Hindawi Publishing Corporation
Journal of Computer Systems, Networks, and Communications
Volume 2008, Article ID 794960, 13 pages
doi:10.1155/2008/794960

Research Article

Service-Oriented Synthesis of Distributed and Concurrent Protocol Specifications

Jehad Al Dallal and Kassem Saleh

Department of Information Sciences, College for Women, Kuwait University, P.O. Box 5969, Safat 13060, Kuwait

Correspondence should be addressed to Jehad Al Dallal, jehad@cfw.kuniv.edu

Received 28 February 2008; Accepted 8 July 2008

Recommended by Nidal Nasser

Several methods have been proposed for synthesizing computer communication protocol specifications from service specifications. Some protocol synthesis methods based on the finite state machine (FSM) model assume that primitives in the service specifications cannot be executed simultaneously. Others either handle only controlled primitive concurrency or have tight restrictions on the applicable FSM topologies. As a result, these synthesis methods are not applicable to an interesting variety of inherently concurrent applications, such as the Internet and mobile communication systems. This paper proposes a concurrent-based protocol synthesis method that eliminates the restrictions imposed by the earlier methods. The proposed method uses a synthesis method to obtain a sequential protocol specification (P-SPEC) from a given service specification (S-SPEC). The resulting P-SPEC is then remodeled to consider the concurrency behavior specified in the S-SPEC, while guaranteeing that P-SPEC provides the specified service.

1. INTRODUCTION

A protocol can be defined as an agreement for the orderly and timely exchange of messages among communicating entities. A full protocol description comprises a precise format for valid messages (syntax), rules for the orderly message exchanges (grammar), and vocabulary of valid messages that can be exchanged with their clear meaning (semantics).

In protocol design, interacting entities are constructed to provide a set of specified services for distributed service users. While designing a communication protocol, semantic and syntactic errors may exist. Semantic design errors cause incorrect services to be provided to distributed service users. Syntactic design errors can cause the protocol to deadlock. There are three common types of syntactic design error, including (1) an unspecified reception error, when the protocol reaches a state where it is not able to handle the message that may arrive; (2) a deadlock error, when the protocol is at a nonfinal state, all channels are empty, and no transmission transition is specified; (3) a livelock error, when the protocol entities are locked in a loop exchanging messages that are not contributing to the provision of the desired service.

A communication system is most conveniently structured in layers. The service access point (SAP) is an interaction point through which a layer can communicate with the upper layer or service users, or with the lower layer. A layer can service several SAPs. The communication is performed using service primitives (SPs). The SP identifies the type of the message and the SAP at which it occurs.

From the user's viewpoint, at a high level of abstraction, the layer can be viewed as a black box where only interactions with the user, identified by SPs, are observable. The specification of the service provided by the layer is defined by the ordering of the visible SPs and is called service specification (S-SPEC). At a refined and lower level of abstraction, the service provided by the layer is performed using a number of cooperating protocol entities. The number of protocol entities is equal to the number of SAPs available at the layer. In a distributed system, the protocol entities are geographically distributed and exchange protocol messages through a communication medium. The protocol specification (P-SPEC) consists of the specification of each of the protocol entities, each describing the interactions needed to deliver the specified service. Figure 1 shows the two levels of

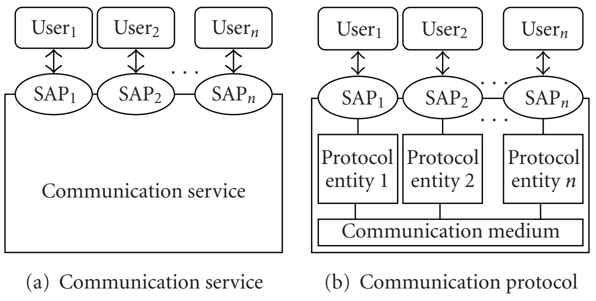


FIGURE 1: A communication system at two levels of abstraction.

abstraction of a communication service. Both S-SPEC and P-SPEC can be modeled using finite state machines (FSMs).

In autonomous communication systems such as the Internet and mobile communication systems, a user can initiate a service at any time. As a result, distributed users at different SAPs may issue simultaneous service requests; consequently, it is possible that two or more users will simultaneously issue service requests to each other. This situation leads to message collision. If the protocol specification is not properly designed to deal with this situation, an unspecified reception error may occur.

For example, mobile games use WiFi and Bluetooth technologies [1]. In such games, distributed players use different machines to play with each other simultaneously and communicate over wireless networks. If the communication protocol providing this game service is improperly designed, the game will be blocked indefinitely or will function improperly.

Concurrent clients and servers can communicate over the Internet using TCP/IP standard or nonstandard protocols [2]. For example, a concurrent client for the TIME service can send requests to multiple servers one at a time and receive responses in an arbitrary order. An unspecified reception error may occur or a deadlock may be reached in the client if its protocol is not designed properly.

Protocol specifications are more complex to specify than service specifications because of their refined nature. Therefore, in a top-down design approach, it is quite natural to start the protocol design process from a complete and unambiguous service specification. The construction or derivation of a protocol specification from a given service specification is called *protocol synthesis*. Instead of applying a sequence of design, analysis, error detection, and correction iteratively until the design is error-free, the protocol synthesis approach does not require any further validation of the synthesized protocol specification. The correctness of the synthesized protocol must be guaranteed as a direct by-product of the synthesis method [3]. The synthesis approach is used to construct or complete a partially specified protocol design, such that the interactions between the constructed or completed protocol entities proceed without encountering any logical error while providing the specified service. Several protocol synthesis methods have appeared in the literature [4–16]. The methods introduced by Saleh and Probert [4], Higashino et al. [7], Yamaguchi et al. [8], Bista et al. [11], Maneerat et al. [12], Yamaguchi et al. [13], Maneerat et al. [14], Dallal [15], and Stakhanova et al. [16] are either not based on the FSM model or support only sequential applications. The method introduced by Saleh and Probert [9] supports only controlled nonsequential applications. The methods introduced by Khoumsi [5], Park and Miller [6], and Kakuda et al. [10] support nonsequential applications that have restrictions on the service specification model topology or on the allowed ordering of service primitives in the service specification.

This paper introduces a method for the synthesis of protocol specifications that do not necessarily start from sequential service specifications. The synthesis method extends the synthesis method introduced in Saleh and Probert [4] to derive the sequential P-SPEC. The synthesis method then remodels the sequential P-SPEC to consider the concurrency behavior specified in the S-SPEC. The synthesis method supports uncontrolled concurrent applications and is free of the restrictions imposed on service specification by the earlier methods. The synthesis method introduced uses FSMs for modeling both service and protocol specifications. This paper does not address the multithreading problem in which an application with different threads runs within one or more processors. Instead, it addresses the more complex problem of concurrency in which an application consists of distributed protocol entities that are running concurrently in geographically distributed processors.

The paper is organized as follows. Related research is overviewed in Section 2. In Section 3, the models used for the service and protocol specifications are defined. In Section 4, the concurrent protocol synthesis method is introduced. Finally, Section 5 concludes the paper and discusses possible extensions in future work.

2. RELATED RESEARCH

Two approaches are used in designing communication protocols: analysis and synthesis. In the analysis approach, a sequence of design, analysis, error detection, and correction is applied iteratively to produce error-free design. In the synthesis approach, the protocol design is constructed or completed in such a way that no further validation is needed. Some protocol synthesis methods start the derivation process from a complete service specification [4–10, 12–15] and others do not [17, 18]. The protocol synthesis methods can be further classified according to the models used, which include finite state machines [4, 6, 7, 9, 10, 15], Petri nets [8, 13], and LOTOS-like [5, 12, 14].

