Composition of Service Specifications*

Gurdip Singh, Ionut Buricea and Zhenyu Mao
Department of Computing and Information Sciences
Kansas State University
Manhattan, KS 66506
email: {singh,ionutb}@cis.ksu.edu

Abstract

The service specification $ss(P)$ of a protocol P defines the services provided by the protocol and its protocol specification $ps(P)$ specifies the rules of message exchange to ensure the service. Protocol composition has been advocated as an attractive way to design complex protocols. Several techniques have been studied for composition of protocol specifications. In these techniques, to combine component protocols P and Q to design R, $ps(P)$ and $ps(Q)$ are first combined to obtain $ps(R)$ and then inference rules are used to derive $ss(R)$. In this paper, we explore an alternative strategy in which we allow composition to be specified at the service specification level (that is, $ss(P)$ and $ss(Q)$ are first combined to obtain $ss(R)$). Given $ss(R)$, we provide an algorithm to mechanically combine $ps(P)$ and $ps(Q)$ to generate $ps(R)$ such that $ps(R)$ satisfies $ss(R)$. We show that analysis of $ss(R)$ is sufficient to ensure that $ps(R)$ satisfies certain safety and liveness properties. This results in efficient validation as state space of $ss(R)$ is typically significantly smaller than that of $ps(R)$.

1 Introduction

The problem of designing a correct network protocol is a challenging task and should be based on formal engineering principles. A protocol can be viewed as a service provider offering some communication services. A protocol has a set of *service access points* (SAPs) via which its services can be accessed. Each service access point may have a set of actions (*service primitives*) associated with it. Service specification and protocol specification are two important stages of the protocol engineering cycle. At the service specification stage, the designer must specify the properties (services) to be provided by the protocol. The service specification, $ss(P)$, of a protocol P gives the possible sequences in which actions at different SAPs may occur.

In the protocol specification stage, the protocol is described as a set of communicating processes, one for each SAP. The designer must specify details such as format of the messages, rules for message exchange, local state maintained by each entity and properties of the communication channels. Protocol validation is a part of this stage to ensure that the protocol specification meets the service specification. Protocol specifications typically are much more complex than service specifications due to factors such as lossy communication channels, locality of the processes, etc.

The complexity of designing correct network protocols has led researchers to propose compositional techniques to design protocols. The main idea in these techniques is to first design the protocols for the subfunctions independently and then combine them in a disciplined manner to obtain the composite protocol. A number of techniques have been proposed for composition of protocol specifications such as sequential composition alternative composition and parallel composition [1, 2, 8, 9, 10, 13, 14, 15, 16, 17].

Let P and Q be two component protocols used to design a composite protocol R. Each of these techniques provide inference rules that ensure certain properties of R by placing some restrictions on the component protocols that can be combined. Since these restrictions are sufficient conditions, we can construct several protocols from component protocols that do not satisfy these restrictions. To increase the applicability of the compositional techniques, such compositions must be allowed. However, for such compositions, the inference rules may not be applicable and therefore, we would have to analyze the composite protocol specification directly, whose state space may be very large.

To overcome this problem, we study composition at the service specification level. We provide a framework

---

*This work was supported by NSF CAREER award CCR9502506 and ARO grant DAAH04-95-1-0464.

in which $ss(R)$ can be obtained by specifying a set of constraints on $ss(P)$ and $ss(Q)$ where the constraints specify how the services offered by P and Q have to be combined. Given $ss(R)$, we give an algorithm to generate the protocol specification, $ps(R)$, by combining $ps(P)$ and $ps(Q)$. In defining these compositions, we do not impose any restrictions on the constraints. This flexible use of constraints increases the applicability of the composition techniques to allow a larger set of protocols to be designed and composed. However, a designer may specify constraints that do not result in the desired composite protocol (for example, the resulting protocol may contain deadlocks). However, we show that the analysis of $ss(R)$ is sufficient to infer certain safety and liveness properties of $ps(R)$. This results in efficient verification as the state space of $ss(R)$ is typically significantly smaller than that of $ps(R)$.

We feel that defining composition of service specification is more intuitive and useful as the main aim of composition is to combine the services provided by P and Q in a certain way. Composition at service specification level also supports a building block approach to protocol construction wherein a library of basic protocols is made available. The library may consists of the service specification and protocol specification of each protocol. A designer can select a subset of protocols from the library (depending on the services they provide) and specify how to combine their services. We are currently building a software tool based on this methodology.

2 Model

We define an extended finite state machine (EFSM) X as a tuple $< S_X, A_X, F_X, M_X, V_X, T_X, s_X >$, where S_X is a set of states, A_X is a set of actions, F_X is a set of terminal states, M_X is a set of messages that may be sent or received, V_X is a set of variables and T_X is a transition function $S_X * A_X \longrightarrow S_X$, and s_X is the initial state. Each action a in A_X has a boolean guard $en(a)$ associated with it which can refer to variables in V_X (we will omit $en(a)$ if it is identically true). A state machine may be viewed as a directed labeled graph where S_X forms the set of nodes, T_X defines the edges and A_X defines the labels of the edges. We use $Seq(X)$ to denote the set of action sequences allowed by X.

