14:  MM \rightarrow Payee : payMM msg means \{payM\};
15: }

guard (14). Middleman is free to pay early. That decision is part of Middleman’s implementation.

6.3 Mapping

Mapping 3 PayViaInt Protocol

1: map M1: Pay \mapsto PayViaInt {
2:   role
3:     Payer \mapsto \{Payer, MM\};
4:     Payee \mapsto \{Payee\};
5:   prop
6:     promise \mapsto promise;
7:     pay \mapsto payP \& payM;
8:   commitment
9:     C_{pay} \mapsto C_{payP} \oplus C_{payM};
10: }

Mapping maps each super-element in Pay to an expression of sub-elements in PayViaInt. Line (3) maps the super-role Payer to the group of sub-roles Payer and MM. Line (7) maps the super-proposition pay to the conjunction of two sub-propositions: payP and payM. Line (9) maps the super-commitment \(C_{pay}\) to the serial composition of sub-commitments \(C_{payP}\) and \(C_{payM}\).

7. PRESERVED PROPERTIES

We adapt the definition of simulation from [Clarke et al. 1999, page 176] for a subprotocol \(Q\) and a superprotocol \(P\). Protocols contain a set of basic actions \(BA\). Each action \(a\) is described by a state transition relation \(R_a(s, s')\). Performing action \(a\) in state \(s\) results in state \(s'\).
**Definition 10** Simulation. Given two structures $Q$ and $P$ with $BA_Q \supseteq BA_P$, a relation $H \subseteq Q \times P$ is a simulation relation between $Q$ and $P$ if and only if for all $q$ and $p$, if $H(q, p)$ then the following conditions hold.

1. $L_Q(q) \cap BA_P = L_P(p)$
2. For every state $q'$ such that $R_{Q,a}(q, q')$, there is a state $p'$ with the property that $R_{P,a}(p, p')$ and $H(q', p')$.

We say $P$ simulates $Q$ (denoted by $Q \preceq P$) if there exists a simulation relation $H$ such that for every initial state $q_0$ in $Q$, there is an initial state $p_0$ in $P$ for which $H(q_0, p_0)$.

**Theorem 11.** If run comparison is true for protocols $P$ and $Q$, then $P$ simulates $Q$.

We do not compare the input superprotocol directly with the input subprotocol (based on guarded messages). Rather, $P$ is the result after decomposing, mapping, and diffusing the superprotocol, and $Q$ is the result after decomposing and diffusing the subprotocol. $P$ and $Q$ are described with the same sub-basic action vocabularies. $P$ uses only sub-basic actions in $BA_P$. $Q$ uses all basic actions in $BA_Q$ with $BA_Q \supseteq BA_P$.

Define $H(q, p)$ iff $L_Q(q) \cap BA_P = L_P(p)$ (both $q$ and $p$ agree on the values of all propositions in $BA_P$). Each sub-state $q$ can match only one super-state $p$. Each super-state $p$ can match multiple sub-states. This satisfies condition (1) of the definition.

**Proof.** Run comparison verifies $L_Q(q) \cap BA_P = L_P(p)$

Consider every action $a \in BA_Q$. If $a$ is also in $BA_P$, then $L_Q(q') \cap BA_P = (L_Q(q) \cup \{a\}) \cap BA_P = (L_Q(q) \cap BA_P) \cup \{a\} = L_P(p) \cup \{a\} = L_P(p')$. And $H(q', p')$.

If $a$ is not in $BA_P$, then $L_Q(q') \cap BA_P = (L_Q(q) \cup \{a\}) \cap BA_P = L_Q(q) \cap BA_P = L_P(p)$. And $H(q', p)$.

Condition (2) of the definition holds, and $H$ is a simulation relation. Our procedure initializes every protocol in the state where no basic actions are true. $L_Q(q_0) \cap BA_P = \{\} = L_P(p_0)$, so $H(q_0, p_0)$. Therefore, $P$ simulates $Q$. □

What properties are preserved by this definition of protocol refinement? Consider super-formula $E f$. If $E f$ is a valid super-formula, then $f$ is valid for some super-run $\pi$. However, sub-runs are a subset of the super-runs, so $\pi$ may not be a sub-run. Therefore, $f$ is not necessarily valid for any sub-run, and $E f$ is not necessarily a valid sub-formula. So the $E$ quantifier cannot be preserved.