Some of the FSM-model-based service-oriented protocol synthesis methods consider the concurrency behavior of the protocol entities, including Park and Miller [6], Saleh and Probert [9], and Kakuda et al. [10]. In Park and Miller [6], a concurrent timed protocol synthesis method is introduced, with three restrictions. The first restriction is that the S-SPEC model is assumed to be cyclic. The second restriction is that if a path p includes a transition associated with an event that passes through a SAP s, any other path executed in parallel with p cannot include a transition associated with an event that passes through the same SAP s. As a result, the main concurrency problem, message collision, is not resolved. The last restriction is that all the outgoing transitions from a choice state have to be associated with events that pass through the same SAP.

Kakuda et al. [10] introduced a concurrent protocol synthesis method, with two restrictions. The first restriction is that the S-SPEC model must be a tree. The second is that if a collision occurs between two protocol entities, the parallel path that has the higher priority proceeds and the events that have not been executed in the other path yet are cancelled. This is called controlled concurrency. In controlled concurrency, if a message collision occurs, the problem is not solved by handling the collision in such a way that all parallel paths proceed without canceling some or all of their events.

Finally, in Saleh and Probert [9], a concurrent protocol synthesis method is introduced. The synthesis method extends the sequentially based synthesis method introduced earlier by them in Saleh and Probert [4]. After applying the three steps of their original method, the extended method added transitions to solve the controlled concurrency problem. If a message collision occurs, either all parallel paths have to be re-executed or the parallel path that has the highest priority proceeds and the events not yet executed in the other paths are cancelled. This solution is not practical because it results in canceling or re-executing some events.

In this paper, the introduced synthesis method solves the true concurrency problem. In our method, if a message collision occurs, all parallel paths proceed without canceling or re-executing any event. In addition, our method eliminates all the restrictions imposed by the methods introduced in Park and Miller [6] and Saleh and Probert [9], and therefore is applicable to a wider range of applications.

3. MODEL DEFINITION

Both the service and protocol specifications are modeled using FSM-based models. A FSM consists of a finite number of states and transitions. It is only suitable for modeling the sequential behavior of systems. The work we are developing here addresses the synthesis of protocol specifications from a service specification exhibiting concurrent behaviors. Therefore, in this section, the FSM is extended to model the concurrent behaviors in the service specification. The extended model is called the extended finite state machine (EFSM). Each protocol entity is a sequentially based subsystem, and therefore, a traditional FSM can be used without extension to model the protocol entities. In this section, the models used are formally defined in the context of the layered communication system introduced in Section 1.

3.1. Service specification model

The service specification described in the EFSM defines sequences of primitives exchanged between users and processes through the service access points at both upper and lower layers. A service specification specifies a concurrent behavior if two or more service primitives pass through different SAPs simultaneously.

Definition 1. A service specification S-SPEC is modeled by an EFSM denoted by a tuple $(S_s,\ T_s,\ \sigma)$, where

(1) S_s is a nonempty finite set of service states; each state $s \in S_s$ is a choice, fork, joint, or leaf state; a choice state is a state that has one or more outgoing transitions and only one transition is executed; a fork state, denoted by $\|$, is a state that has two or more outgoing transitions associated with service primitives that pass through different SAPs, and all these outgoing transitions are executed simultaneously; a joint state is a state that has two or more incoming transitions and one or more outgoing transitions; a state could be both a fork state and a joint state; finally, a leaf state is a terminal sink state that does not have any outgoing transitions; as in [6], we assume in our work that there exists a unique joint state for each fork state, and vice versa;

(2) T_s is a finite set of transitions, such that each transition $t \in T_s$ is a 3-tuple $\langle$*tail(t), head(t), SP*$\rangle$, where *tail(t)* and *head(t)* are, respectively, the tail and the head states of t, and *SP* is the service primitive that defines the service event, its type, and the index of the SAP, denoted by SAP(SP), at which the *SP* is observed; there are two types of service primitive directions; an SP of type $\uparrow$ is directed upward from the protocol entity to the upper SAP (U-SAP) and directed to the service user at the upper layer; an SP of type $\downarrow$ is directed downward from the service user at the U-SAP and directed to a protocol entity; all service primitives in the S-SPEC involve the U-SAP only;

(3) $\sigma \in S_s$ is the initial service state.

A path p in the ESFM is specified by the sequence of states $(s_i, s_j, \dots, s_k)$ traversed by a set of successive transitions, where $s_j = \text{successive}\,(s_i)_p$. The key difference between the EFSM and the traditional FSM is the presence of the fork state. Parallel paths are specified by the sequence $(s_i, s_j, \dots, s_k)$, where s_i is a fork state and s_k is either a joint or leaf state.

Figure 2 shows a service specification S-SPEC. This example demonstrates a simple data transfer application, consisting of three entities: a server and two machines available at three SAPs. The service described in the S-SPEC is a server-controlled transfer of data between two machines. The server and the two machines are processing concurrently. The service is initiated by the server user by issuing a *Next* SP downward to the sever machine. This service request results in two concurrent upward requests *Next* SP at SAP_2 and SAP_3. The user at SAP_2 issues a downward *Data* SP. Similarly, the user at SAP_3 issues a downward *Data* SP. The downward *Data* SP at SAP_2 will result in an upward *Data* SP at SAP_3 and similarly, the downward *Data* SP at SAP_3 will result in an upward *Data* SP at SAP_2. Then, the service cycles back after receiving *Next* SP downward from the server user. In this example, if one or both machines have nothing to exchange, they send NULL data message. In this example, $S_s = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$,

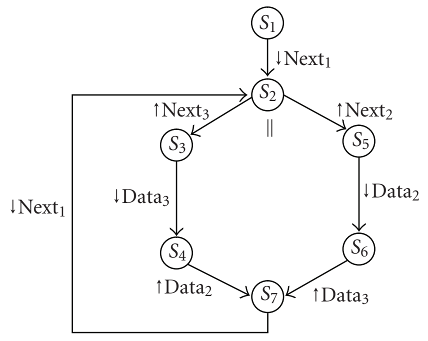


Figure 2: A service specification example.

$T_s = \{\langle s_1, s_2, \downarrow\text{Next}_1\rangle, \langle s_2, s_3, \uparrow\text{Next}_3\rangle, \langle s_3, s_4, \downarrow\text{Data}_3\rangle, \langle s_4, s_7, \uparrow\text{Data}_2\rangle, \langle s_2, s_5, \uparrow\text{Next}_2\rangle, \langle s_5, s_6, \downarrow\text{Data}_2\rangle, \langle s_6, s_7, \uparrow\text{Data}_3\rangle, \langle s_7, s_2, \downarrow\text{Next}_1\rangle\}$, and $\sigma = \{s_1\}$. For the transition $t = \langle s_1, s_2, \downarrow\text{Next}_1\rangle$, the states s_1 and s_2 are, respectively, the tail and head states of t, and the SP is $\downarrow$Next$_1$. For the SP, the event is Next, SAP(SP) = 1, and the type is $\downarrow$. States s_1, s_3, s_4, s_5, and s_6 are choice states and states s_2 and s_7 are fork and joint states, respectively. Paths (s_2, s_3, s_4, s_7) and (s_2, s_5, s_6, s_7) are parallel paths.

3.2. *Protocol specification model*

The protocol specification consists of the specifications of the protocol entities that cooperate to provide the service described in the service specification.