We define a cross product operator $\times$ for P and Q as follows: $G = P \times Q$ is an EFSM such that $A_G = A_P \cup A_Q, V_G = V_P \cup V_Q, M_G = M_P \cup M_Q, F_G = \{(p,q) : (p \in F_P) \wedge (q \in F_Q)\}, S_G = \{(p,q) | (p \in S_P) \wedge (q \in S_Q)\}$ and $s_G = (s_P, s_Q)$. The transition relation T_G consists of tuples of the form $(s1, c, s2)$, where $s1 = (u1, v1)$, $s2 = (u2, v2)$ and $s1, s2 \in S_G$. Tuple $(s1, c, s2) \in T_G$ iff:

$c \in A_P \wedge c \notin A_Q$, $(u1, c, u2) \in T_P$ and $v1 = v2$,
$c \notin A_P \wedge c \in A_Q$, $(v1, c, v2) \in T_Q$ and $u1 = u2$,
$c \in A_P \cap A_Q$, $(u1, c, u2) \in T_P$ and $(v1, c, v2) \in T_Q$.

A protocol has a set of service access points (SAPs) associated with it. Each protocol P has a set of service primitive actions $s\_act(P_i)$ associated with SAPi. For ease of presentation of definition in this section, we consider protocols with two SAPs only (namely, SAP1 and SAP2). We define $s\_act(P) = s\_act(P_1) \cup s\_act(P_2)$. The service specification $ss(P)$ of a protocol P is the cross product of a set of finite state machines $\{l_1, l_2, g_1, \ldots, g_m\}$, where l_i is the local specification at SAPi and g_j is a global service specification. The local specification $seq(l_i)$ defines the possible sequences in which actions at SAPi may occur and we require that $A_{l_i} = s\_act(P_i)$ and $V_{l_1} \cap V_{l_2} = \{\}$. The global specification g_i specifies the order in which actions at different SAPs can occur. We do not allow variables to appear in g_i (therefore, $V_{g_i} = \phi$)[1]. We may omit l_i if the sequence of actions defined by l_i can be derived from the global specifications (that is, some g_j completely defines the possible ordering of actions at SAPi). Figure 2 gives the service specification of a connection establishment protocol $Connect_{12}$. In this protocol, the connection is initiated by site 1 ($CReq_{12}$) and site 2 may either accept it ($CPos_{12}$) or reject it ($CNeg_{12}$). If the connection is established, site 1 can disconnect using primitive $DReq_{12}$.

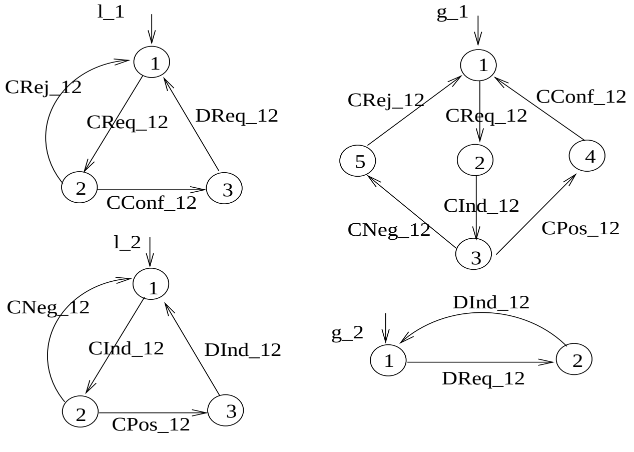


Figure 1: Connection Establishment Protocol: Service specification

The protocol specification, $ps(P)$, is a tuple (P_1, P_2),

---

[1] This assumption has been made to simplify the presentation. Variables in g_i can be used as global variables in the service specification and must be translated into local variables in the protocol specification.

where P_i is an EFSM for SAPi. P_i may contain actions in addition to those associated with SAPi. Thus, $A_{P_i} = s\_act(P_i) \cup A'_i$. Each action a in A'_i may contain local computation and send statements or a receive statement (we use $-m$ to denote "send m" and $+m$ to denote "receive m"). Furthermore, each action in $s\_act(P_i)$ may be modified so that its computation may include local computation and send statements. Figure 2 gives the protocol specification for $Connect_{12}$ assuming reliable and FIFO channels.

