Extension of Protocol Synthesis to Structured Entities

Bhed Bahadur Bista, Atsushi Togashi and Norio Shiratori
email:{bbb,togashi,norio}@shiratori.riec.tohoku.ac.jp
Research Institute of Electrical Communication/Graduate School of Information Science
Tohoku University
2-1-1 Katahira, Sendai 980-77, Japan

Abstract

Entities in communication systems behave under sets of communication rules (protocols) and their behaviours are related to each other. Using this relationship, we present a synthesis algorithm to derive a structured protocol model by generating a peer entity from a given entity automatically. The given entity is in the form of a communicating process, which is natural in the context of communication protocols. We prove that the synthesized protocol model consisting of the generated entity and the peer entity is safe, i.e. logical errors free, collectively represented as deadlock free and is bisimulated by both entities. Unlike many previous works, where FSMs (Finite State Machines) are used to synthesize protocol model, we present our synthesis algorithm for the process-algebraic language LOTOS, which is one of the FDTs developed by ISO.

1 Introduction

Communication protocols are the main parts in computer networks and distributed information processing systems and play a major role for their proper operations. Formal Description Techniques (FDTs) such as Estelle [1], LOTOS [2] and SDL [3] have been developed to write such protocols precisely, correctly and unambiguously.

There are various protocol design techniques proposed in literatures. They can be broadly classified into *analytic methods* [4], [5], [6] and *synthetic methods* [7], [8], [9], [10],[11]. In the former ones, a pre-designed protocol is analyzed to detect errors and if possible correct them whereas in the latter ones, a protocol is derived from a service specification or a partial protocol is completed so that the resultant protocol satisfies the properties for correctness. Since the further verification or validation of the protocol is not required in the synthetic design techniques, they are usually preferred in the design of a protocol specification in its early stage.

Behaviours of entities in communication systems are highly related to each other as they communicate with each other under certain sets of communication rules. By exploiting this relationship, it is possible and desirable to generate a peer entity (a missing entity) from a given entity. In [12], [13], a peer entity is automatically generated in LOTOS from a given entity which is specified as a communicating process in LOTOS. However, the entities are specified in monolithic style [14], which is an expanded form of behaviour expressions. Large and complex systems are usually structured into phases and concurrent entities such a way that they are more understandable, manageable and each phase or entity can be designed separately. LOTOS unlike other FDTs has various operators such as *enabling*, *disabling* and *parallel*, to structure behaviours of systems. A system which is not structured is difficult to understand, manage and is error prone. In this paper, we extend [12], [13] to derive a protocol, which is structured, by generating a peer entity from a given entity. The given entity is defined as a communicating process, which is structured, in LOTOS. The generated peer entity is structured and is in LOTOS also. Furthermore, we show that the resulting protocol is deadlock free and is bisimulated by both entities to show that the protocol is safe and maintains the behaviours of entities.

We will briefly review and compare the previous works which are mostly related to this paper. Most of the works related to this paper are based on FSMs. Our work is based on LOTOS. The main advantages of LOTOS over FSMs are concurrency [15], which is one of the behaviours which can be expressed in LOTOS but not in FSMs. States of a system specified in LOTOS are implicit and in FSMs are explicit, thus state explosion can occur in the synthesis of protocol specifications of a large system in FSMs. Besides, due to its various operators and mathematical base, LOTOS has more expressive power while writing an abstract system behaviour than FSMs. In [16], a protocol service specification and $n-1$ entities are given and the entity n is constructed. Their limitation is that the notion of correctness does not capture some aspects such as potential deadlocks. Thus, an entity satisfying the automatically generated specification may cause deadlocks. In [7], partial specifications of n entities are completed by combining some pre-designed components which are considered as primitive communicating entities. Though the approach is different from ours using FSMs, the formal proof of the correctness of the combination of components is not provided and thus, it is not known formally whether the combinations of components always satisfy the proof of correctness. The correctness is checked by a validation technique in [5].

Most of the protocol synthesis based on LOTOS [17], [18],[19], concentrate on automatic transforma-

0-8186-7216-1/95 $04.00

Name	Syntax	Axioms and inference rules
inaction	**stop**	
successful termination	**exit**	$\mathbf{exit} \xrightarrow{\delta} stop$
action prefix	$a;P$	$a;P \xrightarrow{a} P$
choice	$P_1 [] P_2$	$\frac{P_1 \xrightarrow{a} P_1'}{P_1 [] P_2 \xrightarrow{a} P_1'} \quad \frac{P_2 \xrightarrow{a} P_2'}{P_1 [] P_2 \xrightarrow{a} P_2'}$
enabling	$P_1 \gg P_2$	$\frac{P_1 \xrightarrow{a} P_1',\ a \neq \delta}{P_1 \gg P_2 \xrightarrow{a} P_1' \gg P_2} \quad \frac{P_1 \xrightarrow{\delta} P_1'}{P_1 \gg P_2 \xrightarrow{\mathbf{i}} P_2}$
disabling	$P_1 [> P_2$	$\frac{P_1 \xrightarrow{a} P_1',\ a \neq \delta}{P_1 [> P_2 \xrightarrow{a} P_1' [> P_2} \quad \frac{P_1 \xrightarrow{\delta} P_1'}{P_1 [> P_2 \xrightarrow{\delta} P_1'} \quad \frac{P_2 \xrightarrow{a} P_2'}{P_1 [> P_2 \xrightarrow{a} P_2'}$
parallel composition	$P_1 \|[H]\| P_2$	$\frac{P_1 \xrightarrow{a} P_1',\ a \notin H \cup \{\delta\}}{P_1 \|[H]\| P_2 \xrightarrow{a} P_1' \|[H]\| P_2} \quad \frac{P_2 \xrightarrow{a} P_2',\ a \notin H \cup \{\delta\}}{P_1 \|[H]\| P_2 \xrightarrow{a} P_1 \|[H]\| P_2'}$ $\frac{P_1 \xrightarrow{a} P_1', P_2 \xrightarrow{a} P_2',\ a \in H \cup \{\delta\}}{P_1 \|[H]\| P_2 \xrightarrow{a} P_1' \|[H]\| P_2'}$

Table 1: Basic Lotos syntax and semantics

tion of a service specification into a protocol specification and prove the correctness between the two specifications. The approach is important in the top down design method. However, if there is any mistake in the service specification, it is amplified in all the following transformations. It is also not known whether the correctness between the service specification and the protocol specification is still preserved if any detail is added in the protocol specification. For instance, the Alternative Bit Protocol cannot be derived from its service specification unless the error handling part in the protocol specification is specified in the service specification.

