# Implementation of Natural Language Specifications of Communication Protocols by Executable Specifications

Yasunori ISHIHARA, Hiroyuki SEKI, and Tadao KASAMI

Graduate School of Information Science

Nara Institute of Science and Technology

8916-5, Takayama, Ikoma, Nara 630-01 JAPAN

E-mail: ishihara@is.aist-nara.ac.jp

**ABSTRACT**

First, this paper defines a subclass of algebraic specifications. Each specification of the subclass consists of two sub-specifications: a BE program and a BE interpreter specification. The syntax of BE programs resembles the syntax of LOTOS, and the semantics of BE programs is defined as a behavior of an interpreter, called a BE interpreter, which has a finite number of registers and unbounded I/O buffers. Since BE interpreter specifications are based on a state transition model, each specification of the subclass can be easily compiled into an executable program. Next, the paper proposes a method of implementing logical formulae, which are derived from natural language specifications of communication protocols, by BE programs. Such a natural language specification often specifies valid sequences of actions to be performed by a protocol machine. In this implementation method, the meaning of each predicate that corresponds to a word denoting actions is defined as a BE program and stored as a "lexical item" of the predicate. Then, a BE program for logical formulae is constructed in a bottom-up manner. Thus, a natural language specification of communication protocols can be translated into an executable program in the framework of algebraic specifications.

## 1. Introduction

In a software development process, informal requirements and/or specifications are often written in a natural language since they are readable and the intuitive meanings are understandable. However, it is desirable that such an informal specification is translated into a formal specification so that one can analyze the informal specification, reduce its ambiguity, and derive an efficient program which satisfies the specification. Among various formal specification methods which have been proposed and studied, algebraic specification methods [2] are useful and powerful because of the following reasons:

1. Abstract data types can be defined simply in algebraic specifications;

2. Formal semantics of a specification is simply provided by axioms (equations); and

3. One can write a specification which has arbitrary structure and arbitrary degree of abstraction.

In Refs. [3] and [9], we proposed a translation method from natural language specifications of communication protocols into algebraic specifications. A specification such as a protocol specification defines valid sequences of actions performed by a protocol machine. In the method, the valid sequences of actions are represented by an axiom in the form of a logical formula. However, such an axiom is too abstract to be compiled into an executable program directly.

First, this paper defines a class of interpreters (machines), called BE interpreters, as a model of protocol machines. A BE interpreter has a finite number of registers and unbounded I/O buffers, and performs three kinds of atomic actions: input from a buffer, output to a buffer, and calculation using its registers. An input program for a BE interpreter, called a BE program, specifies the order of actions by means of such operators as action-prefix, choice, conditional, and so on. The syntax of BE programs is also defined within the framework of algebraic specifications. The semantics of BE programs, i.e., the behavior of BE interpreters, is defined by axioms based on a state transition model. Therefore, a BE interpreter specification with a given BE program can be easily compiled into an executable program.

As stated above, we have already proposed a translation method from natural language specifications of communication protocols into logical formulae. In this paper, we also propose a method of implementing such logical formulae by BE programs. Each word which denotes actions in a natural language specification is translated into a predicate, and the meaning of such a predicate is given as a BE program by a human implementor. Based on such BE programs, a BE program for logical formulae is constructed in a bottom-up manner.

The syntax of BE programs is modeled on the syntax of LOTOS [6]. There are several reasons why we do not implement the logical formulae by LOTOS. First of all, it is desirable that the whole translation from natural language specifications into executable programs is handled in the same framework. By defining BE programs in the framework of algebraic specification methods, the whole translation becomes simple and concise. Secondly, the model of processes (machines) in LOTOS is too primitive for our purpose. For example, natural language specifications of communication protocols (e.g., Ref. [5]) often presuppose that protocol machines have registers. However, in LOTOS, the model of processes itself does not have the concept of registers. Therefore, we design BE interpreters so that they have registers. There is another reason that we introduce registers into BE interpreters. It is probable that registers (or substitutes for them) are needed to implement rendezvous (synchronous communication) of LOTOS. For example, to execute a LOTOS specification, Ref. [1] implements rendezvous by using a shared memory, and Ref. [7] does by assignment of values to registers.

## 2. Algebraic Specification Language ASL

In this paper, we adopt ASL [8] as an algebraic specification language. A specification in ASL is a pair *SPEC* = $(G, AX)$ of a context-free grammar $G$ and a set $AX$ of axioms. $G$ specifies the set of expressions and their syntax, and $AX$ specifies their semantics. Let $G = (N, T, P)$ where $N$, $T$ and $P$ are sets of nonterminals, terminals and productions, respectively. For a nonterminal $D \in N$, let $L_G[D]$ denote the set of terminal strings derived from $D$ in $G$, and let $L_G = \bigcup_{D \in N} L_G[D]$. An element in $L_G$ is called an expression (in the specification *SPEC*). $N$ corresponds to the set of sorts (data types); A nonterminal $D$ is sometimes called “data type $D$” and an expression in $L_G[D]$ may be called “an expression of type $D$.”

An axiom is a pair $l == r$ of expressions with variables. A variable of an axiom is denoted by a symbol with the upper bar (e.g., $\bar{x}$). With each variable $\bar{x}$ in an axiom a nonterminal $D_{\bar{x}}$ is associated (declared by “$\bar{x} : D_{\bar{x}}$” in the specification), and an arbitrary expression in $L_G[D_{\bar{x}}]$ can be substituted into $\bar{x}$. The least congruence relation that satisfies all the axioms in $AX$ is denoted by $\equiv_{SPEC}$. See Ref. [8] for details.

Table 1 Specification of sequences.

- Production schemata:

$$\begin{aligned}
\mathsf{Seq\_}D &\rightarrow \lambda, \\
\mathsf{Seq\_}D &\rightarrow \mathsf{Seq\_}D \cdot D, \\
D &\rightarrow \mathsf{head}(\mathsf{Seq\_}D), \\
\mathsf{Seq\_}D &\rightarrow \mathsf{tail}(\mathsf{Seq\_}D), \\
\mathsf{Bool} &\rightarrow \mathsf{member}(D, \mathsf{Seq\_}D).
\end{aligned}$$

