A Compositional Approach for Designing Protocols*

Gurdip Singh
Department of Computing and Information Sciences
Kansas State University
Manhattan, KS 66506

Abstract

The complexity of designing distributed protocols has led to compositional techniques for designing and verifying protocols. We propose a technique based on the notion of parallel composition of protocols. It allows the component protocols to share messages and variables. The composite protocol is an interleaved execution of the component protocols with the constraint that they be synchronized at events updating shared variables or sending/receiving shared messages. All invariants of the component protocols are preserved in the composite protocol. The technique allows modular design and verification of protocols. It can be viewed as an inverse of the projection method for verifying protocols.

1 Introduction

Designing and verifying distributed protocols is difficult as they perform several concurrent activities and are required to cope with variable message delays and processor speeds. The complexity of distributed protocols has led to compositional design and verification techniques in which protocols performing the various subfunctions are designed and verified separately and then integrated to form a more complex protocol. Traditional techniques have viewed a process as the unit of modularity [9] [10]. In these techniques, a protocol is viewed as a parallel composition of processes.

Recently, several proposals have been made which use a protocol as the unit of modularity. [5] proposed a framework for *sequential composition* of protocols, in which a problem is divided into two or more subproblems which can be performed in sequence. Protocols solving the subproblems are designed and verified independently and then combined sequentially to obtain a protocol for the entire problem. It was shown that under certain conditions, properties of composite protocol can be inferred from those of the component protocols. [4] [12] have also proposed techniques based on sequential composition.

Several techniques based on the notion of *parallel composition* of protocols have been proposed [3] [8]. In a parallel composition of protocols, the component protocols may execute concurrently (*i.e.*, their execution at a node may be interleaved). [3] proposed a notion of *superimposition* which is a parallel composition of a basic computation with some control protocol. In this case, the control protocol observes the events of the basic protocol and constrains its execution. For example, a deadlock avoidance protocol can be superimposed on a basic computation (which will control the basic computation to prevent deadlocks). [8] proposed a methodology of constructing a multi-function protocol by combining the protocols performing the various functions. The composition involves imposing operational constraints on the execution of the protocols solving the subproblems. Both techniques can be used to add functionality to a protocol. The technique of using protocols as building blocks to design more complex protocols is a promising one. It facilitates modular design and verification by allowing a designer to focus on one aspect at a time. The techniques discussed above, however, assume that the component protocols do not share messages or update shared variables (they do allow read-only access to variables of the other protocols).

In this paper, we propose a technique which is based

*This work was supported by NSF Research Initiation Award CCR-9211621.

0-8186-3670-X/93 $03.00 © 1993 IEEE

on the notion of parallel composition of protocols. It allows the component protocols to share variables as well as messages. Informally, our composition principle allows an interleaved execution of the component protocols but requires that they be synchronized at events which update shared variables or send/receive common messages. Our methodology involves the following three steps (which are similar to those in [5]):

- In the first step, a designer divides the functionality of a protocol into subfunctions.
- In the second step, a protocol for each subfunction is obtained.
- The third step involves combining the protocols to obtain a protocol for the entire problem. For this purpose, we give an algorithm for obtaining the composite protocol from the component protocols.

We show that all invariants of the component protocols are preserved in the composite protocol. In [6], it was observed that in many multi-function protocols, modules performing the various functions may interact with each other, and share messages and update common variables. This sharing poses a problem in decompositional analysis. To deal with this problem, they proposed a *projection* method to verify a protocol. We show that our technique can be viewed as an inverse of the projection method. [7] presented another version of an inverse of the projection method based on event refinements. [11] presented a technique based on parallel composition which allows shared variables (but not shared messages). This technique, however, requires explicit specification of synchronization points and all invariants of the component protocols may not be preserved in the composite protocol.