This paper is organized as follows. Following the Introduction, section 2 gives an outline of LOTOS, section 3 presents structured protocol model. Section 4 discusses the details of the proposed synthesis algorithm and presents a simple application example using the algorithm. Section 5 concludes the paper.

2 LOTOS

LOTOS (Language Of Temporal Ordering Specification) is an FDT (Formal Description Technique) developed by ISO (International Organization for Standardization) [2] for the formal description of distributed systems. A system, specified in LOTOS, consists of a number of *processes* interacting with each other. A process in LOTOS is considered as an abstract entity which communicates with other processes by synchronizing in *communication actions (events)*. An action is considered atomic (i.e. not divisible in time). Throughout this paper, let $\mathcal{A}$ denote a finite set of actions which are externally observable and controllable. An internal action which can't be externally observed is represented as $\mathbf{i}$ and let $Act = \mathcal{A} \cup \{\mathbf{i}\}$. Act^* denotes the set of all sequences of actions and we write ε as the empty sequence. $\hat{t}$ is the resulting sequence of actions t where $\mathbf{i}$ is removed.

In LOTOS, the *behaviour expression* of a process is defined by stating the temporal relation of actions (events) of the process. There are various LOTOS operators to structure behaviour expression of a system. Operators which are used in this paper are shown in Table 1 with their names, syntax and inference rules. See [2] for other operators.

The description of data structures and data values are based on the algebraic specification of abstract data types (ADTs) for actions involving in exchange of data.

LOTOS has the following two semantic models:

- Label transition systems for behaviour expressions
- Many sorted algebras for data types

In this paper, we are mainly concerned with behaviour expressions, thus label transition systems. A labeled transition system is a state transition graph in which nodes represent states and edges represent state changes. This is formally defined as follows:

Definition 2.1 (Labeled Transition System) A *labeled transition system* L is a quadruple (S, A, T, s_0), where

1. S is a nonempty set of *states*;
2. A is a subset of Act;
3. $T \subseteq S \times A \times S$ is a *transition relation*;
4. $s_0 \in S$ is the *initial state* of L. □

The transition relation defines the dynamic changes of states as actions are performed. For $(s, a, s') \in T$, we normally write $s \xrightarrow{a} s'$ and this may be interpreted as "in the state s an action a can be performed and after the action the state moves to s'". If $t = a_1 a_2 \ldots a_n \in Act^*$, then

- $s \xrightarrow{t} s'$ stands for $s \xrightarrow{a_1} \xrightarrow{a_2} \ldots \xrightarrow{a_n} s'$.

- $s \stackrel{t}{\Rightarrow} s'$ stands for $s \stackrel{\mathbf{i}^*}{\rightarrow} \stackrel{a_1}{\rightarrow} \stackrel{\mathbf{i}^*}{\rightarrow} \stackrel{a_2}{\rightarrow} \stackrel{\mathbf{i}^*}{\rightarrow} \ldots \stackrel{\mathbf{i}^*}{\rightarrow} \stackrel{a_n}{\rightarrow} \stackrel{\mathbf{i}^*}{\rightarrow} s'$.

- $s \stackrel{t}{\rightarrow}$ stands for $s \stackrel{t}{\rightarrow} s'$ for some s' and $\stackrel{t}{\not\rightarrow}$ denotes the negation of $s \stackrel{t}{\rightarrow}$.

Based on the operational semantics given by the transition systems, a *bisimulation relation* is defined as follows:

Definition 2.2 (Bisimulation)
Let $L_1 = \langle S_1, A_1, T_1, s_{10} \rangle$ and $L_2 = \langle S_2, A_2, T_2, s_{20} \rangle$ be labelled transition systems. A binary relation $R \subseteq S_1 \times S_2$ is a *(weak)bisimulation* if $(s_1, s_2) \in R$ implies that, for all $a \in Act$,

1. if $s_1 \stackrel{a}{\rightarrow} s_1'$ for some s_1', then $s_2 \stackrel{\hat{a}}{\Rightarrow} s_2'$ and $(s_1', s_2') \in R$ for some s_2';

2. if $s_2 \stackrel{a}{\rightarrow} s_2'$ for some s_2', then $s_1 \stackrel{\hat{a}}{\Rightarrow} s_1'$ and $(s_1', s_2') \in R$ for some s_1'.

L_1 is *bisimilar* to L_2 (or L_1 is *bisimulated* by L_2) if $(s_{10}, s_{20}) \in R$ for some bisimulation R. □

Define a relation $\approx$ by

$$\approx = \cup\{R \subseteq S_1 \times S_2 \mid R : \text{a bisimulation}\}$$

Then, $\approx$ is an equivalence relation and turns out to be the largest bisimulation. It is clear from the definition that L_1 is bisimilar to L_2 iff $s_{10} \approx s_{20}$.

3 Structured Communication Model

3.1 Protocol Model

In a communication system, protocol entities provide services to their users by interacting among themselves. In this paper, we use LOTOS as our notational vehicle. Protocol entities and service access points are represented in LOTOS by processes and gates respectively. An abstract communication system can be represented as in Figure 1 in which PM represents *protocol model* consisting of interacting processes, P and Q which provide services to their respective users, U_p and U_q.

