Modularization of a specification in LOTOS

Kentaro Goh and Norio Shiratori

Research Institute of Electrical Communication
Tohoku University
2-1-1, Katahira, Sendai, 980 Japan

Abstract

In this paper we propose a transformation algorithm of LOTOS specifications from monolithic style to constraint-oriented style. The objective of this transformation is to decompose a monolithic LOTOS behaviour into two processes composed by the parallel operator. In order to make behaviours observation congruent before and after the decomposition, the algorithm creates an additional process from the given specification and adds to the decomposed processes using parallel operator. The additional process consists of n interleaved subprocesses (n ≥ 1). There are no internal interaction gates among decomposed processes and the added process. The interactions among them occur only at the visible gates. This transformation has the following applications: (1) modularization or decomposition of a specification, (2) restructuring of complex specifications and (3) derivation of protocol specifications from service specifications.

1 Introduction

In the design of large and complex systems like communication protocols and information networks, the use of top-down approach based on the stepwise refinement is effective. On this approach a requirement specification of the system to be developed is decomposed and refined step by step. Formal Description Techniques (FDTs) have the advantages of (1) elimination of ambiguity and (2) verification and validation on specifications, so the use of FDTs on the design steps is very effective. However, on the stepwise approach an equivalence relation between specifications before and after the decomposition or the refinement is necessary.

LOTOS[1] is one of FDTs developed within the ISO and has the mathematical basis based on an equivalence. Therefore the correctness of transformations of specifications on the design steps can be formally proved. The transformations of specifications correspond to the decomposition or the refinement of specifications.

The transformation method preserving a correctness of specifications may be divided into following two approaches:

(a) the approach which after transformation, verifies whether the result is correct or not and if it is not correct the transformation steps are repeated until the correct result is obtained.

(b) the approach which uses transformation tools that always preserve the correctness of the refined specification.

In practice the approach (a) can be difficult for large specifications, therefore in the actual system development the approach (b) is more effective.

In LOTOS, the approach (b) is called the Correctness Preserving Transformation (CPT) [2] and plays an important role in the LOTOS-based development strategy. In order to develop large systems by the stepwise refinement effectively, the systematic CPTs for decomposition methods are desired.

The algorithm presented in this paper is one of the CPTs. The objective of this transformation is to decompose a LOTOS behaviour into parallel modules while the transformation preserves their correctness.

The works on the LOTOS transformation method aimed at the decomposition or the modularization published in the literature include [3], [4], [5], [6] and [7]. The main purpose of [3], [4], [5] and [6] is to derive protocol specifications from service specifications. Thus, these methods transform a specification into resource-oriented style, in which protocol entities and communication channels are expressed by certain processes respectively. Subprocesses decomposed by their methods correspond to protocol entities, and these subprocesses synchronize at internal gates via communication channel processes. Therefore, their methods do not achieve a high degree of parallelism among decomposed processes. In the case of distributed development of the modules these methods lead to low productivity, since negotiations about internal synchronization gates must be necessary. In [5], which is extended in [6], the number of decomposed processes is n. However the proof of correctness is not given. Limitations of a process to be decomposed are as follows: (1) No internal action i and (2) No choice between events of different subprocesses. In [7], the number of decomposed processes is 2, and the proof of correctness is given for the observation equivalence. A limitation of a process to be decomposed is no i. The i should be expressed by the hiding operator. [8] is an extension of [7] to increase the number of decomposed processes to n. Contrary to these methods, [9] decomposes a specification into modules which do not synchronize at internal synchronization gates. Thus the method can achieve a high degree of parallelism between decomposed processes. The proof of
Table 1: Summary of the main features of decomposition methods in LOTOS