We demonstrate the applicability of our technique by designing a leader election protocol and a data transfer protocol:

- For the leader election problem, we first design a two-process leader election protocol and verify it. We then obtain a protocol for N-process leader election by combining two-process leader election protocols, one for each pair of processes. A similar strategy can be used to extend a two-process mutual exclusion protocol to an N-process mutual exclusion protocol.
- To obtain a data transfer protocol with one sender i and two receivers j and k, we first design two data transfer protocols, one for sending a sequence of data items from i to j and the other for sending the same sequence of data items from i to k. We then combine these protocols to obtain the final protocol.

We also generalize the principle so that it takes two parameters, V and M, where V is a subset of shared variables and M is a subset of shared messages. In this case, we require that the component protocols be synchronized only at those events which update variables in V or send/receive messages in M. This is motivated by the fact that in certain compositions, the composite protocol desired might require asynchronous updates to some shared variables (synchronizing updates to a subset of shared variables, however, must be sufficient to preserve the desired set of invariants). For example, we consider two token based mutual exclusion protocols, both of which assume that nodes are arranged logically in a ring. However, request propagation mechanism for a token is different in the two protocols. We combine these protocols to obtain a protocol with a more efficient request propagation mechanism (this protocol requires asynchronous updates to a shared variable). However, the subset of shared variables and messages at which the protocols are synchronized is sufficient to ensure the mutual exclusion property.

This paper is organized as follows. In the next section, we present a formalism to specify protocols and discuss some examples. In Section 3, we present our composition principle and describes the properties of the composite protocols. In Section 4, we discuss some extensions of the composition principle. Section 5 concludes the paper.

2 Model

A distributed protocol, P, is a set of processes $[P_1 \parallel P_2 \parallel \cdots \parallel P_n]$. Each P_i is in the following normal form [1]:

$$*[\ \Box_{l=1}^{m_i}\ en(a_l) \longrightarrow a_l :< s_l >\ ]$$

where a_l is the label of the action, $en(a_l)$ is the enabling condition (which is a boolean expression) and s_l is the computation associated with the action. A

communication channel from P_i to P_j is modeled as a sequence $ch_{i,j}$ of messages. The statement 'send $M(arg\_list)$ to j' executed by P_i causes $M(arg\_list)$ to be appended to the end of $ch_{i,j}$, and the statement 'receive $M(para\_list)$ from i' executed by P_j removes the message at the head of $ch_{i,j}$ and stores the contents of the message in $para\_list$.

Let $variable(P_i)$ denote the set of variables which appear in P_i (except the variables modeling the channels) and $variable(P) = variable(P_1) \cup \cdots \cup variable(P_n)$. Let $mess(P_i)$ denote the set of messages sent or received by P_i and $mess(P) = mess(P_1) \cup \cdots \cup mess(P_n)$. We assume that messages with the same name but sent over different links are different. For this purpose, we will assume that each message is indexed with the channel over which it is sent (for clarity, we may omit the subscripts when they are clear from context).

The state of P_i is defined by the values of variables in $variable(P_i)$. The state of a channel from P_i to P_j is the value of $ch_{i,j}$. The state of the system is a tuple $(s_1, \ldots, s_n, c_1, \ldots, c_k)$, where s_i's represent the states of the processes and c_j's represent the states of the channels. An event is specified by an action and its enabling condition. An event occurs when the computation associated with its action is executed. An *execution* of a protocol is a maximal sequence, $g_0 \rightarrow_{e_0} g_1 \rightarrow_{e_1} \ldots$, such that g_0 is the initial state, e_l is enabled in state g_l and the execution of the computation associated with e_l transforms state g_l to g_{l+1}. We say that g is *reachable* from g' if there exists a sequence of events which takes the system from state g' to g. An assertion I is an *invariant* if I is true in the initial state and in all states reachable from the initial state.