Definition 2. The protocol entity specification PE-SPEC$_i$ is modeled by a FSM denoted by a tuple (S_{pi}, T_{pi}, σ_{pi}), where

(1) S_{pi} is a nonempty finite set of states of protocol entity i; each state $s_{pi} \in S_{pi}$ is an image of one or more S-SPEC states. A state s_{pi} can be an image of more than one S-SPEC state if two or more S-SPEC states are combined during the protocol synthesis process; a path p_p in the PE-SPEC$_i$ is an image of path p_s in the S-SPEC if each state in p_p is an image of one or more states in p_s, and each state in p_s is a preimage of a state in p_p; a parallel path in PE-SPEC is an image of a parallel path in S-SPEC; each state can be either a sending or a receiving state; a sending state is a source state of a transition associated with an SP; a receiving state is a source state of transitions such that all of them are associated with receiving events;

(2) T_{pi} is a finite set of transitions such that each transition $t \in T_{pi}$ is a 3-tuple $\langle$*tail(t)*, *head(t)*, $E_i\rangle$, where *tail(t)* and *head(t)* are, respectively, the tail and the head states of t, and E_i is a protocol event that can be either: (1) an SP that passes through SAP$_i$, (2) an SP that passes through SAP$_i$ and an event message E sent to PE$_j$ denoted by $!e_j$, or (3) an event message E received from PE$_j$ denoted by $?e_j$; an event of the second type is denoted by $E/!e_j$;

(3) $\sigma_{pi} \in S_{pi}$ is the initial protocol state.

Figure 3 shows the three PE-SPECs synthesized from S-SPEC in Figure 2. For PE-SPEC$_1$, $S_{pi} = \{s_1, s_2, s_7\}$, $T_{pi} = \{\langle s_1, s_2, \text{Next}/!\text{next}_{2,3}\rangle, \langle s_2, s_7, ?\text{data}_2\rangle, \langle s_2, s_7, ?\text{data}_3\rangle, \langle s_7, s_2, \text{Next}/!\text{next}_{2,3}\rangle\}$, and $\sigma_{pi} = \{s_1\}$. States s_1 and s_7 are, respectively, the images of states s_1 and s_7 in the S-SPEC shown in Figure 2. State s_2 is the image of the states s_2, s_3, s_4, s_5, and s_6 in the S-SPEC. As a result, the path (s_2, s_7) in PE-SPEC$_1$ is the image of path (s_2, s_3, s_4, s_7) in S-SPEC, and it is therefore a parallel path because it is an image of a parallel path in S-SPEC. Finally, states s_1 and s_7 are sending states, and state s_2 is a receiving state.

In this paper, we assume that the communication medium between the protocol entities is reliable, and the messages are delivered in a first-in-first-out (FIFO) order.

4. SYNTHESIZING CONCURRENT PROTOCOL SPECIFICATIONS

The synthesis of concurrent protocol specifications method introduced in this paper extends a sequentially based synthesis method introduced by Saleh and Probert [4]. In their method, three steps are applied to obtain the PE-SPECs. In the first step, projected service specifications are derived by projecting the S-SPEC onto each SAP. In the second step, a set of transition synthesis rules are applied to the transitions of the projected service specifications to obtain the sequentially based PE-SPECs. In the third step, the PE-SPECs are optimized to obtain the final PE-SPECs as shown in Figure 3. When the S-SPEC includes concurrent behaviors, the PE-SPECs obtained may cause an unspecified reception error. For instance, the S-SPEC given in Figure 2 shows that the transitions outgoing from state s_2 is executed simultaneously. When the PE-SPECs function concurrently, the transitions associated to event *Next* in both PE-SPEC$_2$ and PE-SPEC$_3$ execute simultaneously. After that, the transition associated to event $\downarrow$ Data/!data$_3$ in PE-SPEC$_2$ is executed causing an unspecified reception error to occur in PE-SPEC$_3$. Similarly, the transition associated to event $\downarrow$ Data/!data$_2$ in PE-SPEC$_3$ is executed causing an unspecified reception error to occur in PE-SPEC$_2$.

In our extension, more states and transitions are added to the derived protocol entities to handle the concurrency behaviors. The synthesis algorithm, the additional transition synthesis rules, and the conditions for applying the transition synthesis rules are illustrated below. The syntactic and semantic correctness of our extended synthesis method are provided in Appendix A.

4.1. *The synthesis algorithm*

Starting from a service specification modeled in EFSM, the algorithm outlined in Algorithm 1 automatically derives the concurrent protocol entities that provide the set of services given in the S-SPEC. In the first step, the sequential protocol entities are derived using Saleh and Probert [4] synthesis method. In the second step, using the algorithm outlined in Algorithm 2, the protocol entities are remodeled to consider their concurrent execution.

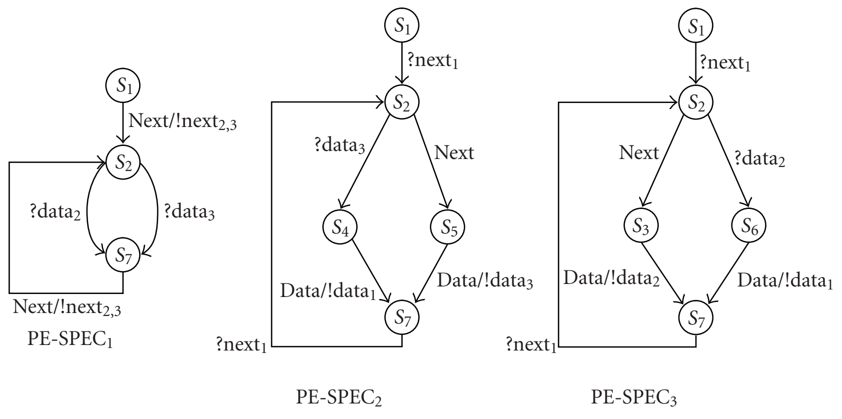


Figure 3: The PE-SPECs synthesized from the S-SPEC given in Figure 2.

Synthesis algorithm. Derivation of concurrent protocol specification from a service specification.
Input: EFSM-based service specification.
Output: Concurrent FSM-based PE-SPECs.
Steps:
(1) apply the sequential-based protocol synthesis method introduced in Saleh and Probert [4],
(2) apply the *From_Sequential_To_Concurrent_PEs* procedure given in Algorithm 2 to obtain the concurrent PE-SPECs.

Algorithm 1: The synthesis algorithm.

4.2. Additional transition rules

To obtain the concurrent PE-SPECs from the sequential ones, the procedure shown in Algorithm 2 is applied. According to this procedure, only the parallel paths are remodeled. First the parallel paths are partitioned according to the type of events associated to the first transition in each path. Then, the paths in each partition are remodeled according to the type of the image of the fork state (i.e., sending or receiving state).

Step 1: partitioning parallel paths

In the protocol entities, the image of the fork state can have one or more outgoing transitions. In the case where there is only one outgoing transition, no message collision can occur. Hence, no modification is required. In the case where there is more than one outgoing transition from a state s, which is an image of a fork state, the outgoing paths from state s are partitioned into two groups according to the type of the event associated to the first transition in each path. There are two possible types of such events: a service primitive or an event reception. Paths in which the first transition is associated to a service primitive are placed in the first group and the others are placed in the second group.

Paths in the first group are initiated by the protocol entity under consideration. On the other hand, the paths in the second group are not initiated until the protocol entity has received the required event. When the protocol entity can execute more than one path in the first group, it has to select one of them. Once the first transition of a parallel path in the first group has been executed, the parallel path is selected and the other parallel paths in the same group are disabled. However, this case is different for the parallel paths in the second group. To summarize, when an image of a fork state in a protocol entity is reached, one of the paths—if there are any—in the first group is executed along with all paths in the second group. The first two steps of the procedure given in Algorithm 3 illustrate how to partition the parallel paths.