<table>
<thead>
<tr>
<th></th>
<th>[5,6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
<th>Our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>style after the application</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallelism</td>
<td>Not maintain</td>
<td>Not maintain</td>
<td>Not maintain</td>
<td>Maintain</td>
<td>Maintain</td>
</tr>
<tr>
<td>No. of comp. proc.</td>
<td>n</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Proof of correctness</td>
<td>None</td>
<td>Observation equivalence</td>
<td>Observation equivalence</td>
<td>Observation congruence</td>
<td></td>
</tr>
<tr>
<td>Limitations</td>
<td>No i (1)</td>
<td>No i (2)</td>
<td>No i (2)</td>
<td>No i</td>
<td>No i (3)</td>
</tr>
<tr>
<td>Application area</td>
<td>(a)</td>
<td>(a)</td>
<td>(a)</td>
<td>(a)</td>
<td>(b)</td>
</tr>
</tbody>
</table>

(1) No choice between events of different subprocesses
(2) i should be expressed by hiding operator
(3) Satisfy a specific predicate (it results from the parallel composition of two subprocesses)
(4) Each gate is used only once

Correctness is given for the observation equivalence. Limitations of processes to be decomposed are as follows:
(1) No i, (2) No recursion and (3) Satisfying a specific predicate, i.e. it results from the parallel composition of two processes. If a process does not satisfy the predicate, the method creates hidden internal gates which are strategically located in the process to be decomposed. Therefore the specification style after decomposition in [9] can be recognized as resource-oriented style.

In this paper, we propose a new decomposition algorithm for LOTOS specifications. The presented algorithm is a transformation algorithm for LOTOS specifications from monolithic style to constraint-oriented style[3][4]. The objective of this transformation is to decompose a monolithic LOTOS behaviour into two processes composed by parallel operator. Decomposed processes model minimal local constraints of the system interactions. In order to make behaviours observation congruent before and after the decomposition, the algorithm creates and adds an additional process to the decomposed processes using parallel operator. The additional process consists of n interleaved subprocesses (n ≥ 1). These subprocesses model minimal remote (or end-to-end) constraints of interactions between decomposed processes. There are no internal interaction gates among decomposed processes and the added process. The interactions among them occur only at the visible gates. The advantage of our method over [8] is that our method handles the increase in number of processes to be decomposed while keeping the high parallelism of decomposed processes. The transformation to constraint-oriented style corresponds to a decomposition of a process without decrease of its abstraction level, while the transformation to resource-oriented style decreases a process abstraction level. Therefore the transformation to constraint-oriented style is useful for an application to the requirement capturing (or the specification) phase in a system development process. The summary of the main features of decomposition methods in LOTOS is shown in Table 1.

The formal language used in this paper is a subset of LOTOS, but the approach could also be adopted to other process algebra formalisms like CCS[10], CSP[11] or ACP[12]. The essential common feature of these formalisms is that they allow Labelled Transition Systems (LTSs)[1][10].

This transformation has many practical applications such as, (1) modularization or decomposition of a specification, (2) restructuring of complex specifications and (3) derivation of protocol specifications from service specifications.

This paper is organized as follows. In section 2, we introduce the decomposition problem and formalize the problem. In section 3, we present our decomposition method and show an application example. Finally in section 4, we conclude the paper.

2 Formalization of the decomposition problem

2.1 Example of the decomposition

In order to get a better intuitive understanding of the decomposition problem we first illustrate it with a simple example. We take the question/answer service[4], for an example.

Example 1. A question is generated by a user Q (action QQ) and forwarded by the service to another user A (AQ). The latter is required to generate an answer (AQ) which is returned by the service to the user Q (QQ). An architecture and action sequence of this service are shown in Figure 1(a). The process 'B' below is a monolithic representation of the question/answer service as given in the informal definition.

\[ B \equiv QQ; AQ; AQ; QQ; \text{stop} \]
In order to refine this specification by a team of specifiers the specification must be decomposed. The first design decision might be to split the specification into two processes, \( LC_1 \) and \( LC_2 \) such that:
- the set of actions of \( LC_1 \) is concerned with actions which occur at the user Q side and
- the set of actions of \( LC_2 \) is concerned with actions which occur at the user A side.
In other words, \( LC_1 \) and \( LC_2 \) decide the temporal relationship between observable interactions at the Q side and the A side respectively. These are local constraints of the service, so these are expressed by separate processes.