2.1 Protocol Examples

In this section, we will discuss some protocols which we will use to illustrate the composition principle. We first consider a protocol for leader election. The purpose of this protocol is to distinguish exactly one process as the leader. Figure 1 gives a leader election protocol, $LD^{\{i,j\}}$, for two processes i and j (the protocol is given for process i; the protocol at process j is similar). We assume that initially $leader_i = sent_{i,j} = sent_{j,i} = win_{i,j} = win_{j,i} =$ false and $ch_{i,j} = ch_{j,i} = \lambda$. To elect itself the leader, each process sends a message containing its identity to the other process. On receiving this message, a process compares its identity with the one received in the message. An *ack* message is sent if the identity received in the message is greater. The process with the larger identity becomes the leader. Process i sets $leader_i$ to *true* if it is elected the leader. Assuming that $id_i \neq id_j$, we can show that $(leader_i \Rightarrow \neg leader_j) \wedge (leader_j \Rightarrow \neg leader_i)$ is an invariant of the protocol.

$LD_i^{\{i,j\}}$::
*[$\neg sent_{i,j}$ $\longrightarrow$
$S_{i,j}$:< send capture(id_i) to j; $sent_{i,j}$:= true >
□
capture(x) = $first(ch_{j,i})$ $\longrightarrow$
$L_{i,j}$:< receive capture(x) from j;
if $x > id_i$ then send ack to j >
□
ack = $first(ch_{j,i})$ $\longrightarrow$
$W_{i,j}$:< receive ack from j; $win_{i,j}$:= true >
□
$win_{i,j}$ $\longrightarrow$ $Elect_{i,j}$:< $leader_i$:= true >
]

Figure 1: Two-Process Leader Election

We next consider a data transfer protocol, $D^{\{i,j\}}$, with process i as the sender and process j as the receiver. Process i has a sequence of data items stored in an array $sdata[1..max]$ which it wants to send to j. Process j stores the items received in an array $rdata[1..max]$. Figure 2 gives a data transfer protocol. Initially, $sent = rec = 0$, ack_recd = true and $ch_{i,j} = ch_{j,i} = \lambda$. Each data item received by $D_j^{\{i,j\}}$ is acknowledged. $D_i^{\{i,j\}}$ can send the next data item only if it has received an *ack* for the previous data item sent. The protocol $D^{\{i,k\}}$ is similar (with k as the receiver instead of j). Assuming that channels do not lose or duplicate messages, we can show that the following are invariants of $D^{\{i,j\}}$:

I1: $0 < x < rec \;\; rdata[x] = sdata[x]$
I2: $sent - 1 \leq rec \leq sent$.

3 A Compositional Approach

In this section, we will present our composition principle. We will consider composition of two compo-

$D_i^{\{i,j\}}$::
*[$ack\_recd_{i,j} \wedge sent < max \longrightarrow$
$Send_{i,j}$:< $sent := sent + 1$; $ack\_recd_{i,j} := false$;
send data($sdata[sent]$) to j >
□
ack = $first(ch_{j,i}) \longrightarrow$
$Ack_{i,j}$:< receive ack from j; $ack\_recd_{i,j} := true$ >;
]

$D_j^{\{i,j\}}$::
*[data(x) = $first(ch_{i,j}) \longrightarrow$
$Rec_{i,j}$:< receive data(x) from i; $rec := rec + 1$;
$rdata[rec] := x$; send ack to i >
]

Figure 2: Data Transfer Protocol

nent protocols only (the technique can be extended to any number of component protocols). Let $P \equiv [P_1 \parallel \cdots \parallel P_n]$ and $Q \equiv [Q_1 \parallel \cdots \parallel Q_n]$ be the two component protocols. We will obtain a composite protocol R, denoted by $Comp(P, Q)$. The composition proposed is local in nature *i.e.*, R_i is dependent only of P_i and Q_i and can be viewed as an interleaved execution of these two processes. Intuitively, the composition requires that R_i satisfy the following conditions:

- (A1): For each i, P_i and Q_i execute asynchronously. For example, access to non-shared variables must be completely asynchronous.
- (A2): For each i, the execution of P_i and Q_i must be synchronized at events updating a shared variable or sending/receiving a common message.

