Formalizing and Verifying Protocol Refinements

Scott N. Gerard and Munindar P. Singh
North Carolina State University

A (business) protocol describes, in high-level terms, a pattern of communication between two or more participants, specifically via the creation and manipulation of the commitments between them. In this manner, a protocol offers both flexibility and rigor: a participant may communicate in any way it chooses as long as it discharges all of its activated commitments. Protocols thus promise benefits in engineering cross-organizational business processes. However, software engineering using protocols presupposes a formalization of protocols and a notion of the *refinement* of one protocol by another. Refinement for protocols is both intuitively obvious (e.g., *PayViaCheck* is clearly a kind of *Pay*) and technically nontrivial (e.g., compared to *Pay*, *PayViaCheck* involves different participants exchanging different messages). This paper formalizes protocols and their refinement. It develops Proton, an analysis tool for protocol specifications that overlays a model checker to compute whether one protocol refines another with respect to a stated mapping. Proton and its underlying theory are evaluated by formalizing several protocols from the literature and verifying all and only the expected refinements.

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

General Terms: Algorithms, Languages, Theory, Verification

Additional Key Words and Phrases: Commitments, Agent communication, Verification of multiagent systems

1. INTRODUCTION

We focus our attention on business service engagements as realized over the Internet. In current practice, such an engagement is defined rigidly and purely in operational terms. Consequently, the software components of the business partners involved are tightly coupled with each other, and depend closely on the engagement specification. Even small changes in one partner's components must be propagated to others, even when such changes are inconsequential to the business being conducted. Conversely, if the model leaves the engagements unstructured, humans must carry out the necessary interactions manually, with concomitant loss in productivity. We motivate protocols as providing a happy middle ground between rigid automation and flexible manual execution.

Specifically, in contrast with traditional approaches, we model each partner as an autonomous *agent*. The agents participate in a *(business) protocol* to realize a service engagement. A protocol describes a pattern of communication between agents. Based on the foregoing, we formulate the following key requirements on a suitable formalization of protocols. First, a protocol is *public*, meaning that it pertains to the messages sent and received by participating agents, not how those agents are implemented. Thus, the semantics of a

Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206, USA. {sngerard,singh}@ncsu.edu. Gerard is also a Senior Technical Staff Member with IBM.

protocol should depend solely on the communications of the agents enacting it, not on their internal policies. Second, the semantics should capture the business meanings of the messages, thereby avoiding operational constraints, and thus enabling the agents to deal better with exceptions and opportunities [Yolum and Singh 2002]. Third, the semantics should be modular: an agent who enacts a protocol correctly may concurrently enact additional protocols. Fourth, designing engagements using protocols presupposes that we support engineering methodologies such as those based on stepwise refinement. We address the above criteria for protocols with an emphasis on their refinement.

We understand a protocol semantically in terms of exactly the set of runs (i.e., computations) that it allows. Following Mallya and Singh [2007], we posit that a putative subprotocol refines a putative superprotocol if and only if each run allowed by the subprotocol is also allowed by the superprotocol. In general, a subprotocol would include additional roles and actions: in determining refinement, we disregard those that do not feature in the superprotocol. Doing so facilitates modularity, enhanceability, and reuse of protocols.

Consider a simple protocol *Pay* consisting of two actions where a payer first commits to paying a payee, and next pays. Now consider a protocol *PayViaMM* where the payer first pays a middleman, who in turn pays the payee. Both *Pay* and *PayViaMM* send a payment from the payer to the payee. Even though *PayViaMM* involves an additional role (middleman) and *PayViaMM* uses different messages (two payment messages instead of one), we expect *PayViaMM* refines *Pay*, because *PayViaMM* makes a payment as *Pay* specifies. Similarly, we expect *PayViaCheck* and *PayViaCredit* also refine *Pay*. We imagine a service engagement design exercise where protocol designers begin by identifying the need for payment as *Pay*, then refine it to *PayViaMM*, and then to *PayViaCheck*. The designers may build or find an existing repository of protocols (analogous to taxonomies of business processes [Malone et al. 2003]). The question we address is how can protocols in such a repository be expressed so that their refinements can be rigorously verified.

1.1 Proton: Approach and Contributions

We formulate refinement in technical terms and show how to compute it via a tool called *Proton*. We specify a protocol declaratively in terms of (i) its roles, (ii) the guarded messages the roles exchange, and (iii) the meaning of each message as a set of actions on the public state of the roles, sometimes termed the *social state* [Baldoni et al. 2010]. Commitments between roles are central to our approach [Singh 1999]. Section 2.4 provides additional details. For now, suffice it to say that a state of a protocol is determined by what atomic propositions hold therein (some propositions specify the states of commitments).

We define the semantics of a protocol precisely in terms of the runs (i.e., sequences of actions) it allows. Informally, a *subprotocol* refines a *superprotocol* if and only if the latter allows all the runs the former allows. However, refinement is nontrivial because the protocols may involve different roles and messages, the messages may have different meanings, and the meanings may be at different levels of abstraction. Hence, we define refinement only with respect to a mapping of meanings from the superprotocol to the subprotocol. For example, the payment in *Pay* maps to two payments in *PayViaMM*.

Our approach for verifying refinement takes three inputs: formal descriptions of a putative superprotocol and subprotocol, and a mapping between them. We reduce the protocol descriptions to their canonical forms, taking into account the mapping provided. We generate an input to an existing model checker consisting of (i) a specification of a temporal logic model and (ii) temporal formulae whose truth in the model verifies refinement.

1.2 Contributions

Our main contributions are as follows. One, we offer the *first* approach that computes the refinement for protocols based on static analysis of protocol specifications. Two, we formulate a notion of the serial composition of commitments, which can have broader applications than this paper, e.g., in the treatment of commitments in coalitions.

$$\begin{tikzcd}
 & & \mathit{Pay} & & \\
\mathit{PayViaSpouse} \arrow[urr] & \mathit{FullPay} \arrow[ur] & \mathit{PayViaMM} \arrow[u] & \mathit{OrderPayShip} \arrow[ul] & \mathit{NetBill}_2 \arrow[l] \\
 & \mathit{PayViaCheck} \arrow[ur] & \mathit{PayViaCredit} \arrow[u] & \mathit{OrderPayViaMMShip} \arrow[ul] \arrow[u] & \mathit{NetBill}_3 \arrow[l]
\end{tikzcd}$$

Fig. 1. Refinements demonstrated by Proton (arrows point from subprotocols to superprotocols).

Further, we have implemented our approach in the Proton tool that overlays the well-known model checker MCMAS (`http://www-lai.doc.ic.ac.uk/mcmas/`). Figure 1 summarizes some protocol refinements that Proton verifies (under the obvious mappings) based on the above and other examples known from the literature.

1.3 Organization

Section 2 overviews our syntax and semantics and briefly reviews commitments. Section 3 introduces our running examples for payment and order protocols. Section 4 formalizes our definitions of protocols, mappings between protocols, and protocol refinement. Section 5 describes how Proton generates input for the MCMAS model checker and the CTL formulae that must be satisfied for protocol refinement to hold. Section 6 pulls the previous sections together, illustrating how protocol *PayViaMM* refines, or fails to refine, protocol *Pay* under various mappings. Section 7 shows that the algorithmic implementation in Section 5 is correct with respect to the theoretical framework of Section 4. Section 8 describes the related literature and important future directions.

2. TECHNICAL FRAMEWORK

We adopt the following terminology. A *subprotocol* refines a *superprotocol*. In hyphenated form, *super-x* and *sub-x* refer to element *x* as it occurs in the superprotocol and subprotocol, respectively. For example, a *super-role* is a role defined in the superprotocol and a *sub-commitment* is a commitment defined in the subprotocol.

2.1 Interpreted Systems

We adopt Lomuscio and colleagues' [Lomuscio and Raimondi 2006; Cohen et al. 2009] formalization of a multiagent system as an *interpreted system*. Importantly, protocols involve roles, not agents. We presume no knowledge of the internals of an agent playing a role and consider all possible strategies that a role may follow in a protocol.

Each role is described by a set of possible local states, a set of local actions, a local strategy listing the legal actions in each local state, and a local progression function defining the progression of the role's local state based on the actions performed by all the roles. To clarify the terminology, our *role* and *strategy* respectively map to *agent* and *protocol* in work by Lomuscio and colleagues.

Definition 2.1 *Interpreted System.* An interpreted system $\mathcal{I}$ is

$$\mathcal{I} \;=\; \langle \Sigma, P, PV, L^i, \mathit{Act}^i, \mathit{AP}^i, t^i, G, G_0, F\rangle$$

$\Sigma = \{1, \dots, n, e\}$ is a set of three or more roles, including a distinguished role e that stands for the environment. P is a set of atomic propositions. Let $i \in \Sigma$ range over all roles and the environment e. L^i is a nonempty set of possible local states for each i. Act^i is a set of actions for each i. $\mathit{AP}^i : L^i \times L^e \mapsto 2^{\mathit{Act}^i}$ is the local strategy for each i. In local state $l \in L^i, l^e \in L^e$, role i can perform only the actions in $\mathit{AP}^i(l, l^e)$. $G \subseteq L^1 \times \cdots \times L^n \times L^e$ is the set of reachable global states. For any global state $g \in G$, we write g^i for the i-th component in g, i.e., the local state of role i in g. $G_0 \subseteq G$ is a nonempty set of initial global states. $PV : P \mapsto 2^G$ is the evaluation function for propositions. The set of joint actions is $\mathit{Act} = \mathit{Act}^1 \times \cdots \times \mathit{Act}^n \times \mathit{Act}^e$. $t^i : L^i \times L^e \times \mathit{Act} \mapsto L^i$ is the local progression function for role $i \in \Sigma \setminus e$, and $t^e : L^e \times \mathit{Act} \mapsto L^e$ is the progression function for the environment. All roles progress simultaneously. The global progression function is $T : G \times \mathit{Act} \mapsto G$ and is defined such that $T(g, a) = g'$ iff $\forall i \in \Sigma \setminus e : t^i(g^i, g^e, a^i) = g'^i$ and $t^e(g^e, a^e) = g'^e$. T must be serial ($\forall g \in G, \exists a \in \mathit{Act}, \exists g' \in G : T(g, a) = g'$). F is a set of Boolean fairness conditions, each of which must be true infinitely often on all legal execution paths. A path π in $\mathcal{I}$ is an infinite sequence of global states $\langle g_0, g_1, \dots\rangle$ in G such that every pair of adjacent states is a legal transition, i.e., $\forall i \geq 0 : \exists a \in \mathit{Act} : T(g_i, a) = g_{i+1}$. The i-th state in path π is denoted π_i, and the set of all paths starting at $g \in G$ is denoted $\Pi(g)$.

Given an interpreted system $\mathcal{I}$, we associate with it a Kripke model $\mathcal{K} = (G, G_0, T, PV)$ where G is the set of possible worlds understood as the reachable states of $\mathcal{K}$, built from the set of initial states G_0 by iterating the global progression function T; and, the temporal relation $T \subseteq G \times \mathit{Act} \times G$ connects global states based on the joint actions. The labeling function *PV* is the propositional labeling function.

The grammar of the temporal language CTL is (*PropName* is an atomic proposition)

$$\phi \;::=\; \mathit{PropName} \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \mathbf{AG}\phi \mid \mathbf{AF}\phi \mid \mathbf{A}[\phi\mathbf{U}\phi] \mid \mathbf{EG}\phi \mid \mathbf{EF}\phi \mid \mathbf{E}[\phi\mathbf{U}\phi]$$

Our temporal formulae use the standard CTL temporal logic operators: **A** (for all paths), **E** (for some path), **G** (on all future states on a path), **F** (eventually on the path), and **U** (until on the path). All operators can be rewritten using $\neg$, $\vee$, **EG**, and **EU** in the standard way. The following is the semantics for CTL, specified relative to the Kripke structure $\mathcal{K}$ at state g. Given a model $\mathcal{K}$ for an interpreted system, $\mathcal{I}$ satisfies a CTL formula ϕ if and only if $\mathcal{K}, g_0 \models \phi$, where $g_0 \in G_0$ is a starting state.

$\mathcal{K}, g \models p$	iff $p \in g$
$\mathcal{K}, g \models \neg\phi$	iff it is not the case $\mathcal{K}, g \models \phi$
$\mathcal{K}, g \models \phi \vee \psi$	iff $\mathcal{K}, g \models \phi$ or $\mathcal{K}, g \models \psi$
$\mathcal{K}, g \models \mathbf{EG}\phi$	iff $\exists \pi \in \Pi(g), \forall i \geq 0, \mathcal{K}, \pi_i \models \phi$
$\mathcal{K}, g \models \mathbf{E}[\phi\mathbf{U}\psi]$	iff $\exists \pi \in \Pi(g), \exists k \geq 0, \mathcal{K}, \pi_k \models \psi$ and $\forall 0 \leq j < k, \mathcal{K}, \pi_j \models \phi$

We write $a \models b$ if and only if for all models $\mathcal{K}$ and states g, $\mathcal{K}, g \models b$ holds whenever $\mathcal{K}, g \models a$ holds.

2.2 Proton Syntax

Proton verifies whether one protocol refines another under a mapping. Figure 2 shows Proton's input syntax (here $\mid$ separates alternatives, $\langle A\rangle^*$ is zero or more repetitions of

Protocol	::=	protocol *ProtoName* {role $\langle$*RoleName*; $\rangle^*$ prop $\langle$*PropName*; $\rangle^*$ commitment $\langle$*Commit*; $\rangle^*$ message $\langle$*Message*; $\rangle^*$ }
Commit	::=	*ComName* = C(*Debtor*, *Creditor*, *Ant*, *Csq*)
Message	::=	*Snd* $\rightarrow$ *Rcv* : $\langle[Guard]\rangle^?$*MsgName* $\langle$means {*Actions*}$\rangle^?$
Debtor	::=	*RoleExp*
Creditor	::=	*RoleExp*
Ant	::=	*ActExp*
Csq	::=	*ActExp*
Snd	::=	*RoleName*
Rcv	::=	*RoleName*
Guard	::=	*MsgExp*
Map	::=	map *MapName* : *ProtoName* $\mapsto$ *ProtoName*{role$\langle$*RoleMap*; $\rangle^*$ prop$\langle$*PropMap*; $\rangle^*$commitment $\langle$*CommitMap*; $\rangle^*$ }
RoleMap	::=	*RoleName* $\mapsto$ *RoleExp*
PropMap	::=	*PropName* $\mapsto$ *ActExp*
CommitMap	::=	*ComName* $\mapsto$ *CommitExp*
RoleExp	::=	*RoleName* $\mid$ {*RoleName* $\langle$, *RoleName*$\rangle^*$ }
MsgExp	::=	*MsgName* $\mid$ $\neg$*MsgExp* $\mid$ *MsgExp* $\vee$ *MsgExp* $\mid$ *MsgExp* $\wedge$ *MsgExp* $\mid$ *MsgExp* $\rightarrow$ *MsgExp*
ActExp	::=	*Action* $\mid$ $\neg$*ActExp* $\mid$ *ActExp* $\vee$ *ActExp* $\mid$ *ActExp* $\wedge$ *ActExp* $\mid$ *ActExp* $\rightarrow$ *ActExp*
CommitExp	::=	*ComName* $\mid$ *CommitExp* $\oplus$ *ComName*
Actions	::=	*Action* $\langle$, *Action*$\rangle^*$
Action	::=	*PropName* $\mid$ CREATE(*ComName*) $\mid$ TRANSFER(*ComName*) $\mid$ RELEASE(*ComName*) $\mid$ CANCEL(*ComName*)

Fig. 2. Proton input syntax in BNF.

A, and $\langle A\rangle^?$ is an optional occurrence of A). Notice that *PropName* is simply an atomic proposition name.