Before the parallel paths are remodeled, one more refinement action is required for the paths in the second group. Each path in the second group is divided into subpaths. Each subpath starts with a transition associated with a receiving event and contains only one such transition. The use of these subpaths is illustrated below. The third step of the procedure given in Algorithm 3 shows how to divide the paths in the second group into subpaths.

For example, in PE-SPEC$_1$ given in Figure 3, state s_2 is an image of a fork state and has more than one outgoing transition; therefore, the outgoing parallel paths from state s_2 are partitioned into two groups: G_1 and G_2. The first group is empty because none of the outgoing transitions from state s_2 is associated with a service primitive. Figure 4(a) shows the content of the second group. The subpaths of the parallel paths given in Figure 4(a) are the same as the parallel paths because each of the parallel paths contains only one transition. For PE-SPEC$_2$ given in Figure 3, the outgoing parallel paths from state s_2 are partitioned into two groups. The first and second groups, G_1 and G_2, include the paths that start with a transition associated with a service primitive

```
Procedure: From_Sequential_To_Concurrent_PEs
Inputs: Sequentially based PE-SPECs modeled using FSMs.
Outputs: Corresponding concurrent-based PE-SPECs modeled using FSMs.
Steps:
for each PE-SPEC do
   for each state $s$ which is an image of a fork state in PE-SPEC do
         if state $s$ has more than one outgoing transition then
                  (1) apply Partition_Parallel_Paths procedure,
                  (2) if $s$ is a sending state then
                        (2.1) apply Re-Model_Parallel_Paths_for_Sending_State procedure,
                        (2.2) else apply Re-Model_Parallel_Paths_for_Receiving_State procedure.
```

Algorithm 2: Procedure for obtaining concurrent-based PE-SPECs from sequentially based PE-SPECs.

```
Procedure: Partition_Parallel_Paths
Inputs: A PE-SPEC and an image of a fork state $s$ in the PE-SPEC model.
Outputs: Groups of parallel pathspartitioned at state $s$ and a set of sub-paths of each path in a group.
Steps:
(1) $G_1 = G_2 = \{ \}$,
(2) for each transition $t$ outgoing from state $s$ do
   (2.1) if $t$, which is the first transition in an image of a parallel path $p$, is associated
         with a service primitive then $G_1 = G_1 \cup p$ else $G_2 = G_2 \cup p$,
(3) for each path $p$ in $G_2$ do
   (3.1) $\mathrm{SUB}_p = \{ \}$   {set of sub-paths of $p$},
   (3.2) $s_0 = s$,
   (3.3) while ($s_0$ is not a joint or free state) do
         (3.3.1) $s_1$ = successive $(s_0)_p$,
         (3.3.2) while ($s_1$ is not a joint or free state and the outgoing transition from $s_1$ is not
                 associated with a receiving event) do
              (3.3.2.1) $s_1$ = successive $(s_1)_p$,
         (3.3.3) $\mathrm{SUB}_p = \mathrm{SUB}_p \cup$ subpath of $p$ from $s_0$ to $s_1$,
         (3.3.4) $s_0 = s_1$.
```

Algorithm 3: Partitioning procedure.

and a receiving event, respectively. Figure 4(b) shows the contents of both groups. The subpaths of the parallel path contained in G_2 are the same as the parallel path, because the parallel path contains only a transition associated with a receiving event, followed by a transition associated with a service primitive.

Step 2.1: remodeling parallel paths starting at a sending image of a fork state

In a PE-SPEC that has a sending image of a fork state, the first group of the parallel paths is not empty. As discussed earlier, one of the parallel paths in the first group can be executed along with all the parallel paths in the second group. This means that at any state for a path in the first group, any subpath of any parallel path in the second group has to be allowed to execute, or an unspecified reception error can occur. This can be modeled by making each state in the first group a source state for the first transition in each subpath of each path in the second group.

This solution solves the unspecified reception error problem until the protocol entities reach the image of the joint state. When any of the protocol entities reach the image of the joint state, they have to wait for the other protocol entities to reach their images of the joint state, or an unspecified reception error can occur because of the divergence between the protocol entities. According to the solution for the unspecified reception error discussed above, the protocol entity can either leave the image of the joint state by executing the outgoing transition from the image of the joint state or execute one of the subpaths added to the image of the joint state. An unspecified reception error can occur if the protocol entity executes the outgoing transition from the joint state without executing the other added transitions. Therefore, it is important to ensure that none of the protocol entities leaves the image of the joint state unless all other protocol entities have reached their corresponding state. In other words, it must be ensured that none of the protocol entities leaves the image of the joint state unless the last subpath of each parallel path has been executed. This can

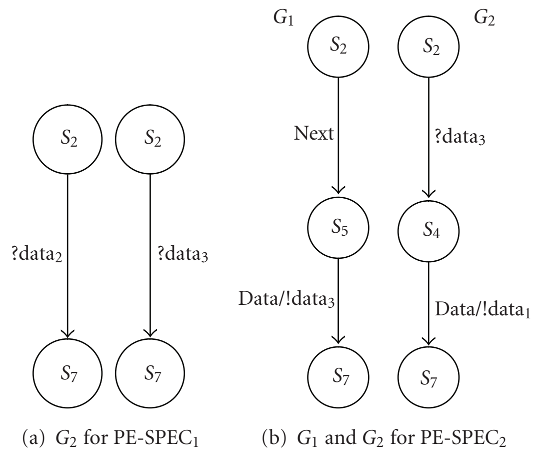


Figure 4: Application of partitioning procedure on the PE-SPECs given in Figure 3.

be modeled by (1) adding copies of the parallel paths in the first group, (2) connecting these copies using the last subpaths of the paths in the second group, and (3) redirecting the outgoing transition from the image of the joint state to be initiated from the state reached by executing all the last subpaths under consideration. Note that if the image of the joint state is a source state for a transition associated to a receiving event, the image of the joint state cannot be left unless the event message is sent by some other protocol entity. Therefore, the solution illustrated above has to be applied only in a protocol entity where the image of the joint state is a source state of a transition associated to a service primitive.

The solutions for the unspecified reception problem and the divergence problem discussed above are given formally in the procedure shown in Algorithm 4. For PE-SPEC$_2$ given in Figure 3, the procedure given in Algorithm 4 is applied because s_2 is a sending state. Step 1 of the procedure is not applicable because joint state s_7 is not a sending state. In Step 2, copies of the subpath of G_2 are added to each state contained in G_1, as shown in Figure 5(a). Step 3 is not applicable because Step 1 is not applicable. Step 4 is not applicable because there are no redundant transitions connected to the images of the fork and joint states. Finally, in Step 5, the path contained in G_2 is removed from PE-SPEC$_2$. The resulting PE-SPEC$_2$ is shown in Figure 5(b). PE-SPEC$_3$ and PE-SPEC$_2$ given in Figure 3 are similar to each other; therefore, the same synthesis steps apply. The resulting PE-SPEC$_3$ is shown in Figure 5(c).

Step 2.2: remodeling parallel paths starting at a receiving image of a fork state

When the image of the fork state is not a source state for any transition associated to a service primitive, the first group, formed using the partitioning procedure given in Algorithm 3, is empty. In this case, to prevent the occurrence of an unspecified reception error, the subpaths of each parallel path in the second group have to be added to each state in each other path in the group. Each state in a parallel path in the group must be made as a source state for the first transition in each subpath of each other path in the group. If the protocol entity under consideration has a sending image of a joint state, the divergence problem has to be solved using the same technique illustrated for Step 2.1.