A PM is a 5-tuple $\langle P, Q, G, Gp, Gq \rangle$. P and Q are processes. G is a gate at which P and Q interact to exchange messages. G_p and G_q are gates at which P and Q provide services to their users U_p and U_q respectively. Another view of looking at the gates is to treat them as buffers in which a process puts in a message and the other process takes out the message. In fact, such an action is atomic as putting the message in the buffer and taking out the message from the buffer is performed by synchronizing in an action in LOTOS. For example, G is interpreted in such a way that *g_in* is the gate at which P sends messages ($\overrightarrow{\alpha}$) to Q (and the gate at which Q receives messages ($\overrightarrow{\alpha}$) from P) and *g_out* is the gate at which Q sends messages ($\overleftarrow{\alpha}$) to P (and the gate at which P receives messages ($\overleftarrow{\alpha}$) from Q).

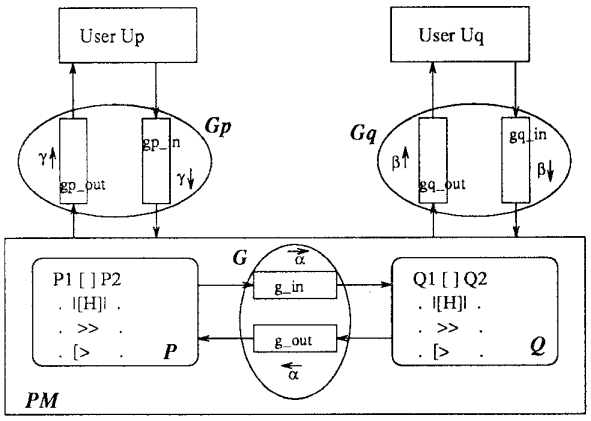

Figure 1: Structured communication model

In [12] and [13], the given entity P and the generated peer entity Q, are both in monolithic style [14] and thus the synthesized protocol consisting of P and Q is also in monolithic style. A specification written in monolithic style is fully expanded in terms of LOTOS prefix and choice operators. On the other hand, a structure specification uses LOTOS parallel, enabling and disabling operators also to distinguish and structure system functionalities thereby making the system behaviour modular, more understandable and manageable. The parallel operator is used to express concurrency and constraint between processes whereas the enabling and disabling are mainly used for expressing the phases of the system. Large and complex systems are structured and if they are in monolithic style they will be very difficult to understand and will be error prone. In this paper, we consider both P and Q as structured processes as shown in Figure 1. The resulting synthesized protocol consisting of P and Q is also structured.

3.2 Notations

Basic LOTOS expresses the system behaviours without data parts and is usually used to abstract the system behaviour, verify and test the properties. Although the complete system specification includes data part also, it is very difficult to verify and test the system properties as the data part overshadows the system behaviour. In this paper we use Basic LOTOS and use arrows in actions or events in order to represent messages and direction of messages flow when the processes synchronize in those actions. In this way, we are able to abstract some data parts of the full LOTOS if data are considered as universe of data. We give the following notations for the protocol model given in Figure 1 to make discussion simpler in following sections.

P : a given entity
Q : the peer entity
$\gamma_1\downarrow, \gamma_2\downarrow, ... \in G_p$: actions of type '$\downarrow$' representing messages from U_p to P
$\gamma_1\uparrow, \gamma_2\uparrow, ... \in G_p$: actions of type '$\uparrow$' representing messages from P to U_p
$\beta_1\downarrow, \beta_2\downarrow, ... \in G_q$: actions of type '$\downarrow$' representing messages from U_q to Q
$\beta_1\uparrow, \beta_2\uparrow, ... \in G_q$: actions of type '$\uparrow$' representing messages from Q to U_q
$\overrightarrow{\alpha_1}, \overrightarrow{\alpha_2}, ... \in G$: actions of type '$\rightarrow$' representing messages from P to Q
$\overleftarrow{\alpha_1}, \overleftarrow{\alpha_2}, ... \in G$: actions of type '$\leftarrow$' representing messages from Q to P

Specification of an entity in the PM is described with respect to its upper and lower interfaces. For instance, P in figure 1 is described with respect to $G_p(\gamma\downarrow, ..., \gamma\uparrow)$ and $G(\overrightarrow{\alpha}, ..., \overleftarrow{\alpha})$. We assume that the given entity is defined as a communicating process which is structured. The synthesis algorithm given in section 4 generates a peer entity which is also structured. In order to define a communicating process, it is assumed that we have denumerable set X of *process variables.* A communicating process for a given entity is defined in the next section.

3.3 Communicating Processes

Definition 3.1 (communicating process) A *communicating process* for a given entity is defined inductively by the following BNF notations:
$P ::= P[\,]P \mid P \gg P \mid P[> P \mid P|[H]|P \mid$ **stop** $\mid$ **exit** $\mid x \mid$ **rec**$x.P \mid (\sum_{i\in I} S_i) \mid (\sum_{j\in J} R_j)$
$S ::= \gamma_i\downarrow; \overrightarrow{\alpha_i}; P \mid a; \gamma_i\downarrow; \overrightarrow{\alpha_i}; P \mid \gamma_i\downarrow; \overrightarrow{\alpha_i}; a; P$
$R ::= \overleftarrow{\alpha_j}; \gamma_j\uparrow; P \mid a; \overleftarrow{\alpha_j}; \gamma_j\uparrow; P \mid \overleftarrow{\alpha_j}; \gamma_j\uparrow; a; P$
where $x \in X$ and $a \in H$ □

The recursive behaviour of the given entity is expressed by the communicating process, **rec**$x.P$, which is interpreted as a process x whose meaning is defined by the recursive equation, $x \stackrel{def}{=} P$. For simplicity, we treat only communicating processes without mutual recursion as above. However, it is easy to extend for the process with mutual recursion.