The *Protocol* nonterminal describes the syntax for a protocol (as Listing 1 exemplifies). A protocol declares roles, propositions, commitments, and messages. Each message m is sent from a sender ($m.\mathit{snd}$) to a receiver ($m.\mathit{rcv}$), has a guard ($m.\mathit{guard}$) which must be true before the message can be sent, a set of actions ($m.\mathit{actions}$) that means a conjunction of these actions ($m.\mathit{actexp}$). Actions are either propositions (being set true) or a commitment operation (being performed). Boolean negation ($\neg$) is allowed in antecedent, consequent, and guard expressions to check state, but a message meaning cannot set a proposition to false using a negated action.

The *Map* nonterminal describes the syntax for a mapping between two protocols (as Listing 3 exemplifies). A mapping maps individual roles, propositions, and commitments from the putative superprotocol to expressions in the putative subprotocol. *ProtoName*, *MapName*, *RoleName*, *PropName*, *ComName*, *MsgName*, and *Action* are names.

The serial composition operator, $\oplus$, chains two commitments together and is described in Section 2.5. We write $\bigoplus_i \mathsf{C}_i$ for a left-associated chain $(\mathsf{C}_1 \oplus \mathsf{C}_2) \oplus \ldots \oplus \mathsf{C}_n$. In Section 2.6, we compare commitments between superprotocol and subprotocol, under an abstraction mapping M, using commitment covering ($\leq_M$).

2.3 Proton Semantics

Proton's semantics is based on interpreted systems: it constructs an interpreted system from an input superprotocol, subprotocol, and mapping.

Each state g is a set of true propositions p_i. All propositions are false in the initial state g_0. Actions cause state transitions. For this paper, we use a simplified model of actions, assuming actions (i) always succeed, (ii) have definite outcomes (no uncertainty), and (iii) have no side-effects. The actions for role i, Act^i, are the propositional and commitment actions $\mathit{Act}^i = \{p_i \in \mathcal{A}\} \cup \{a(\mathsf{C}_j) | a \in \mathit{Act}_\mathsf{C}, \mathsf{C}_j \in \mathcal{C}\} \cup \{\mathit{nop}\}$ where $\mathcal{A}$ is the set of protocol propositions, $\mathcal{C}$ is the set of protocol commitments, $\mathit{Act}_\mathsf{C} = \{\text{CREATE}, \text{TRANSFER}, \text{RELEASE}, \text{CANCEL}\}$, and *nop* represents no-operation.

Operationally, messaging is point-to-point and synchronous. All protocol state is stored in the "environment" (effectively, a distinguished agent), and is globally accessible by all roles (a current simplification). At each time step, the environment schedules one role to execute next (interleaved execution). When scheduled, the role's agent (i) determines which of its messages are currently enabled, by accessing the protocol's global state and evaluating each message's guard expression, (ii) chooses an enabled message to send or chooses "no-operation", (iii) performs all actions in message's meaning, in any order, and (iv) updates the protocol's state.

In every global state g of the interpreted system, each commitment C_i has a state, $\mathsf{C}_i.\mathit{status} \in \mathit{Status}_\mathsf{C}$, where $\mathit{Status}_\mathsf{C} = \{\mathit{null}, \mathit{cond}, \mathit{detached}, \mathit{dis}, \mathit{xfer}, \mathit{rel}, \mathit{can}\}$, whose value can be any of the states in Figure 3. We define propositions for the expressions $\mathsf{C}.\mathit{status} = x$ in each state g. For each commitment C_i, we evaluate the occurrence of the commitment operations using the four propositions:

$$\begin{array}{lcl} \text{CREATE}(\mathsf{C}_i) & \triangleq & \mathsf{C}_i.\mathit{status} \neq \mathit{null} \\ \text{TRANSFER}(\mathsf{C}_i) & \triangleq & \mathsf{C}_i.\mathit{status} = \mathit{xfer} \\ \text{RELEASE}(\mathsf{C}_i) & \triangleq & \mathsf{C}_i.\mathit{status} = \mathit{rel} \\ \text{CANCEL}(\mathsf{C}_i) & \triangleq & \mathsf{C}_i.\mathit{status} = \mathit{can} \end{array}$$

As Section 1.1 mentions and Definition 4.6 formalizes, refinement depends upon a mapping between protocols to account for their different levels of abstraction. Specifically, we must map each (i) super-role to a set of sub-roles, (ii) super-proposition to a Boolean expression of sub-propositions, and (iii) super-commitment to an expression of sub-commitments. Proton combines two protocols and a mapping to (i) construct an interpreted system model $\mathcal{I}$ from the subprotocol's propositions, commitments, and guarded actions, as specified in Definition 4.2 and (ii) generate appropriate CTL formulae as specified in Section 5.6. The refinement in consideration holds if and only if the constructed model satisfies all the CTL formulae that Proton generates.

2.4 Background on Commitments

Commitments are a formal and concise method of describing how agent roles commit to performing future actions [Singh 1999; Yolum and Singh 2002]. We extend previous commitment definitions in two ways. First, we allow both debtors and creditors to be sets of roles, enabling us to handle commitment chains with multiple debtors and multiple creditors. Second, we introduce a new TRANSFER commitment operation to unify and replace prior uses of DELEGATE and ASSIGN.

A commitment $\mathsf{C}_{\{\text{debtors}\},\{\text{creditors}\}}(\mathit{antecedent}, \mathit{consequent})$ means that the debtors commit to the creditors, that if the antecedent holds, they will bring about the consequent. In an active commitment whose antecedent is false, the debtors are conditionally committed to the creditors. When the antecedent becomes true, we say the commitment is *detached*, and the debtors become unconditionally committed to the creditors.

Consider the following two example commitments drawn from the payment scenarios: (i) $\mathsf{C}_{\text{Payer,Payee}}(promise, pay)$: the payer conditionally commits to paying the payee; and (ii) $\mathsf{C}_{\{\text{Payer,MM}\},\text{Payee}}(promise, pay_P \wedge pay_M)$: the payer and middleman commit to paying the payee via pay_P (the payer's payment) and pay_M (the middleman's payment).

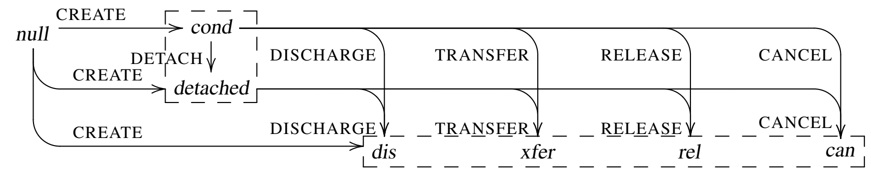


Fig. 3. State transition diagram for commitments (states in lowercase; transitions in small caps).

Figure 3 shows the state transition diagram for a commitment. The states of a commitment are (i) *null*: where the commitment does not yet exist; (ii) *cond*(active and conditional): after CREATE with antecedent and consequent being false, and with no other operations; (iii) *detached*(active and detached): after CREATE with antecedent true, consequent false, and with no other operations; (iv) *dis*(discharged): after CREATE and consequent being true; (v) *xfer*(transferred): after CREATE and TRANSFER; (vi) *rel*(released): after CREATE and RELEASE; and (vii) *can*(canceled): after CREATE and CANCEL. A commitment in states *cond* or *detached* is said to be *active*. A commitment in states *dis*, *xfer*, *rel*, or *can* is said to be *resolved*. The commitment operations are (i) CREATE, performed only by debtors, creates an active commitment; (ii) DETACH, which occurs implicitly when the antecedent becomes true; (iii) DISCHARGE, which occurs implicitly when the consequent becomes true; (iv) TRANSFER, performed by either debtors or creditors, deactivates the current commitment and replaces it by another commitment; (v) RELEASE, performed only by creditors, deactivates the commitment, thus releasing the debtors; and (vi) CANCEL, performed only by debtors, cancels, deactivates, and "breaks" the debtors' commitment.

We understand a protocol Q as refining a protocol P if and only if the correct enactments of Q are also correct enactments of P. As in previous work, in a correct enactment of a protocol each detached commitment must eventually resolve. Debtors may act before they are required to do so and the consequent may become true before the antecedent. In general, there is no guarantee that autonomous debtors do not arbitrarily CANCEL. In practice, the creditors would assume the debtors are trustworthy or the setting would include an external mechanism (such as penalties) to ensure the debtors' compliance.

2.5 Serial Composition of Commitments

In *PayViaMM*, where the payer commits to a middleman who commits to the payee, the two commitments together effectively commit the payer to the payee. We introduce serial composition as a general way to chain commitments over intermediaries, computing a single, resultant commitment. The serial composition of commitments is a static construct, but the resultant commitment dynamically progresses through the states in Figure 3 as the protocol progresses.

Definition 2.2. Let C_A and C_B be two commitments where $\mathsf{C}_A.csq \models \mathsf{C}_B.ant$. Then, the serial composition of C_A and C_B is the commitment $\mathsf{C}_\oplus = \mathsf{C}_A \oplus \mathsf{C}_B$ whose components are specified precisely as follows:

$$\mathsf{C}_\oplus.\mathit{debt} := \mathsf{C}_A.\mathit{debt} \cup \mathsf{C}_B.\mathit{debt} \tag{1}$$
$$\mathsf{C}_\oplus.\mathit{cred} := \mathsf{C}_A.\mathit{cred} \cup \mathsf{C}_B.\mathit{cred} \tag{2}$$
$$\mathsf{C}_\oplus.\mathit{ant} := \mathsf{C}_A.\mathit{ant} \tag{3}$$
$$\mathsf{C}_\oplus.\mathit{csq} := \mathsf{C}_A.\mathit{csq} \wedge \mathsf{C}_B.\mathit{ant} \wedge \mathsf{C}_B.\mathit{csq} \tag{4}$$

The state of $\mathsf{C}_\oplus$ is defined based on the states of C_A and C_B. $\mathsf{C}_\oplus$ is created exactly when both C_A and C_B are created. $\mathsf{C}_\oplus$ is respectively transferred, released, or canceled when at least one of C_A and C_B is transferred, released, or canceled.

Singh's [2008] formalization of commitments includes the similar idea of commitment chaining, but serial composition additionally captures the intuition of a coalition of roles. The above well-definedness condition $\mathsf{C}_A.\mathit{csq} \models \mathsf{C}_B.\mathit{ant}$ follows Singh's [2008] definition for chaining, although serial composition accommodates different roles. The second commitment becomes active ($\mathsf{C}_B.\mathit{ant}$) whenever the first commitment resolves ($\mathsf{C}_A.\mathit{csq}$), including the case where the first debtors perform without being required to do so ($\mathsf{C}_A.\mathit{ant}$ always false).

Informally, we say debtors are *responsible* for their commitments, and creditors are *beneficiaries* of their commitments. In a detached commitment, the debtors are responsible for eventually making the consequent true. 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). We use *several* responsibility so that, in serial composition of commitments, a debtor is never compelled to assume additional responsibilities. The result of serial composition is useful for reasoning about multiple commitments, but the original commitment expression, with its individual commitments, must be retained to determine which role(s) failed to perform if the resultant consequent is not produced.

$\mathsf{C}_\oplus$ states the union of debtors is committed to the union of creditors to bring about the consequent $\mathsf{C}_A.csq \wedge \mathsf{C}_B.ant \wedge \mathsf{C}_B.csq$ when antecedent $\mathsf{C}_A.ant$ is true. Debtors are *severally* responsible for $\mathsf{C}_\oplus$, so that debtors are never compelled to assume additional responsibilities. Every debtor in $\mathsf{C}_A.debt$ is partially responsible for discharging C_A, and thus is partially responsible for discharging $\mathsf{C}_\oplus$. Also, every debtor in $\mathsf{C}_B.debt$ has some responsibility for $\mathsf{C}_\oplus$. Equation 1 captures this intuition for debtors. Equation 2 captures the analogous intuition for creditors.

If we order the states of a commitment as $\mathit{null} < \mathit{can} < \mathit{rel} < \mathit{xfer} < \mathit{cond} < \mathit{detached} < \mathit{dis}$, then the state of a serial composition is the minimum of its constituents' states: $\mathsf{C}_\oplus.\mathit{status} = \mathit{min}(\mathsf{C}_A.\mathit{status}, \mathsf{C}_B.\mathit{status})$. That is, a serial composition progresses no further than its least constituent.

Because of Equations 3 and 4, serial composition is neither commutative nor associative. However, it creates commitments that are at least as strong as, and typically stronger than, their inputs. $\mathsf{C}_A \oplus \mathsf{C}_B$ is typically stronger than C_A 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 ($\mathsf{C}_B.csq$ vs. $\mathsf{C}_A.csq \wedge \mathsf{C}_B.ant \wedge \mathsf{C}_B.csq$). The lemma below shows serial composition is not always stronger, because $\oplus$ is idempotent: a commitment can be usefully added to a commitment chain only once.

LEMMA 2.3. *If* C_k *is any commitment in a commitment chain* $\bigoplus_{1\le i\le n} \mathsf{C}_i$*, then composing* C_k *again does not increase the strength.*

$$(\bigoplus_{1\le i\le n} \mathsf{C}_i) \oplus \mathsf{C}_k = \bigoplus_{1\le i\le n} \mathsf{C}_i$$

PROOF. If $\bigoplus_i \mathsf{C}_i$ is well defined, then so is $(\bigoplus_i \mathsf{C}_i) \oplus \mathsf{C}_k$. By inspection, Equations (1–4) yield the same results for both sides. □

2.6 Commitment Covering

Because commitments are crucial to our semantics of protocols, commitments are also crucial to refinement. And because we need to compare two protocols, we need a mechanism to compare two commitments. Specifically, each super-commitment must be *covered by*, or make at least the same commitment as, another relevant sub-commitment. The commitment comparison accommodates a mapping to account for the commitments being expressed at different levels of abstraction. Definition 2.4 extends Chopra and Singh's [2009] notion of *commitment strength*. In addition to the logical relationships between antecedents and consequents, this definition incorporates the mapping of roles and propositions.

Definition 2.4 *Commitment Covering*. A stronger commitment C_S covers (is stronger than) a weaker commitment C_W with respect to mapping M, written $\mathsf{C}_W \le_M \mathsf{C}_S$, if and only if

$$\forall d \in \mathsf{C}_W.debt \qquad M(d) \cap \mathsf{C}_S.debt \neq \emptyset \tag{5}$$
$$\forall c \in \mathsf{C}_W.cred \qquad M(c) \cap \mathsf{C}_S.cred \neq \emptyset \tag{6}$$
$$M(\mathsf{C}_W.ant) \models \mathsf{C}_S.ant \tag{7}$$
$$\mathsf{C}_S.csq \models M(\mathsf{C}_W.csq) \tag{8}$$

where $M(x)$ maps (super-) element x in C_W to an expression of (sub-) elements in C_S.

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 $M(d)$. We require each super-debtor's responsibilities be passed to one or more of its sub-debtors. Together these sub-debtors assume the super-debtor's responsibilities. Equation 5 captures the requirement that every super-debtor's responsibilities must pass to at least one of its sub-debtors, so that responsibilities are not lost. Similarly, each super-creditor is a partial beneficiary of the super-commitment. Equation 6 captures the requirement that every super-creditor's benefit passes to at least one of its sub-creditors.

In many situations, multiple sub-commitments must be combined to cover a single super-commitment. In those cases, a super-commitment is covered by the serial composition of multiple sub-commitments.

We visualize our explanations by diagramming a commitment as a labeled arrow.

$$\xrightarrow[\text{ant}^{(\wedge)} \qquad\qquad\qquad \text{csq}^{(\wedge)}]{\text{debt}^{(\cup)} \quad \mathsf{C}_{\text{name}}{}^{(\oplus)} \quad \text{cred}^{(\cup)}}$$

The name of the commitment is written in the top center of the arrow. Debtors and creditors are above the arrow and the antecedent and consequent below it. When multiple terms appear in a position, they are implicitly combined using the operator in parentheses.