Moreover, an equivalence relation between before and after the decomposition is necessary. So we add an additional process \( RC \) which acts for a remote constraint of the service. \( RC \) models remote, or end-to-end, constraints of the temporal ordering of observable interactions. It consists of \( n \) interleaved subprocesses \( RC_i \)'s, and a \( RC \) decides minimal temporal ordering of observable interactions. The number of \( RC_i \)'s, \( n \), depends on the frequency of interactions between \( LC_i \). If we write an equivalence relation as \( \equiv \), we can formulate the problem of decomposition of the question/answer service as follows (See Figure 1(b)):

Find two processes \( LC_1 \) and \( LC_2 \), and an additional process \( RC \) such that

\[
(\langle LC_1 \| \| LC_2 \rangle \langle G \rangle) \quad RC \equiv B
\]

where \( G \) which is a subset of \( \{Q_1, Q_2, A_1, A_2, Q_3\} \) is a set of actions at which two \( LC_i \)'s and \( RC \) synchronize, and \( RC \) consists of \( n \) interleaved subprocesses as follows:

\[
RC = RC_1 \| \| RC_2 \| \| \ldots \| \| RC_n \quad (n \geq 1)
\]

We obtain the following problems as solutions of this problem if we take the observation congruence as an equivalence relation:

\[
\begin{align*}
LC_1 &\equiv Q_1; Q_2; \text{stop}, & \quad LC_2 &\equiv A_1; A_2; \text{stop} \\
RC_1 &\equiv Q_4; A_4; \text{stop}, & \quad RC_2 &\equiv A_3; Q_3; \text{stop}
\end{align*}
\]

and, \( G = \{Q_1, Q_2, A_1, A_2\} \).

2.2 Notations and Assumptions

Assumption 1 In this paper we use only the Basic LOTOS as we do not deal with data types and value expressions, and the LOTOS operators that are treated are \( \text{stop} \), action prefix, choice and parallel. We assume that the processes before and after decomposition, i.e. \( B, LC_1 \) and \( LC_2 \), and the additional processes \( RC_i \)'s, make use of only \( \text{stop} \), action prefix and choice.

Assumption 2 In this paper we use observation congruence [10] for a basis of an equivalence relation between processes before and after the transformation, denoted as \( \equiv \).

Assumption 3 In this paper we assume that the specification style of the process \( B \) to be decomposed is the LOTOS monolithic style [4].

The general form of the monolithic style is as follows[7]:

\[
B \equiv \sum \{ a_i; B_i \mid i \in I \}
\]

for some finite index set \( I \), where each \( B_i \) is either a process identifier or an expression in a general form of the monolithic style. By \( \sum \{ P_i \mid i \in \{1, 2, \ldots, n\} \} \) we mean \( P_1 \| \| P_2 \| \| \ldots \| \| P_n \) and by \( \sum \emptyset \) we mean \( \text{stop} \).

Notation 1 The set of all observable actions (so not including \( \delta \) and \( i \)) that occur anywhere in process \( B \) is denoted by \( \text{Act}(B) \).

For example, if \( B \equiv a; b; \text{stop} \| \| c; \text{stop} \) then \( \text{Act}(B) = \{a,b,c\} \).

Now we formalize the decomposition problem.

Definition 1 (Decomposition problem)

Given:
- a monolithic process \( B \)
- a partitioning of \( \text{Act}(B) \) into \( A_1 \) and \( A_2 \), i.e. \( A_1 \cup A_2 = \text{Act}(B) \) and \( A_1 \cap A_2 = \emptyset \)

Problem: find two processes \( LC_1 \) and \( LC_2 \), and an additional process \( RC \) with the following properties:

- \( \text{Act}(LC_1) = A_1 \) and \( \text{Act}(LC_2) = A_2 \) and \( \text{Act}(RC) \subseteq \text{Act}(B) \)
- \( (\langle LC_1 \| \| LC_2 \rangle \langle G \rangle) \quad RC \equiv B \)

where \( RC \equiv RC_1 \| \| RC_2 \| \| \ldots \| \| RC_n \) \( (n \geq 1) \), \( G \subseteq \text{Act}(B) \).