To define these conditions more formally, we first define the notion of matching actions. Actions a_1 :< s; s_1 > and a_2 :< s; s_2 > of P_i and Q_i respectively are *matching actions* if s_1 and s_2 do not update any variable in $variable(P_i) \cap variable(Q_i)$ or send/receive a message in $mess(P_i) \cap mess(Q_i)$ (similarly, a_1 :< $s_1; s$ > and a_2 :< $s_2; s$ > are also matching). For example, the actions

$Send_{i,j}$:< $sent := sent + 1$; $ack\_recd_{i,j} := false$;
send data($sdata[sent]$) to j >
$Send_{i,k}$:< $sent := sent + 1$; $ack\_recd_{i,k} := false$;
send data($sdata[sent]$) to k >

in $D^{\{i,j\}}$ and $D^{\{i,k\}}$ respectively are matching, where s = '$sent := sent + 1$;' (note that although the message data is sent in both protocols, it is sent over different links and therefore, is not considered a common message). Let $shared(P, Q)$ denote the set of actions in P and Q which update a variable in $variable(P) \cap variable(Q)$ or send a message in $mess(P) \cap mess(Q)$. For example, $shared(LD^{\{i,j\}}, LD^{\{i,k\}}) = \{Elect_{i,j}, Elect_{i,k}\}$ and $shared(D^{\{i,j\}}, D^{\{i,k\}}) = \{Send_{i,j}, Send_{i,k}\}$ We say that P and Q are *matching protocols* if for each action a in $shared(P, Q)$ which belongs to P (Q), there exists a distinct matching action in Q (P) (thus, the matching function must be one-to-one). For example, protocols $D^{\{i,j\}}$ and $D^{\{i,k\}}$ are matching since $shared(D^{\{i,j\}}, D^{\{i,k\}}) = \{Send_{i,j}, Send_{i,k}\}$ and $(Send_{i,j}, Send_{i,k})$ is a matching pair of actions.

We will now discuss an algorithm for obtaining R. Let P and Q be matching protocols. The set of guarded commands of R_i is obtained as follows:

- If a :< s > $\notin shared(P, Q)$ then $en(a) \longrightarrow a$:< s > is in R_i. This rule reflects condition A1 (that is, the component protocols must execute asynchronously).
- Let a :< $s; s_a$ > be an action in $shared(P, Q)$ and b :< $s; s_b$ > be the matching action. Then $en(c) \longrightarrow c$:< $s; s_a; s_b$ > is in R_i, where c is a new label and $en(c) = en(a) \wedge en(b)$. In this case, we say that c is obtained by *fusing* actions a and b. This rule reflects condition A2 which indicates that the execution of the protocols must be synchronized at matching actions.

Using the methodology described above, we will obtain a three-process leader election protocol using protocol $LD^{\{i,j\}}$ of Figure 1. We will first compose $LD^{\{1,2\}}$ and $LD^{\{1,3\}}$. We know that $variable(LD^{\{1,2\}}) \cap variable(LD^{\{1,3\}}) = \{leader_1\}$ and $mess(LD^{\{1,2\}}) \cap mess(LD^{\{1,3\}}) = \{\}$. From the specification of the protocol, we have that $shared(LD^{\{1,2\}}, LD^{\{1,3\}}) = \{Elect_{1,2}, Elect_{1,3}\}$ and $Elect_{1,2}$ and $Elect_{1,3}$ are matching. Thus, the protocols are matching and we can combine them to obtain $Comp(LD^{\{1,2\}}, LD^{\{1,3\}})$. We next observe that $Comp(LD^{\{1,2\}}, LD^{\{1,3\}})$ and $LD^{\{2,3\}}$ share the variables in the set $\{leader_2, leader_3\}$ and are matching. Hence, we can combine these protocols to obtain the final protocol, $LD^{\{1,2,3\}}$, which is shown in Figure 3 (the protocol for processes 2 and 3 is similar to the one shown in the figure).

