An Error Detection Method for Recursive Processes for LOTOS Instruction and Its Support System

Byung-ho PARK,* Kaoru TAKAHASHI** and Norio SHIRATORI***

*Graduate School of Information Sciences, Tohoku University, Sendai, Japan
email: sunson@shiratori.riec.tohoku.ac.jp
**Sendai National College of Technology, Sendai, Japan
email: kaoru@cc.sendai-ct.ac.jp
***Research Institute of Electrical Communication/Graduate School of Information Sciences,
Tohoku University, Sendai, Japan
email: norio@shiratori.riec.tohoku.ac.jp

Received October 7, 1998; final version accepted February 8, 1999

LOTOS, one of the formal description techniques, can strictly and unambiguously describe the requirement specifications of distributed systems such as protocols and communication systems. But it is difficult for beginners to learn and understand LOTOS. Thus, for LOTOS instruction support, we have already proposed an algorithm which can detect learner's errors contained in the learner's answer for some problems described in LOTOS. This algorithm is applied to non-recursive LOTOS processes. However, non-recursive processes cannot represent processes that execute events infinitely. In this paper, we propose a new algorithm which can support equivalence decision and error detection in recursive processes. We also present a software support system which aids this new algorithm.

KEYWORDS: recursive processes, LOTOS, equivalence decision, error detection, education support system

1 Introduction

LOTOS, one of the standard Formal Description Techniques (FDTs) developed within ISO, is mainly used in describing Open Systems Interconnection (OSI), especially service definitions and protocol descriptions [2][4]. But LOTOS is considered to be difficult to understand, learn and use it to describe system behaviors because of its complex mathematical concepts and notations, comparing to other FDTs (e.g. Estelle [5] and SDL [3]) [13][14]. Therefore, LOTOS has not been widely used despite some superior advantages such as verification ability of specifications and description ability of parallel behaviors which are absent in other FDTs.

To solve these problems, an education support system for novice users may be necessary. In an education support system for learning LOTOS, it is necessary that the system points out the distinction between the learner's description (learner's answer) and the correct description (correct answer) for some problems described in LOTOS. However, past efforts can only verify whether two descriptions are equivalent or not [1][6][7][12].

We have supplemented the verification ability with some additional features including the error detection, the error classification (what kind of errors are occurred), the error position (where the errors are occurred), etc [10]. Furthermore, we have developed a system which can verify equivalence between a learner's description and the correct description, and can detect the learner's errors [10]. However, these features and the system are limited to finite LOTOS processes.

In practice, general LOTOS processes may perform infinite sequences of events or actions. To support such processes in LOTOS instruction, this paper proposes an error detection algorithm for recursive processes. An education support system which can support the recursive processes is also developed.

The organization of the paper is as follows: In Section 2, we explain the syntax and semantics of LOTOS. An algorithm is also presented in this section which translates the minimal LTS (Labelled Transition System) to the simplest LTS. In Section 3, for recursive (infinite LOTOS) processes, we propose an algorithm which has remarkable features such as verification of equivalence, error detection, error classification, etc. The education support system is shown in Section 4, followed by the conclusion in Section 5.

2 Preliminary

2.1 LOTOS

In this subsection, we outline the basic concept of LOTOS.

In LOTOS, a system is described by defining the temporal relation between actions (or events) which can oc-
cur in the system. The system is considered as a black box, and communication between the system and the environment is performed at interaction points called gates.

LOTOS consists of two descriptive components: The first is the description of process behaviors and interactions. The second is the data structures and value expressions. We focus only on the first component of LOTOS for making the education support system.

First, we define a process behavior expression. We assume a finite set $\mathcal{A}$ of actions and a countable infinite set $\mathcal{X}$ of process variables that can be replaced by any process behavior expression.

**Definition 2.1.** A **process behavior expression** is defined inductively as follows:

1. A process variable $x \in \mathcal{X}$ is a process behavior expression.
2. stop is a process behavior expression.
3. If $a \in \mathcal{A}$ and $P$ is a process behavior expression, then $a;P$ is a process behavior expression.
4. If $P$ and $Q$ are process behavior expressions, then $P;Q$ is a process behavior expression.
5. If $x$ is a process variable and $P$ is a process behavior expression, then rec $x.P$ is a process behavior expression.

Note that, to represent recursive processes, a recursive operator (rec) is adopted in this syntax which is slightly different from the original syntax. This makes the syntax simple.

The semantics of a process behavior expression is given by an LTS (Labelled Transition System).

**Definition 2.2.** An LTS Sys is a 4-tuple $\langle S, \mathcal{A}, \rightarrow, s_0 \rangle$ where

1. $S$ is a non-empty finite set of states,
2. $\mathcal{A}$ is the finite set of actions,
3. $\rightarrow$ is a transition relation over states ($\rightarrow \subseteq S \times \mathcal{A} \times S$), and
4. $s_0 \in S$ is the initial state of Sys.