The procedure given in Algorithm 5 formally illustrates the remodeling technique for the parallel paths starting at a receiving image of a fork state. For PE-SPEC$_1$ given in Figure 3, the procedure given in Algorithm 5 is applied because s_2 is a receiving state. Since the paths in G_2 given in Figure 4(a) end at state s_7, which is a sending state, Step 1 of the procedure is applied. In Step 1.3, a copy of each of the parallel paths is added to PE-SPEC$_1$, as shown in Figure 6(a). In Step 1.4, copies of s_2 and their associated transitions are deleted from the copied paths, as shown in Figure 6(b). In Step 1.5, the parallel paths are connected to their copies, as shown in Figure 6(c). In Step 1.6, states s_7' and s_7'' are joined together in state s_7'. In Step 1.8, the outgoing transition from state s_7 is redirected to initiate from state s_7'. Figure 6(d) shows the application of Steps 1.6 and 1.8. Steps 2, 3, and 4 of the procedure do not make any changes to PE-SPEC$_1$, given in Figure 6(d), because the first group of parallel paths is empty, there are just two paths in the second group, and each of the parallel paths includes one transition.

4.3. Comparison with other methods

In the following, we compare our method for handling concurrency with the two methods we have surveyed earlier in the paper in Section 2.

The synthesis method introduced in Park and Miller [6] cannot be applied to the S-SPEC given in Figure 2 due to restriction 2. This restriction states that if a path p includes a transition associated with an event that passes through a SAP s, any other path executed in parallel with p cannot include a transition associated with an event that passes through the same SAP s. In Figure 2, the events that pass through SAPs 2 and 3 are associated with the transitions of both parallel paths, hence violating the second restriction. As a result, the main concurrency problem due to message collision is not resolved.

The synthesis method described in Kakuda et al. [10] cannot be applied to the S-SPEC in Figure 2 because it violates the first restriction of their method. This restriction states that the S-SPEC model must be a tree. Consequently, their method cannot deal with the specified concurrent behavior.

5. CONCLUSIONS AND FUTURE WORK

In this paper, a synthesis method for concurrent protocol specifications from service specifications is introduced. Both the service and protocol specifications are modeled using FSM-based models. The service specification FSM-based model is extended to model concurrency behaviors. The synthesis method first uses a previously introduced method to synthesize sequentially based protocol specifications, then the synthesis method remodels the derived protocol

```
Procedure: Re-Model_Parallel_Paths_for_Sending_State.
Inputs:
    – a PE-SPEC that has an image state $s$ of a fork state such that $s$ is a sending state,
    – $G_1$, $G_2$, and the subpaths found using the Partition_Parallel_Paths procedure given in Algorithm 5.
Outputs: a remodeled PE-SPEC that has additional states and transitions to consider the
concurrency behavior of the service specification.
Steps:
(1) if the paths in $G_1$ and $G_2$ end with an image of a joint state and this joint image state is a sending state then
    (1.1) LastSUBs = { },
    (1.2) for each path in $G_2$ do
        (1.2.1) LastSUBs = LastSUBs $\cup$ last sub-path in $\text{SUB}_p$,
        (1.2.2) $\text{SUB}_p = \text{SUB}_p -$ last sub-path in $\text{SUB}_p$.
    (1.3) add to the PE-SPEC model $\sum_{i=1}^{n} C(n,i)$ copies of each path in $G_1$, where $n = |G_2|$ (i.e.,
        number of paths in $G_2$) and $C(n,i) = n!/(i!(n-i)!)$,
    (1.4) connect each path in $G_1$ and its copies together using all subpaths in LastSUBs such
        that each combination of the subpaths is covered once. Connecting two paths using a
        subpath includes connecting each state in one of the paths to its corresponding state in
        the other path using the subpath,
    (1.5) join all copied states that have no outgoing transitions together in a state $j$,
    (1.6) delete all redundant transitions connected to state $j$,
    (1.7) redirect the outgoing transitions from the joint state image to be initiated from state $j$,
(2) for each state $sg$ in $G_1$ do
    (2.1) for each path $p$ in $G_2$,
        (2.1.1) for each sub-path in $\text{SUB}_p$ do
            (2.1.1.1) add a copy of the subpath to state $sg$ such that the state $sg$ is, respectively,
                the tail and head state of the first and last transitions in the subpath.
(3) if there are copied paths created using Step 1.3 then
    (3.1) for each copied path $c$ of an original path $p$ do
        (3.1.1) let $q$ be the path from any state in $p$ to its copied state in $c$,
        (3.1.2) for each subpath $sp$ added in Step 2.1.1.1 to a state in $p$ do
            (3.1.2.1) if $sp$ is a subpath of a path $g$ and none of the transitions in $q$ is a subpath in path $g$ then
                (3.1.2.1.1) for each state $sc$ in path $c$ do
                    (3.1.2.1.1.1) add a copy of the subpath $sp$ to state $sc$ such that the state $sc$ is, respectively,
                        the tail and head state of the first and last transitions in the subpath.
(4) Delete redundant transitions connected to the images of the fork and joint states.
(5) Remove all paths in the groups in the set $G_2$ excluding the images of the fork and joint states in these paths.
```

Algorithm 4: A procedure for remodeling the PE-SPEC to consider the concurrency behavior in the case where there is a sending image of fork state.

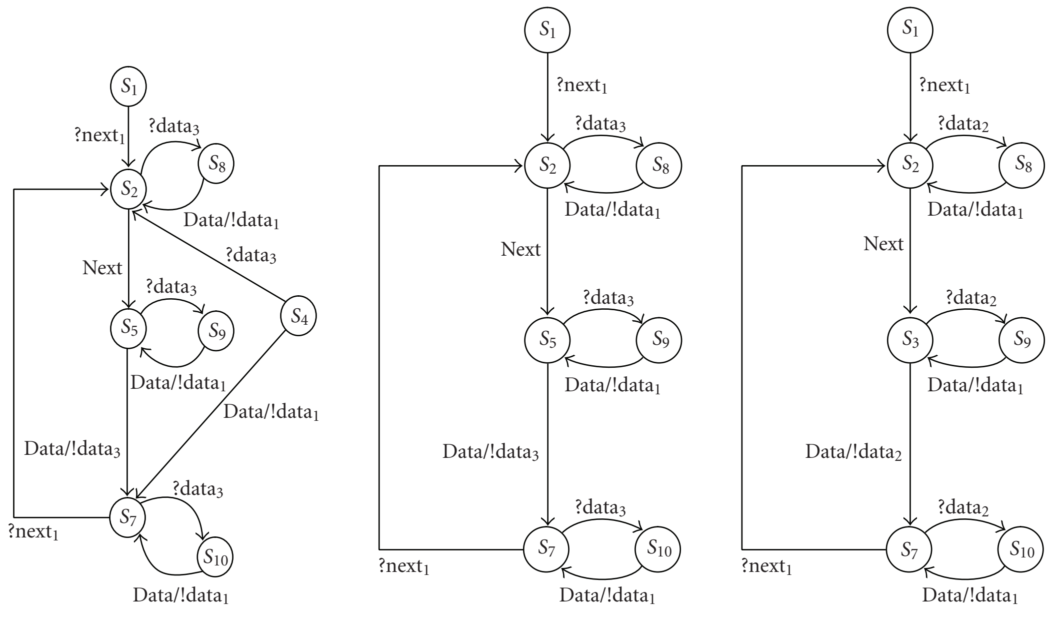
specifications by adding new states and transitions to handle the concurrency behavior specified in the service specification. In contrast with methods introduced earlier, the synthesis method introduced here solves the true concurrency problem such that if a message collision occurs, all parallel paths proceed without canceling any event. In addition, the synthesis method introduced here eliminates all the restrictions imposed by the earlier methods. As a result, the synthesis method is applicable to a wider range of concurrent applications.

The synthesis method introduced in this paper does not consider medium channel delays and timing requirements that can be provided in the service specification. In the future, we plan to study the assignment of timing constraints to a service specification that has concurrency behavior. In addition, we plan to extend the introduced synthesis method to map the timing constraints associated with the transitions of the service specification model to the transitions of the protocol specification models.