- Axioms:

$$\bar{x}_{\mathsf{seq}} : \mathsf{Seq\_}D,\ \bar{x}, \bar{x}' : D$$

$$\begin{aligned}
\mathsf{head}(\bar{x}_{\mathsf{seq}} \cdot \bar{x}) &== \mathsf{if}\ \bar{x}_{\mathsf{seq}} = \lambda\ \mathsf{then}\ \bar{x}\ \mathsf{else}\ \mathsf{head}(\bar{x}_{\mathsf{seq}}), \\
\mathsf{tail}(\bar{x}_{\mathsf{seq}} \cdot \bar{x}) &== \mathsf{if}\ \bar{x}_{\mathsf{seq}} = \lambda\ \mathsf{then}\ \lambda\ \mathsf{else}\ \mathsf{tail}(\bar{x}_{\mathsf{seq}}) \cdot \bar{x}, \\
\mathsf{member}(\bar{x}, \lambda) &== \mathsf{false}, \\
\mathsf{member}(\bar{x}, \bar{x}_{\mathsf{seq}} \cdot \bar{x}') &== \mathsf{if}\ \bar{x} = \bar{x}'\ \mathsf{then\ true\ else}\ \mathsf{member}(\bar{x}, \bar{x}_{\mathsf{seq}}).
\end{aligned}$$

In this paper, we presuppose a fixed specification $SPEC_0 = (G_0, AX_0)$, $G_0 = (N_0, T_0, P_0)$ of primitive data types (e.g., integer, Boolean, set, and so on), which defines the data types of the contents of the registers and I/O buffers of a BE interpreter. We assume that $SPEC_0$ supports the following data types:

1. Boolean; Let $\mathsf{Bool}$ be a nonterminal which generates Boolean expressions.

2. Sequence; Let $\mathsf{Seq\_}$ be a constructor on data types to support sequences of a given data type, i.e., for any data type $D$, $\mathsf{Seq\_}D$ generates sequences of expressions of type $D$. Formally, $SPEC_0$ has the production schemata and axioms shown in Table 1, where $\lambda, \cdot, \mathsf{head}, \mathsf{tail}, \mathsf{member} \in T_0$. Constant function $\lambda$ denotes the empty sequence and function "$\cdot$" denotes the concatenation operation. For a given sequence, $\mathsf{head}$ returns the first element and $\mathsf{tail}$ returns the sequence obtained by eliminating the first element. Predicate $\mathsf{member}$ is true if and only if the first parameter is an element of the second parameter.

## 3. BE Programs

As stated in section 1, a BE interpreter has registers and I/O buffers, and performs three kinds of atomic actions. The syntax and semantics of a BE program are defined to meet the following requirements:

(a) Any number of registers and I/O buffers can be introduced into a BE interpreter;

(b) Data types of contents of the registers and I/O buffers are pre-defined as primitive data types;

(c) All the registers and I/O buffers of a BE interpreter can be directly accessed by performing (atomic) actions; and

(d) The order of actions can be explicitly specified in a BE program.

This section restates these requirements formally, i.e., describes the conditions which a BE program $SPEC_{\mathrm{PRG}} = (G_{\mathrm{PRG}}, AX_{\mathrm{PRG}})$, $G_{\mathrm{PRG}} = (N_{\mathrm{PRG}}, T_{\mathrm{PRG}}, P_{\mathrm{PRG}})$ has to meet.

First, for the requirement (a), the following two data types, $\mathsf{Reg}$ and $\mathsf{Buf}$, are introduced into $G_{\mathrm{PRG}}$:

1. $\mathsf{Reg} \in N_{\mathrm{PRG}}$ generates names of registers of a BE interpreter.

2. $\mathsf{Buf} \in N_{\mathrm{PRG}}$ generates names of I/O buffers.

Define *REG* and *BUF* as $L_{G_{\mathrm{PRG}}}[\mathsf{Reg}]$ and $L_{G_{\mathrm{PRG}}}[\mathsf{Buf}]$, respectively. To ensure that the number of registers and buffers is finite, we simply assume that each element of $REG \cup BUF$ is a terminal symbol. We also assume that $REG \cap BUF = \emptyset$.

Secondly, for the requirement (b), $SPEC_{\mathrm{PRG}}$ must satisfy the following condition:

3. $SPEC_{\mathrm{PRG}} \supset SPEC_0$ (component-wise containment).

4. For each $reg \in REG$, there is a unique nonterminal symbol $D_{reg} \in N_0$ such that $D_{reg} \to reg \in P_{\mathrm{PRG}}$. $D_{reg}$ is denoted by $type[reg]$.

5. For each $buf \in BUF$, there is a unique nonterminal symbol $D_{buf} \in N_0$ such that $\mathsf{Seq\_}D_{buf} \to buf \in P_{\mathrm{PRG}}$. $D_{buf}$ is denoted by $type[buf]$.

Thirdly, for the requirement (c), the following data type, $\mathsf{Action}$, is introduced:

6. $\mathsf{Action} \in N_{\mathrm{PRG}}$ generates actions. For each $buf \in BUF$ and $reg \in REG$, the following productions are in $P_{\mathrm{PRG}}$:

$$\mathsf{Action} \quad \to \quad \mathsf{in}(buf, reg),$$

$$\begin{aligned} \mathsf{Action} &\rightarrow \mathsf{out}(\mathit{buf}, \mathit{reg}), \\ \mathsf{Action} &\rightarrow \mathsf{set}(\mathit{reg} \leftarrow D_{\mathit{reg}}), \end{aligned}$$

where $\mathsf{in}, \mathsf{out}, \mathsf{set}, \leftarrow \in T_{\text{PRG}}$, *type*[*buf*] = *type*[*reg*], and $D_{reg}$ = *type*[*reg*]. $\mathsf{in}(\mathit{buf}, \mathit{reg})$ denotes that a BE interpreter receives a data from buffer *buf* and the data is stored in register *reg*. $\mathsf{out}(\mathit{buf}, \mathit{reg})$ denotes that a BE interpreter transmits a data stored in register *reg* to buffer *buf*. $\mathsf{set}(\mathit{reg} \leftarrow t)$ denotes an assignment of the value of an expression $t$ to register *reg* (the value of an expression is formally defined in section 4).

Lastly, for the requirement (d), we introduce *behavior expressions*, which specify the order of actions. Some behavior expressions are associated with *behavior identifiers* so that a behavior expression can refer (call) another behavior expression, i.e., a behavior identifier corresponds to a procedure name. The syntax of behavior expressions is defined as follows:

7. $\mathsf{B\_id} \in N_{\text{PRG}}$ generates behavior identifiers. There are productions of the following form:

$$\mathsf{B\_id} \rightarrow \pi,$$

where $\pi \in T_{\text{PRG}}$ is a behavior identifier.