In the following, if $(s, a, s') \in \rightarrow$, then we write $s \xrightarrow{a} s'$. Therefore, we can write

$$\rightarrow = \bigcup_{a \in \mathcal{A}} \{ \xrightarrow{a} \mid \xrightarrow{a} \subseteq S \times S \}.$$

**Definition 2.3.** The transition relation between process behavior expressions is derived by the following rules:

$$\begin{align*}
\frac{P \xrightarrow{a;P} P'}{Q \xrightarrow{P;Q} Q'} & \quad \frac{P \xrightarrow{\text{rec } x.P/x} P'}{	ext{rec } x.P \xrightarrow{Q/x} P'}
\end{align*}$$

where $P\{Q/x\}$ represents the process behavior expression which is obtained by simultaneously replacing all free occurrences of variable $x$ in process behavior expression $P$ by process behavior expression $Q$.

By the transition relation of Definition 2.3, a process behavior expression can be represented as a state transition diagram. Thus, an LTS can correspond to the state transition diagram of a process behavior expression corresponding to the initial state of the LTS. In the following, we consider an LTS and the corresponding state transition diagram are the same.

If some occurrence of process variable $x$ in process behavior expression $P$ is within $P'$ of subprocess behavior expression rec $x.P'$ of $P$, then this $x$ is called **bound**; otherwise called **free**.

If each process variable in $P$ is bound, then $P$ is called **closed**. Next, we introduce the concept of guard.

**Definition 2.4.** Process variable $x$ is **guarded** in process behavior expression $P$ if some free occurrence of $x$ is within some subprocess behavior expression $a;P'(a \in \mathcal{A})$ of $P$. When each process variable in $P$ is guarded, $P$ is also called **guarded**.

If a process behavior expression is unguarded and not closed, then it can not be interpreted from its initial state.

In this paper, therefore, we only deal with closed and guarded process behavior expressions because of a strict interpretation of process behavior expressions. In the following, we call such a process behavior expression a (recursive) **process**.

The equivalence between processes in LOTOS is based on observations. We define strong equivalence which is one of the equivalences based on observations [9].

**Definition 2.5.** A relation $\equiv$ over processes is a strong bisimulation if $P \equiv Q$ implies, for all $a \in \mathcal{A}$,

1. whenever $P \xrightarrow{a} P'$ then, for some $Q'$, $Q \xrightarrow{a} Q'$ and $P' \equiv Q'$
2. whenever $Q \xrightarrow{a} Q'$ then, for some $P'$, $P \xrightarrow{a} P'$ and $P' \equiv Q'$.

For processes $P$ and $Q$, if there exists a strong bisimulation $\equiv$ such that $P \equiv Q$, then $P$ and $Q$ are strongly equivalent, written as $P \sim Q$. Obviously, $\sim$ is an equivalence relation and the largest strong bisimulation [9].

The strong equivalence does not consider internal actions which cannot be observed from the environment. Since this paper deals only with observable actions, the strong equivalence is enough for our purpose.

### 2.2 Simplest LTS

Processes which are strongly equivalent with some process $P$ and whose description forms are different from $P$ generally exist infinitely. This fact causes a bad effect when pointing out error location and error classification
by comparing the correct answer process with the learner’s answer process which includes errors. In this subsection, we introduce an algorithm that can derive the unique process whose LTS is the simplest among processes which are strongly equivalent with \( P \).

First, we define a minimal \( \text{LTS} \). In the following, the \( \text{LTS} \) \( Sys = \langle S, \mathcal{A}, T, s_0 \rangle \) corresponding to a process \( P \) means

\[
\begin{align*}
S &= \{ P \uparrow P'(\rightarrow)^n P', \ n \geq 0 \}, \\
\mathcal{A} &= \{ a | P \xrightarrow{\mathcal{A}} P', P, P' \in S \}, \\
T &= \{ \xrightarrow{\mathcal{A}} | P \xrightarrow{\mathcal{A}} P', P, P' \in S \}, \\
s_0 &= P.
\end{align*}
\]

Here, \( P(\rightarrow)^nP' \) means that \( P \xrightarrow{\mathcal{A}^0} P_1 \xrightarrow{\mathcal{A}^1} \cdots \xrightarrow{\mathcal{A}^{n-1}} P_{n-1} \xrightarrow{\mathcal{A}^n} P' \) for some processes \( P_0, \ldots, P_{n-1} \) and actions \( \mathcal{A}_0, \ldots, \mathcal{A}_{n-1}, \mathcal{A}_n \). For notation, \( \mathcal{A}^+ \) represents the whole action sequences of \( \mathcal{A} \) whose length is more than or equal to one, and \( \mathcal{A}^\ast \) represents the whole action sequences of \( \mathcal{A} \) whose length is more than or equal to zero. Let \( P \) and \( P' \) be processes or states, then \( P \xrightarrow{\mathcal{A}^i} P' \) denotes a transition from \( P \) to \( P' \) by an action sequence \( s(s \in \mathcal{A}^+) \). \( P(\rightarrow)^nP' \) and \( P(\rightarrow)^\ast P' \) denote a transition from \( P \) to \( P' \) by zero or more actions and by one or more actions, respectively.