APPENDIX

A. PROOF OF CORRECTNESS

Proving the correctness of the synthesis method requires proving that the synthesis method is syntactically and semantically correct. This proof is necessary to eliminate the need for further validation of the derived protocol specifications. In Saleh [19], it is proved that the protocol entities derived using the first step of the synthesis method are syntactically and semantically correct. Step 2 of the synthesis method given in Algorithm 1 remodels only the paths between the images of the fork and the joint or leaf states. Therefore, before executing outgoing transitions from the images of the fork state, all the protocol entities are proved to be syntactically and semantically correct. In Lemma 2, it is proved that no protocol entity leaves the image of the joint state unless all protocol entities reach the image of the joint state. After reaching the image of the joint state in



(a) PE-SPEC$_2$: Step 2 (b) PE-SPEC$_2$ after Step 5 (c) PE-SPEC$_3$

FIGURE 5: An application of the procedure given in Algorithm 4 for the PE-SPEC$_2$ and PE-SPEC$_3$ given in Figure 3.

all protocol entities, the execution of the protocol entities is also proved to be syntactically and semantically correct [19]. Therefore, it must only be proved that the execution of the remodeled paths between the images of the fork and the joint or leaf states is syntactically and semantically correct.

A.1. Syntactic correctness

Proving the syntactic correctness of the synthesis method requires proving that the derived protocol specifications are free of syntactic design errors, including unspecified reception, deadlock, and livelock.

Lemma 1. *The procedures given in Algorithms 4 and 5 prevent the occurrence of unspecified message reception when parallel paths outgoing from a fork state are executed.*

Proof. In a PE, when any outgoing transition from the fork state is executed, the execution of the path p that contains the transition begins. However, the PE has to behave as if the execution of the other parallel paths is also starting. As a result, at any state in p, the execution of any subpath of any other parallel path must be allowed. Otherwise, an unspecified reception error occurs. This problem is dealt with in the procedures given in Algorithms 4 and 5 by making each state in p a source state for the first transition in each subpath of each path that can be executed in parallel with p (i.e., Steps 2 and 1.4 in the procedure given in Algorithm 4, and Step 2 in the procedure given in Algorithm 5). According to the *Partition_Parallel_Paths* procedure given in Algorithm 3, each subpath starts with a transition associated with a receiving event. Therefore, making each state in p a source state for the first transition in each subpath of each path that can be executed in parallel with p prevents the occurrence of unspecified message reception at any state in p. In the procedure given in Algorithm 4, the copies of the subpaths of the paths in the second group are added to the states of the paths in the first group. Copies of the subpaths of a path in the first group are not added to the states in the other paths in the same group because the paths in the first group cannot be executed in parallel. As a result, using the procedure given in Algorithm 4 prevents the occurrence of unspecified message reception at any state in any path in the first group. After the copies of the subpaths of the paths in the second group have been added to the states in the paths of the first group, the paths in the second group become redundant and have to be removed (i.e., Step 5 of the procedure given in Algorithm 4). Consequently, in the case where there is a sending state that is an image of a fork state, the procedure given in Algorithm 4 prevents the occurrence of unspecified message reception when the parallel paths outgoing from the fork state in the S-SPEC are executed.

In the case where there are no paths in the first group (i.e., the case dealt with in the procedure given in Algorithm 5), the paths in the second group are executed in parallel. To prevent an unspecified message reception error, additional transitions are added in Step 2 of the procedure given in Algorithm 5. These transitions are added to each state in each path in the second group. The transitions added to each state in a path p include all the subpaths of all paths—except p—in the second group. As a result, in the case where there is a receiving state that is an image of a fork state, the procedure given in Algorithm 5 prevents the occurrence of unspecified message reception when the parallel paths outgoing from the fork state in the S-SPEC are executed. As a result, in both cases, the procedures given in Algorithms 4 and 5 prevent unspecified message reception when the parallel paths outgoing from a fork state are executed. □

```
Procedure: Re-Model_Parallel_Paths_for_Receiving_State
Inputs:
  – a PE-SPEC that has an image state s of a fork state such that s is a receiving state,
  – $G_1$, $G_2$, and the subpaths found using the Partition_Parallel_Paths procedure given in Figure 10.
Outputs: a remodeled PE-SPEC that has additional states and transitions to consider the concurrency
behavior of the service specification.
Steps:
(1) if the paths in $G_2$ end with an image of a joint state and this state is a sending state then
    (1.1) LastSUBs = { },
    (1.2) for each path p in $G_2$ do
        (1.2.1) LastSUBs = LastSUBs ∪ last sub-path in $\mathrm{SUB}_p$,
    (1.3) add to the PE-SPEC model $\sum_{i=1}^{n} C(n,i)$ copies for each path in $G_2$, where n is $|G_2| - 1$,
    (1.4) delete the copies of the image of the fork state and their associated transitions from the copied paths,
    (1.5) for each path p in $G_2$ do
        (1.5.1) $\mathrm{LastSUB}_{np}$ = LastSUBs − last sub-path in $\mathrm{SUB}_p$,
        (1.5.2) connect the path p and its copies together using all subpaths in $\mathrm{LastSUB}_{np}$ such that each
                combination of the subpaths is covered once. Connecting two paths using a subpath
                includes connecting each state in one of the paths to its corresponding state in the other
                path using the subpath,
    (1.6) join all copied states that have no outgoing transitions together in a state j,
    (1.7) delete all redundant transitions connected to state j,
    (1.8) redirect the outgoing transitions from the joint state image to be initiated from state j,
    (1.9) for each path in $G_2$
        (1.9.1) $\mathrm{SUB}_p = \mathrm{SUB}_p$ − last subpath in $\mathrm{SUB}_p$,
(2) for each path g in $G_2$ do
    (2.1) for each path p in the set $G_2 - g$ do
        (2.1.1) for each state sg in path g excluding the image of the fork state do
            (2.1.1.1) for each sub-path sp in $\mathrm{SUB}_p$ do
                (2.1.1.1.1) add a copy of the subpath sp to state sg such that the state sg is, respectively,
                            the tail and head state of the first and last transitions in the subpath sp.
(3) if there are copied paths created using Step 1.3 then
    (3.1) for each copied path c of an original path p do
        (3.1.1) let q be any path from any state in p to its copied state in c
        (3.1.2) for each subpath sp added in Step 2.1.1.1.1 to a state in p do
            (3.1.2.1) if sp is a subpath of a path g and none of the transitions in q is a subpath in path g then
                (3.1.2.1.1) for each state sc in path c do
                    (3.1.2.1.1.1) add a copy of the subpath sp to state sc such that the state sc is, respectively,
                                  the tail and head state of the first and last transitions in the subpath.
(4) Delete redundant transitions connected to the image of the joint state.
```

Algorithm 5: A procedure for remodeling the PE-SPEC to consider the concurrency behavior in the case where there is a receiving image of fork state.