8. $\mathsf{B\_exp} \in N_{\text{PRG}}$ generates behavior expressions. The following productions are in $P_{\text{PRG}}$:

$$\begin{aligned} \mathsf{B\_exp} &\rightarrow \mathsf{stop}, \\ \mathsf{B\_exp} &\rightarrow \mathsf{B\_id}, \\ \mathsf{B\_exp} &\rightarrow \mathsf{Action}; \mathsf{B\_exp}, \\ \mathsf{B\_exp} &\rightarrow (\mathsf{B\_exp} \diamond \mathsf{B\_exp}), \\ \mathsf{B\_exp} &\rightarrow (\mathsf{B\_exp}|\mathsf{Seq\_Action}|\mathsf{B\_exp}), \\ \mathsf{B\_exp} &\rightarrow [\mathsf{Bool}] \mathrel{-\!\!>} \mathsf{B\_exp}, \\ \mathsf{B\_exp} &\rightarrow (\mathsf{B\_exp} \gg \mathsf{Seq\_Action} \gg \mathsf{B\_exp}), \\ \mathsf{B\_exp} &\rightarrow (\mathsf{B\_exp} \,[\!> \mathsf{Seq\_Action} \,[\!> \mathsf{B\_exp}), \end{aligned}$$

where $\mathsf{stop}, ;, \diamond, |, [, ], -\!\!>, (, ), \gg, [\!> \in T_{\text{PRG}}$.

Table 2 Meanings of operators.

- stop means that no actions are performed, i.e., a BE interpreter which executes it goes into a dead state.
- Execution of a behavior identifier $\pi$ is equivalent to execution of the behavior expression which is associated with $\pi$.
- Action-prefix: $a; B$ specifies that a BE interpreter performs action $a$, then executes behavior expression $B$.
- Choice: $(B_1 \Diamond B_2)$ specifies that a BE interpreter executes either $B_1$ or $B_2$ nondeterministically. If a BE interpreter performs an action performable in common with $B_1$ and $B_2$, it is considered that the interpreter is executing "both" of $B_1$ and $B_2$ (see the end of section 4 for detail).
- Parallel composition: $(B_1|\lambda \cdot a_1 \cdots a_n|B_2)$ specifies that a BE interpreter executes behavior expressions $B_1$ and $B_2$ in a "time sharing" manner. Here, each action $a_i$ $(1 \leq i \leq n)$ must be simultaneously performed in the executions of $B_1$ and $B_2$.
- Conditional: $[p]$ –> $B$ specifies that a BE interpreter executes behavior expression $B$ if predicate $p$ holds, and goes into a dead state otherwise.
- Enabling: $(B_1 \gg \lambda \cdot a_1 \cdots a_n \gg B_2)$ specifies that a BE interpreter executes behavior expression $B_1$ first. When some action $a_i$ $(1 \leq i \leq n)$ is performed during the execution of $B_1$, the interpreter begins to execute behavior expression $B_2$.
- Disabling: $(B_1 [> \lambda \cdot a_1 \cdots a_n [> B_2)$ specifies that a BE interpreter executes behavior expression $B_1$, and the interpreter can nondeterministically begin to execute behavior expression $B_2$ until some action $a_i$ $(1 \leq i \leq n)$ is performed during the execution of $B_1$.

Table 2 shows the intuitive meanings of the operators used in behavior expressions. The formal semantics is defined in section 4 as the behavior of a BE interpreter.

Now we introduce a predicate := which associates a behavior expression with a behavior identifier.

9. There is a production

$$\text{Bool} \to \text{B\_id} := \text{B\_exp}$$

in $P_{\text{PRG}}$, and for each $\pi \in L_{G_{\text{PRG}}}[\text{B\_id}]$, there are one or more axioms

$$\pi := B == \text{true}$$

in $AX_{\text{PRG}}$, where $:= \in T_{\text{PRG}}$ and $B \in L_{G_{\text{PRG}}}[\text{B\_exp}]$. $\pi := B \equiv_{SPEC_{\text{PRG}}}$ true means that $\pi$ is defined as $B$ in $SPEC_{\text{PRG}}$. An expression in the form of $\pi := B$ is called a *behavior definition* (of $\pi$).

Among the behavior identifiers defined by operator :=, exactly one behavior identifier must be specified as the main (top level) behavior expression, i.e., the one which should be executed first by a BE interpreter.

10. There is a production

$$\textsf{Bool} \rightarrow \textsf{main}(\textsf{B\_id})$$

in $P_{\mathrm{PRG}}$, and there is exactly one axiom

$$\textsf{main}(\pi) == \textsf{true}$$

in $AX_{\mathrm{PRG}}$, where $\textsf{main} \in T_{\mathrm{PRG}}$ and $\pi \in L_{G_{\mathrm{PRG}}}[\textsf{B\_id}]$. $\textsf{main}(\pi) \equiv_{\mathit{SPEC}_{\mathrm{PRG}}} \textsf{true}$ means that $\pi$ is the main behavior expression.

To execute the main behavior expression, the initial values of the registers and I/O buffers must be specified:

11. For each $y \in \mathit{REG} \cup \mathit{BUF}$, the following production is in $P_{\mathrm{PRG}}$:

$$D_y \rightarrow \textsf{initial}(y),$$

where $\textsf{initial} \in T_{\mathrm{PRG}}$, $D_y = \mathit{type}[y]$ if $y \in \mathit{REG}$, and $D_y = \textsf{Seq\_}\mathit{type}[y]$ if $y \in \mathit{BUF}$. Moreover, for each $y \in \mathit{REG} \cup \mathit{BUF}$, there is exactly one axiom

$$\textsf{initial}(y) == c_y$$

in $AX_{\mathrm{PRG}}$, where $c_y \in L_{G_{\mathrm{PRG}}}[\mathit{type}[y]]$ if $y \in \mathit{REG}$ and $c_y \in L_{G_{\mathrm{PRG}}}[\textsf{Seq\_}\mathit{type}[y]]$ if $y \in \mathit{BUF}$. $\textsf{initial}(y) \equiv_{\mathit{SPEC}_{\mathrm{PRG}}} c_y$ means that the initial value of $y$ is $c_y$.

## 4. BE Interpreter Specifications

In this section, the semantics of BE programs is defined in terms of the behavior of a BE interpreter. Let $\mathit{SPEC}_{\mathrm{INT}} = (G_{\mathrm{INT}}, AX_{\mathrm{INT}})$ where $G_{\mathrm{INT}} = (N_{\mathrm{INT}}, T_{\mathrm{INT}}, P_{\mathrm{INT}})$ be a BE interpreter specification, and let $\mathit{SPEC}_{\mathrm{PRG}} = (G_{\mathrm{PRG}}, AX_{\mathrm{PRG}})$ where $G_{\mathrm{PRG}} = (N_{\mathrm{PRG}}, T_{\mathrm{PRG}}, P_{\mathrm{PRG}})$ be a BE program.