The minimal LTS is defined as follows [7]:

**Definition 2.6.** Let \( P \) and \( Sys \) be a process and its LTS, respectively. Among the LTSs of the processes which are strongly equivalent with \( P \), an LTS with the smallest number of states and transitions is called the minimal LTS of \( Sys \). The process corresponding to the minimal LTS of \( Sys \) is called the minimal form of \( P \). □

Suppose that a process \( P \) corresponds to an LTS \( Sys \). A process corresponds to some state of \( Sys \) if the process is a subprocess of \( P \). If two processes which correspond to states \( s \) and \( s' \), respectively, are strongly equivalent, then \( s \) and \( s' \) are also called strongly equivalent, written as \( s \sim s' \).

For a minimal LTS, the following proposition holds [7]:

**Proposition 2.1.** For any minimal LTS \( Sys \), any different two states of \( Sys \) are not strongly equivalent mutually. □

See [7] for an algorithm which obtains the minimal LTS from a finite LTS \( Sys = \langle S, \mathcal{A}, T, s_0 \rangle \). This algorithm satisfies the following:

**Proposition 2.2.** An arbitrary finite LTS can be translated into the minimal LTS by the algorithm of [7]. □

Figure 1 shows a translated example by [7]’s algorithm. An error detection method proposed in the next section is applied only to an LTS which becomes a tree structure by removing transitions forming recursion (loops). Such an LTS is called a simplest LTS. We present an algorithm to translate a minimal LTS into the simplest LTS.

**Definition 2.7 (recursive transition).** Let \( Sys = \langle S, \mathcal{A}, T, s_0 \rangle \) be an arbitrary LTS. Then, if there exist transitions such that \( s_0(\rightarrow)^n s_{i-1} s_i \xrightarrow{\mathcal{A}^i} s_i \) for some states \( s_0, s_i \in S \), then \( (s, a, s') \in T \) is called a recursive transition of \( Sys \) where \( s_i = s, s_i = s' \) and \( 0 \leq i \leq n \). □

**Definition 2.8 (state level).** In an LTS \( Sys = \langle S, \mathcal{A}, T, s_0 \rangle \), the state level of a state \( s (s \in S) \) is the maximum transition number which can transit from the initial state \( s_0 \) to \( s \) without traversing the same states (including \( s \)). It is written as \( \text{level}(s) \) for the state level of \( s \). We assume \( \text{level}(s_0) = 0 \). □

**Definition 2.9 (ancestor/descendant).** In an LTS \( Sys = \langle S, \mathcal{A}, T, s_0 \rangle \), if \( s(\rightarrow)^n s' \) for \( s, s' \in S \), then \( s \) is called an ancestor of \( s' \), and \( s' \) is called a descendant of \( s \). □

**Definition 2.10 (simplest LTS).** In a LTS \( Sys' = \langle S, \mathcal{A}, T, s_0 \rangle \) which is strongly equivalent with a minimal LTS \( Sys \), \( Sys' \) is called the simplest if the recursive transitions included in \( T \) are the same as \( Sys \), the LTS ob-

![Original LTS and Minimal LTS](image)
An algorithm for translating a minimal LTS to the simplest one is presented as follows:

**Algorithm 1.** Let Sys = \( <S, \mathcal{A}, T, s_0> \) be a minimal LTS.

while \( \exists s, s_1, s_2 \in S. a, b \in \mathcal{A}, s_1 \rightarrow \rightarrow s \in T (\text{level}(s_1) < \text{level}(s)), s_2 \rightarrow s \in T (\text{level}(s_2) < \text{level}(s)) \) do

for \( s' \) such that \( s' \neq S \), do the following rewrite:

\[ S = S \cup \{ s' \}; \]

\[ T = (T - \{ s_2 \rightarrow s \}) \cup \{ s_2 \rightarrow s' \} \] (here, we call \( s \) and \( s' \) split states.);

if \( \exists c \in \mathcal{A}. \exists s'' \in S. s \rightarrow s'' \in T \) then

For each \( s'' \), do the following:

\[ T = T \cup \{ s'' \rightarrow s'' \}; \]

however, if \( s'' = s \) then let \( T = T \cup \{ s'' \rightarrow s' \} \) and do not do the following.

if \( \text{level}(s'') < \text{level}(s') \lor \text{level}(s'') < \text{level}(s) \) then

if \( s'' \) is not the ancestor of \( s' \) then

\[ T = T - \{ s'' \rightarrow s'' \}; \]

copy the subgraph which starts from \( s'' \), to the next place (state) to be transited by \( s' \rightarrow \rightarrow \); replace each state in the copied subgraph

translate the subgraph that starts from the split state \( q \) whose level is the smallest and such that \( q(\rightarrow) \) into the minimal LTS;

e else

if \( s'' \) is not the ancestor of \( s \) then

\[ T = T - \{ s \rightarrow s'' \}; \]

copy the subgraph which starts from \( s'' \), to the next place (state) to be transited by \( s \rightarrow \rightarrow \); replace each state in the copied subgraph

translate the subgraph that starts from the split state \( q \) whose level is the smallest and such that \( q(\rightarrow) \) into the minimal LTS;

e end if

e endif

e endif

e endif

end while.