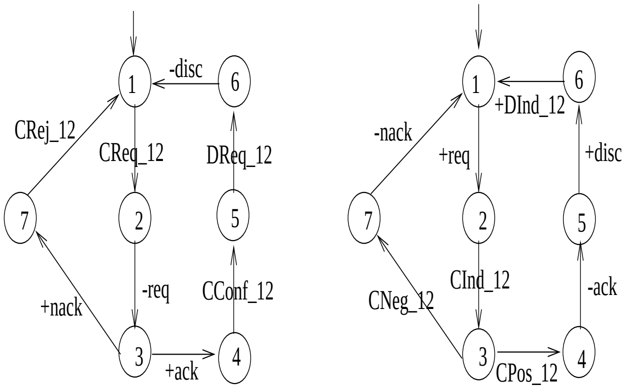


Figure 2: Connection Establishment Protocol: protocol specification

The *global state* of $ps(P)$ is given by $< u, v, x, y >$, where u (v) is the state of P_1 (P_2) and x (y) is the sequence of messages in transit to P_1 (P_2). In this paper, we will assume that communication between processes is asynchronous. The global state may change due to a transition in either P_1 or P_2. If the global state changes from S to S' due to a transition labeled t in either P_1 or P_2, then we say that S' is a *successor* state of S via t. The sequence $ex \equiv S^0 \longrightarrow_{t_1} S^1 \longrightarrow_{t_2} \cdots \longrightarrow_{t_k} S^k$ is an execution of P if S^0 is the initial state and S^i is the successor of S^{i-1} via t_i. Let $transition(ex)$ denote the sequence $t_1; t_2; \ldots; t_k$. We say that ex' is an *extension* of ex via action sequence $s = t_{k+1}; \ldots; t_{k+m}$ if $ex' = ex \longrightarrow_{t_{k+1}} S^{k+1} \cdots \longrightarrow_{t_{k+m}} S^{k+m}$. A state u of P_i is a *receiving state* if all transitions incident from u involve receiving a message. A state u is a *final state* if it has no transitions incident from it. Let λ denote an empty sequence. A state pair (u, v) is *stable* if (u, v, λ, λ) is a reachable global state and if $(u, v, , x, y)$ is a reachable state then $x = y = \lambda$.

Definition 1: $ps(P)$ is *deadlock free* if there does not exist a reachable state (u, v, x, y) such that both u and v are non-terminal states such that no transition is enabled from u and v.

Definition 2: $ps(P)$ is *free from unspecified receptions* if there does not exist a reachable state (u, v, x, y) such that (a) $first(x) = m$ and there is no enabled transition incident from u that receives m, or (b) $first(y) = m$ and there is no enabled transition from v that receives m.

Definition 3: $ss(P)$ is *deadlock free* if there does not exist a non-terminal state s such that there is no enabled transition from s.

$ps(P)$ is $safe$ iff $ps(P)$ is deadlock free and free from unspecified receptions. $ss(P)$ is $safe$ iff $ss(P)$ is deadlock free. We define $proj(seq, A)$ as the sequence of actions obtained by removing from seq all actions not in A.

Definition 4: $ps(P)$ $satisfies$ $ss(P)$ if:

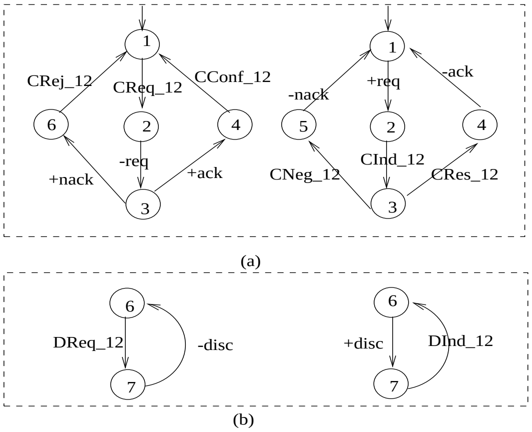
1. for each sequence $seq \in Seq(ss(P))$, there exists an execution ex of $ps(P)$ such that $proj(transition(ex), s\_act(P)) = seq$ and if $seq; a \in Seq(ss(P))$ then there exists an extension ex' of ex via $s; a$, where s does not include any action in $s\_act(P)$.
2. for each execution ex of $ps(P)$, $proj(transition(ex), s\_act(P)) \in Seq(ss(P))$ and if there exists an extension ex' of ex via $s; a$, where s does not include any action in $s\_act(P)$ then $proj(transition(ex'), s\_act(P)) \in Seq(ss(P))$.

3 Problem definition

Several techniques have been proposed for composition at the protocol specification level such as:

- sequential composition, that allows a multiphase protocol to be constructed by composing protocols for the individual phases in sequence [2, 1, 9, 17],
- alternative composition, that combines a set of protocols such that at most one component protocol can be active at any given time [9, 16], and
- parallel composition, that allows protocols in which multiple functions can be performed concurrently to be constructed [8, 14, 15].

In [13], each of these techniques were studied for specifications involving timing information. Each of these techniques can be viewed as imposing some constraints on the combined execution of the component protocols. These constraints may be based on actions or states of the component protocols. For instance, the sequential composition operator in [2] specifies a constraint that Q_i cannot start executing until P_i reaches a final state. Similarly, constraints that inhibit execution of certain actions in Q on the occurrence of a specific action in P have been used in [9, 17] to define transition between protocols. However, to ensure the safety of the composite protocol, each technique provides inference rules that may restrict the actions or states that can be used in specifying the constraints and also the type of protocols that can be combined. For example, the inference rule for sequential composition in [9, 17] requires that the execution of Q may start when P has reached a stable state pair and that the set of stable state pair satisfy a closure property. Finally, the inference rules in [14, 15] restrict the type of invariants and liveness properties of the component protocols that can be inferred in the composite protocol.