In the communicating processes, $\gamma_i\downarrow; \overrightarrow{\alpha_i}$, $a; \gamma_i\downarrow; \overrightarrow{\alpha_i}$ and $\gamma_i\downarrow; \overrightarrow{\alpha_i}; a$ are considered as *sending primitives* and $\overleftarrow{\alpha_j}; \gamma_j\uparrow$, $a; \overleftarrow{\alpha_j}; \gamma_j\uparrow$ and $\overleftarrow{\alpha_j}; \gamma_j\uparrow; a$ are considered as *receiving primitives.* Note that $\overrightarrow{\alpha_i}$ follows $\gamma_i\downarrow$ and $\gamma_j\uparrow$ follows $\overleftarrow{\alpha_j}$ in sending and receiving primitives respectively. The temporal orderings of $\gamma_i\downarrow$, $\overrightarrow{\alpha_i}$, $\overleftarrow{\alpha_j}$ and $\gamma_j\uparrow$ in a communicating process is natural in a sense that the communicating process receives a message ($\gamma_i\downarrow$) from its user and sends it ($\overrightarrow{\alpha_i}$) to its peer entity and, similarly receives a message ($\overleftarrow{\alpha_j}$) from the peer entity and delivers it (γ_j) to the user and then be ready for other actions.

The most difficult part in describing the communicating processes is to find out which actions belong to H in generalized parallel operator, $P_1|[H]|P_2$. We realize that actions $\gamma_i\downarrow$, $\overrightarrow{\alpha_i}$, $\overleftarrow{\alpha_j}$ and $\gamma_j\uparrow$, cannot belong to H as it will be unnecessary because it will mean that P_1 and P_2 will receive the same message either from user U_p and send it Q or vice versa. So we assume that H has some actions which are local to P_1 and P_2 only, i.e. hidden from their environments U_p and Q, and denote them by a. a can appear in every possible places in P as shown by the LTSs of P in Figure 2. For simplicity, we consider Figure 2(a) only. Others can be quite easily accommodated in the definition of communicating processes. $P_1||P_2$ (full synchronization) is not considered because of the same reason as above.

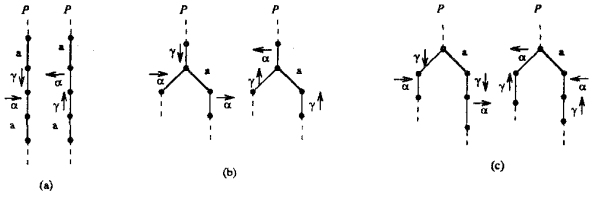

Figure 2: The temporal ordering of a in P

We have two restrictions in communicating processes. Before stating them, we define the set of initial actions of an expression B as $init(B) = \{c \in Act(B) \cup \{\delta\} | B \xrightarrow{c} \}$

Restriction 1
If $P = P_1[\,]P_2$, then either $init(P_1) \cup init(P_2) \subseteq \{\gamma_i\downarrow\} \cup \{\delta\}$ or $init(P_1) \cup init(P_2) \subseteq \{\overleftarrow{\alpha_j}\} \cup \{\delta\}$

Restriction 2
If $P = P_1 \gg P_2$ and $P_1 = B_1[\,]B_2$, then $B_1 \stackrel{\delta}{\not\rightarrow}$ and $B_2 \stackrel{\delta}{\not\rightarrow}$.

3.4 A Safe Protocol Specification

In this section, we discuss what a safe protocol specification is and state that our synthesized protocol model will be safe according to the discussion. A safe protocol specification is the one which is free from logical errors. In general, there are four types of logical errors, (1) *deadlock*, (2) *unspecified reception*, (3) *non-executable transition* and (4) *buffer overflow*, which are well discussed in FSMs model [7]. Since we do not have a real buffer (though indirectly implied), we do not consider the buffer overflow. Other three errors are collectively called as deadlock from actions execution point of view in our model, since the deadlock occurs in LOTOS when processes cannot synchronize in an action [12],[13]. If there are any unspecified or non-executable actions in any process in the communication model then the processes cannot synchronize on those actions and the PM is deadlock. We formally define the deadlock in definition 3.2. Due to the lack

of space we omit examples for deadlocks in this paper, but they can be found in [12],[13].

Definition 3.2 (Deadlock) Let $PM = P|[G]|Q$ be a protocol model. PM is *deadlock* if there exist processes $P'(Q')$ and $a \in Act$ such that

(1) $P \xrightarrow{t} P' \xrightarrow{a} (Q \xrightarrow{t} Q' \xrightarrow{a})$ for some $t \in Act^*$

(2) For any $Q''(P'')$ such that $PM \xrightarrow{t'} P''|[G]|Q''$ $(PM \xrightarrow{t'} P''|[G]|Q')$, for some $t' \in Act^*$ we have $P'|[G]|Q'' \overset{a}{\not\rightarrow} (P''|[G]|Q' \overset{a}{\not\rightarrow})$.

In this case, we say that deadlock *occurs* at P' (Q') and a is the *non-executable* action[1]. □

Now, we can formalize our protocol synthesis problem as follows:
Given: The specification of an entity P, which is a given entity and is expressed as a communicating process.
Problem: Generate the specification of the peer entity Q, which is expressed as a communicating process also, such that

(1) $PM = P|[G]|Q$ is *deadlock* free.

(2) P with an action of the form $\overleftarrow{\alpha}$ replaced by $\mathbf{i}$; $\overleftarrow{\alpha}$ is bisimilar to **hide** $\tilde{\beta}$ **in** $P|[G]|Q$, where $\tilde{\beta}$ denotes the sequence of actions in Q of the form $\beta\uparrow$ and $\beta\downarrow$.

(3) Q with an action of the form $\overrightarrow{\alpha}$ replaced by $\mathbf{i}$; $\overrightarrow{\alpha}$ is bisimilar to **hide** $\tilde{\gamma}$ **in** $P|[G]|Q$, where $\tilde{\gamma}$ denotes the sequence of actions in P of the form $\gamma\uparrow$ and $\gamma\downarrow$.