Assumption 4 The transformation presented here imposes further restrictions, which are given below, to the process to be decomposed.

1. no internal actions \( i \),
2. no recursion
3. each action of the process can be distinguished by a distinct name.

These restrictions are not essential drawback of the transformation since these can be dealt by slight extensions of the algorithm. Same restrictions are adopted by [9]. We discuss about the restrictions on section 4.

Although, by our method a process is decomposed into two processes, it can be decomposed into \( n \) processes by application of the algorithm \( n - 1 \) times.

3 Decomposition method

In this section we develop the decomposition algorithm. First of all, in section 3.1 we give some definitions. In section 3.2 we describe the decomposition algorithm. In section 3.3 we show properties of the
algorithm. In section 3.4 we show an application example of the algorithm. Finally in section 3.5 we show a decomposition example of a recursive process.

3.1 Definitions

Definition 2 A potential process of a process $B$ is defined as a state of LTS of the process $B$. For example, the potential processes of a process $B$ are $B$, $B_1$, $B_2$, and $B_3$, where $B \equiv a; \langle b; \text{stop}\rangle c; \text{stop}$, $B_1 \equiv b; \text{stop}$, $B_2 \equiv \text{stop}$ and $B_3 \equiv \text{stop}$.

Definition 3 A preaction of a process $B$, $\text{Pre}(B)$, is a preceding action of the process $B$.

Definition 4 The set of actions in which a process $B$ can immediately participate at any point in the unfolding of its behaviour is called the initials of the process $B$, denoted by $\text{Init}(B)$.

Definition 5 A contact set of actions of a process $B$, $\text{Cont}(B)$, is defined as a set of $\text{Init}(B) \cup \{\text{Pre}(B)\}$. A $\text{Cont}(B)$ is equal to an $\text{Init}(B)$, if the $B$ has no $\text{Pre}(B)$.

Definition 6 A potential process $B$ except the stop is called a joint process if and only if $\text{Cont}(B) \subseteq A$, where $A$ is a set of actions.

Definition 7 A potential process $B$ except the stop is called a disjoint process if and only if $\text{Cont}(B) \cap A_1 \neq \emptyset$ and $\text{Cont}(B) \cap A_2 \neq \emptyset$, where $A_1$ and $A_2$ are disjoint sets of actions.

Definition 8 If we construct a new potential process $P$, which has the identical preaction and initials as other potential process $Q$, then the potential process $P$ is called the reconstruction of the potential process $Q$.

Definition 9 For a process $B \equiv \sum \{a_i; B_i \mid i \in I\}$ and a set of actions $A$, $\text{Rest}(B, A)$ is defined as follows:

$$\text{Rest}(B, A) \equiv \sum \{a_i; \text{Rest}(B_i, A) \mid a_i \in A, i \in I\}$$

$$\sum \{\text{Rest}(B_i, A) \mid a_i \not\in A, i \in I\}$$

where $I$ is a finite index set. Ordinarily it is assumed that $\text{Rest}(\text{stop}, A) \equiv \text{stop}$.

For example, if $B \equiv a; \langle b; c; \text{stop}\rangle d; e; \text{stop} \rangle f; g; h; \text{stop}$ and $A = \{a, b, e, h\}$, then $\text{Rest}(B, A) \equiv$

![Figure 2: LTSs of examples of a joint process and a disjoint process](image)

![Figure 3: LTSs of an example of a reconstruction of processes](image)

![Figure 4: LTSs of an example of the function Rest](image)
Algorithm

In view of a parallelism of decomposed processes and reconstructions of the processes, let us consider the following strategy.

Strategy

(1) The construction method of LCs

We map \( LC_1 \) and \( LC_2 \) from the given process \( B \) onto processes based on the disjoint sets of actions \( A_1 \) and \( A_2 \) by using the \( Rest(B, A_1) \) and \( Rest(B, A_2) \), respectively.