Figure 2 shows a translation example using **Algorithm 1**.

For **Algorithm 1**, the following proposition holds. This proposition is a simple extension of the result obtained in our previous work. See [10].

**Proposition 2.3.** Let Sys be a minimal LTS, Sys' = \( <S, \mathcal{A}, T, s_0> \) be the LTS obtained by **Algorithm 1**, and \( P \) and \( P' \) be the corresponding processes, respectively. Then:

1. \( P \sim P' \)

2. Let \( Z \) be the set which includes all tree structure LTSs obtained by removing recursive transitions from the LTSs corresponding to processes that are strongly equivalent with \( P \). Then, Sys' belongs to \( Z \), and the number of states and transitions of Sys' is the minimum among LTSs in \( Z \).

From the definition of the simplest LTS, the following proposition holds:

**Proposition 2.4.** Let Sys = \( <S, \mathcal{A}, T, s_0> \) be a simplest LTS. Then:

\[ \forall s, s', s'' \in S. s \rightarrow s', s \rightarrow s'', s' \neq s'' \text{ implies } s' \not\sim s''. \]
3 Error Detection Method for Recursive Processes

A special feature of a recursive process is that it can execute events infinitely. For verification of recursive processes, a simple comparison between two recursive processes may incur the non-termination of the comparison algorithm. Although recursive processes have transitions which can occur infinitely, they have finite number of states, actions and transitions. By aiming at this point, if we compare two systems by using simplest LTSs, then we can avoid the non-termination problem of the algorithm. Another problem is that processes strongly equivalent with each other generally exist infinitely. This is also difficult to verify equivalence because of comparison of multi-action between two systems. But this problem can also be solved by using a simplest LTS because it has only one form except commutative one based on strong equivalence.

In this paper, we classify errors to be detected and point out what kind of errors learner's make.

3.1 Definitions for Error Detection and Classification

We prepare several definitions for the error detection and classification. An error detection is done by comparing multi-actions at each level of each process.

Definition 3.1. $\text{succ}(P)$, which is a set of successors of process $P$, is a multi-set of actions to be able to make a non-recursive transition from $P$. That is:

$$\text{succ}(P) = \{a|\exists P'. P \xrightarrow{a} P', P \xrightarrow{a} P' \text{ is a non-recursive transition}\}$$

If $n$ occurrences ($n \geq 2$) of the identical action are included in $\text{succ}(P)$, then it indicates possible transitions to $n$ kinds of subprocesses from $P$ by the action.

Definition 3.2. $\text{recur\_succ}(P)$, which is a set of successors of process $P$, is a multi-set of actions to be able to make a recursive transition from $P$. That is:

$$\text{recur\_succ}(P) = \{a|\exists P'. P \xrightarrow{a} P', P \xrightarrow{a} P' \text{ is a recursive transition}\}$$

If $n$ occurrences ($n \geq 2$) of the identical action are included in $\text{recur\_succ}(P)$, then it indicates possible transitions to $n$ kinds of subprocesses from $P$ by the action.

Definition 3.3. The action set $\text{action\_set}_k(P)$ of level $k$ ($k \geq 0$) for process $P$ is defined as follows:

$$\text{action\_set}_k(P) = \bigcup_{\text{level}(Q)=k} \text{succ}(Q) \cdot (P(\rightarrow)^k Q)$$

where $\bigcup_M$ means a union of multi-sets.

An action of $\text{action\_set}_k(P)$ is called a non-recursive action.

Definition 3.4. The action set $\text{recur\_action\_set}_k(P)$ of level $k$ ($k \geq 0$) for process $P$ is defined as follows:

$$\text{recur\_action\_set}_k(P) = \bigcup_{\text{level}(Q)=k} \text{recur\_succ}(Q) \cdot (P(\rightarrow)^k Q)$$

An action of $\text{recur\_action\_set}_k(P)$ is called a recursive action.

Figure 3 shows an example of $\text{action\_set}_k(P)$ and $\text{recur\_action\_set}_k(P)$.

In the rest of the paper, we assume that $P$ is a correct answer process and $Q$ is a learner's answer process. When $\text{action\_set}_k(P) \neq \text{action\_set}_k(Q)$ or $\text{recur\_action\_set}_k(P) \neq \text{recur\_action\_set}_k(Q)$ for some $k$ ($k \geq 0$), there are excess or shortage errors between $P$ and $Q$. In the following, these errors are defined.