(a)

(b)

Figure 3: (a) Connection protocol, (b) Disconnection protocol

We find these restrictions to be conservative in most cases. Since they are not necessary conditions, one might still be able to construct composite protocols from a set of component protocols that do not satisfy these conditions. To take full advantage of the compositional techniques, we must allow such compositions. This would however require a more flexible use of constraints in defining compositions. The following examples illustrate these compositions:

Example 3.1: Figure 3 gives the protocol specifications of the connection and disconnection phases separately. We may wish to combine these phases to obtain the connection management protocol shown in Figure 2. To perform this composition, we must specify that each execution of $DReq_{12}$ must be preceded by an occurrence of $CConf_{12}$. In addition, a new request $CReq_{12}$ can be made only after $DReq_{12}$ has occurred. This protocol cannot be designed directly using existing sequential composition techniques (it is possible to derive it indirectly by first converting the component protocols to non-iterative versions and combining them). □

Example 3.2: Consider the protocol, $Connect_{12}$, shown in Figure 2 in which site 1 establishes a connection with site 2. We can obtain a similar protocol $Connect_{21}$ in which site 2 establishes a connection with site 1. Consider the alternative composition of these two protocols in which connection can only be established in one direction at a time. If both attempt to establish a connection at the same time, $Connect_{12}$ is given priority (thus, in $Connect_{12}$, we want a positive response, $CPos_{12}$, to be generated whereas in $Connect_{21}$, we want a negative response, $CNeg_{21}$ to be generated). This can be accomplished by imposing a constraint that if site 1 has sent a request to site 2 ($CReq_{12}$), then site 1 is not allowed to execute $CPos_{21}$. Only after the occurrence of $CRej_{12}$ or $DReq_{12}$ (which marks the termination of the connection from site 1 to site 2), $CPos_{21}$ is allowed to execute. Similar restrictions have to be placed for other cases. This is an example of a constraint that allow temporary inhibition of actions in one protocol on the occurrence of an action in the other protocol. □

In the full paper, we also give an example to illustrate the flexible use of constraints in the context of parallel composition. These examples illustrate the need for more flexibility in specifying constraints than allowed by the existing techniques. [8] proposes a similar approach in which constraints are defined by strengthening guards in one protocol using proposition involving variables of the other protocol. However, it is possible that if we allow constraints to be imposed in an arbitrary manner, it may lead to deadlocks or other undesirable behavior in the composite protocol. Hence, protocols constructed using such an approach would have to be analyzed directly. Since the state space of a composite protocol is typically large, this analysis will be expensive.

In this paper, we discuss an alternative approach in which composition is performed at the service specification level. In defining these compositions, we do not impose any restrictions in defining the constraints. Although the protocol specifications are complicated (many times due to message loss, timeouts, etc), the service specifications are relatively much simpler. Thus, even though we allow arbitrary compositions of service specifications, we can analyze them as their state space is not very large.

More formally, the contributions can be stated as follows: Assume that we are given $ss(P)$, $ss(Q)$, $ps(P)$ and $ps(Q)$ such that $ps(P)$ satisfies $ss(P)$ and $ps(Q)$ satisfies $ss(P)$. We give a constraint based methodology to combine service specifications of P and Q. Thus, if R is the composite protocol, then $ss(R)$ is given by $ss(P) \times ss(Q)$ subject to a set of constraints

SC. Given this composition, we give an algorithm to construct $ps(R)$ from $ps(P)$, $ps(Q)$ and SC. We show that if $ss(R)$ is safe, then we can conclude that $ps(R)$ is safe and satisfies $ss(R)$. Thus, the technique avoids the analysis of $ps(R)$.

In performing compositions of service specifications, we can only impose constraints on actions belonging to $s\_act(P)$ and $s\_act(Q)$ (thus, we cannot impose constraints on sending and receiving actions). However, we find that in most cases, imposing constraints on service primitives is not a stringent restriction as the main aim of protocol composition is to combine the services of P and Q in a certain manner. This combination can be defined in a more intuitive and clear manner at the service specification level.

There has been significant amount of work in the area of synthesizing protocol specifications from service specifications [3, 6, 12]. In some techniques, service specifications have been structured using composition operators and algorithms to implement these operators at the protocol specification level have been provided. We defer the detailed comparison to these synthesis techniques after the presentation of our approach.

4 Composition of service specifications

In this section, we discuss our framework for composition of service specifications. We first present a basic set of constraints to illustrate the technique. More complex constraints can be added to the framework, some of which are discussed subsequently. Given two service specifications, $ss(P)$ and $ss(Q)$, we define their composite specification using a set of constraints. Let $ss(R) = ss(P) \times ss(Q)$. Then, the constraints are imposed on the set of executions of $ss(R)$. Let $ex = S^0 \longrightarrow_{t_1} S^1 \longrightarrow_{t_2} \cdots \longrightarrow_{t_k} S^k$ be an execution sequence of R, where S^0 is the initial state of $ss(R)$. Let l_a and l_b be the number of occurrences of a and b in ex, respectively.

1. The synchronizing constraint is specified as $synch(a, b)$, where $a \in s\_act(P_i)$ and $b \in s\_act(Q_i)$ (both actions must belong to the same SAP). Constraint $synch(a, b)$ requires that if a or b occurs in ex, then except for the last occurrence of a or b when $l_a \neq l_b$, the execution of a and b in ex be delayed until a and b are enabled and then both are executed in either order but atomically, i.e., no other action at site i can be interleaved between the execution of a and b.
2. The ordering constraint is specified as $order(A, B)$, where $A, B \subseteq s\_act(P_i) \cup s\_act(Q_i)$. Constraint $order(A, B)$ requires that the execution of actions in A and B must alternate and the first action in B cannot occur until an action in A occurs.
3. The interrupt constraint is specified as $interrupt(A, B, C)$, where $A, C \subseteq s\_act(P_i)$ and $B \subseteq s\_act(Q_i)$. Constraint $interrupt(A, B, C)$ requires that after the occurrence of any action in A, an action in B cannot be executed until an action in C occurs. The interrupt constraint in which $A, C \subseteq s\_act(Q_i)$ and $B \subseteq s\_act(P_i)$ is defined similarly.
4. The enabling constraint is specified as $enable(A, Q_i)$, where $A \subseteq s\_act(P_i)$. Constraint $enable(A, Q_i)$ requires that Q_i can begin execution only after the occurrence of an action in A. The constraint $enable(B, P_i)$ is defined similarly.