$LD_1^{\{1,2,3\}}$::
*[$\neg sent_{1,2}$ $\longrightarrow$ $S_{1,2}$:< send capture(id_1) to 2;
$sent_{1,2}$:= true >
□
$\neg sent_{1,3}$ $\longrightarrow$ $S_{1,3}$:< send capture(id_1) to 3;
$sent_{1,3}$:= true >
□
capture(x) = $first(ch_{2,1})$ $\longrightarrow$
$L_{1,2}$:< receive capture(x) from 2;
if $x > id_1$ then send ack to 2 >
□
capture(x) = $first(ch_{3,1})$ $\longrightarrow$
$L_{1,3}$:< receive capture(x) from 3;
if $x > id_1$ then send ack to 3 >
□
ack = $first(ch_{2,1})$ $\longrightarrow$ $W_{1,2}$:< receive ack from 2;
$win_{1,2}$:= true >
□
ack = $first(ch_{3,1})$ $\longrightarrow$ $W_{1,3}$:< receive ack from 3;
$win_{1,3}$:= true >
□
$win_{1,2} \wedge win_{1,3}$ $\longrightarrow$ $Elect_1$:< $leader_1$:= true>
]

Figure 3: Three Process Leader Election Protocol

$D_i^{\{i,j,k\}}$::
[$ack\_recd_{i,j} \wedge ack\_recd_{i,k} \wedge sent < max$ $\longrightarrow$
$Send_i$:< $sent := sent + 1$;
$ack\_recd_{i,j}$:= false; $ack\_recd_{i,k}$:= false;
send data($sdata[sent]$) to j;
send data($sdata[sent]$) to k >
□
ack = $first(ch_{j,i})$ $\longrightarrow$
$Ack_{i,j}$:< receive ack from j; $ack\_recd_{i,j}$:= true >
□
ack = $first(ch_{k,i})$ $\longrightarrow$
$Ack_{i,k}$:< receive ack from k; $ack\_recd_{i,k}$:= true >
]

Figure 4: Data Transfer Protocol with Two Receivers

We will now combine the data transfer protocols $D^{\{i,j\}}$ and $D^{\{i,k\}}$ to obtain a protocol $D^{\{i,j,k\}}$ in which i sends the same sequence of data items to both j and k. $D^{\{i,j\}}$ and $D^{\{i,k\}}$ share the variables $\{sent, sdata, max\}$ but no messages. It can be seen that they are matching. The composite protocol is shown in Figure 4. $D_j^{\{i,j,k\}}$ and $D_k^{\{i,j,k\}}$ are identical to $D_j^{\{i,j\}}$ and $D_k^{\{i,k\}}$ respectively and are not shown in Figure 4.

The technique can also be used, for example, to extend a 2-process mutual exclusion protocol to an N-process mutual exclusion protocol or a 2-process synchronizer [2] to an N-process synchronizer. In many multi-function protocols, it is the case that protocols performing the subfunctions specify different constraints under which an action must be performed in order to ensure certain properties. By synchronizing the action in different protocols, we ensure that the action is performed when constraints imposed by each individual protocol are satisfied, thus preserving the properties of each protocol.

3.1 Properties of the Composite Protocols

In this section, we present results which enable us to infer properties of the composite protocol from those of the component protocols.

Theorem 3.1 *Let $R = Comp(P, Q)$. If I is an invariant of P (Q) which refers only to variables in variable(P) (variable(Q)) then I is an invariant of R.*

Proof of Theorem 3.1: We show that for each execution of R, there exists a corresponding execution of P. From this, we can conclude that if I is an invariant of P then it is an invariant of R. The proof is given in the full paper. □

[6] proposed a projection technique to verify invariant properties of a protocol. For this purpose, a set V of variables and a set M of messages is selected (M is actually the set of messages whose receiving causes an update to variables in V), and the protocol is projected to obtain an image protocol using these sets (all actions updating variables in V or sending/receiving messages in M are retained; rest are mapped to null actions). It was shown that if I is an invariant of the image protocol then I is an invariant of the orig-

inal protocol. It can be shown that the projection of $Comp(P, Q)$ on to the set $variable(P)$ and $mess(P)$ is the protocol P except that some actions may have strengthened enabling conditions (because of the fusing of actions). Thus, the set of executions of the image protocol is a subset of the set of executions of P.