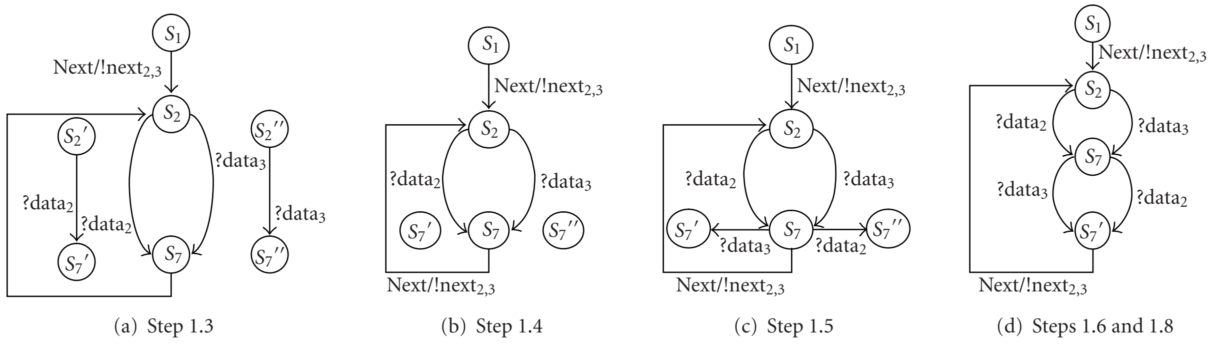


Figure 6: An application of the procedure given in Algorithm 5 for the PE-SPEC$_1$ given in Figure 3.

Lemma 2. *The procedures given in Algorithms 4 and 5 ensure that no protocol entity leaves the image of the joint state unless all the protocol entities reach the image of the joint state.*

Proof. A protocol entity that can leave the image of the joint state first and cause divergence is one that has an outgoing transition from the image of the joint state, associated with a service primitive (i.e., a sending state that is an image of a joint state). This protocol entity can send the service primitive, which means executing the transition and leaving the image of the joint state. If all the outgoing transitions from the image of the joint state in a PE are associated with receiving events (i.e., the image of the joint state is a receiving state), the protocol entity is guaranteed not to leave the image of the joint state unless the event to be received is sent by another PE, which means that the other PE left the image of the joint state. As a result, the divergence problem has to be solved only in the PEs in which the images of the joint states are sending states. For the sake of simplicity, these PEs are denoted here as sending PEs. In sending PEs, it is important to ensure that all transitions immediately preceding the joint state in the S-SPEC are executed before leaving the image of the joint state. When the second step of the synthesis algorithm given in Algorithm 1 is applied, the transitions incoming to the image of the joint state in each sending PE are associated with events. This means that ensuring the execution of all incoming transitions to the image of the joint state in each sending PE, before leaving the images of the joint states, guarantees the execution of all transitions immediately preceding the joint state in the S-SPEC. Consequently, this means that all protocol entities reach the image of the joint state. Therefore, to prove the lemma, it must be proved that all incoming transitions to the image of the joint state in each sending PE are executed before leaving the images of the joint states. This is proved here for both cases: (1) when the image of the fork state is a sending state (i.e., the case dealt with in the procedure given in Algorithm 4), and (2) when the image of the fork state is a receiving state (i.e., the case dealt with in the procedure given in Algorithm 5).

For the case given in Algorithm 4, only one path in the first group can be executed. The paths in the second group are executed as subpaths within the executed path in the first group. As a result, if the PE is a sending PE, it is necessary to ensure that for each path p in the first group, the last subpath in each path that can be executed in parallel with p is executed before leaving the joint state. The paths in the second group can be executed in parallel with the paths in the first group. Therefore, if the PE is a sending PE, it is necessary to ensure that for each path p in the first group, the last subpath in each path in the second group is executed before leaving the joint state.

If only one path exists in the second group, the last subpath of this path can be executed at any state in the paths of the first group. To model the execution of the last subpath ls at any state s in each path of the first group, a copy of s is added to the PE and the subpath ls is added, such that the state s and its copy are, respectively, the tail and head states of the subpath. In this case, when the PE is initially at state s and the subpath ls is executed, the PE reaches the added state. Since the subpath can be executed at any state in any path of the first group, a copy of each path in the first group has to be added so that subpath ls connects each state in the original path with its corresponding state in the added path. When adding a copied state, all subpaths connected as self-loop subpaths to state s are also added to the copied state, except the subpaths that belong to the same path of ls. These subpaths are not copied because executing the last subpath ls ensures that all the preceding subpaths that belong to the same path of ls have already been executed and they cannot be executed again. To ensure not leaving the joint state without executing the last subpath, the outgoing transitions from the image of the joint state are redirected to be outgoing from the copy of the image of the joint state. The new image of the joint state cannot be reached without executing the last subpath ls. Consequently, the new image of the joint state cannot be left without executing the last subpath ls.

The discussion in the previous paragraph applies to the case where there is only one path in the second group. If there are more paths, the solution is generalized to consider the execution of the last subpath of each path in the second group before leaving the image of the joint state. In this case, in the first stage, the number of added paths for each path in the first group is equal to the number of paths in the second group. If the number of paths in the second group is n, the number of created copies of each path in the first group is mathematically equal to $C(n,1) = n!/(1!(n-1)!) = n$. In the second stage, it is important to consider the execution of the last subpaths of two paths. In this case, the number of different combinations of the last two subpaths is considered. The total number of copies in this case (i.e., considering the two stages) is equal to $C(n,1) + C(n,2)$. Consequently, in general, if there are n paths in the first group, the number of copies to be created for each path in the first group is $\sum_{i=1}^{n} C(n,i)$. In this case, the new image of the joint state is the copy of the image of the joint state reached by executing the last subpaths of all paths in the second group. The creation of the copied paths and the connections between them, according to the above discussion, are performed in Step 1 of the procedure given in Algorithm 4. In addition, in Step 3 of the procedure, new transitions are added to the states of the created paths to prevent unspecified receptions occurring. As a result, in the case where there is a sending image of a fork state in a sending PE, the procedure given in Algorithm 4 ensures that no protocol entity leaves the image of the joint state unless all the protocol entities reach the image of the joint state.

For the case given in Algorithm 5, the paths in the second group are executed in parallel and there are no paths in the first group. In the case where there is a sending PE, Step 1.3 of the procedure given in Algorithm 5 adds $\sum_{i=1}^{n} C(n,i)$ copies of each path in the second group, where n is equal to (*number of paths in the second group* -1) (i.e., n is the number of paths in the second group that can be executed in parallel with one of the paths in the same group). In Step 1.5, each original path and its copies are connected using the same method discussed earlier for the case where there is at least one path in the first group. In addition, the image of the joint state is modified as before. Finally, in Step 3 of the procedure given in Algorithm 5, new transitions are added to the states of the created paths to prevent unspecified receptions occurring. As a result, in the case where there is a receiving image of a fork state in a sending PE, the procedure given in Algorithm 5 ensures that no protocol entity leaves the image of the joint state unless all the protocol entities reach the image of the joint state. □

Lemma 3. *The protocol entities derived using the synthesis method introduced in this paper are free of unspecified reception errors.*

Proof. According to Lemma 1, the execution of the remodeled paths between the images of the fork and the joint or leaf states is free of unspecified reception errors. Therefore, the protocol entities derived using the synthesis method introduced in this paper are free of unspecified reception errors. □

Lemma 4. *The protocol entities derived using the synthesis method introduced in this paper are free of deadlock errors.*

Proof. Deadlock errors occur when the protocol is at a nonfinal state, all channels are empty, and no transmission transition is specified. In other words, deadlock occurs when the protocol is at a state in which all its outgoing transitions are associated with receiving events and these events are for messages that are not to be sent by any other protocol entity. This case cannot occur in the extended paths because the added subpaths are added to the original states as self-loop subpaths. Therefore, it is inevitable that the original path (i.e., the path that exists before any new states or transitions are added) will be executed by the added transitions, and therefore these paths are guaranteed, by the proof provided in Saleh [19], to proceed without a deadlock error. The only case where an added transition can prevent the original parallel path from reaching the image joint state is in the case where there is a sending PE. In this case, the last subpaths of all other paths to be executed in parallel are added to prevent them leaving the image of the joint state without having all PEs reach the image of the joint state. Since all the parallel paths are executed, the execution of the last subpaths of each of these paths added by the procedures in Algorithms 4 and 5 is guaranteed. Therefore, these added subpaths cannot cause deadlock errors. Finally, the self-loop subpaths added to the states of the parallel paths by the procedures in Algorithms 4 and 5 always start by transitions associated with receiving events. Therefore, the subpaths are not executed unless the events are sent. All the succeeding transitions in the subpaths are associated with either service primitives or sending events, and no deadlock can therefore occur when the PE is at a state within the added subpaths. As a result, the added transitions and states do not cause any deadlock error. Therefore, the protocol entities derived using the synthesis method introduced in this paper are free of deadlock errors. □