First, we assume that $G_{\mathrm{INT}} \supset G_{\mathrm{PRG}}$ (component-wise containment). $\mathit{SPEC}_{\mathrm{INT}}$ will be defined so that $\mathit{SPEC}_{\mathrm{INT}} \cup \mathit{SPEC}_{\mathrm{PRG}}$ (component-wise union) specifies the behavior of a BE interpreter when $\mathit{SPEC}_{\mathrm{PRG}}$ is given as its input program (see Fig. 1). Define $\mathit{REG}$ and $\mathit{BUF}$ as $L_{G_{\mathrm{PRG}}}[\textsf{Reg}]$ and $L_{G_{\mathrm{PRG}}}[\textsf{Buf}]$, respectively.

$$\underbrace{\boxed{\begin{array}{c}\text{BE Program}\\ \mathit{SPEC}_{\text{PRG}}\end{array}}\ \boxed{\begin{array}{c}\text{BE Interpreter Specification}\\ \mathit{SPEC}_{\text{INT}}\end{array}}}_{\text{``Executable'' Specification } \mathit{SPEC}}$$

Fig. 1 Executable specification *SPEC*.

To define the semantics of the operators used in behavior expressions, a quadruple relation *EXEC* is introduced. Let $B, B' \in L_{G_{\text{INT}}}[\mathsf{B\_exp}]$, $p \in L_{G_{\text{INT}}}[\mathsf{Bool}]$, and $a \in L_{G_{\text{INT}}}[\mathsf{Action}]$. $\langle B, p, a, B' \rangle \in \mathit{EXEC}$ means that "if a BE interpreter is about to execute behavior expression $B$, and the values of the registers and I/O buffers of the BE interpreter satisfy predicate $p$, then, the BE interpreter is allowed to perform action $a$, and it executes behavior expression $B'$ after $a$." In $\mathit{SPEC}_{\text{INT}}$, relation *EXEC* is represented by a predicate $\mathsf{exec}$. A production

$$\mathsf{Bool} \to \mathsf{exec}(\mathsf{B\_exp}, \mathsf{Bool}, \mathsf{Action}, \mathsf{B\_exp})$$

is included in $P_{\text{INT}}$, and the axioms shown in Table 3 is included in $AX_{\text{INT}}$.

We introduce $\mathsf{State} \in N_{\text{INT}}$, which is a data type representing states of the BE interpreter. The productions whose left-hand side is $\mathsf{State}$ are as follows:

$$\begin{aligned}\mathsf{State} &\to \mathsf{s}_{\mathsf{init}},\\ \mathsf{State} &\to \delta(\mathsf{State}, \mathsf{Action}),\end{aligned}$$

where $\mathsf{s}_{\mathsf{init}}$, $\delta \in T_{\text{INT}}$. $\mathsf{s}_{\mathsf{init}}$ denotes the initial state of the BE interpreter, and $\delta(s, a)$ denotes the state immediately after action $a$ is performed at state $s$.

By using the notion of states of a BE interpreter, we define the semantics of each action $a$ as relation between the values of the registers and I/O buffers before $a$ is performed and their values after $a$ is performed. To express this relation in $\mathit{SPEC}_{\text{INT}}$, for each $D \in N_0$, a production

$$D \to \mathsf{val}(D, \mathsf{State})$$

is introduced into $P_{\text{INT}}$, where $\mathsf{val} \in T_{\text{INT}}$. For any expression $t$ which includes some of

Table 3 Axioms for exec.

$\overline{B}, \overline{B'}, \overline{B_1}, \overline{B_1'}, \overline{B_2}, \overline{B_2'}$ : B_exp, $\overline{a}$ : Action,
$\overline{p}, \overline{p'}, \overline{p_1}, \overline{p_2}$ : Bool, $\overline{\pi}$ : B_id, $\overline{A}$ : Seq_Action

- Action-prefix:

$$\mathsf{exec}(\overline{a}; \overline{B}, \mathsf{true}, \overline{a}, \overline{B}) == \mathsf{true}.$$

- Choice:

$$\mathsf{exec}(\overline{B_1}, \overline{p_1}, \overline{a}, \overline{B_1'}) \supset \mathsf{exec}((\overline{B_1} \diamond \overline{B_2}), \overline{p_1}, \overline{a}, \overline{B_1'}) == \mathsf{true},$$
$$\mathsf{exec}(\overline{B_2}, \overline{p_2}, \overline{a}, \overline{B_2'}) \supset \mathsf{exec}((\overline{B_1} \diamond \overline{B_2}), \overline{p_2}, \overline{a}, \overline{B_2'}) == \mathsf{true}.$$

- Behavior identifier:

$$((\overline{\pi} := \overline{B}) \wedge \mathsf{exec}(\overline{B}, \overline{p}, \overline{a}, \overline{B'})) \supset \mathsf{exec}(\overline{\pi}, \overline{p}, \overline{a}, \overline{B'}) == \mathsf{true}.$$

- Parallel composition:

$$(\mathsf{exec}(\overline{B_1}, \overline{p_1}, \overline{a}, \overline{B_1'}) \wedge \neg \mathsf{member}(\overline{a}, \overline{A})) \supset$$
$$\mathsf{exec}((\overline{B_1} | \overline{A} | \overline{B_2}), \overline{p_1}, \overline{a}, (\overline{B_1'} | \overline{A} | \overline{B_2})) == \mathsf{true},$$
$$(\mathsf{exec}(\overline{B_2}, \overline{p_2}, \overline{a}, \overline{B_2'}) \wedge \neg \mathsf{member}(\overline{a}, \overline{A})) \supset$$
$$\mathsf{exec}((\overline{B_1} | \overline{A} | \overline{B_2}), \overline{p_2}, \overline{a}, (\overline{B_1} | \overline{A} | \overline{B_2'})) == \mathsf{true},$$
$$(\mathsf{exec}(\overline{B_1}, \overline{p_1}, \overline{a}, \overline{B_1'}) \wedge \mathsf{exec}(\overline{B_2}, \overline{p_2}, \overline{a}, \overline{B_2'}) \wedge \mathsf{member}(\overline{a}, \overline{A})) \supset$$
$$\mathsf{exec}((\overline{B_1} | \overline{A} | \overline{B_2}), \overline{p_1} \wedge \overline{p_2}, \overline{a}, (\overline{B_1'} | \overline{A} | \overline{B_2'})) == \mathsf{true}.$$

- Conditional:

$$\mathsf{exec}(\overline{B}, \overline{p}, \overline{a}, \overline{B'}) \supset \mathsf{exec}([\overline{p'}] \text{ -> } \overline{B}, \overline{p} \wedge \overline{p'}, \overline{a}, \overline{B'}) == \mathsf{true}.$$