As a simple example, consider commitments $\mathsf{C}_1 = \mathsf{C}_{\text{Payer},\text{Payee}}(promise, pay)$, $\mathsf{C}_2 = \mathsf{C}_{\text{Buyer},\text{Seller}}(order, pay)$, and an abstraction mapping M that is defined on roles and propositions as follows: (i) *Payer* $\mapsto \{$*Buyer*$\}$, (ii) *Payee* $\mapsto \{$*Seller*$\}$, (iii) *promise* $\mapsto$ *order*, and (iv) *pay* $\mapsto$ *pay*. The diagram shows C_2 covers C_1.

$$\begin{tikzcd}[row sep=large]
\underset{\text{promise}}{\text{Payer}} \arrow[r, "\mathsf{C}_1"] & \underset{\text{pay}}{\text{Payee}} \\
\underset{\text{order}}{\text{Buyer}} \arrow[r, "\mathsf{C}_2"] \arrow[u, bend left, "\leq_M"] & \underset{\text{pay}}{\text{Seller}}
\end{tikzcd}$$

As another example, we obtain $\mathsf{C}(order, ship) \leq \mathsf{C}(order \vee freeCoupon, ship)$ by Equation 7, since the stronger commitment detaches when *order* or *freeCoupon* is true. And, likewise $\mathsf{C}(order, ship) \leq \mathsf{C}(order, ship \wedge expressDelivery)$ holds by Equation 8, since to discharge the stronger commitment requires *expressDelivery* in addition to *ship*.

3. RUNNING EXAMPLES OF PROTOCOLS

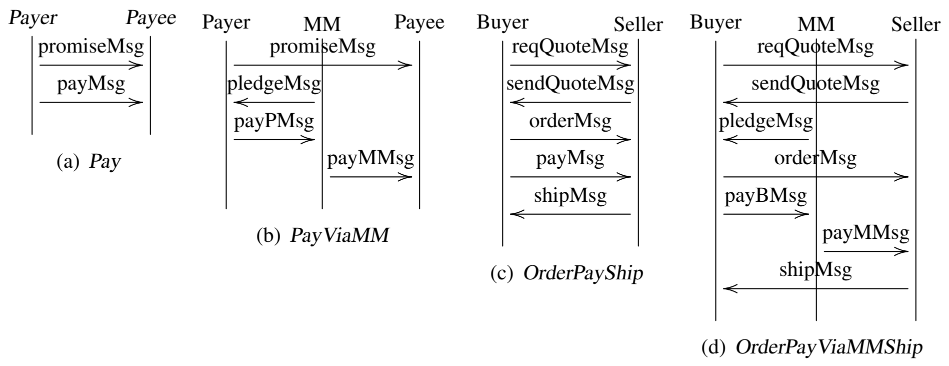


Fig. 4. Suggestive sequence diagrams for selected protocols (in general, alternative sequences may occur).

3.1 Protocol Descriptions

We introduce four running examples. *Pay* and *PayViaMM* are payment protocols whereas *OrderPayShip* and *OrderPayViaMMShip* are order protocols involving payments. Suggestive sequence diagrams are presented in Figure 4. Protocol *Pay* in Figure 4(a) is a basic payment protocol between a payer and a payee. The payer may commit to pay the payee by sending the promise message. Later, it sends a payment message directly to the payee.

Listing 1 shows the Proton specification of protocol *Pay*. 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. A message may be sent only if its guard (the expression between [and]) is true. The guard for *payMsg* in Line 8 is *promiseMsg*. If no guard is explicitly specified, as is the case for *promiseMsg* in Line 7, it is implicitly true. A message's meaning is expressed as a set of actions after means and between { and }.

In protocol *PayViaMM* (pay via middleman), if the payer chooses to do so, it commits to paying the payee by sending the promise message. The middleman commits to sending

Listing 1 *Pay* Protocol

```
protocol Pay {
    role Payer; Payee;
    prop promise; pay;
    commitment
        $C_{pay} = C(Payer, Payee, promise, pay)$;
    message
        $Payer \rightarrow Payee$ : promiseMsg means {promise, CREATE($C_{pay}$)};
        $Payer \rightarrow Payee$ : [promiseMsg] payMsg means {pay};
}
```

payM if the payer sends *payP* by sending *pledgeMsg*. The payer then sends a payment indirectly to the payee, first paying the middleman, who in turn pays the payee. The sequence diagram in Figure 4(b) shows a typical interaction. In this case, other acceptable runs also exist: for example, the middleman may send the *payM* message before the *payP* message. The Proton specification for *PayViaMM* is shown in Listing 2. Middleman commits to Payer to pass along any payment it receives (Line 11). Payer will not pay Middleman without this commitment (Line 12). Since *payMMsg* has an implicit guard of true (Line 14), Middleman is allowed to pay early.

Listing 2 *PayViaMM* Protocol

```
 1: protocol PayViaMM {
 2:     role Payer; MM; Payee;
 3:     prop promise;
 4:         payP;    //payment from Payer to MM
 5:         payM;    //payment from MM to Payee
 6:     commitment
 7:         $C_{payP} = C(Payer, Payee, promise, payP)$;
 8:         $C_{payM} = C(MM, Payer, payP, payM)$;
 9:     message
10:         $Payer \rightarrow Payee$ : promiseMsg means {promise, CREATE($C_{payP}$)};
11:         $MM \rightarrow Payer$ : pledgeMsg means {CREATE($C_{payM}$)};
12:         $Payer \rightarrow MM$ : [promiseMsg $\wedge$ pledgeMsg]
13:                             payPMsg means {payP};
14:         $MM \rightarrow Payee$ : payMMsg means {payM};
15: }
```

Protocol *OrderPayShip* in Figure 4(c) supports a buyer placing an order with 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 seller's offer by placing an order, which creates its commitment to pay for the good if it ships. The seller can ship first, or the buyer can pay first. Protocol *OrderPayViaMMShip* in Figure 4(d) is similar to *OrderPayShip* except that it incorporates *PayViaMM* for payment.

3.2 Mapping Abstractions across Protocols

Since superprotocols represent higher-level abstractions than subprotocols, comparing protocols must address differences in abstraction level. To this end, we map elements (roles, propositions, and commitments) of a putative superprotocol to elements of a putative subprotocol. We map every super-element to an expression of sub-elements, but a subprotocol may contain sub-elements that do not correspond with any super-element.

Listing 3 Mapping M_1: *Pay* to *PayViaMM*

```
1:  map $M_1$: Pay $\mapsto$ PayViaMM {
2:      role
3:          $Payer \mapsto \{Payer\}$;
4:          $Payee \mapsto \{Payee\}$;
5:      prop
6:          $promise \mapsto promise$;
7:          $pay \mapsto payP \wedge payM$;
8:      commitment
9:          $\mathsf{C}_{pay} \mapsto \mathsf{C}_{payP} \oplus \mathsf{C}_{payM}$;        //requires $\mathsf{C}_{pay} \leq_{M_1} \mathsf{C}_{payP} \oplus \mathsf{C}_{payM}$
10: }
```

Consider mapping M_1 in Listing 3 from *Pay* to *PayViaMM*. Each super-role is mapped to a set of sub-roles. Line 3 maps the Payer super-role in *Pay* to the Payer sub-role in *PayViaMM*. Each super-proposition in *Pay* is mapped to a Boolean expression of sub-propositions in *PayViaMM*. Line 7 maps *pay* to the conjunction of *payP* and *payM*. Notice that *payP* and *payM* are messages sent by different roles in *PayViaMM*; thus even the simple Line 7 demonstrates the generality of our mapping approach. Line 9 maps super-commitment $\mathsf{C}_{\mathrm{pay}}$ to the serial composition of sub-commitments $\mathsf{C}_{\mathrm{payP}}$ and $\mathsf{C}_{\mathrm{payM}}$.

Listing 4 Alternative Mapping M_2: *Pay* to *PayViaMM*

```
1:  map $M_2$: Pay $\mapsto$ PayViaMM {
2:      role
3:          $Payer \mapsto \{Payer, MM\}$;
4:          $Payee \mapsto \{Payee\}$;
5:      ...
6:  }
```

Listing 5 Alternative Mapping M_3: *Pay* to *PayViaMM*

```
1:  map $M_3$: Pay $\mapsto$ PayViaMM {
2:      role
3:          $Payer \mapsto \{Payer\}$;
4:          $Payee \mapsto \{Payee, MM\}$;
5:      ...
6:  }
```

There can be multiple mappings between some protocol pairs. The Middleman role does not appear in mapping M_1. We can construct alternative mappings that group the Middleman into coalitions with different super-roles. Mapping M_2 in Listing 4 and Mapping M_3 in Listing 5 are each the same as M_1 except for their role mappings: M_2 groups the Middleman into a coalition with Payer and M_3 into a coalition with Payee. *PayViaMM* refines *Pay* under all three mappings M_1, M_2, and M_3.

We require each commitment to be explicitly mapped. A commitment mapping must not violate the role and proposition mappings, but that is not always sufficient to uniquely determine the commitment mapping. It is possible that a super-commitment can be mapped to multiple serial compositions that meet all constraints. In a hypothetical *PayViaTwoMM* protocol, where the payment can be made through either Middleman$_1$ or Middleman$_2$, the super-commitment $\mathsf{C}_1 = \mathsf{C}_{\mathit{Payer},\mathrm{Payee}}(\mathit{promise}, \mathit{pay})$ can be mapped to either of two serial compositions (the protocol designer chooses between them based on other factors).

$$\begin{aligned}\mathsf{C}_1 &\mapsto \mathsf{C}_{\mathit{Payer},\mathit{MM}_1}(\mathit{promise}, \mathit{payMM}_1) \oplus \mathsf{C}_{\mathit{MM}_1,\mathit{Payee}}(\mathit{payMM}_1, \mathit{pay})\\ \mathsf{C}_1 &\mapsto \mathsf{C}_{\mathit{Payer},\mathit{MM}_2}(\mathit{promise}, \mathit{payMM}_2) \oplus \mathsf{C}_{\mathit{MM}_2,\mathit{Payee}}(\mathit{payMM}_2, \mathit{pay})\end{aligned}$$

Mapping B_1 in Listing 6 shows a possible mapping between *Pay* and *PayViaMM*, similar

Listing 6 Nonrefining Mapping B_1: *Pay* to *PayViaMM*

```
 1: map $B_1$: Pay $\mapsto$ PayViaMM {
 2:     role
 3:         $Payer \mapsto \{Payer\}$;
 4:         $Payee \mapsto \{Payee\}$;
 5:     prop
 6:         $promise \mapsto promise$;
 7:         $pay \mapsto payP \wedge payM$;
 8:     commitment
 9:         $\mathsf{C}_{pay} \mapsto \mathsf{C}_{payM} \oplus \mathsf{C}_{payP}$;        //wrong order
10: }
```

to M_1 except the serial composition in Line 9 combines the commitments in the wrong order. In Section 6, we show *PayViaMM* does *not* refine *Pay* under mapping B_1.

4. FORMALIZING PROTOCOLS AND THEIR REFINEMENT

We assume a set of atomic propositions that describe the state of the world and states of relevant commitments. We define actions as atomic propositions (being made true) and commitment operations (being performed). Messages set propositions true, but not false.

Definition 4.1. A *protocol* is a sextuple $\langle \mathcal{R}, \mathcal{M}, \mathcal{C}, \mathcal{A}, \mathcal{S}, \mathcal{G} \rangle$ corresponding respectively to (i) a set $\mathcal{R}$ of roles; (ii) a set $\mathcal{M}$ of message names; (iii) a set $\mathcal{C}$ of commitments; (iv) a set $\mathcal{A}$ of Boolean propositions and commitment states; (v) a set $\mathcal{S}$ of states, $\mathcal{S} \subseteq 2^{\mathcal{M}}$ such that $\emptyset \in \mathcal{S}$ and if $s \in \mathcal{S}$, $gs \in \mathcal{G}$, and $s \in gs.\mathit{guard}$ then $s \cup gs.\mathit{msg} \in \mathcal{S}$; and (vi) a set $\mathcal{G}$ of guarded statements of the form $\langle \mathit{snd}, \mathit{rcv}, \mathit{guard}, \mathit{msg}, \mathit{actions} \rangle$ with $\mathit{snd}, \mathit{rcv} \in \mathcal{R}$, $\mathit{guard} \subseteq \mathcal{S}$, $\mathit{msg} \in \mathcal{M}$, and $\mathit{actions} = \{a_i \in \mathcal{A}\} \cup \{\mathit{Act}_{\mathsf{C}}(\mathsf{C}_j \in \mathcal{C})\} \cup \{\mathit{nop}\}$. In addition, we impose the *no overlap* constraint: $\forall gs_1, gs_2 \in \mathcal{G}$, if $gs_1.\mathit{actions} \cap gs_2.\mathit{actions} \neq \emptyset$ then $gs_1.\mathit{guard} \cap gs_2.\mathit{guard} = \emptyset$.

Each message corresponds to an atomic proposition recording whether the message has been sent. Each global state $s \in \mathcal{S}$ is a set of (the atomic propositions corresponding to) the messages that have been sent in that state (Item v). Each guarded statement $gs \in \mathcal{G}$ has a guard $gs.\mathit{guard}$ which is a set of states, and a meaning $gs.\mathit{actexp}$—a conjunctive expression of actions. A message msg can be sent by the sender ($gs.\mathit{snd}$) to the receiver ($gs.\mathit{rcv}$) in state s only if $s \in gs.\mathit{guard}$. When m is sent, the action expression $gs.\mathit{actexp}$ becomes true in the next state. The actions corresponding to different messages may be interleaved. The *no overlap* constraint ensures that if two or more super-actions contain the same sub-action, and both super-actions are enabled in a state, then the occurrence of the common sub-action in a sub-run is unambiguous as to which super-action it corresponds to, which recall is key to our notion of refinement.

4.1 Protocol Enactment

We introduce a *run*, a possible computation through our model, as a basis for our semantics. A run, notated π, is an alternating sequence of states and actions $\langle s_0, a_1, s_1, a_2, s_2, \ldots \rangle$ such that s_{i+1} results from performing a_{i+1} in s_i. The length of π is written $|\pi|$.

We can now express two key intuitions. First, the semantics of a protocol is simply the set of runs it allows. Underlying each run is a coarser *message enactment*: a sequence of states and messages where each message's guard is true in the state where the message occurs. Second, a protocol refines another if and only if the runs of the first are also runs of the second, with the proviso that the putative subprotocol may involve roles and actions that are absent in the putative superprotocol. To capture the above, we need to relate protocols to models. Our approach generates a model from the putative subprotocol and then verifies (using suitable mapping) whether the putative superprotocol relates correctly with the subprotocol in that model, that is, whether the runs of the two protocols relate as explained above. Definition 4.2 specifies such a model.

Definition 4.2 *Proton Model*. Let $P = \langle \mathcal{R}, \mathcal{M}, \mathcal{C}, \mathcal{A}, \mathcal{S}, \mathcal{G} \rangle$ be a protocol. Then, the *Proton model for* P is $\mathcal{I} = \langle \Sigma, P, PV, L^i, \mathit{Act}^i, \mathit{AP}^i, t^i, G, G_0, F \rangle$ where (i) $\Sigma = \mathcal{R} \cup \{e\}$, with e being the environment, (ii) $P = \mathcal{A}$, (iii) $\forall p \in \mathcal{A}\ : PV(p) = \{s | p \in s\}$, (iv) $\forall i \in \mathcal{R} : L^i = \{l\}$ and $L^e = \prod_{m_i \in \mathcal{M}} m_i \times \prod_{\mathsf{C}_i \in \mathcal{C}} \mathsf{C}_i.\mathit{status}$, (v) $\forall i \in \mathcal{R} : \mathit{Act}^i = \{m | m.\mathit{snd} = i\} \cup \{\mathit{nop}\}$, and $\mathit{Act}^e = \{\mathit{sched} = r | r \in \mathcal{R}\}$. (vi) $\forall i \in \mathcal{R}, \forall s \in \mathcal{S} : \mathit{AP}^i(s) = \{m | \mathit{sched} = i \wedge m.\mathit{snd} = i \wedge s \in m.\mathit{guard}\}$, (vii) $\forall i \in \mathcal{R} : t^i(l) = l$, and $t^e = \prod_{m_i \in \mathcal{M}} t^{m_i} \times \prod_{\mathsf{C}_i \in \mathcal{C}} t^{\mathsf{C}_i}$, (viii) G is the set of all states reachable from G_0 by transition function T in $\mathcal{I}$, (ix) $G_0 = \emptyset$, and (x) $F = \{\mathsf{C}_i.\mathit{status} \neq \mathit{detached} | \mathsf{C}_i \in \mathcal{C}\}$.