Definition 3.5 (excess error). There exists an excess error if at some level $k$, an action which is not included in the action set of $P$ is included in the action set of $Q$. That is:

$$\text{action\_set}_k(Q) - M \text{action\_set}_k(P) \neq \emptyset \lor \text{recur\_action\_set}_k(Q) - M \text{recur\_action\_set}_k(P) \neq \emptyset$$

then there exists an excess error, where $- M$ means the difference of multi-sets.

Figure 4 shows an example of excess error.

```
# Example of action sets

action_set_0(P) = \{a, b, c, d\}

action_set_1(P) = \{a, d\}

recur_action_set_1(P) = \{a, b\}

recur_action_set_2(P) = \{b\}
```

Fig. 3 Non-recursive and recursive action sets.
**Definition 3.6 (shortage error).** There exists a shortage error if at some level $k$, an action which is included in the action set of $P$ is not included in the action set of $Q$. That is:

$$\text{action}_s(P) -_M \text{action}_s(Q) \neq \emptyset \lor \text{recur}_s(P) -_M \text{recur}_s(Q) \neq \emptyset$$

then there exists a shortage error.

Figure 5 shows an example of shortage error.

However, there exists an error which cannot be detected by difference of $\text{recur}_s$ of $P$ and $Q$ at the same level. This type of errors can occur by the difference of the levels of transition destinations. This error is called a recursive precedence error and defined as follows:

**Definition 3.7 (recursive precedence error).** At some level $k$, $a$ of a recursive transition $Q' \xrightarrow{a} Q''(Q(\rightarrow)^*Q')$ is a recursive precedence error if the following condition is satisfied:

$$\text{level}(Q') \neq \{\text{level}(P') | \exists P'. P(\rightarrow)^*P' \xrightarrow{a} P'' \text{, } P' \xrightarrow{a} P'' \text{ is a recursive transition and } \text{level}(P') = k\}$$

Figure 6 shows an example of recursive precedence error.

Even though the action sets of $P$ and $Q$ are equal in all levels, they may not be strongly equivalent. For example, there exists a sequence error shown in Figure 7.

In the following, we give some definitions for a sequence error. To investigate the sequence relation between $n+1$ ($n \geq 1$) consecutive levels, $n_{\text{Trace}_s}$ and $n_{\text{Trace}_s}$ are defined below:

**Definition 3.8 ($n_{\text{Trace}_s}$ and $n_{\text{Trace}_s}$).** For a process $R$ and an integer $k$ ($k \geq 1$), an $n_{\text{Trace}_s}(R)$ ($1 \leq n \leq k$) is an action sequence $sa$ satisfying the following condition:

$$\exists R', \exists R'', \exists R'''. R(\rightarrow)^*R' \xrightarrow{s} R'' \xrightarrow{s} R'''$$

$s$ is a sequence of non-recursive actions of length $n$,

$a$ is an action, and

$\text{level}(R'') = k$.

We define $n_{\text{Trace}_s}(R)$ as a multi-set of $n_{\text{Trace}_s}(R)$.

Figure 8 shows an example of $1_{\text{Trace}_s}(P)$.

Though a sequence error is checked by comparing $n_{\text{Trace}_s}$ of $P$ and $Q$ at all levels, this is not sufficient because of non-determinism of processes. For example, in Figure 9, though $n_{\text{Trace}_s}(P) =_M n_{\text{Trace}_s}(Q)$ for each $n$ and $k$ ($1 \leq n \leq k$), $P \neq Q$. Here, $=_M$ stands for $=$ of multi-sets.

To supplement it, we introduce $m_{\text{Ans}_s \text{St}_s}$ and $m_{\text{Ans}_s \text{St}_s}$.  

---

**Figures:**

- Fig. 4 Example of excess error.
- Fig. 5 Example of shortage error.
- Fig. 6 Example of recursive precedence error.
- Fig. 7 Typical sequence error.
**Definition 3.9** (\(m\_\text{Ans}_k\text{St}_h\) and \(m\_\text{Ans}_k\text{St}_k\)). For a process \(R\) and integers \(m, k\) \((1 \leq m \leq k)\), an \(m\_\text{Ans}_k\text{St}_h(R)\) is \(R''\) satisfying the following condition:

\[\exists R', \exists R^*, \exists a.R(\rightarrow)^+R' \rightarrow R^*, \text{level}(R') = k, R'' \rightarrow R',\]

and \(s\) is a non-recursive action sequence of length \(m\).

We define \(m\_\text{Ans}_k\text{St}_k(R)\) as a multi-set of \(m\_\text{Ans}_k\text{St}_h(R)\).

As an example, in Figure 10, \(1\_\text{Ans}_k\text{St}_k(P) = \{P_1, P_2\}\).