- Enabling:

$$(\mathsf{exec}(\overline{B_1}, \overline{p_1}, \overline{a}, \overline{B_1'}) \wedge \neg \mathsf{member}(\overline{a}, \overline{A})) \supset$$
$$\mathsf{exec}((\overline{B_1} \gg \overline{A} \gg \overline{B_2}), \overline{p_1}, \overline{a}, (\overline{B_1'} \gg \overline{A} \gg \overline{B_2})) == \mathsf{true},$$
$$(\mathsf{exec}(\overline{B_1}, \overline{p_1}, \overline{a}, \overline{B_1'}) \wedge \mathsf{member}(\overline{a}, \overline{A})) \supset$$
$$\mathsf{exec}((\overline{B_1} \gg \overline{A} \gg \overline{B_2}), \overline{p_1}, \overline{a}, \overline{B_2}) == \mathsf{true}.$$

- Disabling:

$$(\mathsf{exec}(\overline{B_1}, \overline{p_1}, \overline{a}, \overline{B_1'}) \wedge \neg \mathsf{member}(\overline{a}, \overline{A})) \supset$$
$$\mathsf{exec}((\overline{B_1} \,[\!> \overline{A} \,[\!> \overline{B_2}), \overline{p_1}, \overline{a}, (\overline{B_1'} \,[\!> \overline{A} \,[\!> \overline{B_2})) == \mathsf{true},$$
$$(\mathsf{exec}(\overline{B_1}, \overline{p_1}, \overline{a}, \overline{B_1'}) \wedge \mathsf{member}(\overline{a}, \overline{A})) \supset$$
$$\mathsf{exec}((\overline{B_1} \,[\!> \overline{A} \,[\!> \overline{B_2}), \overline{p_1}, \overline{a}, \overline{B_1'}) == \mathsf{true},$$
$$\mathsf{exec}(\overline{B_2}, \overline{p_2}, \overline{a}, \overline{B_2'}) \supset \mathsf{exec}((\overline{B_1} \,[\!> \overline{A} \,[\!> \overline{B_2}), \overline{p_2}, \overline{a}, \overline{B_2'}) == \mathsf{true}.$$

Table 4 Axioms for $\mathsf{val}$.

$\overline{s}$ : State

- Calculation: For each $c \in T_0$,

$$\mathsf{val}(c, \overline{s}) == c,$$

and for each $f \in T_0$ such that $A \to f(A_1, \ldots, A_n) \in P_0$,

$\overline{t_1} : A_1, \ \ldots, \ \overline{t_n} : A_n$

$$\mathsf{val}(f(\overline{t_1}, \ldots, \overline{t_n}), \overline{s}) == f(\mathsf{val}(\overline{t_1}, \overline{s}), \ldots, \mathsf{val}(\overline{t_n}, \overline{s})).$$

- Initial value: For each $reg \in REG$ and $buf \in BUF$,

$$\mathsf{val}(reg, \mathsf{s_{init}}) == \mathsf{initial}(reg),$$
$$\mathsf{val}(buf, \mathsf{s_{init}}) == \mathsf{initial}(buf).$$

- $\mathsf{in}(buf, reg)$: For each $reg, reg' \in REG$ such that $reg \not\equiv reg'$, and for each $buf, buf' \in BUF$ such that $buf \not\equiv buf'$,

$$\mathsf{val}(reg, \delta(\overline{s}, \mathsf{in}(buf, reg))) == \mathsf{head}(\mathsf{val}(buf, \overline{s})),$$
$$\mathsf{val}(reg', \delta(\overline{s}, \mathsf{in}(buf, reg))) == \mathsf{val}(reg', \overline{s}),$$
$$\mathsf{val}(buf, \delta(\overline{s}, \mathsf{in}(buf, reg))) == \mathsf{tail}(\mathsf{val}(buf, \overline{s})),$$
$$\mathsf{val}(buf', \delta(\overline{s}, \mathsf{in}(buf, reg))) == \mathsf{val}(buf', \overline{s}).$$

- $\mathsf{out}(buf, reg)$: For each $reg, reg' \in REG$, and for each $buf, buf' \in BUF$ such that $buf \not\equiv buf'$,

$$\mathsf{val}(reg', \delta(\overline{s}, \mathsf{out}(buf, reg))) == \mathsf{val}(reg', \overline{s}),$$
$$\mathsf{val}(buf, \delta(\overline{s}, \mathsf{out}(buf, reg))) == \mathsf{val}(buf, \overline{s}) \cdot \mathsf{val}(reg, \overline{s}),$$
$$\mathsf{val}(buf', \delta(\overline{s}, \mathsf{out}(buf, reg))) == \mathsf{val}(buf', \overline{s}).$$

- $\mathsf{set}(reg \leftarrow t)$: For each $reg, reg' \in REG$ such that $reg \not\equiv reg'$, and for each $buf' \in BUF$,

$\overline{t} : type[reg]$

$$\mathsf{val}(reg, \delta(\overline{s}, \mathsf{set}(reg \leftarrow \overline{t}))) == \mathsf{val}(\overline{t}, \overline{s}),$$
$$\mathsf{val}(reg', \delta(\overline{s}, \mathsf{set}(reg \leftarrow \overline{t}))) == \mathsf{val}(reg', \overline{s}),$$
$$\mathsf{val}(buf', \delta(\overline{s}, \mathsf{set}(reg \leftarrow \overline{t}))) == \mathsf{val}(buf', \overline{s}).$$

members of $REG \cup BUF$, $\mathsf{val}(t, s)$ denotes the value of $t$ at state $s$. The semantics of the actions is defined by the axioms shown in Table 4.

Lastly, a production

$$\mathsf{Bool} \to \mathsf{bexp}(\mathsf{B\_exp}, \mathsf{State})$$

is introduced into $P$, where $\mathsf{bexp} \in T_{\mathrm{INT}}$, and the axioms shown in Table 5 into $AX_{\mathrm{INT}}$. Let $SPEC$ (= $(G, AX)$) be $SPEC_{\mathrm{PRG}} \cup SPEC_{\mathrm{INT}}$. Intuitively, $\mathsf{bexp}(B, s) \equiv_{SPEC} \mathsf{true}$ means that the BE interpreter can execute behavior expression $B$ at state $s$. By using $\mathsf{bexp}$, define the behavior of BE interpreters as follows:

**Definition 1:** A BE interpreter has to perform, at state $s$, an action $a$ such that

Table 5 Axioms for bexp.