Where $\times$ is binary cross-product and $\prod$ is set cross-product. The protocol's state is the cross-product of the state of each message and commitment. Since both messages and commitments involve multiple roles, each role has just a single state l and all state is in the environment (iv). Proton supports interleaved rather than concurrent actions with the environment scheduling one role at each step (v). Every role can perform the nop action (no-operation) at every step (v). t^{m_i} is the transition function that tracks the past occurrence of message m_i, and t^{C_i} is the transition function that tracks the commitment state of commitment C_i as defined by Figure 3 (vii).

Through a slight abuse of notation, for simplicity, we treat guards and actions as expressions in the following.

Definition 4.3 *Enactment*. Let $P = \langle \mathcal{R}, \mathcal{M}, \mathcal{C}, \mathcal{A}, \mathcal{S}, \mathcal{G} \rangle$ be a protocol and $\mathcal{I}$ be its Proton model. Then, an alternating sequence of states and messages $\langle h_0, m_1, h_1, m_2, h_2, \ldots\rangle$ is a *message enactment of* P if and only if $h_0 = \emptyset$ and $(\forall j \geq 0 : h_j \in \mathcal{S}, m_{j+1} \in \mathcal{M} : \mathcal{I}, h_j \models m_{j+1}.\mathit{guard}$ and $\mathcal{I}, h_{j+1} \models m_{j+1}.\mathit{actexp})$.

A message enactment yields one or more runs with different interleavings of each message's actions. We define a function μ that maps each index in the message enactment to the index in the run where the corresponding message expression $m_j.\mathit{actexp}$ becomes true. Each message expression occurs in the same order in every run, and becomes true precisely at the state where its execution completes.

Definition 4.4 *Run*. Let $P = \langle \mathcal{R}, \mathcal{M}, \mathcal{C}, \mathcal{A}, \mathcal{S}, \mathcal{G}\rangle$ be a protocol and $\mathcal{I}$ be its Proton model. Then, an alternating sequence of states and actions $\langle s_0, a_1, s_1, a_2, s_2, \ldots\rangle$ is a *run of* P if and only if $s_0 = \emptyset$ and $(\forall j \geq 0 : s_j \in \mathcal{S}, a_{j+1} \in \mathcal{A} : \mathcal{I}, s_j \models a_{j+1}.\mathit{guard}$ and $\mathcal{I}, s_{j+1} \models a_{j+1}.\mathit{actexp})$.

We say a run is *well defined* to emphasize that it satisfies the guard and action expression conditions above: that it is more than just an alternating sequence of states and actions. The empty run $\langle\emptyset\rangle$ is always well defined, since no agent is required to perform any action.

Definition 4.5 *Generated Runs*. Let $P = \langle \mathcal{R}, \mathcal{M}, \mathcal{C}, \mathcal{A}, \mathcal{S}, \mathcal{G}\rangle$ be a Proton protocol and $\mathcal{I}_P$ its model. Then, a run $\pi = \langle s_0, a_1, s_1, \ldots\rangle$ is *generated by* P if and only if there exists a message enactment $\langle h_0, m_1, h_1, \ldots\rangle$, and there exists a strictly increasing function on the natural numbers $\mu : \mathbb{N} \mapsto \mathbb{N}$ such that $(\forall j \geq 0 : \mathcal{I}, s_{\mu(j)} \models m_{j+1}.\mathit{guard}$ and $\mathcal{I}, s_{\mu(j+1)} \models m_{j+1}.\mathit{actexp})$.

We write **runs**(P) for the set of all runs generated by protocol P in $\mathcal{I}_P$.

4.2 Protocol Refinement

Definition 4.6 *Mapping*. M maps protocol $P = \langle \mathcal{R}_P, \mathcal{M}_P, \mathcal{C}_P, \mathcal{A}_P, \mathcal{S}_P, \mathcal{G}_P\rangle$ to protocol $Q = \langle \mathcal{R}_Q, \mathcal{M}_Q, \mathcal{C}_Q, \mathcal{A}_Q, \mathcal{S}_Q, \mathcal{G}_Q\rangle$ if and only if $M = \langle M_R, M_P, M_C\rangle$, where

$$\begin{aligned} M_R &= \{\langle r, R\rangle \mid r \in \mathcal{R}_P, R \subseteq \mathcal{R}_Q\} \\ M_P &= \{\langle p, e\rangle \mid p \in \mathcal{A}_P, e \subseteq 2^{\mathcal{A}_Q}\} \\ M_C &= \{\langle \mathsf{C}, \oplus_i \mathsf{C}_i\rangle \mid \mathsf{C} \in \mathcal{C}_P, \mathsf{C}_i \in \mathcal{C}_Q, \mathsf{C} \leq_M \oplus_i \mathsf{C}_i\} \end{aligned}$$

Informally, a run π_Q *embeds* a run π_P if all of π_P lies within π_Q. In effect, π_Q does everything that π_P does, and possibly more: as Mallya and Singh [2007] propose, a protocol Q refines a protocol P if and only if every run of Q embeds some run of P. This captures the intuition that any computation (run) allowed by Q is allowed by P as well.

Consider the mapping from *Pay* to *OrderPayShip*. In protocol *Pay*, *promiseMsg* means $\{\text{CREATE}(\mathsf{C}_{pay})\}$ and *payMsg* means $\{pay\}$. In protocol *OrderPayShip*, *orderMsg* means $\{\text{CREATE}(\mathsf{C}_{pay})\}$ and *payMsg* means $\{pay\}$. Therefore, *promiseMsg* and *payMsg* in *Pay* mean the same, respectively, as *orderMsg* and *payMsg* in *OrderPayShip*.

$$\begin{aligned} \mathit{Pay} &\mapsto \mathit{OrderPayShip} \\ \mathit{promiseMsg} &\mapsto \mathit{orderMsg} \\ \mathit{payMsg} &\mapsto \mathit{payMsg} \end{aligned}$$

Pay has two message enactments: $\langle\rangle$ and $\langle \text{promiseMsg}, \text{payMsg}\rangle$. *OrderPayShip* has five message enactments, which embed *Pay*'s runs as follows.

$\langle\rangle$	: $\langle\rangle$
$\langle\rangle$	: $\langle \textit{reqQuoteMsg} \rangle$
$\langle\rangle$	: $\langle \textit{reqQuoteMsg}, \textit{sendQuoteMsg} \rangle$
$\langle$promiseMsg, payMsg$\rangle$	: $\langle \textit{reqQuoteMsg}, \textit{sendQuoteMsg}$, orderMsg, payMsg, $\textit{shipMsg} \rangle$
$\langle$promiseMsg, payMsg$\rangle$	: $\langle \textit{reqQuoteMsg}, \textit{sendQuoteMsg}$, orderMsg, $\textit{shipMsg}$, payMsg$\rangle$

We define a *mapped run* where each sub-state s is enriched to a state $M(s)$ by including values for all super-propositions and super-commitments. We now compare enriched sub-states $M(s)$ in mapped sub-runs with super-states in super-runs. Below, we write $exp\langle\langle x \mapsto y\rangle\rangle$ to mean the expression resulting from the uniform substitution of symbol x by expression y in *exp*.

Definition 4.7 *Mapped Run*. Let $\pi = \langle s_0, a_1, s_1, \ldots\rangle$ be a run and $M = \langle M_R, M_P, M_C\rangle$ be a protocol mapping. Then the *M-map of* π, $M(\pi) = \langle M(s_0), a_1, M(s_1), \ldots\rangle$, is a run where for all s, $M(s) \supseteq s$ and $M(s)$ is the minimal set for which the following conditions hold:

—(Propositions) if $\langle m, E\rangle \in M_P$ and $s \models E$, then $m \in M(s)$.

—(Commitments) if $\langle \mathsf{C}, \bigoplus_i \mathsf{C}_i\rangle \in M_C$ and $\forall i : s \models \mathsf{C}_i.\textit{status}$, then $\mathsf{C}.\textit{status} \in M(s)$ where $\mathsf{C}.\textit{status} = \textit{min}(\mathsf{C}_i.\textit{status})$.

Continuing with the above discussion, we map each sub-run and verify that it embeds some super-run. The following definition captures the intuition that the embedding sub-run steps through each of the states of the embedded super-run, but may potentially include additional states. We ignore the transitions in each run.

To simplify the notation, we also introduce a projected mapping function $\widehat{M}(q) = M(q) \cap \mathcal{A}_P$ that is the set of just the propositions and states in a (super-)protocol P.

Definition 4.8 *Embedding*. Let P and Q be two protocols. A run $\pi_Q = \langle q_0, \cdot, q_1, \ldots\rangle \in$ **runs**(Q) *embeds* a run $\pi_P = \langle p_0, \cdot, p_1, \ldots\rangle \in$ **runs**(P), written **emb**(π_Q, π_P), if and only if there exists a strictly increasing function on natural numbers $\tau : \mathbb{N} \mapsto \mathbb{N}$ such that $(\forall i : 0 \leq i \leq |\pi_P| : p_i = \widehat{M}(q_{\tau(i)})$ and $(\forall j : \tau(i) \leq j < \tau(i+1) : \widehat{M}(q_{\tau(i)}) = \widehat{M}(q_j))$, where $\widehat{M}(q) = M(q_j) \cap \mathcal{A}_P$.

Function τ maps from indices of π_P to indices of π_Q, and the conditions ensure every time $\widehat{M}(q_j)$ changes, the new value matches the next p_i.

Now we can define refinement in purely semantic terms that capture our intuition that each mapped sub-run must embed some super-run. Notice that this definition implicitly uses Proton models $\mathcal{I}_P$ and $\mathcal{I}_Q$ respectively for P and Q.

Definition 4.9 *Refinement*. Let P and Q be two protocols, and M a mapping from P to Q. Then Q refines P under M if and only if $(\forall \pi_Q \in$ **runs**$(Q) : (\exists \pi_P \in$ **runs**$(P) :$ **emb**$(M(\pi_Q), \pi_P)))$.

5. VERIFYING PROTOCOL REFINEMENT

Figure 5 shows the high-level process flow for verifying protocol refinement. The Proton preprocessor reads the subprotocol, superprotocol, and mapping specifications and constructs (as Section 6 details) the input for the MCMAS model checker in the Interpreted Systems Programming Language (ISPL) [Lomuscio et al. 2009].

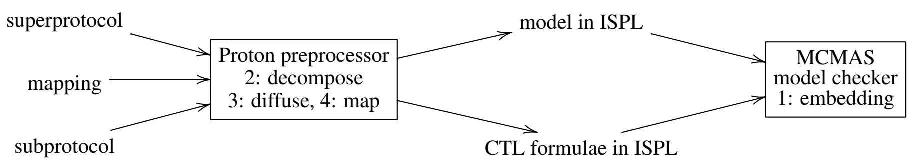


Fig. 5. Proton process flow. Activities are shown in boxes and specification documents are not. Transformations and comparisons are numbered 1 to 4 in sequence.

The input to MCMAS is a set of guarded statements for each role. Internally, MCMAS implicitly generates a state transition system such as that shown in Figure 6. The system starts in initial state s_0. Action *requestQuote* transitions to state s_1, action *sendQuote* transitions to state s_2, and so on. There is an edge for every action enabled in a state.

$$\begin{tikzcd}
 & & & & s_4 \arrow[dr, "\text{ship}"] & \\
s_0 \arrow[r, "\text{requestQuote}"] & s_1 \arrow[r, "\text{sendQuote}"] & s_2 \arrow[r, "\text{order}"] & s_3 \arrow[ur, "\text{pay}"] \arrow[dr, "\text{ship}"'] & & s_6 \\
 & & & & s_5 \arrow[ur, "\text{pay}"'] &
\end{tikzcd}$$

Fig. 6. A schematic of some possible enactments of the *OrderPayShip* protocol.

The Proton preprocessor generates an interpreted system model for the subprotocol. There is one ISPL agent definition for each sub-role, and the state of all sub-elements (propositions and commitments) are expressed as model state variables. The model checker simulates the subprotocol's actions. Because each super-element is mapped to an expression of sub-elements, the state of every super-element can be inferred from the subprotocol's state. As Section 5.6 shows, protocol refinement conditions are expressed as CTL formulae. If all these CTL formulae are true, the subprotocol refines the superprotocol.

5.1 Intuition: Decomposition

A message can mean multiple things. To better understand and characterize a message, we decompose each *message* into its meaning as a set of primitive, well-defined *actions*. The meaning of a message is then the conjunction of all its constituent actions.

An action is either a Boolean proposition or a commitment operation. A propositional action sets the value of the proposition to true. We do not support setting propositions to false. Commitment actions are the operations CREATE, TRANSFER, RELEASE, and CANCEL that change the state of a commitment.

We replace all message terms with a conjunction of their actions throughout a protocol, *decomposing* protocol messages, converting from a "protocol of guarded messages" to a "protocol of guarded actions." Each *msg* term is replaced by a conjunction of its actions in both the guard and the action expression of every guarded statement.

$$\mathbf{means}([\mathit{guard}]\ \mathit{msg}\ \mathit{means}\{\mathit{act}_i\}) \ \Rightarrow\ [\mathit{guard}\langle\langle \mathit{msg} \mapsto \bigwedge_i \mathit{act}_i \rangle\rangle\,]\ \bigwedge_i \mathit{act}_i$$

5.2 Intuition: Diffusion

The result of decomposition is a set of guarded action expressions, but the model checker executes actions, not action expressions. Therefore, each guarded action expression must be converted to an equivalent set of guarded actions. However, computing the equivalent guarded actions is nontrivial. For example, consider the guarded action expression

$$[\mathit{guard}]\ \mathit{payP} \wedge \mathit{payM}$$

A naïve approach would be to apply the guard to each action separately: $[\mathit{guard}]\ \mathit{payP}$ and $[\mathit{guard}]\ \mathit{payM}$. Doing so would be overly restrictive because neither payP nor payM can occur before the guard becomes true. Greater flexibility is needed.

Given a guarded conjunction of actions $[guard]\, a_1 \wedge a_2 \wedge \cdots \wedge a_n$, the action expression becomes true when the *last* (in time) of the a_is becomes true. Given a guarded disjunction of actions $[guard]\, a_1 \vee a_2 \vee \cdots \vee a_n$, the action expression becomes true when the *first* (in time) of the a_is becomes true. We minimally constrain when the a_i become true so the overall expression becomes true at exactly the same point, relative to the other actions, in both the super-runs and the sub-runs. For conjunctions, all the a_i, except the *last*, can move to any *earlier* point in time. For disjunctions, all the a_i, except the *first*, can move to any *later* point in time. The a_i can even move all the way to the run's beginning (conjunction) or end (disjunction). Decomposition (Section 5.1) generates guarded action expressions with conjunctions; abstraction mappings (Section 5.4) can generate guarded action expressions with both conjunctions and disjunctions.

The recursive *diffusion* function **dif** transforms a guarded action $\underline{\text{expression}}$ to a set of guarded actions.

$$\mathbf{dif}([\mathit{guard}]\ \bigvee_i \mathit{exp}_i) \ \Rightarrow\ \{\mathbf{dif}([\mathit{guard}]\ \mathit{exp}_i)\} \tag{9}$$

$$\mathbf{dif}([\mathit{guard}]\ \bigwedge_i \mathit{exp}_i) \ \Rightarrow\ \{\mathbf{dif}([\mathit{guard} \vee \bigvee_{j \neq i} \neg \mathit{exp}_j]\ \mathit{exp}_i)\} \tag{10}$$

$$\mathbf{dif}([\mathit{guard}]\ \mathit{act}) \ \Rightarrow\ [\mathit{guard}]\ \mathit{act} \tag{11}$$

where *guard* and exp_i are Boolean expressions of actions. Diffusion transforms the guarded expression example above to

$$[\mathit{guard} \vee \neg \mathit{payM}]\ \mathit{payP}$$
$$[\mathit{guard} \vee \neg \mathit{payP}]\ \mathit{payM}$$

Both actions can still occur after the guard becomes true, so it allows at least all of the runs allowed in the naïve approach. Additionally, the first action to fire can fire at any time. If *payP* fires before *payM*, then $\neg$*payM* is true when *payP* fires, so *payP* can fire at any time. Diffusion thus covers possibilities that the naïve approach omits.