Since $(leader_1 \Rightarrow \neg leader_2) \wedge (leader_2 \Rightarrow \neg leader_1)$ is an invariant of $LD^{\{1,2\}}$, from Theorem 3.1, it is an invariant of $Comp(LD^{\{1,2\}}, LD^{\{1,3\}})$. Similarly, from invariants of $LD^{\{1,3\}}$ and $LD^{\{2,3\}}$, we can deduce that $\forall i, j, k \in \{1, 2, 3\}\ i \neq j \neq k$ $(leader_i \Rightarrow \neg(leader_j \vee leader_k))$ is an invariant of $LD^{\{1,2,3\}}$. We can also infer that invariants $I1$ and $I2$ of $D^{\{i,j\}}$ will be preserved in $D^{\{i,j,k\}}$.

4 Extensions to the Composition Principle

In this section, we propose an extension to the composition principle discussed above. In composition of some protocols, we may be interested in synchronizing the execution of the component protocols at a subset of shared actions. For example, consider two mutual exclusion protocols, $MUT1$ and $MUT2$, each involving sites arranged logically in a ring. In both protocols, there exists a token which circulates in the ring in the clockwise direction. A process can enter the critical section only if it possesses the token. The propagation of requests for token is different in $MUT1$ and $MUT2$. In $MUT1$, if i needs the token, it sends a request directly to all other processes (we assume that processes can also directly communicate with each other). On receiving this request, any other process j sets a variable req_j to true. If req_j is true and j holds the token, it sends the token to its left neighbor. In $MUT2$, if i needs a token, it sends a request to its right neighbor in the ring. On receiving a request, j sets req_j to true. If req_j is true and j does not hold the token, it sends a request to its right neighbor. Thus, the request travels counterclockwise in the ring until it reaches a process which holds the token. If req_j is true and j holds a token, it sends the token to its left neighbor. Thus, in $MUT1$, requests are propagated directly while in $MUT2$, requests are propagated counterclockwise along the ring. Figure 5 and Figure 6 shows $MUT1$ and $MUT2$ respectively

$MUT1_i$::
*[$ext_i \wedge hold_i \longrightarrow$
$Enter_1 :<$ $in_i := $ true; $ext_i := $ false $>$
□
$ext_i \wedge \neg hold_i \wedge \neg sent_i \longrightarrow$
$Sreq_1 :<$ send request to $i+1$;
send request to $i-1$; $sent_i := $ true $>$
□
request $= first(ch_{j,i}) \longrightarrow$
$Rec_1 :<$ receive request from j; $req_i := $ true $>$
□
token $= first(ch_{i-1,i}) \longrightarrow$
$Rtoken_1 :<$ receive token from $i-1$;
$hold_i := $ true $>$
□
$req_i \wedge hold_i \wedge \neg in_i \wedge \neg ext_i \longrightarrow$
$Stoken_1 :<$ send token to $i+1$; $hold_i := $ false;
$req_i := $ false; $sent_i := $ false $>$
]

Figure 5: Protocol MUT1

for three sites with indices 0, 1, and 2. We assume that i has $i+1$ $(i-1)$ as the left (right) neighbor (operations are assumed modulo 2). In both protocols, it is assumed that when process i wants to enter the critical section, ext_i is set to true. On entering the critical section, it sets ext_i to false and in_i to true. On exiting the critical section, it sets in_i to false. $hold_i$ = true indicates that i has the token and $sent_i$ = true in $MUT1$ ($sent_i'$ = true in $MUT2$) implies that i has sent a request for which it has not received a response. Setting ext_i to true and in_i to false are done external to the protocol (by the protocol using the mutual exclusion protocol) and are not shown in the figures.

We may want to obtain a protocol in which requests are propagated as specified by both protocols. Thus, req_i is set to true on receiving the request along any one of the paths: either directly or along the ring (thus, propagation of request will be faster since i can receive it via either paths). Even though req_i is a shared variable, we do not want to synchronize the assignments to req_i since that will involve waiting for the request to arrive via both paths before req_i is set to true. Also, we find that to preserve the mutual exclusion property, it is not necessary to perform this synchronization. To allow such compositions, we modify the composition rule as described in the following.