$\bar{\pi}$ : B_id, $\bar{B}, \bar{B}'$ : B_exp, $\bar{s}$ : State, $\bar{p}$ : Bool, $\bar{a}$ : Action

$$\text{main}(\bar{\pi}) \supset \text{bexp}(\bar{\pi}, \text{s}_{\text{init}}) == \text{true},$$
$$(\text{bexp}(\bar{B}, \bar{s}) \wedge \text{exec}(\bar{B}, \bar{p}, \bar{a}, \bar{B}') \wedge \text{val}(\bar{p}, \bar{s})) \supset \text{bexp}(\bar{B}', \delta(\bar{s}, \bar{a})) == \text{true}.$$

bexp$(B, \delta(s, a)) \equiv_{SPEC}$ true for some $B \in L_G$[B_exp]. If such an action does not exist, the BE interpreter goes into a dead state. □

Suppose that bexp$((B_1 \diamond B_2), s) \equiv_{SPEC}$ true, where $s \in L_G$[State] and $B_1, B_2 \in L_G$[B_exp] such that

$$\begin{aligned} \text{exec}(B_1, p_1, a, B_1') &\equiv_{SPEC} \text{true}, \\ \text{exec}(B_2, p_2, a, B_2') &\equiv_{SPEC} \text{true} \end{aligned}$$

for some $p_1, p_2 \in L_G$[Bool], $a \in L_G$[Action], and $B_1', B_2' \in L_G$[B_exp]. If val$(p_1, s) \equiv_{SPEC}$ true and val$(p_2, s) \equiv_{SPEC}$ true, both of

$$\begin{aligned} \text{bexp}(B_1', \delta(s, a)) &\equiv_{SPEC} \text{true}, \\ \text{bexp}(B_2', \delta(s, a)) &\equiv_{SPEC} \text{true} \end{aligned}$$

hold by the definition of bexp. That is, if $a$ is a performable action in common with $B_1$ and $B_2$, nondeterministic choice $(B_1 \diamond B_2)$ at state $s$ is replaced by choice between $B_1'$ and $B_2'$ at state $\delta(s, a)$. Therefore, unlike LOTOS, executing

$$(a_1; \cdots; a_n; [p_1] \text{->} B_1' \diamond a_1; \cdots; a_n; [p_2] \text{->} B_2')$$

at state $s$ is equivalent to executing

$$a_1; \cdots; a_n; ([p_1] \text{->} B_1' \diamond [p_2] \text{->} B_2')$$

at $s$, in the sense that for both of these behavior expressions, a BE interpreter chooses between $[p_1]$ -> $B_1'$ and $[p_2]$ -> $B_2'$ at state $\delta(\cdots \delta(s, a_1), \cdots, a_n)$.

Similarly, this property holds in the case that bexp$(\pi, s) \equiv_{SPEC}$ true, where $\pi \in L_G$[B_id] has more than one behavior definitions. In section 6, we propose an implementation method which uses this property.

## 5. Translation from Natural Language Specifications of Communication Protocols into Logical Formulae

Refs. [3] and [9] propose a translation method from natural language specifications into algebraic specifications. As an example of an input specification, a communication protocol specification is considered. In such a specification, a sentence often specifies an action which a program (or a protocol machine in the case of protocol specifications) has to perform. However, it often specifies implicitly when the action should be performed.

**Example 1:** Consider the following consecutive sentences in Ref. [5]:

(1) A valid incoming MAJOR SYNC POINT SPDU (with . . . ) results in an S-SYNC-MAJOR indication.

(2) If Vsc is false, V(A) is set equal to V(M).

"MAJOR SYNC POINT SPDU" and "S-SYNC-MAJOR indication" are names of data, and "Vsc," " V(A)," and "V(M)" are names of registers of a protocol machine. A protocol machine has to perform the actions specified by (2) immediately after it performs the actions specified by (1). However, sentence (2) does not specify explicitly when the actions has to be performed. □

In Ref. [3], a state of the program which is specified implicitly in a natural language specification is called a *situation*. Moreover, for a constituent (i.e., a phrase, clause or sentence) $X$, the pre-situation of $X$ is defined as the situation at which the action(s) specified by $X$ has to be performed, and the post-situation of $X$ is defined as the one immediately after the action(s) is performed. The pre-/post-situations are also defined for a sequence of sentences.

It is assumed that a natural language specification is a set of paragraphs (sequences of sentences) and there exists no contextual dependency between distinct paragraphs (i.e., for any constituent $X$ in a paragraph, the pre-situation of $X$ is either the pre-situation of the paragraph or the post-situation of another constituent). In most of protocol specifications, for each kind of input data, there is a paragraph which specifies sequences of actions to be performed when a protocol machine receives an input data of that kind at the pre-situation of the paragraph. Therefore, the pre-situation of a paragraph usually denotes a state in which a protocol machine is waiting for an input. And, if a protocol machine reaches the post-situation of a paragraph, then the machine waits for a next input. Under these assumptions, each paragraph in a natural language specification is independently translated into an algebraic axiom.

In Ref. [3], situations are formalized as a data type Situation, which is represented by a sequence of "events" which the program has performed from the initial situation. By using type Situation and other primitive data types, a paragraph $P$ of a natural language specification is translated into an axiom in the form of

$$\begin{array}{l} \bar{\sigma}_0 : \mathsf{Situation},\ \bar{x}_1 : A_1,\ \ldots,\ \bar{x}_m : A_m \\ \qquad\qquad [R_1 \wedge \cdots \wedge R_m] \supset \bigwedge_{S\in P} \mathit{pred}_S \quad == \quad \mathsf{true}, \end{array} \tag{1}$$

where $\mathit{pred}_S$ is a logical formula denoting the meaning of sentence $S$, $\bar{\sigma}_0, \bar{x}_1, \ldots, \bar{x}_m$ are all the distinct variables appearing in $\bigwedge_{S\in P} \mathit{pred}_S$, and $A_j$ $(1 \leq j \leq m)$ is the data type of $\bar{x}_j$. Each sub-formula $R_j$ $(1 \leq j \leq m)$ is called the restriction on $\bar{x}_j$.

Each sub-predicate of $\mathit{pred}_S$ which denotes actions has two extra parameters: One denotes the pre-situation and the other does the post-situation. Since each paragraph is "contextually closed," the expression representing the pre-situation of a paragraph is denoted by a variable $\bar{\sigma}_0$. And, the expression representing each of other situations is denoted by $\tau_k(\bar{\sigma}_0, \bar{x}_1, \ldots, \bar{x}_j)$, where $\bar{x}_1, \ldots, \bar{x}_j$ represent input data received up to situation $\tau_k(\bar{\sigma}_0, \bar{x}_1, \ldots, \bar{x}_j)$.