In order to realize (2) and (3) we have to consider the semantical meaning of entities' behaviours. For instance, the behaviour of P in (2) may be expressed in different form (i.e. $\overleftarrow{\alpha}$ is replaced by $\mathbf{i}$; $\overleftarrow{\alpha}$) because for P, any message sent by user U_q to Q seems invisible thus non-deterministic, i.e. it is not known to P which message is sent to Q by U_q and hence to P. In order to realize this non-determinism, we replace receiving actions $\overleftarrow{\alpha}$ of P by $\mathbf{i};\overleftarrow{\alpha}$.

4 Synthesis Algorithm and Results

In this section, we present the details of our proposed algorithm and prove some results. Before presenting the details of the algorithm, we first discuss the basic idea of the algorithm in the next subsection.

4.1 Basic Idea of the Algorithm

The basic idea of the synthesis algorithm is as follows. For each sending primitive of the given entity, we generate an appropriate receiving primitive for the peer entity. Similarly, for each receiving primitive of the given entity, we generate an appropriate sending primitive for the peer entity. In other words, when P receives a message $(\gamma\downarrow)$ from its user and sends it $(\overrightarrow{\alpha})$ to the peer entity, i.e. $P = \gamma\downarrow;\overrightarrow{\alpha};P'$, we generate the peer entity Q such a way that it receives the message $(\overrightarrow{\alpha})$ from P and delivers the message $(\beta\uparrow)$ to its user, i.e. $Q = \overrightarrow{\alpha};\beta\uparrow;Q'$. In the same manner, when P receives a message $(\overleftarrow{\alpha})$ from the peer entity and delivers it $(\gamma\uparrow)$ to its user, i.e. $P = \overleftarrow{\alpha};\gamma\uparrow;P'$, we make Q such a way that it receives the message $(\beta\downarrow)$ from its user and sends it $(\overleftarrow{\alpha})$ to P, i.e. $Q = \beta\downarrow;\overleftarrow{\alpha};Q'$. Similarly, we consider other primitives of P to generate the appropriate primitives of Q. The concept is natural for communication protocols. For instance, in a simple connection establishment protocol, if given entity is defined as $P = ConReq\downarrow;\overrightarrow{CR};$**exit** then the generated peer entity is $Q = \overrightarrow{CR};ConInd\uparrow;$**exit**, where $ConReq\downarrow$ is *Connection Request* service primitive, $\overrightarrow{CR}$ is the corresponding *Protocol Data Unit* and $ConInd\uparrow$ is *Connection Indication* service primitive.

4.2 Synthesis Algorithm

The synthesis algorithm we will propose in this section is based on the communication patterns discussed in the previous section. Basically, the algorithm is a three steps procedure. In step (1), the communication patterns of the sending and receiving primitives of the given entity are checked. In step (2), the corresponding communication patterns of the sending and receiving primitives of the peer entity are obtained. In step (3), the primitives of the peer entity are composed together to obtain the complete peer entity.

In the Algorithm 4.1, *peer* is the function (procedure) whose input is a given entity P. *peer* checks the communication pattern of primitives of P recursively and returns the corresponding primitives of Q. *peer* has auxiliary procedure, $aux(S)$ and $aux(R)$, which generate receiving primitives and sending primitives of Q respectively. The peer entity is structurally generated if the given entity is structured. In case of the given entity consisting of sub-entities composed by enabling and disabling operators, $P_1 \gg P_2$ and $P_1[> P_2$, we transform them into $P_1 \gg nc;P_2$ and $P_1[> nc;P_2$ respectively. In the protocol model PM, P and Q synchronize in nc (next phase) also. The reason for this transformation is that enabling and disabling operators are mainly used for expressing phases in a system as it is already explained in the previous sections. Here in this paper, we would like the phases of P and Q progress synchronously by synchronizing in nc before executing the actions of the next phases, P_2 and Q_2. The advantage of this transformation is that phases of P and Q do not lag or advance from each other while communicating to each other. We state the algorithm as shown below.

[1]Note that a must be a synchronization action.

Algorithm 4.1 (Synthesis Algorithm)
procedure $peer(P)$
begin
case P **of**
stop: **return stop**; (c1)
exit: **return exit**; (c2)
x: **return** x; (c3)
rec $x.P$: **return rec** $x.peer(P)$; (c4)
$(\sum_{i\in I} S_i)$: **return** $(\sum_{i\in I} aux(S_i))$; (c5)
$(\sum_{j\in J} R_j)$: **return** $(\sum_{j\in J} aux(R_j))$; (c6)
$P_1[\,]P_2$: **return** $peer(P_1)[\,]peer(P_2)$; (c7)
$P_1 \gg nc; P_2$:
return $peer(P_1) \gg nc; peer(P_2)$; (c8)
$P_1[> nc; P_2$:
return $peer(P_1)[> nc; peer(P_2)$; (c9)
$P_1|[H]|P_2$:
return $peer(P_1)|[H']|peer(P_2)$; (c10)
end
procedure $aux(S_i)$
begin
case S_i **of**
$\gamma_i\downarrow; \overrightarrow{\alpha_i}; P$:
return $\overrightarrow{\alpha_i}; \beta_i\uparrow; peer(P)$; (S1)
$a; \gamma_i\downarrow; \overrightarrow{\alpha_i}; P$:
return $b; \overrightarrow{\alpha_i}; \beta_i\uparrow; peer(P)$; (S2)
$\gamma_i\downarrow; \overrightarrow{\alpha_i}; a; P$:
return $\overrightarrow{\alpha_i}; \beta_i\uparrow; b; peer(P)$; (S3)
end
procedure $aux(R_j)$
begin
case R_j **of**
$\overleftarrow{\alpha_j}; \gamma_j\uparrow; P$:
return $\beta_j\downarrow; \overleftarrow{\alpha_j}; peer(P)$; (R1)
$a; \overleftarrow{\alpha_j}; \gamma_j\uparrow; P$:
return $b; \beta_j\downarrow; \overleftarrow{\alpha_j}; peer(P)$; (R2)
$\overleftarrow{\alpha_j}; \gamma_j\uparrow; a; P$:
return $\beta_j\downarrow; \overleftarrow{\alpha_j}; b; peer(P)$; (R3)
end
where $a \in H$ and $b \in H'$. □