**Definition 3.10** (sequence error). For all level of \(P\) and \(Q\), suppose that the both action sets are equal. Then, there exists a sequence error if \(n\_\text{Trace}_k(P) \neq m\_\text{Trace}_k(Q)\) for some integers \(k, n\) \((1 \leq n \leq k)\) or \(m\_\text{Ans}_k\text{St}_k(P) \neq m\_\text{Ans}_k\text{St}_k(Q)\) for some integers \(m, k\) \((1 \leq m \leq k)\). Here, \(Z = Z'\) means that there exists a bijection function \(f:Z \rightarrow Z'\) over multi-sets such that \(z \sim f(z)\) for all \(z \in Z\).

**3.2 Algorithm for Error Detection**

In this subsection, we propose a new algorithm for error detection and equivalence verification, based on the concept of strong equivalence between two recursive processes (i.e. the correct answer process \(P\) and the learner’s answer process \(Q\)).

For an education support for LOTOS, error detection in the learner’s answer is realized by comparing the simplest LTSSs of \(P\) and \(Q\), and by seeking four kinds of errors (i.e. excess error, shortage error, recursive precedence error and sequence error) in the learner’s answer. An excess error and a shortage error can be detected by comparing the multi-action sets at each level. Next, a recursive precedence error is detected if the levels of the transition destinations of \(P\) and \(Q\) are different for a recursive action which is not excess error. A sequence error is detected by comparison of subprocesses and traces for a non-recursive action which is not excess error, or for a recursive action which is neither excess error nor recursive precedence error. If no errors are detected in \(Q\), then \(P\) and \(Q\) are strongly equivalent and the learner’s answer is perfect.

**Algorithm 2.**

Let \(P\) be the simplest process of the correct answer, \(Q\) be the simplest process of learner’s answer, \(\text{seq.err.chk}\) be an array of action sets of undecided sequence errors, and \(\text{rec.pre.err.chk}\) be an array of action sets of undecided recursive precedence errors.

For the fixed \(n\_\text{Trace}_k(Q)\), let \(n\_\text{Trace}_k(Q)\) be the \(s_{ij}\).
begin
  \( k := 0; n := 1; j := 0; r := 0; \)
  \( \text{while} (\text{action}_s(P) \neq \emptyset \lor \text{action}_s(Q) \neq \emptyset \lor \text{recur}_s(P) \neq \emptyset \lor \text{recur}_s(Q) \neq \emptyset) \text{ do} \)
  \( \text{if} (\text{action}_s(Q) - \text{action}_s(P)) \neq \emptyset \text{ then} \)
  \( \text{excess}_e := \text{action}_s(Q) - \text{action}_s(P); \) (*excess errors of non-recursive actions*)
  \( \text{seq}_e := \text{action}_s(Q) - \text{excess}_e; \)
  \( \text{else} \)
  \( \text{seq}_e := \text{action}_s(Q); \)
  \( \text{end if} \)
  \( \text{if} (\text{recur}_s(Q) - \text{recur}_s(P)) \neq \emptyset \text{ then} \)
  \( \text{recursive}_e := \text{recur}_s(Q) - \text{recur}_s(P); \) (*excess errors of recursive actions*)
  \( \text{rec}_e := \text{recur}_s(Q) - \text{recursive}_e; \)
  \( \text{else} \)
  \( \text{rec}_e := \text{recur}_s(Q); \)
  \( \text{end if} \)
  \( \text{if} (\text{action}_s(P) - \text{action}_s(Q)) \neq \emptyset \text{ then} \)
  \( \text{shortage}_e := \text{action}_s(P) - \text{action}_s(Q); \) (*shortage errors of non-recursive actions*)
  \( \text{end if} \)
  \( \text{if} (\text{recur}_s(Q) - \text{recur}_s(P)) \neq \emptyset \text{ then} \)
  \( \text{recursive}_e := \text{recur}_s(Q) - \text{recur}_s(P); \) (*shortage errors of recursive actions*)
  \( \text{end if} \)
  \( \text{while} (\text{rec}_e \neq \emptyset) \text{ do} \)
  \( \text{if} \left( \{\text{level}(P') | P' \in \text{level}_s(P)\} = \{\text{level}(Q') | Q' \in \text{level}_s(Q)\} \right) \text{ then} \)
  \( \text{seq}_e := \text{rec}_e; \)
  \( \text{else} \)
  \( \text{recursive}_e := \text{rec}_e; \) (*recursive precedence errors*)
  \( \text{end if} \)
  \( r := r + 1; \)
  \( \text{end while} \)
  \( \text{if} (k = 0) \text{ then} \)
  \( \text{correct}_e := \text{seq}_e; \)
  \( \text{seq}_e := \emptyset; \)
  \( \text{end if} \)
  \( \text{while} (\text{seq}_e \neq \emptyset) \text{ do} \)
  \( \text{if} (\text{n}_s(P) \neq \text{n}_s(P)) \text{ then} \)
  \( \text{sequence}_e := \text{seq}_e; \) (*sequence errors*)
  \( \text{else} \left( 1 = \left[ \{s \in \text{n}_s(P) \} \land 1 = \left[ \{s \in \text{n}_s(Q) \} \right] \right) \text{ then} \)
  \( \text{correct}_e := \text{seq}_e; \)
  \( \text{else} \left( 2 \leq \left[ \{s \in \text{n}_s(P) \} \lor 2 \leq \left[ \{s \in \text{n}_s(Q) \} \right] \right) \right) \)
  \( m := n; \)
  \( \text{if} (\text{n}_s(P) \neq \text{n}_s(P)) \text{ then} \)
  \( \text{sequence}_e := \text{seq}_e; \) (*sequence errors*)
  \( \text{else} \left( 1 = \left[ \{\text{m}_s(P) \} \land 1 = \left[ \{\text{m}_s(Q) \} \right] \right) \text{ then} \)
  \( \text{correct}_e := \text{seq}_e; \)
  \( \text{else} \left( 2 \leq \left[ \{\text{m}_s(P) \} \lor 2 \leq \left[ \{\text{m}_s(Q) \} \right] \right) \right) \)
  \( n := n + 1; \)
  \( \text{end if} \)
  \( \text{end if} \)
  \( j := j + 1; \)
  \( \text{end while} \)
  \( \text{all arrays} := \emptyset; \)
  \( k := k + 1; \)
  \( \text{end while} \)
end.