Lemma 5. *The protocol entities derived using the synthesis method introduced in this paper are free of livelock errors.*

Proof. A livelock error occurs when the protocol entities exchange messages that are meaningless for the provision of the desired service. There are two types of messages exchanged within the parallel paths. The first type is a message derived directly from the service specification using the algorithm introduced by Saleh and Probert [4], and therefore these messages are meaningful for the provision of the desired service. The other type is a message added in Step 2 of the synthesis algorithm given in Algorithm 1. All these messages are associated with transitions of subpaths of other paths. These other paths are derived directly from the service specification using the algorithm introduced in Saleh and Probert [4], and therefore the messages associated with these paths are also meaningful for the provision of the desired service. As a result, the protocol entities derived using the synthesis method introduced in this paper are free of livelock errors. □

Theorem 1. *The protocol entities derived using the synthesis method introduced in this paper are syntactically correct.*

Proof. Since the protocol entities derived using the synthesis method introduced in this paper are free of unspecified reception (Lemma 3), deadlock (Lemma 4), and livelock errors (Lemma 5), the protocol entities are syntactically correct. □

A.2. Semantic correctness

Proving the semantic correctness of the synthesis method requires proving that the interactions among the derived protocol entities through a reliable underlying FIFO communication medium provide the service specified in the S-SPEC. In other words, it must be proved that all possible orderings of the service primitives noticed when the protocol entities are executed are consistent with the ordering of the service primitives in the S-SPEC.

Theorem 2. *The protocol entities derived using the synthesis method introduced in this paper are semantically correct.*

Proof. The remodeled paths consist of three parts: (1) the original paths, (2) the copied paths, and (3) the subpaths added to the former two types of paths. Step 2 of the synthesis algorithm does not modify the order of transitions in the original paths. Therefore, the original paths are semantically correct. The copied paths have the same order of transitions in the original paths, and therefore they are semantically correct. Finally, each of the added subpaths starts with a transition associated with a receiving event, succeeded by a transition associated with a service primitive *sp*, if any. The receiving event cannot be executed unless the message to be received has already been sent. This means that the service primitive *sp* cannot be executed unless its former service primitive in the S-SPEC has already been executed. Therefore, the order of the service primitives in the paths executed as subpaths connected to states of other paths is also maintained. As a result, the remodeled paths using Step 2 of the synthesis method given in Algorithm 1 are also semantically correct. □

ACKNOWLEDGMENT

The authors would like to acknowledge the support of this work by Kuwait University Research Grant WI03/05.

REFERENCES

[1] A. Davison, *Killer Game Programming in Java. Killer game programming in Java, Chapter B.1. Echoing Clinet/Server Application Using Bluetooth*, O'Reilly Media, Sebastopol, Calif, USA, 2006.

[2] D. Comer and D. Stevens, *Internetworking with TCP/IP. Vol. 3*, Prentice-Hall, Englewood Cliffs, NJ, USA, 2001.

[3] R. L. Probert and K. Saleh, "Synthesis of communication protocols: survey and assessment," *IEEE Transactions on Computers*, vol. 40, no. 4, pp. 468–476, 1991.

[4] K. Saleh and R. Probert, "Automatic synthesis of protocol specifications from service specifications," in *Proceedings of the 10th Annual International Phoenix Conference on Computers and Communications (IPCC '91)*, pp. 615–621, Scottsdale, Ariz, USA, March 1991.

[5] A. Khoumsi, "New results for deriving protocol specifications from service specifications for real-time applications," in *Proceedings of the Maghrebian Conference on Software Engineering and Arterial Intelligent (MCSEAI '98)*, Tunis, Tunisia, December 1998.

[6] J.-C. Park and R. E. Miller, "Synthesizing protocol specifications from service specifications in timed extended finite state machines," in *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*, pp. 253–260, Baltimore, Md, USA, May 1997.

[7] T. Higashino, K. Okano, H. Imajo, and K. Taniguchi, "Deriving protocol specifications from service specification in extended FSM models," in *Proceedings of the 13th International Conference on Distributed Computing Systems*, pp. 141–148, Pittsburgh, Pa, USA, May 1993.

[8] H. Yamaguchi, K. Okano, T. Higashino, and K. Taniguchi, "Protocol synthesis from time Petri net based service specifications," in *Proceedings of the Internatoinal Conference on Parallel and Distributed Systems (ICPADS '97)*, pp. 236–243, Seoul, South Korea, December 1997.

[9] K. Saleh and R. Probert, "An extended service-oriented method for the synthesis of protocols," in *Proceedings of the 6th International Symposium on Information and Computer Sciences*, pp. 547–557, Antalya, Turkey, October 1991.

[10] Y. Kakuda, M. Nakamura, and T. Kikuno, "Automated synthesis of protocol specifications from service specifications with parallelly executable multiple primitives," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E77-A, no. 10, pp. 1634–1645, 1994.

[11] B. B. Bista, K. Takahashi, and N. Shiratori, "Composition of service and protocol specifications," in *Proceedings of the 15th International Conference on Information Networking*, pp. 171–178, Beppu City, Japan, January-February 2001.

[12] N. Maneerat, R. Varakulsiripunth, D. Seki, et al., " Composition method of communication system specifications in asynchronous model and its support system," in *Proceedings of the 9th IEEE International Conference on Networks*, pp. 64–69, Bangkok, Thailand, October 2001.

[13] H. Yamaguchi, K. El-Fakih, G. von Bochmann, and T. Higashino, "Protocol synthesis and re-synthesis with optimal allocation of resources based on extended Petri nets," *Distributed Computing*, vol. 16, no. 1, pp. 21–35, 2003.

[14] N. Maneerat, R. Varakulsiripunth, B. B. Bista, K. Takahashi, Y. Kato, and N. Shiratori, "Composition of service and protocol specifications in asynchronous communication system," *IEICE Transactions on Information and Systems*, vol. E87-D, no. 10, pp. 2306–2317, 2004.

[15] J. A. Dallal, "Automatic synthesis of timed protocol specifications from service specifications," *WSEAS Transactions on Computers*, vol. 5, no. 1, pp. 105–112, 2006.

[16] N. Stakhanova, S. Basu, W. Zhang, X. Wang, and J. Wong, "Specification synthesis for monitoring and analysis of MANET protocols," Tech. Rep. 07-02, Department of Computer Science, Iowa State University, Ames, Iowa, USA, 2007.

[17] T. Y. Choi, "Sequence method for protocol construction," in *Proceedings of the 6th IFIP International Symposium on Protocol Specification, Testing, and Verification*, pp. 307–321, Gray Rocks Inn, Canada, 1986.

[18] Y. X. Zhang, K. Takahashi, N. Shiratori, and S. Noguchi, "An interactive protocol synthesis algorithm using a global state transition graph," *IEEE Transactions on Software Engineering*, vol. 14, no. 3, pp. 394–404, 1988.

[19] K. Saleh, *Synthesis method for the design and validation of communication protocols*, Ph.D. dissertation, University of Ottawa, Ottawa, Canada, 1991.