4.3 Results

In this section, we prove that the synthesized protocol is safe, i.e. deadlock free (Theorem 4.1) and is bisimulated by both entities P and Q (Theorem 4.2). We give brief outlines of proofs of the theorems. Before proving the theorems, we prove the following lemmas

Lemma 4.1 *If $P_i|[G]|Q_i$ are deadlock free for all $i \in I$, then the protocol model, $PM = (\sum_{i\in I} \gamma_i\downarrow; \overrightarrow{\alpha_i}; P_i)|[G]|(\sum_{i\in I} \overrightarrow{\alpha_i}; \beta_i\uparrow; Q_i)$ is deadlock free.*

Proof. Suppose that $P_i|[G]|Q_i$ are deadlock free for all $i \in I$. The resulting protocol model is equivalently expanded as

$$PM = \sum_{i\in I} \gamma_i\downarrow; \overrightarrow{\alpha_i}; (P_i|[G]|\beta_i\uparrow; Q_i)$$

Since the process $\beta_i\uparrow; Q_i$ can always execute $\beta_i\uparrow$ in $P_i|[G]|\beta_i\uparrow; Q_i$ (P_i may be able to execute some non-synchronized actions in $P_i|[G]|\beta_i\uparrow; Q_i$), if $P_i|[G]|\beta_i\uparrow; Q_i$ is deadlock for some i then $P_i|[G]|Q_i$ must be deadlock that contradicts the assumption. Thus, $P_i|[G]|\beta_i\uparrow; Q_i$ is deadlock free for all i. Therefore, we can conclude that PM is deadlock free. □

Lemma 4.2 *Let P_1 and P_2 be processes having no synchronized actions in common with processes Q_2 and Q_1, respectively. If protocol models $P_1|[G]|Q_1$ and $P_2|[G]|Q_2$ are deadlock free, then $PM = (P_1 \,|||\, P_2)|[G]|(Q_1 \,|||\, Q_2)$ is deadlock free.*

Proof. If PM is deadlock, then either $P_1|[G]|Q_1$ or $P_2|[G]|Q_2$ must be deadlock. This contradicts the assumption. Thus, we get the lemma. □

Note: it is assumed that no synchronization actions are common between P_1 and Q_2, and P_2 and Q_1.

Lemma 4.3 *Let P_1 and P_2 be processes having no synchronized actions in common and let $a \in H$ and $b \in H'$. If $(P_1 \,|||\, P_2)|[G]|(Q_1 \,|||\, Q_2)$ is deadlock free, then $PM = (a; P_1|[H]|a; P_2)|[G]|(b; Q_1|[H']|b; Q_2)$ is deadlock free.*

Proof. The PM can be equivalently expanded as $PM = (a; (P_1 \,|||\, P_2))|[G]|(b; (Q_1 \,|||\, Q_2))$. If PM is deadlock then $(P_1 \,|||\, P_2)|[G]|(Q_1 \,|||\, Q_2)$ is deadlock. This contradicts the assumption and $(P_1 \,|||\, P_2)|[G]|(Q_1 \,|||\, Q_2)$ is already proved to be deadlock free in Lemma 4.2. □

Theorem 4.1 (syntactic property)
Let $Q = peer(P)$ be the peer entity of a given entity P obtained by the synthesis algorithm. Then $PM = P|[G]|Q$ is deadlock free.

Proof. The proof is by structural induction. For each binary operator $*$ we prove that $(P_1 * P_2)|[G]|(Q_1 * Q_2)$ is deadlock free where $Q_1 = peer(P_1)$ and $Q_2 = peer(P_2)$. As an example we prove a simple case of parallel operator. Similar proof technique can be applied to other operators.

In the base cases, i.e. $P =$ **stop**, **exit**, or x for some variable $x \in X$, the proof is trivial.

Now suppose

$P = (\sum_{i\in I} a; \gamma_i\downarrow; \overrightarrow{\alpha_i}; P_i)|[a]|(\sum_{j\in J} a; \overleftarrow{\alpha_j}; \gamma_j\uparrow; P_j)$.

In this case, the deadlock freeness of PM is derived by Lemma 4.1 Lemma 4.2 and 4.3.

For recursive cases we suppose that $P =$ **rec** $x.P_1'(x)|[H]|$**rec** $x.Q_1'$. It is sufficient to show that $PM =$ (**rec** $x.P_1'(x)|[H]|$**rec** $x.P_2'$) $|[G]|$(**rec** $x.Q_1'(x)|[H']|$**rec** $x.Q_2$) is deadlock free, where $Q_1'(x) = peer(P_1'(x))$ and $Q_2'(x) = peer(P_2'(x))$. Let $P_1'^{(n)}(x)$ and $P_2'^{(n)}(x)$ be the entities $P_1'(x)$ and $P_2'(x)$ where variable x is simultaneous unfolded n times according to the defining equations $x \stackrel{\text{def}}{=} P_1'(x)$ and $x \stackrel{\text{def}}{=} P_2'(x)$ respectively. Let $Q_1'^{(n)}(x) = peer(P_1'^{(n)}(x))$ and $Q_2'^{(n)}(x) =$

$peer(P_2'^{(n)}(x))$. We prove the claim by showing that $(P_1'^{(n)}(x)|[H]|P_2'^{(n)}(x))|[G]|(Q_1'^{(n)}(x)|[H']|Q_2'^{(n)}(x))$ is deadlock free by induction on n. Now the proof can be obtained by applying structural induction hypothesis on n. The similar proof is given in [13], [12] and we omit the details here.