Given a set of constraints SC, the service specification of the composite protocol R is a state machine whose executions are those of $ss(P) \times ss(Q)$ that satisfy SC. The set of constraints discussed above only restrict the set of executions of $ss(P) \times ss(Q)$ (each constraint either delays the execution of an action or limits the choice of actions available in a state). Hence, we have the following lemma (the proof of the lemma is straightforward).

Lemma 4.1 *If $ss(R)$ is deadlock free then for any execution sequence $ex \in Seq(ss(R))$, $proj(ex, act(P)) \in Seq(ss(P))$ and $proj(ex, act(Q)) \in Seq(ss(Q))$.*

Although we can explicitly construct $ss(R)$ to perform its analysis, we use an alternative approach. To perform analysis of $ss(R)$, we have implemented an algorithm that performs reachability analysis by taking $ss(P)$, $ss(Q)$ and SC as input. These constraints are enforced during the generation of the state space. Thus, the composite state machine for $ss(R)$ is generated on-the-fly. For each constraint, the algorithm keeps track of appropriate state information (for instance, for the constraint $enable(A, Q_i)$, it keeps track of whether an action of A has already occurred and allows execution of an action in Q_i only if this condition holds). At present, the algorithm is able to detect deadlocks and verify invariants.

In the following, we give examples of compositions of protocols (in specifying the constraints, for simplicity, a singleton set $\{a\}$ will be written as a):

Example 4.1: We can combine $ss(Connect)$ and $ss(Disconnect)$ of Example 3.1 using the following constraints: We have to specify that iterations of *Connect* and *Disconnect* must alternate. The constraint $order(CConf_{12}, DReq_{12})$ ensures that the x^{th}

iteration of $Disconnect$ is initiated only after the x^{th} establishment of the connection. We also need to specify $order(CReq_{12}, DReq_{12})$ to ensure that the x^{th} connection request can be generated only after the termination of $x - 1^{st}$ iteration of $Disconnect$. However, this constraint results in a deadlock. We must impose the constraint $order(CReq_{12}, \{DReq_{12}, CRej_{12}\})$ instead as each connection request may not result in a connection establishment. □

Example 4.2: Consider the composition of the protocol $Connect_{12}$ and $Connect_{21}$ as discussed in Example 3.2. The constraint $interrupt(CReq_{12}, CPos_{21}, \{DReq_{12}, CRej_{12}\})$ specifies that after site 1 has requested connection to site 2, site 1 cannot accept the connection request from 2 until the connection from site 1 to site 2 is either terminated or denied. Similarly, $interrupt(CInd_{21}, CReq_{12}, \{CNeg_{21}, DInd_{21}\})$ specifies that if site 1 has already received a connection request from site 2, then site 1 cannot send a connection request to site 2 until the connection from site 2 to site 1 is terminated or denied. $interrupt(CInd_{12}, CRes_{21}, \{CNeg_{12}, DInd_{12}\})$ specifies a similar constraint for the opposite direction. These constraints result in a deadlock free service specification that has the desired properties. □

Example 4.3: In many complex protocols such as those for multimedia collaboration, several sessions may have to be established. Each session may be controlled by a separate connection management protocol. These protocols could have been designed independently and may have different rules for connection establishment and disconnection. Composing such protocols can be a non-trivial task as these sessions may interact with one another in a complex manner. Specifying these interactions require the type of constraints that we have proposed. In the full paper, we design a multimedia protocol by composing protocols to managing a control session, an audio session and a text session. We show how constraints can lead to deadlocks in the composite session management protocol. □

Example 4.4: Consider the sequential composition of the protocols in Figure 4. Sequential composition of P and Q implies that at each site, all actions of P must precede any action of Q. However, for the specification in Figure 4, we cannot define $ps(R_i)$ as $ps(P_i); ps(Q_i)$ (after reaching state 4, P_1 cannot switch to Q_1 because it does not know whether $m2$ will arrive or not). On the other hand, we can define $ss(R_i)$ as $ss(P_i); ss(Q_i)$ using the enabling constraints $enable(\{A, B\}, Q_1)$ and $enable(\{C, D\}, Q_2)$ (as discussed in the algorithm later, this sequential execution of $ss(R)$ is not translated into a sequential execution at the protocol specification level). This example illustrates the case where existing composition operators cannot be applied to the protocol specifications but are applicable to the corresponding service specifications. □