By a simple extension of the result obtained in our previous work [10], the following proposition holds for this algorithm:

**Proposition 3.1.** Let \( P \) be the correct answer process and \( Q \) be a learner’s answer process. Then, Algorithm 2 detects no errors if and only if \( P \) and \( Q \) are strongly equivalent. \( \square \)
We introduce a simple example for comprehension of the algorithm mentioned above. The example we use is shown in Figure 11.

For $k = 0$, $\text{action} \_ \text{set}_0(P) = \{a, c\}$ and $\text{action} \_ \text{set}_0(Q) = \{a, c, e\}$. Therefore, $\text{action} \_ \text{set}_0(P) \neq \emptyset \lor \text{action} \_ \text{set}_0(Q) \neq \emptyset$ is true. Then, do the following things: First, check an excess error. For $\text{action} \_ \text{set}_0(Q) - M \text{action} \_ \text{set}_0(P) = \{e\}$, $\{e\}$ is assigned to $\text{excess} \_ \text{error}$. That is, $e$ is detected as an excess error. For actions which are not excess errors, put them to the array for detecting a sequence error (i.e. $\text{seq} \_ \text{err} \_ \text{chk}_k \leftarrow \{a, c\}$). Next, check a shortage error. For $\text{action} \_ \text{set}_0(P) - M \text{action} \_ \text{set}_0(Q) = \emptyset$, there is no shortage error. For $\text{seq} \_ \text{err} \_ \text{chk}_k$, because $k = 0$ (initial level), $\{a, c\}$ are the correct actions. Because $\text{seq} \_ \text{err} \_ \text{chk}_k (j = 0)$ becomes $\emptyset$, the sequence error check part is skipped. All arrays are emptied and $k$ becomes 1.

For $k = 1$, $\text{action} \_ \text{set}_1(P) = \{d\}$ and $\text{action} \_ \text{set}_1(Q) = \{d\}$. Using the same methods mentioned above, there are no excess error and shortage error. $\text{action} \_ \text{set}_1(Q) (\{d\})$ is assigned to $\text{seq} \_ \text{err} \_ \text{chk}_0$. For recursive action, $\text{recur} \_ \text{action} \_ \text{set}_1(P) = \{b\}$ and $\text{recur} \_ \text{action} \_ \text{set}_1(Q) = \{b\}$. Using the same methods mentioned above, there are no excess error and shortage error also. Next, for $b \in \text{recur} \_ \text{action} \_ \text{set}_1(Q) \land b \notin \text{excess} \_ \text{error}$, put them to the array for detecting a recursive precedence error (i.e. $\text{rec} \_ \text{pre} \_ \text{err} \_ \text{chk}, \leftarrow \{b\} (r = 0)$). Because $\text{rec} \_ \text{pre} \_ \text{err} \_ \text{chk}_0 \neq \emptyset$, do the following things: For $\{\text{level} (P)\} \neq M \{\text{level} (Q)\}$ (i.e. $\{1\} \neq M \{0\}$), $\text{recursive} \_ \text{precedence} \_ \text{error} \leftarrow \text{rec} \_ \text{pre} \_ \text{err} \_ \text{chk}_0 (\{b\})$. That is, a recursive action $b$ of $Q$ is detected as a recursive precedence error. For $\text{seq} \_ \text{err} \_ \text{chk}_0 = \{d\}$, check the following things: Because $1 = |\{cd\}| \land (\{cd\} = \{s_1 a_r \mid s_1 a_r \in \text{Trace} \_ \text{set}_1(P))$ and $1 = |\{e\}| \land (\{e\} = \{s_1 \mid s_1 b \mid s_1 b \in \text{Trace} \_ \text{set}_1(P))$, $d$ is a correct action. All arrays are emptied and $k$ becomes 2.