**Example 2:** The paragraph which consists of the two sentences in Example 1 is translated into an axiom of $F$ == true, where $F$ is the following logical formula:

$$\begin{array}{l} \bar{\sigma}_0 : \mathsf{Situation}, \quad \bar{x}_1 : \mathsf{SPDU} \\ [\mathsf{valid}(\bar{x}_1) \wedge \mathsf{incoming}(\bar{x}_1) \wedge \mathsf{MAP}(\bar{x}_1)] \supset \\ \quad [\mathsf{receive}(\bar{x}_1, \hat{\sigma}_0, \hat{\sigma}_1) \wedge \mathsf{send}(\mathsf{SSYNMind}(\bar{x}_1), \hat{\sigma}_1, \hat{\sigma}_2)] \wedge \\ \quad [\mathsf{if\_then}(\mathsf{Vsc} = \mathsf{false}, \\ \qquad\qquad \mathsf{set\_equal\_to}(\mathsf{Va}, \mathsf{Vm}, \hat{\sigma}_3, \hat{\sigma}_4), \\ \qquad\qquad \hat{\sigma}_2, \hat{\sigma}_5)]]]. \end{array}$$

In the above logical formula, variable $\bar{x}_1$ represents an input data MAJOR SYNC POINT SPDU. And, $\hat{\sigma}_0, \ldots, \hat{\sigma}_5$ are expressions of type Situation, where $\hat{\sigma}_0 = \bar{\sigma}_0$ represents the pre-

Table 6    Meanings of subexpressions.

- valid($\bar{x}_1$): $\bar{x}_1$ has a valid data format.
- incoming($\bar{x}_1$): $\bar{x}_1$ is an incoming object.
- MAP($\bar{x}_1$): $\bar{x}_1$ is a data unit MAJOR SYNC POINT SPDU.
- SSYNMind($\bar{x}_1$): The service primitive S-SYNC-MAJOR indication to be sent when a protocol machine receives $\bar{x}_1$. Contents of the S-SYNC-MAJOR indication depends on $\bar{x}_1$.
- receive($\bar{x}_1, \hat{\sigma}_0, \hat{\sigma}_1$): At situation $\hat{\sigma}_0$, the event "receipt of $\bar{x}_1$" is allowed to occur and the situation immediately after the event is $\hat{\sigma}_1$.
- send(SSYNMind($\bar{x}_1$)$, \hat{\sigma}_1, \hat{\sigma}_2$): At situation $\hat{\sigma}_1$, the event "transmission of SSYNMind($\bar{x}_1$)" has to occur and the situation immediately after the event is $\hat{\sigma}_2$.
- set_equal_to(Va, Vm$, \hat{\sigma}_3, \hat{\sigma}_4$): At situation $\hat{\sigma}_3$, the event "setting the value of V(A) equal to the value of V(M)" has to occur and the situation immediately after the event is $\hat{\sigma}_4$.
- if_then($q$, *pred*$, \hat{\sigma}_2, \hat{\sigma}_5$): At situation $\hat{\sigma}_2$, the events specified by *pred* occur if $q$ is true, and no events occur otherwise. The situation immediately after these events is $\hat{\sigma}_5$.

situation of the paragraph and $\hat{\sigma}_k = \tau_k(\bar{\sigma}_0, \bar{x}_1)$ $(1 \leq k \leq 5)$ represent the other situations. Intuitive meanings of subexpressions in the formula are presented in Table 6. □

The semantics of $\tau_k$ is defined by the semantics of the predicate which includes $\tau_k(\bar{\sigma}_0, \ldots)$. Consider the logical formula in Example 2. By the semantics of if_then, $\hat{\sigma}_2$ is equal to $\hat{\sigma}_3$ and $\hat{\sigma}_4$ is equal to $\hat{\sigma}_5$ if Vsc = false holds, and $\hat{\sigma}_2$ is equal to $\hat{\sigma}_5$ otherwise. And, by the semantics of set_equal_to, $\hat{\sigma}_4$ is equal to the situation immediately after a protocol machine assigns the value of V(M) to V(A) at situation $\hat{\sigma}_3$ (see Table 6). As explained in the next section, the semantics of predicates which denote actions is defined in terms of behavior definitions.

## 6. Implementation of Logical Formulae by Behavior Definitions

In Ref. [3], an "event" is considered as an "atomic action" of a protocol machine (in the case of protocol specifications) such as transmitting data or updating a particular register (see Table 6). However, "atomic action" is informally used in Ref. [3] since protocol machines are not defined formally.

In section 4, we defined a BE interpreter, which can be a formal model of protocol machines. Now we consider "atomic actions" as expressions of type Action. Then, Situation is represented by a sequence of Action, i.e., type Situation is identified with type State introduced in this paper.

Let $\mathit{SPEC}_{\mathrm{NL}}$ be a natural language specification, i.e., a set of paragraphs. Let $\mathit{SPEC}_{\mathrm{LF}} = (G_{\mathrm{LF}}, AX_{\mathrm{LF}})$ ($G_{\mathrm{LF}} = (N_{\mathrm{LF}}, T_{\mathrm{LF}}, P_{\mathrm{LF}})$) be the algebraic specification derived from $\mathit{SPEC}_{\mathrm{NL}}$, where $AX_{\mathrm{LF}}$ consists of:

- axioms on primitive data types; and
- axioms in the following form:

$$\begin{aligned} &\bar{s}_0 : \mathsf{Situation},\ \bar{x}_1 : A_1,\ \ldots,\ \bar{x}_m : A_m \\ &\qquad \bigwedge_{S \in P} \Big[(R_1 \wedge \cdots \wedge R_m) \supset \mathit{pred}_S\Big] \quad == \quad \mathsf{true}, \end{aligned} \tag{2}$$

  where $P \in \mathit{SPEC}_{\mathrm{NL}}$ is a paragraph, and for each $j$ $(1 \leq j \leq m)$, $A_j$ denotes a primitive data type and $R_j$ denotes the restriction on $\bar{x}_j$. For any paragraph $P \in \mathit{SPEC}_{\mathrm{NL}}$, the pre-situation of $P$ is denoted by $\bar{s}_0$, and an input data to be received at the pre-situation of $P$ is denoted by $\bar{x}_1$.

In what follows, a method of implementing $\mathit{SPEC}_{\mathrm{LF}}$ by a BE program $\mathit{SPEC}_{\mathrm{PRG}} = (G_{\mathrm{PRG}}, AX_{\mathrm{PRG}})$ is presented. Let $\mathit{SPEC}_{\mathrm{INT}}$ be a BE interpreter specification for $\mathit{SPEC}_{\mathrm{PRG}}$, and $\mathit{SPEC} = \mathit{SPEC}_{\mathrm{PRG}} \cup \mathit{SPEC}_{\mathrm{INT}}$ (see Fig. 2).