(2) The construction method of joint processes and disjoint processes

For each potential process of the given process \( B \), if the potential process is a joint process of \( A_1 \), then we reconstruct the potential process to \( LC_1 \), else if the potential process is a joint process of \( A_2 \), then we reconstruct the potential process to \( LC_2 \), and it is a joint process of \( A_1 \) and \( A_2 \) then we reconstruct the potential process to one of \( RC_i \)s.

(3) The construction method of sets of actions of which \( RC_i \)s consist

We make a set of actions \( C_i \) of which \( RC_i \) consists as follows:

\[
\bigcup_{i \in \{1, 2, \ldots, n\}} C_i \subseteq \text{Act}(B)
\]

where \( C_i \)s are pairwise disjoint.

(4) The construction method of the set of synchronization gates

We make the set of synchronization gates \( G \) as follows:

\[
G = \bigcup_{i \in \{1, 2, \ldots, n\}} C_i
\]

where \( C_i \) is a set of actions of which \( RC_i \) consists.

We shall now describe the decomposition algorithm, which is a solution of the problem mentioned in section 2.2, based on the strategy.

Algorithm

Input: A monolithic process \( B \) and a partitioning of \( \text{Act}(B) \) into \( A_1 \) and \( A_2 \).

Output: Two decomposed processes \( LC_1 \) and \( LC_2 \), and an additional process \( RC \equiv (RC_1 \parallel \cdots \parallel RC_n) \) with \( n \geq 1 \).

Step 1: Find action sets \( C_1, C_2, \ldots, C_n \) which construct \( RC \) based on the strategy by using following four substeps:

Step 1.1: For each potential process of \( B \) except for stop, make the contact set of actions of the potential process. These sets are named by suffix 1, 2, 3, ... respectively.

Step 1.2: Eliminate sets which are subsets of \( A_1 \) or \( A_2 \) from the sets made on step 1.1.

Step 1.3:

(i) Make a new set by taking union of sets of step 1.2 if the intersection of those sets is not an empty set. Eliminate old sets and suffix new sets with the smallest suffix number of old sets.

(ii) Repeat (i) until the intersection of sets is an empty set.

Step 1.4: Rename the sets which made until step 1.3 by \( C_1, C_2, \ldots, C_n \) in order of small suffix respectively.

Step 2: Construct the processes \( LC_1, LC_2 \) and \( RC_1, RC_2, \ldots, RC_n \) using the function \( Rest \) as follows:

\[
\begin{align*}
LC_1 &\equiv Rest(B, A_1), \\
LC_2 &\equiv Rest(B, A_2) \\
RC_1 &\equiv Rest(B, C_1), \\
&\quad \ldots, \\
RC_n &\equiv Rest(B, C_n)
\end{align*}
\]

3.3 Properties of the algorithm

The processes \( LC_i \)s and \( RC_i \)s generated by the algorithm have the following properties.

Proposition 1 For each potential process of \( B \) to be decomposed,

(1) if the potential process is a joint process of \( A_1 \), then the reconstruction of the potential process exists only in \( LC_1 \), else if the potential process is a joint process of \( A_2 \), then the reconstruction of the potential process exists only in \( LC_2 \), and

(2) if the potential process is a disjoint process then the reconstruction of the potential process exists only in one of \( RC_i \)s.

(proof) This is obvious by the algorithm based on strategy (2) and by the definition of the function \( Rest \).

Proposition 2

(1) The number of joint processes of \( A_1 \) in \( B \) is equal to the number of reconstructions of the joint processes in \( LC_1 \), and the number of joint processes of \( A_2 \) in \( B \) is equal to the number of reconstructions of the joint processes in \( LC_2 \).

(2) The number of disjoint processes in \( B \) is equal to the number of reconstructions of the disjoint processes in \( RC \).

(proof) This is obvious by the algorithm based on the strategy and by the definition of the function \( Rest \).