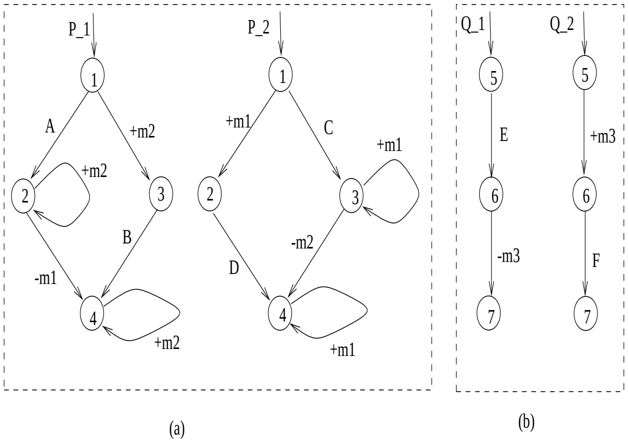


Figure 4: Sequential composition of protocols

5 Composition of protocol specifications

In this section, we give an algorithm to combine $ps(P)$ and $ps(Q)$ to obtain $ps(R)$ using the set of constraints SC specified at the service specification level. When combining P_i and Q_i, we require that the send and receive statements from P_i and Q_i operate on the same set of input channels and output channels of R_i. Without loss of generality, we have the following two assumptions regarding communications in R: (1) The message sets of P_i and Q_i are disjoint; and (2) The bound on a channel in R is the sum of the bounds on the same channel in P and Q. We further assume that the local variable sets of P_i and Q_i are disjoint. Finally, an action $a \in act(P)$ may appear as a label in more than one transition in $ss(P)$ or $ps(P)$. We assume that a constraint on a refers to all transitions with label a.

Due to property shown in Lemma 4.1, the enforcement of the constraints at the protocol specification level is simple. The algorithm involves three steps. The first step introduces new variables and states for each constraint. The next step transforms P_i and Q_i by adding new conjuncts and/or local statements to their respective transitions.

Step 1:

- If $synch(a, b) \in SC$, then add variable syn_{ab} with possible values $(0,1,2)$ and with initial value 0.

- If $order(A,B) \in SC$, then add a boolean variable ord_{AB} with initial value $false$.
- If $interrupt(A,B,C) \in SC$, then we add a new variable inh_{ABC} with initial value $false$.
- If $enable(A,Q_i) \in SC$, then we add a new variable enb_{Q_i} with initial value $false$.

In addition, we introduce a variable $synch_i$ for each site i.

Step 2: We modify each action in P_i and Q_i by adding conjunct(s) and/or local statement(s) to its enabling condition and computation. Specifically, for each $a \in A_{P_i}$, we do the following:

1. if a is not involved in a synchronization constraint then add $\neg synch_i$ to $en(a)$ as a conjunct.
2. $synch(a,b) \in SC$. Then for a, add $(synch_i \wedge syn_{ab} = 2) \vee (\neg synch_i)$ and $en(b)$ as conjuncts to $en(a)$ and add statement:
   "if $syn_{ab} = 0$ then $syn_{ab} := 1$; $synch_i = true$ else $syn_{ab} := 0$; $synch_i = false$"
   to its computation. For b, add $(synch_i \wedge syn_{ab} = 1) \vee (\neg synch_i)$ and $en(a)$ as conjuncts to $en(b)$ and add statement
   "if $syn_{ab} = 0$ then $syn_{ab} := 2$; $synch_i = true$ else $syn_{ab} := 0$; $synch_i = false$"
   to its computation.
3. $order(A,B) \in SC$. Then for all $a \in A$, add conjunct $\neg ord_{AB}$ to $en(a)$ and add statement "$ord_{AB} := true$" to its computation. For all $b \in B$, add conjunct ord_{AB} to $en(b)$ and add statement "$ord_{AB} := false$" to its computation. A similar action is taken if $order(B,A) \in C$.
4. $interrupt(A,B,C) \in SC$. Then, for all $a \in A$, add the statement "$inh_{ABC} := true$". For all $c \in C$, add the statement "$inh_{ABC} := false$". For all $b \in B$, add $\neg inh_{ABC}$ as conjunct to $en(b)$.
5. $enable(A,Q_i) \in SC$. Then, for all $a \in A$, add the statement "$enb_{Q_i} := true$". For all actions $b \in s\_act(Q_i)$ such that b is reachable from the initial state via a sequence of send or receive statements, add enb_{Q_i} as conjunct to $en(b)$.

Step 3: We compute $R'_i = P_i \times Q_i$ (note that P_i is different from the original one. For notational simplicity, we still denote it as P_i instead of P'_i. The same remark applies to Q_i). Each state in R'_i is of the form $u.v$ where u is a state of P_i and v is a state of Q_i. We construct R_i from R'_i by removing the following transitions: Let $synch(a,b) \in SC$. Let $head(a)$ $(tail(a))$ be the set of states in P_i with transitions labeled a incident from (to) them. $head(b)$ and $tail(b)$ are defined similarly. Then, we remove all transitions from R'_i labeled a except those incident from states of the form $u.v$, where $u \in head(a)$ and $v \in head(b) \cup tail(b)$. Similarly, we remove all transitions from R'_i labeled b except those incident from states of the form $u.v$, where $v \in head(b)$ and $u \in head(a) \cup tail(a)$.