Consider super-formula $X f$. We simulate the subprotocol and ensure all of its transitions are allowed by the superprotocol. Any sub-runs containing an action $a$ in $BA_Q$, but not in $BA_P$, would invalidate $X$ operators in sub-formulae. So the $X$ operator can not be preserved.

Let ACTL-X denote the fragment of CTL logic that includes the $A$ path quantifier (but not $E$), and the temporal operators $F$, $G$, and $U$ (but not $X$). The ACTL-X language is

$$\mathcal{L} = p|\mathcal{L} \land \mathcal{L} \lor \mathcal{L} | AF \mathcal{L} | AG \mathcal{L} | A(\mathcal{L} U \mathcal{L})$$

where $p$ is a proposition. We do not include a negation operator applied only to literals. Protocol designers can define “negative propositions” (e.g. define an additional proposition close instead of ¬open).

Clarke proves simulation preserves ACTL formulas. So our protocol refinement definition preserves all valid ACTL-X super-formulae as valid sub-formulae.
Fig. 4. Demonstrated Refinements where there is an arrow from the superprotocol to the superprotocol

<table>
<thead>
<tr>
<th>Superprotocol</th>
<th>Subprotocol</th>
<th>Map</th>
<th>@ cover</th>
<th>formulae</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pay</td>
<td>PayViaSpouse</td>
<td>M1</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Pay</td>
<td>PayViaInt</td>
<td>M1</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Pay</td>
<td>PayViaInt</td>
<td>M2</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Pay</td>
<td>PayViaCheck</td>
<td>M1</td>
<td>2</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>Pay</td>
<td>PayViaCredit</td>
<td>M1</td>
<td>2</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>Pay</td>
<td>OrderPayShip</td>
<td>M1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Pay</td>
<td>OrderPayViaIntShip</td>
<td>M1</td>
<td>2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>PayViaInt</td>
<td>PayViaCheck</td>
<td>M1</td>
<td>3</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>PayViaInt</td>
<td>PayViaCredit</td>
<td>M1</td>
<td>3</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>PayViaInt</td>
<td>OrderPayViaIntShip</td>
<td>M1</td>
<td>0</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>OrderPayShip</td>
<td>OrderPayViaIntShip</td>
<td>M1</td>
<td>1</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>OrderPayShip</td>
<td>NetBill2</td>
<td>M1</td>
<td>0</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>OrderPayViaIntShip</td>
<td>NetBill3</td>
<td>M1</td>
<td>0</td>
<td>3</td>
<td>12</td>
</tr>
</tbody>
</table>

Fig. 5. Information about each demonstrated Refinement. ⊕ is the number of serial compositions. covers is the number of commitment covering checks. formulae is the number of CTL formulae verified by the model checker. time is the elapsed time in seconds for the complete refinement verification.

8. RESULTS

With our tool, we demonstrated the refinements shown in Figures 4 and 5. OrderPayShip is identical to the first NetBill scenario in Mallya and Singh. NetBill2 and NetBill3 are scenarios 2 and 3 in the same paper.

9. EVALUATION

Given a superprotocol, subprotocol, and mapping, if the procedure returns positive result, then the subprotocol does refine the superprotocol. However, the procedure returns a negative result, this does not prove the subprotocol can never refine the superprotocol. Refinement could be possible using a different mapping function.

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 currently use deduction to check the covering relations and serial composition well-defined condition. This can be overly restrictive, since we only need to ensure those conditions hold for reachable states, not all possible states. While it would be possible to explicitly build super- and sub-transition systems, and test only reachable states, this could easily lead to state space explosion. Modern model checkers have invested considerable effort on theory and implementation to minimize state space explosion. Using a model
checker leverages that investment and eliminates some false negatives where checks fail on unreachable states.

Model checkers have also been extended to handle epistemic and strategic modal operators. We have experimented with including such concepts into our definitions, but more work remains ([Alur et al. 2002], [Fagin et al. 1995]). Building on top of a model checker that already handles those concepts, like MCMAS, will simplify our future extensions.

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

10. DISCUSSION

A protocol framework that includes commitments is inherently more flexible than traditional computer protocols. Without commitments, agents are limited to making decisions with just two stages of proposition completion. With commitments, agents can base decisions on a commitment’s four stages of completion.