$MUT2_i$::
*[$ext_i \wedge hold_i \longrightarrow$
$Enter_2 :< in_i := \text{true}; ext_i := \text{false} >$
□
$ext_i \wedge \neg hold_i \longrightarrow Req_2 :< req_i := \text{true} >$
□
request = $first(ch_{i+1,i}) \longrightarrow$
$Rec_2 :<$ receive request from $i+1$; $req_i := \text{true} >$
□
$req_i \wedge \neg sent'_i \longrightarrow$
$Sreq_2 :<$ send request to $i-1$; $sent'_i := \text{true} >$
□
token = $first(ch_{i-1,i}) \longrightarrow$
$Rtoken_2 :<$ receive token from $i-1$;
$hold_i := \text{true} >$
□
$req_i \wedge hold_i \wedge \neg in_i \wedge \neg ext_i \longrightarrow$
$Stoken_2 :<$ send token to $i+1$; $hold_i := \text{false}$;
$req_i := \text{false}$; $sent'_i := \text{false} >$
]

Figure 6: Protocol MUT2

Let $shared(P, Q, V, M)$ denote the set of actions in P and Q which update a variable in V or send/receive a message in M. Actions $a :< s; s_1 >$ and $b :< s; s_2 >$ are matching over (V, M) if s_1 and s_2 do not update a variable in V or send a message in M. We say that P and Q are matching protocols over (V, M) if for each action a in $shared(P, Q, V, M)$ which belongs to P (Q), there exists a distinct action b in Q (P) such that a and b match over (V, M). The composite protocol R, denoted by $Comp(P, Q, V, M)$, is obtained as follows:

1. If $a :< s >\notin shared(P, Q, V, M)$ then $en(a) \longrightarrow a :< s >$ is in R_i.
2. Let $a :< s; s_a >$ be an action in $shared(P, Q, V, M)$ and $b :< s, s_b >$ be the matching action. Then $en(c) \longrightarrow c :< s; s_a; s_b >$ is in R_i, where c is a new label and $en(c) = en(a) \wedge en(b)$.

Let
$V = \{ext_0, ext_1, ext_2, in_0, in_1, in_2, hold_0, hold_1, hold_2\}$
and $M = \{token_{0,1}, token_{0,2}, token_{1,2}\}$ (messages are indexed by the link over which they are sent). Then, $MUT1$ and $MUT2$ match over (V, M). Figure 7 gives the composite protocol, $MUT3 = Comp(MUT1, MUT2, V, M)$.

$MUT3_i$::
*[$ext_i \wedge hold_i \longrightarrow$
$Enter_3 :< in_i := \text{true}; ext_i := \text{false} >$
□
$ext_i \wedge \neg hold_i \longrightarrow$
$Req_2 :< req_i := \text{true} >$
□
$ext_i \wedge \neg hold_i \wedge \neg sent_i \longrightarrow$
$Sreq_1 :<$ send request to $i+1$;
send request to $i-1$; $sent_i := \text{true}; >$
□
request = $first(ch_{j,i}) \longrightarrow$
$Rec_1 :<$ receive request from j; $req_i := \text{true} >$
□
request = $first(ch_{i+1,i}) \longrightarrow$
$Rec_2 :<$ receive request from $i+1$; $req_i := \text{true} >$
□
$req_i \wedge \neg sent'_i \longrightarrow$
$Sreq_2 :<$ send request to $i-1$; $sent'_i := \text{true} >$
□
token = $first(ch_{i-1,i}) \longrightarrow$
$Rtoken_3 :<$ receive token from $i-1$;
$hold_i := \text{true} >$
□
$req_i \wedge hold_i \wedge \neg in_i \wedge \neg ext_i \longrightarrow$
$Stoken_3 :<$ send token to $i+1$; $hold_i := \text{false}$;
$req_i := \text{false}$; $sent_i := \text{false}$;
$sent'_i := \text{false} >$
]

Figure 7: Protocol MUT3

Let $var(I)$ and $mess(I)$ denote the set of variables and messages respectively sufficient to verify I (*i.e.*, they provide sufficient resolution) using the projection technique of [6]. For example, to verify that $in_i \Rightarrow \neg(in_{i-1} \vee in_{i+1})$ is an invariant of $MUT1$, it is sufficient to project the protocol over the set $V = \{in_0, in_1, in_2, hold_0, hold_1, hold_2\}$ and $M = \{token_{0,1}, token_{0,2}, token_{1,2}\}$.