Lemma 5.1 *Let $ss(R)$ be a specification obtained by composing $ss(P)$ and $ss(Q)$ using a set SC of constraints, and $ps(R)$ be obtained from $ps(P)$, $ps(Q)$ and SC using our algorithm. Then, $ps(R)$ is safe and satisfies $ss(R)$ if*

- *$ps(P)$ satisfies $ss(P)$ and $ps(Q)$ satisfies $ss(Q)$*
- *$ps(P)$ and $ps(Q)$ are safe*
- *$ss(R)$ is deadlock free.*

Lemma 5.1 allows us to infer properties of $ps(R)$ by analyzing $ss(R)$. In particular, we can infer that $ps(R)$ is deadlock free by analyzing $ss(R)$. In the examples discussed earlier, the analysis of the composite protocol specification for deadlock freedom directly would require a much larger state space search as compared to the analysis of the service specification (especially if the protocol specifications are designed under assumptions of message loss or reordering etc). Another advantage is that we can derive safety and liveness properties of $ps(R)$ by analyzing $ss(R)$ since $ps(R)$ satisfies $ss(R)$.

6 Extensions of the framework

In this section, we discuss several ways in which our framework can be extended:

- **Alternative composition**: A protocol designed using alternative composition of P and Q provides the functionality of either P or Q, one at a time. Several different semantics of alternative composition have been studied in the literature. For example, consider the composition of $Connect_{12}$ and $Connect_{21}$ of Example 3.2. Consider the case in which both protocols attempt to establish a connection at the same time. In the composition discussed in Example 3.2, we assign priority to $Connect_{12}$ so that if a request is made in this protocol, a concurrent request by $Connect_{21}$ can either be denied (by $CNeg_{21}$) or withheld until the connection in $Connect_{12}$ terminates. Other type of compositions have been discussed in which both protocols are denied the request [13] or only one of them is denied by assinging priorities [9]. In these compositions [9, 13], if a protocol is denied the connection, its execution is "aborted" and restarted from the initial state at a later time. The notion of "aborting" introduces new executions in the composite protocol (if Q is aborted then there may exist a partial execution of Q) and hence, Lemma 4.1 is not satisfied. To ensure the safety of the composite protocol, the abort actions must be performed in a disciplined manner (for example, in [9], if P_i can abort Q_i, then Q_i is aborted on the initiation of P_i and before it receives any response from its peer process). In our framework, we can introduce a new constraint of the form $abort(a, Q_i, b)$, where $a, b \in s\_act(P_i)$, such that Q_i is aborted on the occurrence of Q_i and is reset to its initial state on the occurrence of b. Such a constraint can not only model the alternative composition but other types of compositions in which, for instance, P and Q may initially execute concurrently and if certain actions occur in P, Q is aborted. The implementation of the *abort* constraint at the protocol specification level requires introduction of new messages and new states. In particular, on the execution of a, a special message *Abort* is sent to Q_j and Q_i enters a special abort state. In this state, it discards all message it may receive from Q_j. Process Q_j responds with an *Abort_ack* message on receiving the *Abort* message.

- **Conditional and state-based constraints**: At present, our framework allows specification of constraints on actions only. In some cases, we may want to use a combination of states and actions to specify constraints. For instance, we may want to specify that a constraint between actions be imposed only when the protocol is in a specific state. As another example, we may want to specify that an action a in P_i be enabled only when Q_i has reached a specific state. In the full paper, we discuss the incorporation of these constraints in our framework.

- **Shared Actions, Global Variables and Timing Information**: Our framework currently assumes that the actions and variables of P and Q are disjoint. In some cases, we may want to allow $ss(P)$ and $ss(Q)$ to share actions and variables. We can incorporate this feature in our framework with the following provisions: If $a \in s\_act(P)$ and $b \in s\_act(Q)$ are shared then $synch(a, b) \in SC$. This restriction is similar to the one in [14] where we studied sharing of actions and variables at the protocol specification level.

We also assume that an EFSM for a global specification does not include variables. Allowing variables in the global specification is advantageous as it allows very high-level specifications. Incorporating global variables is a subject of future research (the main problem encountered is that with global variables, an action in $ss(P)$ may be broken into several actions in $ps(P)$, and therefore, there may not exist a one-to-one correspondence between actions in $ss(P)$ and $ps(P)$). Finally, we would like to extend our work to include timed extended finite state machines. In [13], the problem of composing timed specifications at the protocol specification level has been studied. As in techniques for non-timed specifications, some restrictions have been placed to ensure safe compositions. For example, if $synch(a, b) \in SC$ then [13] requires the time interval associated with a and b to be $[0, \infty]$. We believe that by studying composition of timed specifications at the service specification level, we can weaken such restrictions.

7 Related Work

Protocol composition operators have been studied extensively in the literature. In particular, Lotos provides enabling, disabling and choice operators that can be used to combine service specifications [5]. For example, the enabling operator, $P >> Q$, specifies that after the completion of P, Q must be enabled (this requires actions at all SAPs in P to precede all actions in Q). [18] proposed the idea of "specification engineering" which advocates more flexibility in combining specifications. In [18], a number of extensions of Lotos operators were proposed. For instance, consider the Lotos process $A = A\_Req;\ A\_Ind;\ A\_Res;\ A\_Conf$. To use the enabling operator to specify that process B is enabled after the occurrence of A_Req, we have to decompose A into two subprocesses: $A1 = A\_Req$ and $A2 = A\_Ind; A\_Res; A\_Conf$. To overcome this problem, [18] defined operators such as $enables\_after\_ack$, $enables\_after\_try$, $interrupts\_after\_ack$, $interrupts\_after\_try$, $interleaves$, $alternate$, $overtakes$, etc. For example, $enables\_after\_try(A, B)$ specifies that service $B\_Req;\ B\_Ind;\ B\_Res;\ B\_Conf$ can be enabled after action $AReq$ has occurred (this action represents the fact that A has attempted to provide the service). The work in [18] share the same goals as our approach of providing more flexibility in combining services. We find that several of the operators in [18] can be specified using constraints in our framework.