Our definition of protocol refinement does not mean agents that can participate in a superprotocol can necessarily participate unchanged in a subprotocol. Agents work with guarded messages, but our definition for run comparison is based on guarded 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 needs to change to handle payments via check or credit card.

10.1 Relevant Literature

In personal communication, Chopra suggested the approach of describing complex messages as a set of more primitive actions. [Chopra 2008] is a good description of this approach. We used this idea in our decomposition step.

[Mallya and Singh 2007] propose a definition of protocol refinement (there called subsumption) that compares the order of state pairs in runs of states. For every pair of states in the superprotocol, there must be some matching pair of states in the subprotocol with the same order. However, this definition can lead to false positives. False positives can occur when multiple state pairs in the superprotocol each match the same, single state pair in the subprotocol. False positives can also occur when one super-state matches two different sub-states, as demonstrated by the example where all state pairs in the super-run ⟨1, 2, 3⟩ have matching state pairs in the sub-run ⟨2, 1, 3, 2⟩ even though the two runs are very different. Because our approach is related to simulation, it compares runs element by element, preventing these kinds of failure.

We believe we are the first to propose a protocol inheritance mechanism that also includes the ability to map super-elements to expressions of sub-elements. We support mapping super-propositions to Boolean expressions of sub-propositions as well as mapping super-commitments to serial compositions of multiple sub-commitments.

[Singh 2008] states rules similar to those proposed here for commitment strength. Our equation 8 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. The concrete commitment created by serial composition provides a halfway point in commitment reasoning chain, and will likely make the process more understandable for less sophisticated users.

We demonstrated we preserve all ACTL-X super-formulae, but we do not propose specific properties. Others ([Yolum 2007] and [El-Menshawy et al. 2010]) have proposed var-
ious desirable properties for commitment protocols. Some of their properties (fairness, safety, and some of the liveness properties) are in ACTL-X, and are preserved by our protocol refinement. Other properties (other liveness and the reachability properties) use the existential operator \( E \) and are not necessarily preserved. For example, that property that some value will eventually become true would not be preserved.

Constructing a mapping function from a super-commitment to a sub-commitment can be a challenging task. Advice to guide protocol designers, in the form of a basic mapping methodology, would be a valuable addition to this work. Winikoff [2006] and [2007] proposed a methodology for the related task of designing commitment-based protocols. We believe some of their ideas could be valuably adapted into a future commitment mapping methodology. They begin with easily understood interaction diagrams, but also take specific steps to generalize the protocol and expand its set of runs.

This paper used guards to constrain actions, but other mechanisms might be used. Baldoni et al. proposed constraints based on regulative specifications [2010]. Regulative specifications constrain the execution flow based on state values, not on actions. Gabby proposed using past-temporal expressions for controlling when actions can occur and future-temporal expressions for controlling which actions must occur in the future [1987]. Past-temporal expressions are more expressive than guards.

10.2 Research Directions
A proper definition of protocol refinement is an important foundation for my future research directions.

(1) This paper uses guards to constrain actions. There are no known problems with this approach, but alternative mechanisms have also been proposed (for example, regulative specifications and past-temporal expressions). What are the strengths and weaknesses of these various approaches for protocol design? How do these different approaches effect protocol design?

(2) Are there best practices and methodologies for designing good protocols? For example, many commitment examples use a reciprocal commitment style: “If you do A, I’ll do B” and “If you do B, I’ll do A.” But in multiparty protocols, a contract commitment style is simpler and more natural: “if we all agree, I’ll fulfill my part.” When are early vs. late stages of commitment completion most appropriate for guards? Are guards based on early stages of completion less risky when the creditor controls the commitment’s antecedent? What properties make one protocol design approach better, or simpler, than another?

(3) I have investigated describing protocol properties using strategic ([Alur et al. 2002]) and epistemic ([Fagin et al. 1995]) operators. Strategic concepts allow reasoning about an agent’s ability to fulfill its commitments. Epistemic concepts allow reasoning about an agents’ knowledge of the state of its commitments in a distributed environment. For example, an agent cannot be expected to fulfill a commitment if the protocol does not have messages to tell the agent the antecedent is true. But these concepts should only be pursued to the extent they facilitate expressing desirable protocol properties.

REFERENCES