5.3 Intuition: Collection

Diffusion can generate multiple guarded statements for the same action, but MCMAS requires a single guarded statement for each action. Therefore, we introduce the *collection* function **col** that converts a set of guarded statements back to canonical form, where there is a single guarded statement for each action. Consider each individual, input action. Here, **col** collects potentially multiple guarded statements for that action in its input, and generates a single guarded statement for that action in its output. The output guard for the action is the conjunction of all the input guards. If an action appears in only one guarded statement in the input, that guarded statement appears unmodified in the output.

$$\mathbf{col}(\{\ [guard_i]\ act_i\ \}) \ \Rightarrow\ \{[\bigwedge_j guard_j]\ act_i \quad | \, act_j = act_i\} \tag{12}$$

Consider a partial protocol containing these two guarded statements. Since the messages overlap on action *ship*, condition *freeCoupon* ensures the no overlap constraint of Definition 4.1.

$$[orderMsg \wedge \neg freeCoupon]\ paidShipMsg\ \textsf{means}\ \{bill, ship\};$$
$$[orderMsg \wedge freeCoupon]\ freeShipMsg\ \textsf{means}\ \{ship\};$$

Diffusion transforms those to the following three guarded statements.

$$[(orderMsg \wedge \neg freeCoupon) \vee \neg ship]\ bill$$
$$[(orderMsg \wedge \neg freeCoupon) \vee \neg bill]\ ship$$
$$[orderMsg \wedge freeCoupon]\ ship$$

Collection merges the two statements for *ship*, giving these two, final guarded statements.

$$[(orderMsg \wedge \neg freeCoupon) \vee \neg ship]\ bill$$
$$[(orderMsg \wedge freeCoupon \wedge \neg bill]\ ship$$

Figure 7(a) schematically shows how the foregoing developments of decomposition, diffusion, collection, and embedding combine together to check whether one protocol refines another. However, by themselves, they do not address the fact that different protocols can be written at different layers of abstraction.

$$\begin{tikzcd}
\text{super-gMsg} \arrow[r, "\textbf{means}"] & \text{super-gExp} \arrow[r, "\textbf{dif}"] & \text{super-gAct} \arrow[r, "\textbf{col}"] & \text{super-gAct} \arrow[d, dash, "\textbf{emb}"'] \\
\text{sub-gMsg} \arrow[r, "\textbf{means}"'] & \text{sub-gExp} \arrow[r, "\textbf{dif}"'] & \text{sub-gAct} \arrow[r, "\textbf{col}"'] & \text{sub-gAct}
\end{tikzcd}$$

(a) Decomposition, diffusion, collection, and run embedding

$$\begin{tikzcd}
\text{super-gMsg} \arrow[r, "\textbf{means}"] & \text{super-gExp} \arrow[r, "\textbf{map}"] & \text{super-gExp} \arrow[r, "\textbf{dif}"] & \text{super-gAct} \arrow[r, "\textbf{col}"] & \text{super-gAct} \arrow[d, dash, "\textbf{emb}"'] \\
\text{sub-gMsg} \arrow[rr, "\textbf{means}"'] & & \text{sub-gExp} \arrow[r, "\textbf{dif}"'] & \text{sub-gAct} \arrow[r, "\textbf{col}"'] & \text{sub-gAct}
\end{tikzcd}$$

(b) Adds mapping between layers of abstraction

Fig. 7. Protocol refinement defined as transformations (horizontal lines) and comparisons (vertical lines) between protocols of guarded messages (gMsg), guarded action expressions (gExp), and guarded actions (gAct). In each subfigure, the top row refers to a superprotocol and the bottom row to a subprotocol.

5.4 Accommodating Abstraction Mapping

Since superprotocols represent higher-level abstractions than subprotocols, comparing protocols must address differences in levels of abstraction. 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 compare them.

An important type of abstraction difference is the introduction of intermediaries or middlemen in lower-level abstractions. 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-level abstractions. 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), and one super-commitment could be fulfilled through multiple sub-commitments and their intermediate sub-debtors.

We say one protocol refines another protocol *under a given mapping*, because mapping functions are an essential element for protocol refinement, and must be an explicit input. A subprotocol might refine a superprotocol under one mapping, but not under a different mapping. Our approach does not determine whether it is impossible for one protocol to refine another protocol under any possible mapping.

A mapping expresses how terms in a putative superprotocol map to expressions in a putative subprotocol. The *mapping* function **map** converts guarded action expressions written with high-level abstractions x_i in a putative superprotocol, to expressions e_i of low-level terms in a putative subprotocol. (Below, $\langle\langle x_i \mapsto e_i \rangle\rangle$ is a mapping assertion.)

$$\mathbf{map}([\mathit{guard}]\ \mathit{exp}) \ \Rightarrow\ [\mathit{guard}\langle\langle x_i \mapsto e_i\rangle\rangle]\ \mathit{exp}\langle\langle x_i \mapsto e_i\rangle\rangle$$

5.5 Verifying Refinement: Summary

Figure 7(b) schematically shows the transformations and comparison required to demonstrate protocol refinement. In both subfigures, horizontal lines show the transformations of a single protocol: decomposition (**means**), mapping (**map**), diffusion (**dif**), and collection (**col**). Vertical lines show the comparison between two protocols: run embedding (**emb**). The nodes in the figure show how guarded messages (gMsg) are transformed first to guarded action expressions (gExp), and then to guarded actions (gAct). In each subfigure, the top row refers to a superprotocol and the bottom row refers to a subprotocol.

5.6 Generating CTL Formulae for Verification

MCMAS checks whether an interpreted system model satisfies specified CTL formulae. In this section, we describe how Proton expresses conditions for commitment resolution, overlapping messages, serial composition, and commitment covering as CTL formulae. All such formulae must be satisfied for protocol refinement to hold.

5.6.1 *Verify Run Embedding by Checking Guards.* Protocol comparison is fundamentally based on run embedding. Run embedding means, at every state, if the subprotocol can perform an action then the superprotocol must also be able to perform that action. That is, when an action's sub-guard is true, its super-guard must also be true. Since run embedding ignores actions not in the superprotocol, Proton generates CTL formulae for all actions that result from mapping all super-actions ($\forall a \in M(\mathcal{A}_{\mathit{super}})$):

$$\mathbf{AG}(a.\mathit{sub\text{-}guard} \rightarrow a.\mathit{super\text{-}guard}) \tag{13}$$

5.6.2 *Verify that Messages Do Not Overlap.* So that every action a in a sub-run can be uniquely associated with a message, we verify the no overlap constraint of Definition 4.1.

For every pair of guarded statements gs_1 and gs_2 that share a common action meaning a,

$$\mathbf{AG}(\neg(gs_1.\mathit{guard} \wedge gs_2.\mathit{guard})) \tag{14}$$

5.6.3 *Verify that Detached Commitments Eventually Resolve.* We require each detached commitment must eventually resolve in every correct protocol enactment. We employ model checker fairness constraints (expressions that must be true infinitely often on any run) to eliminate sub-runs in which the sub-roles fail to act properly and resolve their detached commitments. Doing so restricts our verification to correct enactments of the given protocols, thus avoiding false negatives due to incorrect enactments.

$$\mathbf{Fairness}\ \mathsf{C}_{sub}.\mathit{status} \neq \mathit{detached} \tag{15}$$

The states of super-commitments can be inferred from the states of sub-commitments.

5.6.4 *Verify Commitment Covering.* The truth or falsity of a statement in an unreachable state has no bearing on the enactment of a protocol, so we can replace $a \models b$ statements by the CTL formula $\mathbf{AG}(a \rightarrow b)$. Doing so enables us to use the model checker to verify commitment covering, which would otherwise need to be handled separately, as indeed it was in a previous version of Proton.

Verifying one commitment covers another under map M, $\mathsf{C}_W \leq_M \mathsf{C}_S$, is done in two parts. First, the preprocessor verifies the debtor and creditor conditions (Equations 5–6). Second, the model checker verifies the antecedent and consequent conditions (Equations 7–8) hold in all (reachable) states with the CTL formulae

$$\mathbf{AG}(M(\mathsf{C}_W.ant) \rightarrow \mathsf{C}_S.ant) \tag{16}$$

$$\mathbf{AG}(\mathsf{C}_S.csq \rightarrow M(\mathsf{C}_W.csq)) \tag{17}$$

5.6.5 *Verify Serial Compositions are Well Defined.* For serial compositions $\mathsf{C}_\oplus = \mathsf{C}_A \oplus \mathsf{C}_B$, the model checker verifies the well-definedness condition holds in all (reachable) states on all paths with the CTL formula

$$\mathbf{AG}(\mathsf{C}_A.csq \rightarrow \mathsf{C}_B.ant) \tag{18}$$

6. TOOLING, DETAILED EXAMPLES, AND EXPERIMENTAL RESULTS

In this section, we pull together the many elements: commitments, serial composition of commitments, and commitment covering; the example payment and order protocols, and various mappings between them; and the formal definitions. We concretely demonstrate how *PayViaMM* refines, or fails to refine, *Pay* under various mappings.

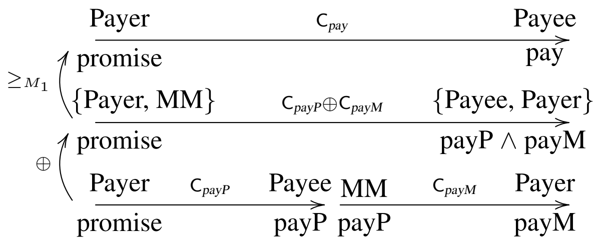
Proton verifies protocol refinement using the process flow as shown in Figure 5 and the pseudocode for **refines**(super, map, sub) shown in Listing 7. The inputs P and Q are protocols, which in our syntax are in terms of guarded messages. The first lines of the algorithm transform these into protocols expressed in terms of guarded actions. Proton generates an interpreted system model from the guarded actions of the subprotocol. There is one MCMAS agent definition for each sub-role, and the state of the sub-elements (propositions and commitments) are MCMAS state variables. The MCMAS model checker then simulates the subprotocol's actions. Because each super-element is mapped to an expression of sub-elements, the superprotocol's state can be inferred from the subprotocol's state. Refinement requires the model of the subprotocol to satisfy a set of CTL formulae. If all

Listing 7 Calculate **refines**(P, M, Q)

```
$Q_{gMsg} = Q$                                   ▷ Input Q is a protocol of guarded messages
$P_{gMsg} = P$                                   ▷ Input P is a protocol of guarded messages
$Q_{gAct} = \mathbf{col}(\mathbf{dif}(\mathbf{means}(Q_{gMsg})))$          ▷ protocol of guarded sub-actions
$P_{gAct} = \mathbf{col}(\mathbf{dif}(\mathbf{map}_M(\mathbf{means}(P_{gMsg}))))$    ▷ protocol of guarded sub-actions
model = genModel($Q_{gAct}$)                     ▷ generate ISPL model
for all $act_P \in P_{gAct}.actions$ do          ▷ For all super-actions
    genCTL($\mathbf{AG}(act_P.sub\text{-}guard \rightarrow act_P.super\text{-}guard)$)
end for
for all $\mathsf{C}_Q \in Q_{gAct}.\mathcal{C}$ do                 ▷ For all sub-commitments
    genFairness($\mathsf{C}_Q.status \neq detached$)
end for
for all $\mathsf{C}_P \in P_{gAct}.\mathcal{C}$ do                 ▷ For all super-commitments
    $\mathsf{C}_Q = \mathsf{C}_P.coveringCommitment$
    genCTL($\mathbf{AG}(M(\mathsf{C}_P.ant) \rightarrow \mathsf{C}_Q.ant)$)
    genCTL($\mathbf{AG}(\mathsf{C}_Q.csq \rightarrow M(\mathsf{C}_P.csq))$)
end for
for all $\mathsf{C}_A \oplus \mathsf{C}_B$ do                       ▷ For all serial composition pairs
    genCTL($\mathbf{AG}(\mathsf{C}_A.csq \rightarrow \mathsf{C}_B.ant)$)
end for
for all overlapping message pairs $m_1$ and $m_2$ in P and Q do
    genCTL($\mathbf{AG}(\neg(m_1.guard \wedge m_2.guard))$)
end for
ctl = all generated CTL formulae
return MCMAS(model, ctl)                         ▷ Are all CTL formulae satisfiable?
```

formulae are true, the subprotocol refines the superprotocol.

$$\mathsf{C}_{pay} \;\mapsto\; \mathsf{C}_{payP} \oplus \mathsf{C}_{payM} \qquad \text{if } \mathsf{C}_{pay} \leq_{M_1} \mathsf{C}_{payP} \oplus \mathsf{C}_{payM}$$



We now check whether *PayViaMM* refines *Pay* under map M_1. Using the commitment diagrams from Section 2.6, this diagram demonstrates commitment C_{pay} from *Pay* is covered by the serial composition of C_{payP} and C_{payM} from *PayViaMM* under mapping M_1 in Listing 3. The bottom-left arrow states that if promise becomes true, Payer commits to making payP true. The bottom-right arrow states that if payP becomes true, MiddleMan commits to making payM true. In the serial composition (middle arrow), if promise becomes true, then Payer and MiddleMan (severally) commit to making both payP and payM true. The well-definedness condition ensures that the discharge of the first commitment entails the antecedent of the second commitment, thus detaching it. The Payee and Payer are creditors of the input commitments, and are thus creditors of the serial composition. The serial composition covers the commitment in *Pay* (top arrow).

Proton generates the following eight CTL formulae to verify *PayViaMM* refines *Pay* under M_1. Equation 13, which verifies whether sub-guards imply super-guards, for actions *promise*, pay_P, pay_M, $\textsc{create}(\mathsf{C}_{\mathrm{payP}})$, and $\textsc{create}(\mathsf{C}_{\mathrm{payM}})$ generates Equations 19–23, respectively. Equation 24 verifies that $\mathsf{C}_{\mathrm{payP}} \oplus \mathsf{C}_{\mathrm{payM}}$ is well defined. Equations 7–8, which verify the antecedent and consequent conditions of $\mathsf{C}_{\mathrm{payP}} \oplus \mathsf{C}_{\mathrm{payM}}$ covers $\mathsf{C}_{\mathrm{pay}}$, generate Equation 25–26, respectively (the debtor and creditor conditions in Equations 5–6 are checked directly by the Proton preprocessor, not by MCMAS).

$$\mathbf{AG}(\top \rightarrow \top) \tag{19}$$

$$\mathbf{AG}(\mathit{promise} \wedge \textsc{create}(\mathsf{C}_{\mathrm{payP}}) \wedge \textsc{create}(\mathsf{C}_{\mathrm{payM}}) \rightarrow (\mathit{promise} \wedge \textsc{create}(\mathsf{C}_{\mathrm{payP}}) \wedge \textsc{create}(\mathsf{C}_{\mathrm{payM}})) \vee \neg \mathit{pay}_M) \tag{20}$$

$$\mathbf{AG}(\top \rightarrow (\mathit{promise} \wedge \textsc{create}(\mathsf{C}_{\mathrm{payP}}) \wedge \textsc{create}(\mathsf{C}_{\mathrm{payM}})) \vee \neg \mathit{pay}_P) \tag{21}$$

$$\mathbf{AG}(\top \rightarrow \top) \tag{22}$$

$$\mathbf{AG}(\top \rightarrow \top) \tag{23}$$

$$\mathbf{AG}(\mathit{pay}_P \rightarrow \mathit{pay}_P) \tag{24}$$

$$\mathbf{AG}(\mathit{promise} \rightarrow \mathit{promise}) \tag{25}$$

$$\mathbf{AG}(\mathit{pay}_P \wedge \mathit{pay}_M \rightarrow \mathit{pay}_P \wedge \mathit{pay}_M) \tag{26}$$

All of the above formulae are obviously true, except Equation 21, which requires further consideration. Equation 21 can be rewritten $\mathbf{AG}(\mathit{pay}_P \rightarrow \mathit{promise} \wedge \textsc{create}(\mathsf{C}_{\mathrm{payP}}) \wedge \textsc{create}(\mathsf{C}_{\mathrm{payM}}))$. It is true because the progression of the model, controlled by message guards, ensures pay_P becomes true only after *promiseMsg*. MCMAS verifies each generated CTL formula holds in the model, so *PayViaMM* refines *Pay* under map M_1.