A number of algorithms to synthesize protocol specifications from service specifications have been proposed [3, 6, 4, 5]. Using this approach, $ss(R)$ can be first derived from $ss(P)$ and $ss(Q)$ and then $ps(R)$ can be derived from $ss(R)$ using a synthesis algorithm. Our work differs in two respects. First, the synthesis algorithms have been proposed for a fixed set of operators. For example, [4] considers the enabling, disabling and choice operators of Lotos. Similarly, [12] discussed sequential, iterative and alternative composition operators for combine service specifications in the FSM model. The motivation of our work is to allow more flexible compositions than those allowed by such operators. It may be possible to encode some of the constraints using Lotos expressions (which can be composed in parallel with $ss(P)$ and $ss(Q)$). However, this may require extensive decomposition of $ss(P)$ and $ss(Q)$ into smaller expressions. Second, our framework allows the use of any protocol specification $ps(P)$ that satisfies $ss(P)$. In particular, we can use manually designed protocol specifications that may be more efficient. The synthesis approach derives a protocol specification from the composite service specification, and are therefore restricted to only those protocols that can be synthesized (the proposed algorithms typically place restrictions on the service specification that can be translated into protocol specification).

Another approach is the refinement methodology proposed in [11, 7]. In this approach, one starts with a high-level specification A that is refined in a stepwise manner to a more detailed specification $refined(A)$, such that $refined(A)$ satisfies A. This approach can be used to refine $ss(P)$ to $ss(R)$. In [11], specifications are given as I/O automata and a parallel composition operator ($\|$) is defined, where $A \parallel B$ is an I/O automata in which A and B are synchronized on common actions. It is shown that $A \parallel B$ satisfies $refined(A) \parallel refined(B)$. We believe that the I/O automata model can be extended to include composition using constraints (in addition to synchronization) discussed in our framework.

8 Conclusion

Protocol composition has been advocated as an attractive way to design complex protocols. Several techniques have been studied to perform composition at the protocol specification level. In this paper, we studied the problem of combining protocols at the service specification level. The composition of $ss(P)$ and $ss(Q)$ is defined with respect to a set of constraints. These constraints provide a very flexible approach to combine protocols. Given the composite service specification $ss(R)$, we provided an algorithm to derive $ps(R)$ from $ps(P)$ and $ps(Q)$. We show that analysis of $ss(R)$ is sufficient to ensure that $ps(R)$ satisfies certain safety and liveness properties. This results in efficient validation as state space of $ss(R)$ is typically significantly smaller than that of $ps(R)$.

References

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

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

[3] P. Chu and M.T. Liu. Protocol synthesis in a state transition model. In *Proceedings of COMPSAC*, 1988.

[4] C. Kant, T. Higoashino, and G.v. Bochmann. Deriving protocol specifications from service specifications written in lotos. Technical Report 805, University of Montreal, 1992.

[5] R. Khendek, G.v. Bochmann, and C. Kant. New results on deriving protocol specifications from service specifications. In *Proceedings of the SIGCOMM Symposium*, 1989.

[6] A. Khoumsi and G.v. Bochmann. Protocol synthesis using basic Lotos and global variables. In *Proceedings of the International Conference on Network Protocols*, 1995.

[7] S. Lam and A. Udaya Shankar. A theory of interfaces and modules i–composition theorem. *IEEE Transactions on Software Engineering*, 20(1):55–71, 1994.

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

[9] H. Lin. Constructing protocols with alternative functions. *IEEE Transactions on Computers*, 40(4), 1991.

[10] H. Lin and C. Tarng. An improved method for constructing multiphase communications protocols. *IEEE Transactions on Computers*, 42(1), 1993.

[11] N. Lynch and M. Tuttle. Hierarchical correctness proofs of distributed algorithms. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 1987.

[12] M. Nakamura, Y. Kakuda, and T. Kikuno. On constructing communication protocols from component-based service specifications. *Computer Communications*, 19, 1996.

[13] J. Park and R. Miller. A compositional approach for designing multifunction time-dependent protocols. In *Proceedings of the International Conference on Network Protocols*, 1997.

[14] G. Singh. A compositional approach for designing protocols. In *Proceedings of the IEEE Int'l Conference on Network Protocols*, 1993.

[15] G. Singh. A methodology for designing communication protocols. In *Proceedings of the ACM SIGCOMM conference*, 1994.

[16] G. Singh and Z. Mao. Structured design of communication protocols. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, 1986.

[17] G. Singh and M. Sammeta. On the construction of multiphase communication protocols. In *Proceedings of the IEEE Int'l Conference on Network Protocols*, 1994.

[18] K.J. Turner. An engineering approach to formal methods. In *Proc. of the Protocol Specification, Testing and Verification*, 1993.