Because $\text{action} \_ \text{set}_2(P) = \emptyset$, $\text{action} \_ \text{set}_2(Q) = \emptyset$, $\text{recur} \_ \text{action} \_ \text{set}_2(P) = \emptyset$ and $\text{recur} \_ \text{action} \_ \text{set}_2(Q) = \emptyset$, the algorithm is terminated.

4 Education Support System

In this section, we show an education support system based on the error detection algorithm mentioned above.

4.1 Outline

This education support system is utilized as follows: First, to improve the knowledge of LOTOS for novice users, the correct answer is entered (stored) to the system. The learner selects an education course which is suitable for his ability and then solves the LOTOS description problem which is given by the system. The system checks the syntax and semantics for the learner’s answer by LOTOS analyzer. After completing above things, the learner’s answer is converted into an LTS and changed to a simplest LTS by Algorithm 1. Then, the simplest LTS is compared with the simplest LTS of the correct answer by Algorithm 2. If no errors are detected, the learner’s answer is perfect. Otherwise, the detected errors are marked and displayed on the LTSS graphically. The learner can know where his errors exist and what kind of errors he made. By repeating these processes, he will gradually acquire knowledge of LOTOS.

4.2 Development of the System

In this subsection, we discuss the software development of the LOTOS education support system. It has been developed with:

- CPU: Pentium 200 MHz
- operating system: Microsoft Windows NT
- development language: Visual C++ 5.0

The system has the software structure as shown in Figure 12. A rough description of each software module is as follows:

- GUI: It provides a standard interface via the keyboard, mouse and display. The learner can select the prob-
lem level (i.e. beginner, intermediate and advanced) by the buttons at the top-left corner on the initial screen (see Figure 13), and then the system gives a problem (see the upper part of Figure 14). This system has an editor as shown in the lower part of Figure 14. Therefore, it provides the learner with an interactive interface.

- LOTOS Analysis Part: It analyzes a LOTOS description (process) by using the LOTOS Specification Libraries, parses the syntax, and checks the semantics.
- LTS Convert Part: It converts the LOTOS descriptions of the learner’s answer and correct answer into LTSs, and changes them to the simplest LTSs by Algorithm 1.
- Verification and Error Detection Part: It detects various errors in the Error Type Libraries by using Algorithm 2, and marks them on the LTSs.
- LOTOS Graphical Display Part: It graphically displays the LTSs of the correct answer and learner’s answer with the detected errors if any (see Figure 15).
4.3 Example

Let us use the education support system to learn LOTOS and to detect errors using a simple example. The example we use here is the description of a vending machine which is frequently used to explain LOTOS behavior expressions of a system.

Figure 13 shows the initial screen for LOTOS education. If the learner selects the beginner course by pressing the button at the top-left corner on the initial screen, the system presents the description of the behavior of vending machine in natural language. This problem is displayed in the Question box shown at the upper part of Figure 14. Note that in the Question box of this figure, although LOTOS identifiers should start with an alphabetical letter, but to make it more natural, here we have used Arabic numerals for the gate identifiers of coin. The learner describes his answer in LOTOS into the Answer box shown at the lower part of the figure.

Then the system converts the LOTOS description into an LTS, and changes it to a simplest LTS. After comparing it with the simplest LTS of the correct answer, if any errors are detected, the system displays the posi-
tions of errors on the LTSs in colors (see Figure 15). Note that in this figure, instead of using colors, we use a box to represent a shortage error and an underline to represent other errors.

5 Conclusion

In this paper, we have presented a new algorithm which can detect learner’s errors and verify equivalence, based on the concept of strong equivalence, between the learner’s answer process and the correct answer process. For LOTOS instruction, it is important not only to verify equivalence but also to detect error types and positions. In general, if a process includes recursion, this type of algorithm suffers from two kinds of problems: first one is a termination problem of the algorithm and second one is infinite existence of LTSs equivalent with each other. This paper has solved these problems by introducing the simplest LTS which has minimum number of states and transitions among LTSs equivalent with each other.

Our algorithm compares the multi-sets of actions at each level between two simplest LTSs corresponding to the correct answer process and the learner’s answer process. If an error is detected, it is classified depending on its type. If no errors are detected, the two processes can be said to be strongly equivalent and the learner’s answer is perfect.

Furthermore, we have developed a education support system for beginners of LOTOS based on the new algorithm. If an error is detected, its type and position are displayed to the learner in a comprehensive manner. The learner can gradually learn LOTOS by using the system.

Future study includes:
• an improvement and enrichment of the education support system (e.g. improvement of GUI and enrichment of problems),
• an extensive application and evaluation of the education support system, and
• an extension of the algorithm to full LOTOS.

Acknowledgement

We would like to thank Dr. S. Kimura, Tsukuba University, for his helpful discussions and comments for this study. We are also grateful to Dr. B. B. Bista, Iwate Prefectural University, for his support for improving the paper.

REFERENCES