Proton generates exactly the same input to MCMAS when checking whether *PayViaMM* refines *Pay* under map M_2 or map M_3, because the subprotocol models are derived from exactly the same *PayViaMM* protocol, the superprotocol *Pay* contains exactly the same propositional and commitment super-elements, and exactly the same CTL conditions must be checked. Therefore, *PayViaMM* refines *Pay* under both maps M_2 and M_3.

$$\mathbf{AG}(\mathit{pay}_M \rightarrow \mathit{promise}) \tag{27}$$

$$\mathbf{AG}(\mathit{promise} \rightarrow \mathit{pay}_P) \tag{28}$$

$$\mathbf{AG}(\mathit{promise} \wedge \mathit{pay}_P \wedge \mathit{pay}_M \rightarrow \mathit{pay}_P \wedge \mathit{pay}_M) \tag{29}$$

Proton correctly reports failures. Proton generates these CTL formulae when checking whether *PayViaMM* refines *Pay* under mapping B_1 in Listing 6. Recall that B_1 maps the super-commitment to a serial composition in the wrong order. Equations 19–23 are also generated, and all hold in the model. Equations 29 obviously holds. But Equation 27 does not hold because pay_M has a true guard in Listing 2, so the Middleman can send pay_M at any time, even before *promise*. Equation 28 comes from the antecedent of $\mathsf{C}_{\mathrm{pay}}$ (which is *promise*) and the antecedent of $\mathsf{C}_{\mathrm{payM}} \oplus \mathsf{C}_{\mathrm{payP}}$ (which is pay_P). In the states between the Payer promising and the Payer actually paying, the formula does not hold, meaning that $\mathsf{C}_{\mathrm{pay}}$ can become detached without $\mathsf{C}_{\mathrm{payM}} \oplus \mathsf{C}_{\mathrm{payP}}$ also becoming detached. The result is $\mathsf{C}_{\mathrm{pay}}$ is not covered by $\mathsf{C}_{\mathrm{payM}} \oplus \mathsf{C}_{\mathrm{payP}}$. MCMAS correctly reports these two formulae as false in the model, and *PayViaMM* does *not* refine *Pay* under mapping B_1.

Superprotocol	*Subprotocol*	*Map*	*refines*	$\oplus$	*cover*	*formulae*	*time*
Pay	*PayViaSpouse*	M_1	*Yes*	0	1	5	0.28
Pay	*FullPay*	M_1	*Yes*	0	1	5	0.28
Pay	*PayViaMM*	M_1	*Yes*	1	1	8	0.34
Pay	*PayViaMM*	M_2	*Yes*	1	1	8	0.35
Pay	*PayViaMM*	M_3	*Yes*	1	1	8	0.36
Pay	*PayViaMM*	B_1	*No*	1	1	8	0.36
Pay	*PayViaCheck*	M_1	*Yes*	2	1	12	0.41
Pay	*PayViaCredit*	M_1	*Yes*	2	1	12	0.41
Pay	*OrderPayShip*	M_1	*Yes*	1	1	7	0.34
Pay	*OrderPayViaMMShip*	M_1	*Yes*	2	1	10	0.41
PayViaMM	*PayViaCheck*	M_1	*Yes*	3	2	15	0.41
PayViaMM	*PayViaCredit*	M_1	*Yes*	3	2	15	0.46
PayViaMM	*OrderPayViaMMShip*	M_1	*Yes*	0	2	9	0.40
OrderPayShip	*OrderPayViaMMShip*	M_1	*Yes*	1	2	14	0.43
OrderPayShip	*NetBill*$_2$	M_1	*Yes*	0	2	11	0.39
OrderPayViaMMShip	*NetBill*$_3$	M_1	*Yes*	0	3	15	0.45

Table I. Information about each demonstrated Refinement. The columns are: the name of the superprotocol; the name of the subprotocol; the name of the map; whether subprotocol refines superprotocol under map; $\oplus$ 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; and *time* is the elapsed time (in seconds) for refinement verification. Run on a MacBook Pro computer with 2.3 GHz Intel Core i7 processor and 8 GB memory, using Java 1.6.0 and MCMAS v1.0.2.

As in Mallya and Singh's [2007] approach, an interesting consequence of our treatment of refinement is that aggregation functions like refinement. For example, consider a protocol *OrderPayShip*. Because all enactments of *OrderPayShip* necessarily include an enactment of *Pay*, *OrderPayShip* is naturally a refinement of *Pay*. The foregoing coheres with the notion of subtype in object-oriented programming.

Proton verified all the refinements shown in Figure 1 and Table I. The first three columns are the superprotocol, subprotocol, and mapping, respectively. *OrderPayShip* is identical to the first NetBill scenario described by Mallya and Singh [2007]. NetBill$_2$ and NetBill$_3$ are scenarios 2 and 3 in the same paper. *PayBySpouse* is a new, simple, payment protocol where one person promises and then his or her spouse pays. *FullPay* is similar to *Pay*, but exercises all the commitment operations: CREATE, TRANSFER, RELEASE, and CANCEL.

Each superprotocol-subprotocol pair has a unique set of mapping names, so M_1 for *Pay* and *PayViaMM* is a different mapping than M_1 for *Pay* and *PayViaCheck*. Mappings M_1, M_2, M_3, and B_1 for *Pay* and *PayViaMM* are shown, respectively, in Listings 3, 4, 5, and 6. Column *refines* is Yes if subprotocol refines superprotocol under map; column $\oplus$ is the number of serial compositions in the mapping; column *covers* is the number of commitment covering checks; column *formulae* is the number of CTL formulae verified by the model checker; and column *time* is the elapsed time in seconds for the complete refinement verification, including Proton preprocessing and MCMAS model checking.

7. CORRECTNESS OF OUR REFINEMENT VERIFICATION METHOD

We now establish the correctness of the verification method of Section 5 with respect to the formal definitions of Section 4. Theorem 7.1 is our main correctness result: it ties the algorithm of Listing 7 with Definition 4.9. An interesting subtlety is that whereas Definition 4.9 "maps up," enriching the states in sub-runs, the algorithm "maps down," expanding expressions and ignoring elements not in $\mathcal{A}_P$.

THEOREM 7.1. *Let P and Q be two protocols, and let M be a mapping from P to Q. Then, **refines**(P, M, Q) returns true if and only Q refines P under M.*

PROOF. Let $\mathcal{I}_Q$ be the Proton model generated from Q. Define mappings M_Q = **means**(Q) and $M_P = M($**means**$(P))$.

$\Rightarrow$ Assume **refines**(P,M,Q). Then all of the CTL formulae in Listing 7 are true. Because of Line 7, all sub-guards imply their super-guards. Because of the fairness condition at Line 10, all detached commitments eventually resolve. Because of Lines 14–15, all commitment coverings are valid. Because of Line 18, all serial compositions are valid. Because of Line 21, no guarded statement pairs overlap.

Let π_Q be a run in **runs**(Q). By Theorem A.7, there is a run $\pi'_Q \in$ **runs**(**col**(**dif**$(M_Q(Q))$)). By Theorem A.2, there exists a run $\pi'_P \in$ **runs**(**col**(**dif**$(M_P(P))$)) And, by Theorem A.7, there exists a run $\pi_P \in$ **runs**(P). Therefore, for every π_Q there is a π_P and Q refines P under M by Definition 4.9.

$\Leftarrow$ Assume Q refines P under M. For any $\pi_Q \in$ **runs**(Q) there exists a run $\pi_P \in$ **runs**(P) such that **emb**$(M(\pi_Q), \pi_P)$ by Definition 4.9. By Theorem A.7, there exists a $\pi'_Q \in$ **runs**(**col**(**dif**$(M_Q(Q))$)) and a $\pi'_P \in$ **runs**(**col**(**dif**$(M_P(P))$)). By Theorem A.2, **emb**$(M(\pi_Q), \pi_P)$ implies $(\forall a_i \in \mathcal{A}_P : \mathcal{I}, g_0 \models \mathbf{AG}(a_i.\textit{sub-guard} \rightarrow a_i.\textit{super-guard}))$, which implies the CTL formulae at Line 7 are true.

Because all detached commitments must eventually resolve, the fairness formulae at Line 10 are true. Because all commitment coverings must be valid, the formulae at Lines 14-15 are true. Because all serial compositions must be valid, the formulae at Line 18 are true. Because protocols must be well defined, the formulae at Line 21 are true. Because all of the CTL formulae evaluate to true, **refines**(P,M,Q) returns true. □

8. DISCUSSION

Commitments support finer guard granularity than propositions can. A proposition divides time into two stages: before and after it holds. A commitment divides time into four stages: null, active and conditional, active and detached, and resolved.

$$\textit{null} \xrightarrow{\text{CREATE}} \textit{cond} \xrightarrow{ant} \textit{detached} \xrightarrow{csq} \textit{resolved}$$

Rather than waiting for final resolution, a protocol can make progress sooner if an action's guard is enabled after one of the first three stages. Commitments increase protocol flexibility, because guards can specify earlier stages.

For example, suppose the Buyer role in *OrderPayShip* decides whether to pay based on the state of proposition ship. Since ship has only two stages, the role's decision can only be "all" (*ship* complete) or "nothing" (*ship* not complete). The "all" choice is represented by the guarded statement [*ship*] *pay*, and the "nothing" choice is represented by [*true*] *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 <u>commits</u> to make *ship* true.

$$[\text{CREATE}(\mathsf{C}_{\text{Seller,Buyer}}(pay, ship))]\ \textit{pay}$$

Incorporating commitments can improve flexibility over traditional protocol frameworks.

A necessary prerequisite of employing protocols is that the participants of a service engagement agree on the format and meanings of the messages they exchange. Note that such agreement is unavoidable: it is just that in today's practice the meanings are not expressed clearly and explicitly and any agreements are hardcoded in implementations.

Our definition of protocol refinement does not mean agents that can participate in a superprotocol can necessarily participate unchanged in a subprotocol. 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 handle the additional messages required in paying via check or credit card.

8.1 Relevant Literature

Proton is the first approach for protocol refinement that incorporates mapping super-elements to expressions of sub-elements. Proton supports mapping super-roles to sets of sub-roles, super-propositions to Boolean expressions of sub-propositions, and super-commitments to serial compositions of chains of sub-commitments.

Mallya and Singh [2007] propose a definition of protocol refinement (which they call subsumption) that compares the order of state pairs in state runs. For every pair of states in the superprotocol run, there must be *some* matching pair of states in the subprotocol run with the same order. However, this definition can create false positives when multiple state pairs in the super-run each match the same state pair in the sub-run or when one super-state matches different sub-states. For example, all state pairs in the super-run $\langle 1, 2, 3 \rangle$ have matching state pairs in the sub-run $\langle 2, 1, 3, 2 \rangle$ even though the two runs are very different. Our definition compares runs step-by-step and thereby avoids the above problems, even if protocol looping is allowed.

Our definition of commitment covering is an extension of *commitment strength* as defined by Chopra and Singh [2009], who identify the basic requirements in Equations 7 and 8. We extend Chopra and Singh's definition with the role requirements in Equations 5 and 6, and we allow commitments to be at different levels of abstraction by including an abstraction mapping function.

Singh [2008] states rules for commitment chaining similar to those proposed here, but does not directly state a rule for stronger consequents, and does not directly state a rule similar to serial composition. The concrete commitment created by serial composition provides a midpoint in commitment reasoning, and can potentially make the comparison of commitments across protocols more explicit.

When we say a group of debtors are jointly and severally responsible for eventually making the consequent true, we mean this in the sense of Rescher's [1998] *legal* responsibility where "individual agents are responsible only for their own individual acts." We do not mean Rescher's notion of legal responsibility where the group as a whole becomes a legal person, nor his notion of *moral* responsibility where intentions are crucial (intentions are absent from our formulation). In Norman and Reed [2002], group imperatives can be addressed *distributively* (as a list of individuals) or as a *collective*. In both cases, group imperatives imply more than just a collection of individual imperatives. While joint and several responsibility is similar to distributive responsibility, because only one member of the group is required to act, it is different, because joint and several responsibility is only a summary of individual responsibilities, and does not impose additional responsibilities on roles that are members of a group.

Our work on protocols builds on the fundamental intuition that protocol states can be effectively characterized in terms of the commitments of the participants, and that such characterization can be used as a basis for correct enactments and for further reasoning.

The earliest works that developed the above theme include the commitment machines approach of Yolum and Singh [2002] for business protocols and McBurney and Parsons' [2002] framework for sequencing multiple dialogue games, allowing one dialogue game to be embedded inside another partially completed dialogue game.

We do not propose specific, desirable properties of protocols, but others have. Yolum [2007], Singh and Chopra [2010], and El-Menshawy et al. [2010] describe desirable properties of protocols in general and commitment protocols in particular, including fairness, safety, liveness, operability, and transparency.

We use Boolean guards to constrain actions, but other representations are possible. Baldoni et al. [2010] proposed constraints based on regulative specifications. Regulative specifications constrain the execution flow using special-purpose operators on state values. Gabbay [1987] proposes using past-temporal expressions for controlling when actions can occur and future-temporal expressions for controlling which actions must occur in the future. Past-temporal expressions are more expressive than our guard expressions.

8.2 Directions for Future Research

Constructing a mapping function from a superprotocol to a subprotocol 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; 2007] proposed a methodology for the related task of designing commitment-based protocols. Some of these ideas could be valuably adapted into a future commitment mapping methodology. The approach begins with an easily understood, but not exhaustive, set of sequence diagrams, and then specifies specific steps to generalize the protocol and expand its set of runs.

Our model and formulation of refinement respects intuitions similar to those of Mallya and Singh [2007]. Mallya and Singh describe protocol composition and establish results relating composition and refinement. We expect that similar results would apply in our framework, and plan to formalize and reason about composition in future work.

Model checkers have been extended to handle epistemic and strategic modal operators [Alur et al. 2002; Fagin et al. 1995]. We have begun investigating the inclusion of such concepts into our definitions. Building on top of a model checker that already handles those concepts, such as MCMAS, will simplify our future extensions. Another important enhancement would be to expand the class of protocols to those that support iteration.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for valuable comments and Amit Chopra for his suggestion of describing complex messages as sets of primitive actions.

REFERENCES

ALUR, R., HENZINGER, T. A., AND KUPFERMAN, O. 2002. Alternating-time temporal logic. *Journal of the ACM 49,* 5 (Sept.), 672–713.

BALDONI, M., BAROGLIO, C., AND MARENGO, E. 2010. Constraints among commitments: Regulative specification of interaction protocols. In *Proceedings of the International Workshop on Agent Communication*. Toronto, 10–29.

CHOPRA, A. K. AND SINGH, M. P. 2009. Multiagent commitment alignment. In *Proceedings of the 8th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Budapest, 937–944.

COHEN, M., DAM, M., LOMUSCIO, A., AND RUSSO, F. 2009. Abstraction in model checking multi-agent systems. In *Proceedings of the 8th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Budapest, 945–952.

EL-MENSHAWY, M., BENTAHAR, J., WAN, W., AND DSSOULI, R. 2010. Verifying conformance of commitment protocols via symbolic model checking. In *Proceedings of the International Workshop on Agent Communication*. Toronto, 53–72.

FAGIN, R., HALPERN, J. Y., MOSES, Y., AND VARDI, M. Y. 1995. *Reasoning about Knowledge*. MIT Press, Cambridge, MA, USA.