Theorem Let \( LC_1, LC_2 \) and \( RC_1, RC_2, \ldots, RC_n \) be generated by the algorithm. Let \( B \) be a monolithic process restricted as Assumption 4. Then \( B \) is transformed into the process \( D \):

\[
D = (LC_1 \parallel \cdots \parallel LC_n) \parallel (RC_1 \parallel \cdots \parallel RC_n)
\]

where \( G = \bigcup_{i \in \{1, 2, \ldots, n\}} C_i \), and \( B \cong D \).

(proof) See Appendix A.
to the medium. Then the receiver receives recmsg. If the message is correct then the receiver gives the user magind and sends an acknowledgement sndack to the medium, else if the message is corrupt then the receiver sends a negative acknowledgement sndnak. Finally the sender receives recack or recnak. $SP$ can be described as the following monolithic process.

$$SP \equiv \text{msgreq}; \text{sndmsg}; \text{recmsg};$$
$$\quad (\text{magind}; \text{sndack}; \text{recack}; \text{stop})$$
$$\quad \text{sndnak}; \text{recnak}; \text{stop}$$

As a basis of a decomposition, we partition a set of actions of the process $SP$ into following two sets of actions:

- $\{\text{msgreq, sndmsg, recack, recnak}\}$, which occur at Sender side, and
- $\{\text{magind, recmsg, sndack, sndnak}\}$, which occur at Receiver side.

Sender Medium Receiver  
<table>
<thead>
<tr>
<th>magind</th>
<th>sndmsg</th>
<th>recmsg</th>
<th>magind</th>
<th>sndmsg</th>
<th>recmsg</th>
</tr>
</thead>
<tbody>
<tr>
<td>recack</td>
<td>sndack</td>
<td>recnak</td>
<td>recack</td>
<td>sndack</td>
<td>recnak</td>
</tr>
</tbody>
</table>

(a) Normal sequence  
(b) Abnormal sequence

Figure 5: Sequence charts of Simple protocol

Here, if we input the process $SP$ and the above two sets of actions to the algorithm then the algorithm decomposes the process $SP$ into two processes $LC_1$ and $LC_2$ with an additional process $RC$ via following steps:

**Step 1:** In order to find action sets $C_1, C_2, ..., C_n$ which construct $RC$'s, following four substeps are carried out.

**Step 1.1:** On this step, following seven sets of actions are made.

$$\{\text{msgreq}\}, \{\text{msgreq, sndmsg}\},$$
$$\{\text{sndmsg, recmsg}\}, \{\text{recmsg, sndnak, magind}\},$$
$$\{\text{sndnak, recnak}\}, \{\text{msgind, sndack}\},$$
$$\{\text{sndack, recack}\}$$

**Step 1.2:** The four sets of actions suffixed 1, 2, 4 and 6 are eliminated from the sets of actions made on the step 1.1, since the sets suffixed 1 and 2 are subsets of the set of Sender side actions and the set suffixed 4 and 6 are subsets of the set of Receiver side actions. Thus the following sets of actions are left.

$$\{\text{sndmsg, recmsg}\}, \{\text{sndnak, recnak}\},$$
$$\{\text{sndack, recack}\}$$

**Step 1.3:** The sets of actions suffixed 3, 5 and 7 satisfy the exit condition of the step 1.3, thus the algorithm is proceeded to next steps.

**Step 1.4:** On this step, the sets of actions suffixed 3, 5 and 7 are renamed by $C_1, C_2$ and $C_3$ respectively.

$$\{\text{sndmsg, recmsg}\}_3 = C_1,$$
$$\{\text{sndnak, recnak}\}_5 = C_2,$$
$$\{\text{sndack, recack}\}_7 = C_3$$

**Step 2:** Construct the processes $LC_1$, $LC_2$ and $RC_1$, $RC_2$ and $RC_3$ using the function $Rest$ as follows:

$$LC_1 \equiv Rest(SP, A_1)$$
$$\equiv \text{msgreq}; \text{sndmsg}; (\text{recnak}; \text{stop} | \text{recack}; \text{stop})$$
$$\equiv \text{recmsg}; (\text{sndnak}; \text{stop} | \text{msgind}; \text{sndack}; \text{stop})$$
$$RC_1 \equiv Rest(SP, C_1)$$
$$\equiv \text{sndmsg}; \text{recmsg}; \text{stop}$$
$$RC_2 \equiv Rest(SP, C_2)$$
$$\equiv \text{sndnak}; \text{recnak}; \text{stop}$$
$$RC_3 \equiv Rest(SP, C_3)$$
$$\equiv \text{sndack}; \text{recack}; \text{stop}$$