Theorem 4.1 *Let $R = Comp(P, Q, V, M)$ and I be an invariant of P. If $var(I) \subseteq V$ and $mess(I) \subseteq M$ then I is an invariant of R.*

The proof of this theorem is a minor modification of the proof of Theorem 3.1. From Theorem 4.1, we can infer that the invariant $in_i \Rightarrow \neg(in_{i-1} \vee in_{i+1})$ of the component protocols is preserved in the com-

posite protocol.

5 Conclusion and Discussion

In this paper, we have presented a technique for parallel composition of protocols. The technique allows protocols to share messages and variables. We first considered the case where the component protocols are synchronized at all events which update shared variables or send/receive shared messages. In this case, all invariants of the component protocols are preserved in the composite protocol. We then generalized the technique so that component protocols are synchronized only at those events which update variables in V or send/receive messages in the set M. In this case, not all invariants of the component protocols may be preserved in the composite protocol.

We have discussed inference rules only for invariants. There are other safety and liveness properties which are of interest. In [6], it was shown that if all actions of the image protocol are well-formed then any safety or liveness property of the image protocol is preserved in the original protocol. Future work involves adapting this technique to derive sufficient conditions to infer safety and liveness properties of R from those of P and Q (as discussed above, the image protocol obtained by projecting R on to the sets $variables(P)$ and $mess(P)$ is protocol P with some of its enabling conditions strengthened).

For component protocols satisfying certain properties, we have obtained a sufficient condition for the composite protocol to be deadlock free given that the component protocols are deadlock-free (the condition only requires reachability analysis of the component protocols). Future works involves generalizing this result to a larger class of component protocols and determining the relationship between this sufficient condition and well-formedness of the actions.

References

[1] Apt, K.R. and Clermont, P. Two normal forms theorems for csp programs. In *Report No. RC 10975, IBM Reserach Center, Yorktown Heights, N.Y.*, 1985.

[2] Awerbuch, B. Complexity of network synchronization. *JACM*, 32(4), Oct. 1985.

[3] Bouge, L. and Francez, N. A compositional approach to superimposition. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1988.

[4] Choi, T.Y. and Miller, R.E. A decomposition method for the analysis and design of finite state protocols. In *Proceedings of the 8th Data Communication Symposium*, 1983.

[5] Chow, C., Gouda, M., and Lam, S. A discipline for constructing multi-phase communicating protocols. *ACM Transactions of Computer Systems*, 3(4), 1985.

[6] Lam, S. and Udaya Shankar, A. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10(4), July 1984.

[7] Lam, S. and Udaya Shankar, A. A relational notation for state transition systems. *IEEE Transactions on Software Engineering*, SE-16(7), July 1990.

[8] Lin, H. A methodology for constructing communication protocols with multiple concurrent functions. *Distributed Computing*, 3, 1988.

[9] Owicki, S. and Gries, D. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6, 1976.

[10] Sabnani, K., Lapone, A., and Umit Uyar, M. An algorithmic procedure for checking safety properties of protocols. *IEEE Transactions on Communications*, 37(9), September 1989.

[11] Singh, G. and Bernstein, A.J. A framework for parallel composition of protocols. In *Parallel Architectures and Languages Europe*, 1992.

[12] Stomp, F. and Roever, W. de. Designing distributed algorithms by means of formal sequentially phased reasoning. In *Proceedings of the 3rd Int'l Workshop on Distributed Algorithms*, 1989.