GABBAY, D. 1987. The declarative past and imperative future: Executable temporal logic for interactive systems. In *Proceedings of the Colloquium on Temporal Logic in Specification*. LNCS, vol. 398. Springer-Verlag, 409–448.

LOMUSCIO, A., QU, H., AND RAIMONDI, F. 2009. MCMAS: A model checker for the verification of multi-agent systems. In *Proceedings of the International Conference on Computer Aided Verification*. LNCS. 682–688.

LOMUSCIO, A. AND RAIMONDI, F. 2006. Model checking knowledge, strategies, and games in multi-agent systems. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. ACM, Hakodate, Japan, 161–168.

MALLYA, A. U. AND SINGH, M. P. 2007. An algebra for commitment protocols. *Journal of Autonomous Agents and Multi-Agent Systems 14,* 2 (Apr.), 143–163.

MALONE, T. W., CROWSTON, K., AND HERMAN, G. A., Eds. 2003. *Organizing Business Knowledge: The MIT Process Handbook*. MIT Press, Cambridge, MA.

MCBURNEY, P. AND PARSONS, S. 2002. Games that agents play: A formal framework for dialogues between autonomous agents. *Journal of Logic, Language and Information 11*, 315–334.

NORMAN, T. J. AND REED, C. 2002. Group delegation and responsibility. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems*. AAMAS. ACM, Bologna, 491–498.

RESCHER, N. 1998. Collective responsibility. *Journal of Social Philosophy 29,* 3 (Dec.), 46–58.

SINGH, M. P. 1999. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law 7,* 1 (Mar.), 97–113.

SINGH, M. P. 2008. Semantical considerations on dialectical and practical commitments. In *Proceedings of the 23rd Conference on Artificial Intelligence*. AAAI Press, 176–181.

SINGH, M. P. AND CHOPRA, A. K. 2010. Correctness properties for multiagent systems. In *Proceedings of the 6th AAMAS Workshop on Declarative Agent Languages and Technologies (DALT 2009)*. LNAI, vol. 5948. Springer, Budapest, 192–207.

WINIKOFF, M. 2006. Designing commitment-based agent interactions. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology*. IEEE Computer Society, Washington, DC, USA, 363–370.

WINIKOFF, M. 2007. Implementing commitment-based interactions. In *Proceedings of the 6th International Conference on Autonomous Agents and Multiagent Systems*. ACM, Honolulu, 1–8.

YOLUM, P. 2007. Design time analysis of multiagent protocols. *Data and Knowledge Engineering Journal 63*, 137–154.

YOLUM, P. AND SINGH, M. P. 2002. Commitment machines. In *Proceedings of the 8th International Workshop on Agent Theories, Architectures, and Languages (ATAL 2001)*. LNAI, vol. 2333. Springer, Seattle, 235–247.

Received August 2010; revised January 2011, May 2011

THIS DOCUMENT IS THE ONLINE-ONLY APPENDIX TO:

Formalizing and Verifying Protocol Refinements

Scott N. Gerard and Munindar P. Singh
North Carolina State University

DRAFT, Vol. V, No. N, July 2011, Pages 1–28.

A. SUPPLEMENTARY THEOREMS

The first theorem connects interpreted system models with our definitions.

THEOREM A.1. *Let $P = \langle \mathcal{R}, \mathcal{M}, \mathcal{C}, \mathcal{A}, \mathcal{S}, \mathcal{G} \rangle$ be a protocol and let $\mathcal{I}$ be its Proton model. A run is allowed by a Proton model $\mathcal{I}$ (Definition 4.2) if and only if it is a well-defined run (Definition 4.3).*

PROOF SKETCH. Proton models allow interleaved, but not concurrent, messages. At each step, the environment schedules some role $r \in \mathcal{R}$. Role r chooses some enabled message $m \in \mathit{Act}^r$, and the ISPL joint action Act is equal to m,

Runs for both ISPL and Definition 4.3 begin in state $\emptyset$. At every step in a run, a message is enabled in ISPL by local strategy AP^r if and only if that message's guard is enabled. Therefore, a message can be appended to an ISPL run if and only if it can be appended to a Proton run. □

The next theorem shows embedding from Definition 4.8 is equivalent to the model checker verifying guards with Equation 13. The idea behind this theorem is that it assumes the two protocols are already mapped, so the guards of the superprotocol can be evaluated in the Proton model generated from the subprotocol.

Let π^i denote the path consisting of the first i steps of π, let $|\pi|$ be the length of path π, and let $\pi + \langle a, s \rangle$ be path π extended by action a resulting in new state s.

THEOREM A.2. *Let P and Q be two protocols, and M a mapping between them. Let $\mathcal{I}_Q$ be the Proton model for Q as specified in Definition 4.2. Let $\pi_P = \langle p_0, b_1, p_1, \ldots \rangle$, and $\pi_Q = \langle q_0, c_1, q_1, \ldots \rangle$. Then, for all runs $\pi_Q \in$ **runs**(Q), there exists a run $\pi_P \in$ **runs**(P) such that **emb**$(M(\pi_Q), \pi_P)$ if and only if $(\forall a_i \in \mathcal{A}_P : \mathcal{I}, g \models \mathbf{AG}(a_i.\text{sub-guard} \rightarrow a_i.\text{super-guard}))$*

PROOF. From Theorem A.1, checking for well-defined runs is the same as checking the interpreted system model. Figure 8 diagrams the relationships between entities in π_P and π_Q.

For simplicity, let *EMBED* be **emb**$(M(\pi_Q), \pi_P)$, and let *GUARD* be $(\forall a_i \in \mathcal{A}_P : \mathcal{I}, g \models \mathbf{AG}(a_i.\textit{sub-guard} \rightarrow a_i.\textit{super-guard}))$.

$$\begin{tikzcd}
\pi_P : \arrow[r, "b_i"] & p_i \arrow[rrrr, "b_{i+1}"] \arrow[d, "\tau"'] & & & & p_{i+1} \arrow[r, "b_{i+2}"] \arrow[d, "\tau"'] & {} \\
\pi_Q : \arrow[r, "c_{\tau(i)}"] & q_{\tau(i)} \arrow[r, "c_{\tau(i)+1}"] & q_{\tau(i)+1} \arrow[r, "c_{\tau(i)+2}"] & \cdots \arrow[r, "c_{\tau(i+1)-1}"] & q_{\tau(i+1)-1} \arrow[r, "c_{\tau(i+1)}"] & q_{\tau(i+1)} \arrow[r, "c_{\tau(i+1)+1}"] & {}
\end{tikzcd}$$

Fig. 8. The mapping between entities in π_P and π_Q.

$\Rightarrow$ Assume *EMBED*. Define *GUARD*^{j} $= (\forall i : \tau(i) \leq j, a_i \in \mathcal{A}^j : \mathcal{I}, g \models \mathbf{AG}(a_i.\textit{sub-guard} \rightarrow a_i.\textit{super-guard}))$ where set $\mathcal{A}^j = \{b_k \,:\, j = \tau(i) \wedge b_k \in \pi_P^i\}$ We construct a function $\tau(i)$ and prove *GUARD*^{j} by induction on path length j in π_Q.

—Base case:
- Define $\mathcal{A}^0 = \emptyset$.
- Define $\tau(0) = 0$.
- *GUARD*^{0} is vacuously true.

—Inductive Step: Assume *GUARD*^{j}. Consider the next action c_j in π_Q.
- —Case: There are no more actions c_j. We are at the end of π_Q, $j = |\pi_Q|$, *GUARD*^{j} $=$ *GUARD*$^{|\pi_Q|}$, and induction is complete.
- —Case: $c_j.\textit{actexp} \notin \mathcal{A}_P$. This corresponds to the case $\tau(i) < j < \tau(i+1)$.
  - c_j has no effect on P.
  - $\widehat{M}(q_{\tau(i)}) = \widehat{M}(q_j)$ by Definition 4.8.
  - Define $\mathcal{A}^{j+1} = \mathcal{A}^j$ (no new a_i were added).
  - Therefore, *GUARD*$^{j+1}$ $=$ *GUARD*^{j} is true.
- —Case: $c_j \in \mathcal{A}_P$. This corresponds to the case $j = \tau(i+1)$.
  - Define $j = \tau(i+1)$.
  - $p_i = \widehat{M}(q_{\tau(i+1)-1})$ by Definition 4.8.
  - $p_{i+1} = \widehat{M}(q_{\tau(i+1)})$ by *EMBED*.
  - Since $q_{\tau(i+1)} = q_{\tau(i+1)-1} \cup c_{\tau(i+1)}.\textit{actexp}$ and $p_{i+1} = p_i \cup b_{i+1}.\textit{actexp}$, then $\widehat{M}(c_{\tau(i+1)}.\textit{actexp}) = b_{i+1}.\textit{actexp}$, and $c_{\tau(i+1)} = b_{i+1}$.
  - Let $a_{i+1} = c_{\tau(i+1)} = b_{i+1}$ be an action described by *GUARD*.
  - $q_{\tau(i+1)-1} \models c_{\tau(i+1)}.\textit{guard}$, since π_Q is well defined.
  - $q_{\tau(i+1)-1} \models a_{\tau(i+1)}.\textit{sub-guard}$.
  - $p_i \models b_{i+1}.\textit{guard}$, since π_P is well defined,
  - $p_i \models a_{i+1}.\textit{super-guard}$.
  - $\widehat{M}(q_{\tau(i+1)-1}) \models a_{i+1}.\textit{super-guard}$, by Definition 4.8.
  - $q_{\tau(i+1)-1} \models a_{i+1}.\textit{super-guard}$, since $q_i \models \widehat{M}(q_i)$ for all i.
  - Since both $a_{i+1}.\textit{sub-guard}$ and $a_{i+1}.\textit{super-guard}$ hold in state $q_{\tau(i+1)-1}$, then $a_{i+1}.\textit{sub-guard} \rightarrow a_{i+1}.\textit{super-guard}$ holds in state $q_{\tau(i+1)-1}$ for the newly added a_{i+1} on path π_Q.
  - Define $\mathcal{A}^{j+1} = \mathcal{A}^j \cup b_{i+1}$. b_{i+1} might already be a member of $\mathcal{A}^j$.
  - Therefore, *GUARD*$^{j+1}$ is true.

Since there is at least one path $\pi_Q \in \mathbf{runs}(Q)$ though every true guard, *GUARD*$^{|\pi_Q|}$ holds for all reachable states. (Note, since we do not consider all runs $\pi_P \in \mathbf{runs}(P)$, there may be states where $b_{i+1}.\textit{guard}$ is true, but $c_{\tau(i+1)}.\textit{guard}$ is not true.) Therefore, *GUARD* holds.

$\Leftarrow$ Assume *GUARD*. Define $\mathit{EMBED}^j = (\forall i : \tau(i) \leq j : \pi_P^i$ *is well defined* $\wedge$ **emb**$(M(\pi_Q^j), \pi_P^i))$. Given any π_Q, we construct a well-defined path π_P and function $\tau(i)$, and prove EMBED^j by induction on path length j in π_Q.

—Base case:
- $q_0 = \widehat{M}(q_0) = \emptyset = p_0$.
- Define $\tau(0) = 0$.
- Define $\pi_P^i = \langle p_0 \rangle$ which is well defined.
- Since $\widehat{M}(q_0) = p_0$, then **emb**$(M(\pi_Q^0), \pi_P^0) = \mathit{EMBED}^0$.

—Inductive Step: Assume EMBED^j. Then **emb**$(M(\pi_Q^{\tau(i)}), \pi_P^i)$ so that $p_i = \widehat{M}(q_{\tau(i)})$. Consider the next action c_{j+1} in π_Q.

—Case: There are no more actions c_{j+1}. $j = |\pi_Q|$, $\mathit{EMBED}^j = \mathit{EMBED}^{|\pi_Q|}$, and the induction is complete.

—Case: Action $c_{j+1} \notin \mathcal{A}_P$. This corresponds to the case $\tau(i) < j+1 < \tau(i+1)$.
- c_{j+1} does not change $\widehat{M}(q_j)$ by Definition 4.8.
- $\mathit{EMBED}^{j+1} = \mathit{EMBED}^j$ is true.

—Case: $c_{j+1} \in \mathcal{A}_P$. This corresponds to the case $j+1 = \tau(i+1)$.
- Define $\tau(i+1) = j+1$.
- Since $c_{j+1} \in \mathcal{A}_P$, $c_j.\mathit{actexp}$ is also an element in $\mathcal{A}_P$.
- Let $a_{i+1} = c_{j+1} = c_{\tau(i+1)}$ in *GUARD*, and let $b_{i+1} = c_{j+1}$ in EMBED^j, so that $a_{i+1} = c_{\tau(i+1)} = b_{i+1}$. a_{i+1}'s sub-guard is $c_{\tau(i+1)}.\mathit{guard}$ and a_{i+1}'s super-guard is $b_{i+1}.\mathit{guard}$.
- Define $p_{i+1} = p_i \cup b_{i+1}.\mathit{actexp}$.
- Define $\pi_P^{i+1} = \pi_P^i + \langle b_{i+1}, p_{i+1} \rangle$.

Show π_P^{i+1} is well defined.

(1)
- $q_{\tau(i+1)-1} \models c_{\tau(i+1)}.\mathit{guard}$, since π_Q is well defined.
- $q_{\tau(i+1)-1} \models a_{i+1}.\mathit{sub\text{-}guard}$.
- $a_{i+1}.\mathit{sub\text{-}guard} \rightarrow a_{i+1}.\mathit{super\text{-}guard}$ at state $q_{\tau(i+1)-1}$ by *GUARD*.
- $q_{\tau(i+1)-1} \models a_{i+1}.\mathit{super\text{-}guard}$.
- $\widehat{M}(q_{\tau(i+1)-1}) \models \widehat{M}(a_{i+1}.\mathit{super\text{-}guard})$.
- $\widehat{M}(q_{\tau(i+1)-1}) \models a_{i+1}.\mathit{super\text{-}guard}$, because $a_{i+1}.\mathit{super\text{-}guard}$ is expressed entirely in terms of $\mathcal{A}_P$.
- $\widehat{M}(q_{\tau(i)}) \models a_{i+1}.\mathit{super\text{-}guard}$ by Definition 4.8.
- $p_i \models a_{i+1}.\mathit{super\text{-}guard}$.
- $p_i \models b_{i+1}.\mathit{guard}$.

(2) $p_{i+1} = p_i \cup b_{i+1}.\mathit{actexp}$ by definition above.

Show **emb**$(\pi_Q^{\tau(i+1)}, \pi_P^{i+1})$.
- $p_i = \widehat{M}(q_{\tau(i+1)-1})$ by Definition 4.8 and EMBED^j.
- Since $c_{\tau(i+1)} = a_{i+1} = b_{i+1}$ are all the same action, $p_i \cup b_{i+1}.\mathit{actexp} = \widehat{M}(q_{\tau(i+1)-1}) \cup c_{\tau(i+1)}.\mathit{actexp}$.
- This reduces to $p_{i+1} = \widehat{M}(q_{\tau(i+1)})$.

Therefore, EMBED^{j+1}.

□

The next definition characterizes when an action expression $a.\mathit{actexp}$ causes $m.\mathit{actexp}$ to become true.

Definition A.3 *Decisive*. Let P be a protocol. Let e be a Boolean expression over state variables of P, and let e' be any subexpression of e. Let $t = \langle s, s' \rangle$ be any transistion between adjacent states s and s'. Therefore, $s' = s \cup e'$. e' is *decisive* for e, on transition t if and only if $s \not\models e'$, and $s' \models e'$ implies $s \not\models e \wedge s' \models e$.

State s' is where e becomes true. An expression e' is decisive for expression e on transition t if and only if, making e' true also makes or causes e to be true. Clearly, e is decisive for itself on all transitions.

The smallest possible subexpression e' or an expression e are its atomic actions. In particular, we say an action a is decisive for message m at state s exactly when expression $a.\mathit{actexp}$ is decisive for expression $m.\mathit{actexp}$ at state s.