As a result, the decomposed processes are $LC_1$ and $LC_2$, thus by the theorem, the following process:

$$SP' \equiv \{LC_1 \mid \mid LC_2\} \mid \mid G\mid \{RC_1 \mid \mid RC_2 \mid \mid RC_3\}$$

with $G = \{\text{sndmsg, recmsg, sndnak, recnak, sndack, recack}\}$ is observation congruent with the process $SP$.

The process $SP$ whose specification style is a monolithic style is transformed into the process $SP'$ whose specification style is a constraint-oriented style.

LTSs of Example 2 are shown in Figure 6.

Figure 6: LTSs of Example 2

3.5 Recursive processes

As presented in Assumption 4(2), the algorithm does not deal with recursive processes. However this limitation can be easily eliminated by checking parts of a recursive call of a process and by reconstructing...
it to decomposed processes. We consider the following example.

**Example 3** The simple protocol with a recursion, $SR$, is based on $SP$ on Example 2. If the sender receives $recna$ from the medium then sends the message $sndmsg$ again. $SR$ can be described as the following monolithic process.

$$SR \equiv msgreq; RE$$
$$RE \equiv sndmsg; recmsg;$$
$$\left(\begin{array}{l}
msgind; sndack; recack; stop \\
[] sndnak; recnak; RE
\end{array}\right)$$

As a basis of a decomposition, we partition a set of actions of the process $SR$ into following two sets of actions:

- $\{msgreq, sndmsg, recack, recnak\}$
- $\{msgind, recmsg, sndack, sndnak\}$

We input the process $SR$ and the above two sets of actions to the algorithm and then we obtain the following processes if we modify the function $Rest$ into which it can check recursive calls and halt.

$$LC_1 \equiv msgreq; RP_1$$
$$RP_1 \equiv sndmsg; (recnak; RP_1 [] recack; stop)$$
$$LC_2 \equiv recmsg; (sndnak; LC_2 [] msgind; sndack; stop)$$
$$RC_1 \equiv sndmsg; recmsg; RC_1$$
$$RC_2 \equiv sndnak; recnak; RC_2$$
$$RC_3 \equiv sndack; recack; stop$$

It can be confirmed that the following process:

$$SR' \equiv (LC_1 \parallel LC_2) \parallel RC$$

with $G = \{sndmsg, recmsg, sndnak, recnak, sndack, recack\}$ is observation congruent with the process $SR$. LTSs of Example 3 are shown in Figure 7.

4 Discussions and Conclusions

In this paper we have presented a transformation algorithm of LOTOS specifications from a monolithic style to a constraint-oriented style. The objective of this transformation is to decompose a monolithic LOTOS behavior into two processes composed by the parallel operator. In order to make behaviors observation congruent before and after the decomposition, the algorithm creates and adds an additional process to the decomposed processes using parallel operator. The additional process consists of $n$ interleaved sub-processes ($n \geq 1$). There are no internal interaction gates among decomposed processes and the added process. The interactions among them occur only at the visible gates.

The decomposition method presented in this paper is a transformation method of LOTOS specifications from monolithic style to constraint-oriented style. When we want to change a general parallel operator between $LC$s and $RC$ to a full synchronization operator, we should, for each action $a$ where $a \in Act(B)$ and $a \notin Act(BC)$, construct a process "$a$; stop" and add this process to $RC$s in parallel.

A process which can be decomposed by our method has the following restrictions:

1. no internal actions $i$,

2. no recursion and

3. each action of the process can be distinguished by a distinct name.