□

Note that in this paper, $*$ is LOTOS choice, enabling, disabling and parallel operators and for enabling and disabling operators we have,
$PM = (P_1 \gg nc; P_2)|[G \cup \{nc\}]|(Q_1 \gg nc; Q_2)$ and
$PM = (P_1[> nc; P_2)|[G \cup \{nc\}]|(Q_1[> nc; Q_2)$ respectively.

Theorem 4.2 (Semantic Property)
Let $Q = peer(P)$ be the peer entity of a given entity P obtained by the synthesis algorithm. Then the following semantic properties are satisfied, where $\tilde{\beta}$ denotes the sequence $\beta_1\uparrow, \ldots, \beta_n\uparrow, \beta_1\downarrow, \ldots, \beta_m\downarrow$ and $\tilde{\gamma}$ denotes the sequence $\gamma_1\uparrow, \ldots, \gamma_n\uparrow, \gamma_1\downarrow, \ldots, \gamma_m\downarrow$ of non-synchronized actions appearing in P and Q respectively.

(1) $\hat{P} \approx$ **hide** $\tilde{\beta}$ **in** $P|[G]|Q$, *where P is the resulting P with an action of the form $\overleftarrow{\alpha}$ replaced by* $\mathbf{i};\overleftarrow{\alpha}$, *i.e.* $\hat{P} = P\{\mathbf{i};\overleftarrow{\alpha}/\overleftarrow{\alpha}\}$

(2) $\hat{Q} \approx$ **hide** $\tilde{\gamma}$ **in** $P|[G]|Q$, *where Q is the resulting Q with an action of the form $\overrightarrow{\alpha}$ replaced by* $\mathbf{i};\overrightarrow{\alpha}$, *i.e.* $\hat{Q} = Q\{\mathbf{i};\overrightarrow{\alpha}/\overrightarrow{\alpha}\}$.

Proof. The proof is by structural induction for each cases of LOTOS operators (enabling, disabling and parallel). In some cases, the proof is done with the help of expanded tree diagrams of behaviour expressions. The proof is very similar to the ones in [12], [13] and we omit the details here.

□

Note that for enabling and disabling operators we have,
$\hat{P} \approx$ **hide** $\tilde{\beta}$ **in** $(P_1 \gg nc; P_2)|[G \cup \{nc\}]|(Q_1 \gg nc; Q_2)$ and
$\hat{P} \approx$ **hide** $\tilde{\beta}$ **in** $(P_1[> nc; P_2)|[G \cup \{nc\}]|(Q_1[> nc; Q_2)$.
Similarly for $\hat{Q}$

4.4 Example of an Application of the Synthesis Algorithm

We briefly present application example of the synthesis algorithm to the small portion of simplified version of the protocol handler of the Transport Layer [20]. The details can be found in [12],[13], [20]. The protocol handler has two entities, a *calling* entity and a *called* entity. Entities exchange messages in three phases, *connection phase*, *data phase* and *disconnect phase*. After the connection is established successfully, the entities enter data phase. During the data phase the entities can be disabled by the disconnect phase. Therefore the given entity is: P = Connect_Phase_P $\gg$ (Data_Phase_P [> Disconnect_Phase_P) The given entity is transformed as shown below.
P = Connect_Phase_P $\gg$ nc; (Data_Phase_P [> nc;Disconnect_Phase_P). The peer entity Q is obtained by applying the synthesis algorithm as follows:
$Q = peer$(Connect_Phase_P $\gg$ nc; (Data_Phase_P [> nc;Disconnect_Phase_P))

We use notations given in section 3.2 to denote primitives (service primitives (SPs) and protocol data units (PDUs)) used in the Transport Layer. We use the following abbreviations:

For SPs, $ConReq\downarrow : \gamma_1\downarrow, ConCnf\uparrow : \gamma_2\uparrow, DisIn\uparrow : \gamma_3\uparrow, DatReq\downarrow : \gamma_4\downarrow, DatIn\uparrow : \gamma_5\uparrow, DisReq\downarrow : \gamma_6\downarrow$.

For PDUs, $\overrightarrow{CR} : \overrightarrow{\alpha_1}, \overleftarrow{CC} : \overleftarrow{\alpha_2}, \overleftarrow{DR} : \overleftarrow{\alpha_3}, \overrightarrow{DT} : \overrightarrow{\alpha_4}, \overleftarrow{DT} : \overleftarrow{\alpha_5}, \overrightarrow{DR} : \overrightarrow{\alpha_6}$.

The given entity, P, is *calling* entity. The phases of the given entity are as follows

Connect_Phase_P = $\gamma_1\downarrow;\overrightarrow{\alpha_1};(\overleftarrow{\alpha_2};\gamma_2\uparrow;$**exit**
$[\,]\overleftarrow{\alpha_3};\gamma_3\uparrow$;Connect_Phase_$P$)
Data_Phase_P = $\gamma_4\downarrow;\overrightarrow{\alpha_4}$;Data_Phase_$P$
$|||\ \overleftarrow{\alpha_5};\gamma_5\uparrow$;Data_Phase_$P$
Disconnect_Phase_P = $\gamma_6\downarrow;\overrightarrow{\alpha_6}$; **exit** $|||\ \overleftarrow{\alpha_3};\gamma_3\uparrow$; **exit**

Here, we apply the algorithm to the Connect_Phase_P of the given entity only. The application of the algorithm is shown as follows;