The next theorem shows that diffusion and collection properly maintain the guards and action expressions as a message is decomposed from a protocol P to its derived protocol $P' = \textbf{col}(\textbf{dif}(\textbf{means}(M(P))))$. We prove it for mappings that contain individual actions and conjunctions, as well as mapping that contain disjunctions even though disjunction is not required by later proofs. This theorem is used between the superprotocol's super-gMsg and super-gAct, and between the subprotocol's sub-gMsg and sub-gAct in Figure 7(b).

THEOREM A.4 DIFFUSION AND COLLECTION PRESERVE GUARDS. *Let P be any protocol, and let $gs \in \mathcal{G}$ be any guarded statement in P. $gs.actexp$ may be an expression. If gs_i is any guarded statement derived from gs by diffusion and collection, and if gs_i is decisive for gs on transition $t = \langle s, s' \rangle$, then*

$$s \models gs_i.guard \leftrightarrow s \models gs.guard$$

PROOF. First we show diffusion preserves guards, then we show collection preserves guards. Diffusion breaks one guarded statement *gs* into a set of guarded statements gs_i. Let

$$H^i = ((gs_i \textit{ is decisive for } gs \textit{ at } s) \rightarrow (s \models gs_i.guard \leftrightarrow s \models gs.guard))$$

where gs_i is derived from gs_{i-1} by diffusion. We prove H^i by induction on the structure of $gs_i.actexp$, paralleling the three cases in Equations 9-11.

—Base case: Since $gs_0 = gs$ and $gs_0.guard = gs.guard$ is trivially true for all states s, then H^0 is certainly true at the start of all decisive transitions.

—Inductive Step: Assume H^i.

—Case: the outermost operator of $gs_i.actexp$ is disjunction. For all transitions $t = \langle s, s' \rangle$ where $gs_i.actexp$ is decisive on t. Let gs_{i+1} be the first conjunct in $gs_i.actexp$ to become true. Then, gs_{i+1} must be false at s and true at s', so gs_{i+1} is also decisive on t. $gs_{i+1}.guard = gs_i.guard$ by Equation 9. That implies $s \models gs_{i+1}.guard \leftrightarrow s \models gs_i.guard$. Combining this with H^i gives H^{i+1}.

—Case: the outermost operator of $e.actexp$ is conjunction. For all transitions $t = \langle s, s' \rangle$ where $gs_i.actexp$ is decisive on t. Let gs_{i+1} be the last conjunct to become true. Then, all conjuncts of $gs_i.actexp$ are true at s', and gs_{i+1} is false at s, so gs_{i+1} is also decisive on t. Since all other conjuncts are true at s, $gs_{i+1}.guard = gs_i.guard$ by

Equation 10. That implies $s \models \mathit{gs}_{i+1}.\mathit{guard} \leftrightarrow s \models \mathit{gs}_i.\mathit{guard}$. Combining this with H^i gives H^{i+1}.

—Case: there is no outermost operator in $\mathit{gs}_i.\mathit{actexp}$. $\mathit{gs}_i.\mathit{actexp}$ is a single guarded action a. Combining $a.\mathit{guard} = \mathit{gs}_i.\mathit{guard}$ from Equation 11, with $\mathit{gs}_i.\mathit{guard} = \mathit{gs}.\mathit{guard}$ from H^i. H^{i+1} since this holds in all states s.

Since Equation 12 uses conjunction, collection enables an action only when it is enabled on all paths. Therefore, collection also preserves guards. □

The next three theorems relate the runs of a protocol P expressed in terms of messages and the runs of its derived protocol $P' = \mathbf{col}(\mathbf{dif}(M(P)))$ expressed in terms of actions. These theorems relate runs between both (1) the super-gMsg and super-gAct protocols, and (2) the sub-gMsg and sub-gAct protocols as shown in Figure 7. The first theorem proves every run of P embeds a run of P'.

THEOREM A.5. *If Let $P = \langle \mathcal{R}, \mathcal{M}, \mathcal{C}, \mathcal{A}, \mathcal{S}, \mathcal{G} \rangle$ be a protocol, possibly containing guarded action expressions, and let M be any mapping function. Let $P' = \boldsymbol{col}(\boldsymbol{dif}(M(P)))$ be the protocol derived from P by mapping, diffusion, and collection. Then*

$$\forall \pi_r \in \boldsymbol{runs}(P) \; : \; (\exists \pi_s \in \boldsymbol{runs}(P') : \boldsymbol{emb}(\pi_s, \pi_r))$$

PROOF. Choose any $\pi_r = \langle h_0, m_1, h_1, \dots \rangle \in \mathbf{runs}(P)$ with message $m_i \in \mathcal{M}$. Construct a well-defined run $\pi_s = \langle g_0, a_1, g_1, \dots \rangle \in \mathbf{runs}(P')$, with mapped action $a_i \in M(\mathcal{A})$, and function $\mu(i)$ such that $0 \leq i \leq |\pi_r|$ such that $h_i = \widehat{M}(g_{\mu(i)})$. Define the induction hypothesis

$$H^i = (\pi_s^{\mu(i)} \textit{ is well defined} \wedge \mathbf{emb}(\pi_s^{\mu(i)}, \pi_r^i))$$

and show H^i by induction on $0 \leq i \leq |\pi_r|$.

—Base case:
- Define $g_0 = \emptyset$.
- Define $\pi_s^0 = \langle g_0 \rangle$ which is well defined.
- Define $\mu(0) = 0$.
- Then $h_0 = \emptyset = \widehat{M}(g_0) = \widehat{M}(g_{\mu(0)})$.
- H^0 holds.

—Inductive Step: Assume H^i. Let m_{i+1} be the next message in π_r.

—Case: No such m_{i+1} exists. All messages in π_r have been considered. $\pi_s = \pi_s^{\mu(i)}$ is a well-defined run and μ demonstrates $\mathbf{emb}(\pi_s^{|\pi_s|}, \pi_r^{|\pi_r|}) = \mathbf{emb}(\pi_s, \pi_r)$. $H^{|\pi_r|}$ holds, so the theorem is true.

—Case: m_{i+1} exists.
- Let $n = |\mathbf{means}(m_{i+1})|$ be the number of actions in m_{i+1}'s meaning.
- Define $g_{k+1} = g_k \cup a_{k+1}.\mathit{actexp}\ \forall k : \mu(i) \leq k < \mu(i) + n$. This creates n new values of g.
- Define $\pi_s^{\mu(i+1)} = \pi_s^{\mu(i)} + \sum_{k:\mu(i) \leq k < \mu(i)+n} \langle a_{k+1}, g_{k+1} \rangle$ by appending all the actions a_k in $\mathbf{means}(m_{i+1})$ onto the end, in any order.
- Let a_d be the decisive action for m_{i+1} in π_r where $\mu(i) \leq d \leq \mu(i) + n$.

- Define $\mu(i+1) = d$.

  Show $\pi_s^{\mu(i+1)}$ is well defined.
  (1) Show $g_k \models a_{k+1}.\textit{guard} \forall k : \mu(i) \leq k < \mu(i) + n$
    - $h_i \models m_{i+1}.\textit{guard}$ because π_r is well defined.
    - For each action $a_j \in \textbf{means}(m_{i+1})$, $m_{i+1}.\textit{guard} \rightarrow a_j.\textit{guard}$, because transition $t = \langle h_i, h_{i+1} \rangle$ is decisive for m_{i+1} and applying Theorem A.4.
    - Therefore, $h_i \models a_k.\textit{guard}\ \forall k : \mu(i) \leq k < \mu(i) + n$.

  (2) $g_{k+1} = g_k \cup a_k.\textit{actexp}\ \forall k : \mu(i) \leq k < \mu(i) + n$ by the definition above.

  Show $\textbf{emb}(\pi_s^{\mu(i+1)}, \pi_r^{i+1})$.
  - $g_{\mu(i+1)} = g_{\mu(i)} \bigcup_{\mu(i) \leq k < \mu(i)+n} a_k.\textit{actexp}$ by definition of $g_{\mu(i+1)}$ above.
  - $\widehat{M}(g_{\mu(i+1)}) = \widehat{M}(g_{\mu(i)} \bigcup_{\mu(i) < k \leq \mu(i)+n} a_k.\textit{actexp})$.
  - For all j, $\widehat{M}(g_j \cup a_{j+1}.\textit{actexp}) = \widehat{M}(g_{j+1})$ if a_{j+1} is not decisive for m_{i+1} in π_s, because $a_{j+1}.actexp$ is not in $\mathcal{A}$.
  - For all j, $\widehat{M}(g_j \cup a_{j+1}.\textit{actexp}) = \widehat{M}(g_{j+1}) \cup m_{j+1}.\textit{actexp}$ if a_{j+1} is decisive for m_{i+1} in π_s, because $a_{j+1}.actexp$ is decisive for $m_{i+1}.actexp$ and $m_{i+1}.actexp$ is in $\mathcal{A}$.
  - $\widehat{M}(g_{\mu(i+1)}) = \widehat{M}(g_{\mu(i)}) \cup m_{i+1}.\textit{actexp}$, by repeatedly applying the previous two reductions for all a_{j+1}.
  - $h_{i+1} = h_i \cup m_{j+1}.\textit{actexp}$, since $h_{i+1} = \widehat{M}(g_{\mu(i+1)})$ by the definition of h_{i+1} and $h_i = \widehat{M}(g_{\mu(i)})$ by H^i.
- Therefore, H^{i+1} holds.

□

The next theorem shows the reverse: every run in P' embeds a run in P.

THEOREM A.6. *If Let $P = \langle \mathcal{R}, \mathcal{M}, \mathcal{C}, \mathcal{A}, \mathcal{S}, \mathcal{G} \rangle$ be a protocol, possibly containing guarded action expressions, and let M be any mapping function. Let $P' = \textbf{\textit{col}}(\textbf{\textit{dif}}(M(P)))$ be the protocol derived from P by mapping, diffusion, and collection. Then*

$$\forall \pi_s \in \textbf{\textit{runs}}(P') \ : \ (\exists \pi_r \in \textbf{\textit{runs}}(P) : \textbf{\textit{emb}}(\pi_s, \pi_r))$$

PROOF. Choose any run $\pi_s = \langle g_0, a_1, g_1, \dots \rangle \in \textbf{runs}(P')$, with mapped action $a_i \in M(\mathcal{A})$. Construct a well-defined run $\pi_r = \langle h_0, m_1, h_1, \dots \rangle \in \textbf{runs}(P)$ with message $m_i \in \mathcal{M}$ and function $\mu(i)$ such that $0 \leq i \leq |\pi_r|$ such that $h_i = \widehat{M}(g_{\mu(i)})$. Define the induction hypothesis

$$H^j = (i = \textit{argmax}_k\ \mu(k) \leq j : \pi_r^i \textit{ is well defined} \wedge \textbf{emb}(\pi_s^j, \pi_r^i))$$

and show H^i by induction on the path length $0 \leq j \leq |\pi_s|$. We allow additional actions in π_s after $\mu(i)$ as long as they have no effect on P.

—Base case:
- Define $h_0 = \emptyset$,
- Define $\pi_r^0 = \langle h_0 \rangle$,
- π_r^0 is well defined.

- Define $\mu(0) = 0$.
- Then $\widehat{M}(g_0) = \widehat{M}(g_{\mu(0)}) = \emptyset = h_0$ implies $\mathbf{emb}(\pi_s^0, \pi_r^0)$ with $\mu(0) \leq 0$.
- H^0 is true.

—Inductive Step: Consider the next action $a_{j+1} \in \pi_s$.

—Case: No such a_{j+1} exists. All actions in π_s have been considered. $\pi_r = \pi_r^i$ is a well-defined run, and μ demonstrates $\mathbf{emb}(\pi_s^{|\pi_s|}, \pi_r^{|\pi_r|}) = \mathbf{emb}(\pi_s, \pi_r)$ with $\forall i : \mu(i) \leq j$. $H^{|\pi_r|} = H$.

—Case: a_j exists but it is *not* decisive for any $m \in \mathcal{A}_P$.

- Leave π_r^i unchanged which is still well defined.
- Leave μ unchanged.
- No additional $h_i = \widehat{M}(g_{\mu(i)})$ conditions are required to established $\mathbf{emb}(\pi_s^{j+1}, \pi_r^i)$ with $\forall i : \mu(i) \leq j$, and the existing conditions are true by the induction hypothesis.
- H^{j+1} is true.

—Case: a_{j+1} exists and it is decisive for some $m \in \mathcal{A}_P$.

- There is at most one such message m by the no overlap constraint of Definition 4.1. Denote the message by $m_{i+1} = m$.
- Define $h_{i+1} = h_i \cup m_{i+1}.\mathit{actexp}$.
- Define $\pi_r^{i+1} = \pi_r^i + \langle m_{i+1}, h_{i+1}\rangle$.
- Define $\mu(i + 1) = j + 1$.

Show π_r^{i+1} is well defined.

(1) Show $h_i \models m_{i+1}.\mathit{guard}$.
- $g_j \models a_{j+1}.\mathit{guard}$ because π_s is well defined.
- $\widehat{M}(g_j) \models \widehat{M}(a_{j+1}.\mathit{guard})$.
- $\widehat{M}(g_j) \models m_{i+1}.\mathit{guard}$ because $a_{j+1}.\mathit{guard} = m_{i+1}.\mathit{guard}$ by Theorem A.4.
- $\widehat{M}(g_{\mu(i+1)-1}) \models m_{i+1}.\mathit{guard}$ by definition of $\mu(i + 1)$.
- $\widehat{M}(g_{\mu(i)}) \models m_{i+1}.\mathit{guard}$ by Definition 4.8.
- Therefore, $h_i \models m_{i+1}.\mathit{guard}$ by H^j.

(2) $h_{i+1} \models m_{i+1}.\mathit{actexp}$ by definition of h_{i+1} above.

Show $\mathbf{emb}(\pi_s^{j+1}, \pi_r^{i+1})$.

- $g_{j+1} = g_j \cup a_{j+1}.\mathit{actexp}$ since π_s is well defined.
- $\widehat{M}(g_{j+1}) = \widehat{M}(g_j \cup a_{j+1}.\mathit{actexp})$.
- For all j, $\widehat{M}(g_j \cup a_{j+1}.\mathit{actexp}) = \widehat{M}(g_j) \cup m_{j+1}.\mathit{actexp}$ if a_{j+1} is decisive for $m_{j+1}.\mathit{actexp}$ in π_s.
- $\widehat{M}(g_{j+1}) = \widehat{M}(g_j) \cup m_{j+1}.\mathit{actexp}$, by applying the previous reduction.
- $\widehat{M}(g_{\mu(i+1)}) = \widehat{M}(g_{\mu(i+1)-1}) \cup m_{j+1}.\mathit{actexp}$, by definition of $\mu(i + 1)$.
- $\widehat{M}(g_{\mu(i+1)}) = \widehat{M}(g_{\mu(i)}) \cup m_{j+1}.\mathit{actexp}$, by Definition 4.8.
- $\widehat{M}(g_{\mu(i+1)}) = h_i \cup m_{j+1}.\mathit{actexp}$, since $h_i = \widehat{M}(g_{\mu(i)})$ by H^j.
- $\widehat{M}(g_{\mu(i+1)}) = h_{i+1}$ by the definition of h_{i+1}.

—H^{i+1} is true.

□

THEOREM A.7. *If Let $P = \langle \mathcal{R}, \mathcal{M}, \mathcal{C}, \mathcal{A}, \mathcal{S}, \mathcal{G}\rangle$ be a protocol, possibly containing guarded action expressions, and let M be any mapping function. Let $P' =$ **col**(**dif**$(M(P))$)*

be the protocol derived from P by mapping function M, diffusion, and collection. Then

$$\forall \pi_r \in \textbf{\textit{runs}}(P) \ : \ (\exists \pi_s \in \textbf{\textit{runs}}(P') : \textbf{\textit{emb}}(\pi_s, \pi_r))$$
$$\forall \pi_s \in \textbf{\textit{runs}}(P') \ : \ (\exists \pi_r \in \textbf{\textit{runs}}(P) : \textbf{\textit{emb}}(\pi_s, \pi_r))$$

PROOF. Follows immediately from Theorems A.5 and A.6. □