(1) is not a serious restriction because the internal actions $i$ can be expressed by using the hiding operator. (2) which has mentioned section 3.5 can be easily eliminated by checking parts of a recursive call of a process and by reconstructing it to decomposed processes. Elimination of (3) is now under research, however it is possible that processes without choice operator and actions with same names can be decomposed by the algorithm presented in this paper.

Although, by our method a process is decomposed into two processes, it can be decomposed into $n$ processes by application of the algorithm $n - 1$ times.

The presented transformation has the following applications: (1) modularization or decomposition of a specification, (2) restructuring of complex specifications and (3) derivation of protocol specifications from service specifications.

Finally, the directions of further research are summarized as follows:

(a) elimination of all restriction of processes to be decomposed,

(b) incorporation of full LOTOS, i.e., including data types and value passing, and

(c) development of a support environment for decomposition of LOTOS specifications implementing the algorithm.

References

A Proof of Theorem

In order to prove that $B$ is observation congruent with $D$, we prove that the LTS of $B$ is identical with the LTS of $D$ by showing that:

(A) the initials of $B$ are identical with the initials of $D$, and

(B) for any potential process $B'$ of $B$ and for any potential process $D'$ of $D$ if the preaction of $B'$ is identical to the preaction of $D'$ then the initials of $B'$ are identical with the initials of $D'$.

First of all, by Definition 6 and 7, each potential process of $B$ is divided into three categories: (a) joint process, (b) disjoint process, and (c) stop. stop is a completely inactive process. It cannot perform any action. Thus we only consider (a) and (b) here.

**Proof of (A)**

By Definition 6 and 7, there are two cases with subcases.

**Case 1:** $B$ is a joint process.

An action which can be occurred at $B$ is one of the actions of initials of $B$. Here, $B$ is only reconstructed in $LC_1$ or $LC_2$ of $D$ with the same initials, and is not reconstructed in $RC$ of $D$ from Proposition 1 and 2.

Regarding the succeeding process of an action, the action which can be occurred at $B$ is divided into three subcases:

(i) preaction of a joint process,

(ii) preaction of a disjoint process, and

(iii) preaction of stop.

In subcase (i), the action only exists in either $LC_1$ or $LC_2$ of $D$, therefore only occurs at $D$ and does not synchronize with other actions. In subcase (ii), the action exists not only in either $LC_1$ or $LC_2$ but also in $RC$ of $D$. Thus the action which occurs at $D$ is the synchronization of either $LC_1$ or $LC_2$ and $RC$. The subcase (iii) is the same as (i).

From the above, by considering the parallel composition of $LC_1$ and $LC_2$, the parallel composition among $RC_1$, $RC_2$, ..., $RC_n$, and the parallel composition of $LC$s and $RC$s, the initials of $B$ are identical with the initials of $D$.

**Case 2:** $B$ is a disjoint process.

An action which can be occurred at $B$ is one of the actions of initials of $B$. Here, $B$ is only reconstructed in one of the $RC$s of $D$ with the same initials, and is not reconstructed in $LC$s of $D$ from Proposition 1 and 2.

Regarding the succeeding process of an action, the action which can be occurred at $B$ is divided into three subcases like Case 1. However in all subcases, the action exists not only in either $LC_1$ or $LC_2$ but also in $RC$ of $D$. Thus the action which occurs at $D$ is the synchronization of either $LC_1$ or $LC_2$ and $RC$. From the above, by considering the parallel composition of $LC_1$ and $LC_2$, the parallel composition among $RC_1$, $RC_2$, ..., $RC_n$, and the parallel composition of $LC$s and $RC$s, the initials of $B$ are identical with the initials of $D$.

**Proof of (B)**

Let $B'$ and $D'$, which are potential processes of $B$ and $D$ respectively, have the same preactions. According to the succeeding process of $B'$, the action which can be occurred at $B'$ is divided into three categories like (A). It is obvious, by similar discussion to (A), that the initials of $B'$ are identical with the initials of $D'$ by parallel compositions among processes of $B'$.

Therefore, the LTS of $B$ is identical with the LTS of $D$, and $B$ is observation congruent with $D$. □