Connect_Phase_Q = $peer$(Connect_Phase_P)
= (**rec** x.$peer(\gamma_1\downarrow;\overrightarrow{\alpha_1};(\overleftarrow{\alpha_2};\gamma_2\uparrow;$**exit**
$[\,]\overleftarrow{\alpha_3};\gamma_3\uparrow;x)))$ from (c4)
= **rec** $x.\overrightarrow{\alpha_1};\beta_1\uparrow;peer((\overleftarrow{\alpha_2};\gamma_2\uparrow;$**exit**
$[\,]\overleftarrow{\alpha_3};\gamma_3\uparrow;x))$ from (S1)
= **rec** $x.\overrightarrow{\alpha_1};\beta_1\uparrow;(\beta_2\downarrow;\overleftarrow{\alpha_2};peer($**exit**$)$
$[\,]\beta_3\downarrow;\overleftarrow{\alpha_3};peer(x))$ from (R1)
= **rec** $x.\overrightarrow{\alpha_1};\beta_1\uparrow;(\beta_2\downarrow;\overleftarrow{\alpha_2};$**exit**
$[\,]\beta_3\downarrow;\overleftarrow{\alpha_3};peer(x))$ from (c2)
= **rec** $x.\overrightarrow{\alpha_1};\beta_1\uparrow;(\beta_2\downarrow;\overleftarrow{\alpha_2};$**exit**
$[\,]\beta_3\downarrow;\overleftarrow{\alpha_3};x)$ from (c3)

The result is: Connect_Phase_Q = $\overrightarrow{\alpha_1};\beta_1\uparrow;(\beta_2\downarrow;\overleftarrow{\alpha_2};$**exit**$[\,]\beta_3\downarrow;\overleftarrow{\alpha_3};$ Connect_Phase_Q)

5 Conclusions

We proposed a protocol synthesis algorithm for the process-algebraic language LOTOS to derive a structured protocol model by generating a peer entity from a given entity. Synthesized protocol is proved to be deadlock free and is bisimulated by both entities if the given entity is expressed as a communicating process, which is natural for communicating entities. The bisimulation between entities and protocol model

shows that the protocol model coincides with the behaviours of the entities. We also proposed the definition of *deadlock* which is different from the deadlock defined in the protocol synthesis based on FSMs. The protocol synthesis procedure we have presented is applicable to every structure of protocol layer which provides services to its upper layer. To show the effectiveness of the proposed synthesis algorithm, we have applied it to a simplified version of the protocol handler of the Transport Layer. As reported in the introduction, our work is the extension of our previous works [12], [13]. We are now developing a tool to support our synthesis algorithm. The future directions of research can be to extend the synthesis algorithm for a protocol model with under lying medium and other variables such as time.

References

[1] ISO. Estelle: A formal Description Technique based on an Extended State Transition Model. ISO 9074, 1989.

[2] ISO. Information Processing Systems — Open Systems Interconnection — LOTOS — A formal description technique based on the temporal ordering of observational behaviour. IS 8807, 1989.

[3] CCITT. SDL: Specification and Description Language. CCITT Z.100, 1988.

[4] G.J. Holzmann. Automated protocol validation in argos: Assertion proving and scatter searching. *IEEE Transactions on Software Engineering*, 13(6):683–695, June 1987.

[5] Y. Kakuda and H. Saito. An integrated approach to design of protocol specifications using protocol validation and synthesis. *IEEE Transactions on Computers*, 40(4):459–467, April 1991.

[6] N. Shiratori, H.Kaminaga, K. Takahashi, and S. Noguchi. A verification method for LOTOS specifications and its application. In C.A. Vissers E. Brinksma, G. Scollo, editor, *Protocol Specification, Testing, and Verification, IX*, pages 59–70. North-Holland, 1990.

[7] Y. Kakuda and Y. Wakahara. Component-based synthesis of protocols for unlimited number of processes. In *Proc. IEEE COMPSAC'87*, pages 721–730, October 1987.

[8] C. V. Ramamoorthy, S. T. Dong, and Y. Usuda. An implementation of an automated protocol synthesizer(APS) and its application to the X.21 protocol. *IEEE Transactions on Software Engineering*, SE-11(9):886–908, 1985.

[9] B.V. Bochmann and R. Gotzhein. Deriving protocol specification from service specification. In *Proc. SIGCOMM'86*, volume 14, pages 144–156, 1986.

[10] Yao-Xue Zhang, Kaoru Takahashi, Norio Shiratori, and Shoichi Noguchi. An interactive protocol synthesis algorithm using a global state transition graph. *IEEE Transactions on Software Engineering*, 14(3), March 1988.

[11] B.B. Bista, Zixue Cheng, and Norio Shiratori. A LOTOS based synthesis method for protocol specification. Technical Report 2, Multimedia Communication and Distributed Processing Workshop, IPS of Japan, Nov 1993.

[12] B.B. Bista, Zixue Cheng, Atsushi Togashi, and Norio Shiratori. A new approach for protocol synthesis based on LOTOS. *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences*, E77-A(10):1646–1655, Oct 1994.

[13] B.B. Bista, Zixue Cheng, Atsushi Togashi, and Norio Shiratori. A synthesis algorithm of a protocol model from a single entity. In Dieter Hogrefe and Stefan Leue, editors, *Seventh International Conference on Formal Description Techniques, FORTE'94*, pages 467–482, Berne, Switzerland, 4-7 October 1994.

[14] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89:179–206, 1991.

[15] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[16] P. Merlin and G.V. Bochmann. On the construction of submodule specifications and communication protocols. *ACM Trans. Program. Language Systems*, 5(1):1–25, Jan 1983.

[17] Rom Langerak. Decomposition of functionality : a correctness preserving LOTOS transformation. In Luigi Logrippo, Robert L. Probert, and Hasan Ural, editors, *Proceedings of the Tenth International IFIP WG 6.1 Symposium on Protocol Specification, Testing, and Verification*, pages 203–218, Ottawa, Ontario, Canada, June 1990. North-Holland.

[18] Teruo Higashino. Service specification and its protocol specifications in lotos-a survey for synthesis and execution-. *IEICE Trans. Fundamentals*, E75-A(3):330–338, March 1992.

[19] Peter van Eijk and Jeroen Schot. An exercise in protocol synthesis. In K.R. Parker and G.A. Rose, editors, *Formal Description Techniques, IV*, pages 117–131, Sydney, Australia, November 1991. Elsevier Science Publishers B.V.

[20] A. Tanenbaum. *Computer Networks.* Prentice-Hall International Editions, 1989.