In our implementation method, each variable of a primitive data type in Axiom (2) corresponds to a register, and each expression of type Situation which includes variables in Axiom (2) (i.e., $\bar{s}_0$ or $\tau_k(\bar{s}_0, \ldots)$ for some $k$) corresponds to a behavior identifier. To do this, the following registers and behavior identifiers are introduced into $\mathit{SPEC}_{\mathrm{PRG}}$:

1. $\mathit{REG}_{\mathrm{VAR}} = \{\mathit{var}_1, \ldots, \mathit{var}_m\}$: Each register $\mathit{var}_j$ is used for storing the value of variable $\bar{x}_j$ in Axiom (2);
2. $\mathit{REG}_{\mathrm{PRED}} = \{\mathit{reg}_1, \ldots, \mathit{reg}_n\}$: Each register $\mathit{reg}_i$ has been defined in $\mathit{SPEC}_{\mathrm{LF}}$ (e.g., Vsc, Va, and Vm in Example 2);
3. $\mathit{REG}_{\mathrm{TMP}}$: A register in $\mathit{REG}_{\mathrm{TMP}}$ is a temporary or dummy one, and denoted by *tmp* with some subscripts; and

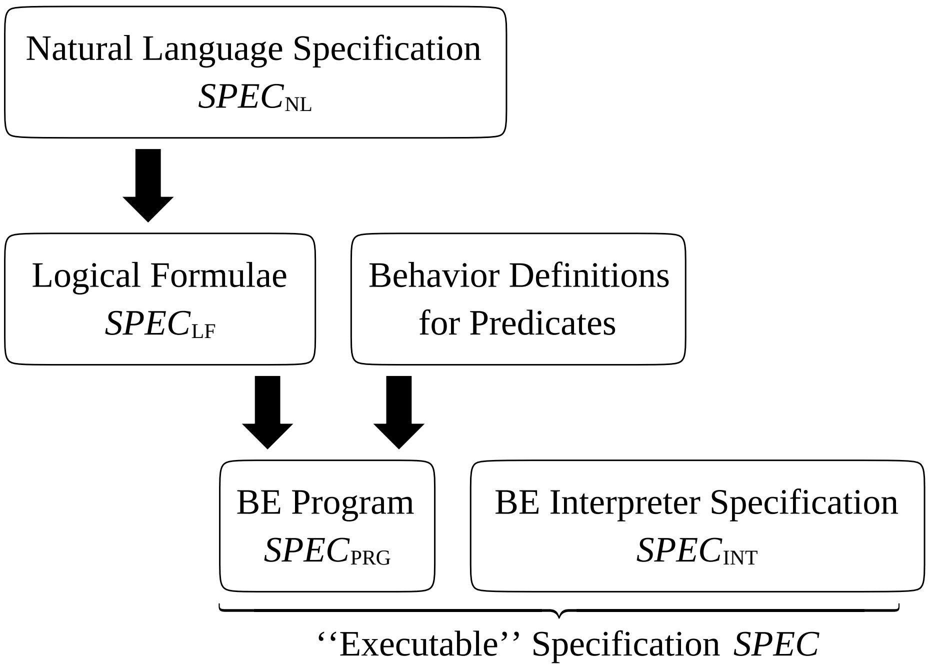
Natural Language Specification
$SPEC_{\mathrm{NL}}$
Logical Formulae
$SPEC_{\mathrm{LF}}$
Behavior Definitions for Predicates
BE Program
$SPEC_{\mathrm{PRG}}$
BE Interpreter Specification
$SPEC_{\mathrm{INT}}$
''Executable'' Specification $SPEC$

Fig. 2 Implementation method.

4. $\pi_{\hat{s}}$: For each pre-/post-situation $\hat{s}$ appearing in Axiom (2), a behavior identifier $\pi_{\hat{s}}$ is introduced into $SPEC_{\mathrm{PRG}}$. A human implementor specifies behavior definitions so that for any instantiated term $s$ of $\hat{s}$ which satisfies Axiom (2),

   - $\mathsf{bexp}(\pi_{\hat{s}}, s) \equiv_{SPEC} \mathsf{true}$, or
   - $\mathsf{bexp}(\pi_{\hat{s}'}, s) \equiv_{SPEC} \mathsf{true}$ for some $\pi_{\hat{s}'}$ such that

   $$(\pi_{\hat{s}'} := \pi_1) \wedge (\pi_1 := \pi_2) \wedge \cdots \wedge (\pi_{l-1} := \pi_l) \wedge (\pi_l := \pi_{\hat{s}}) \equiv_{SPEC} \mathsf{true}$$

   for some $\pi_1, \ldots, \pi_l \in L_G[\mathsf{B\_id}]$ $(l \geq 0)$.

   That is, at such a situation $s$, a BE interpreter can always execute $\pi_{\hat{s}}$. For example, if a predicate $p(\ldots, \hat{s}, \hat{s}')$ ($\hat{s}$ and $\hat{s}'$ are the pre- and post-situations respectively) denotes a sequence $a_1, \ldots, a_k$ of actions, a human implementor specifies $\pi_{\hat{s}} := a_1; \cdots; a_k; \pi_{\hat{s}'}$ as the semantics of $p$ (the details are explained in Step 2.2 below).

The implementation method consists of the following three steps:

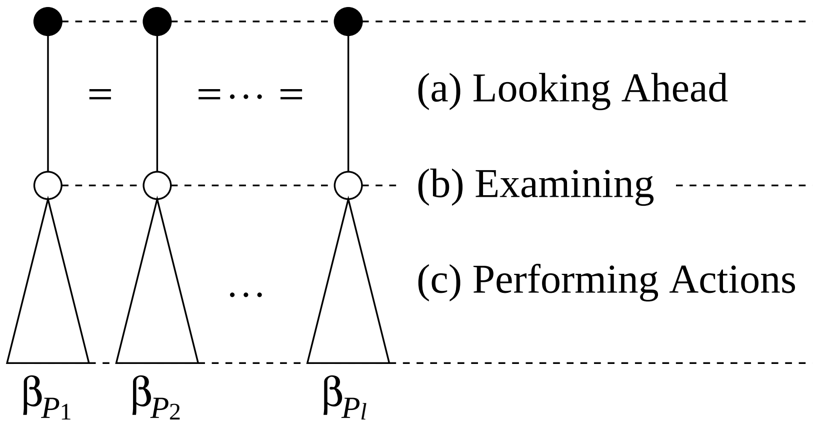

Fig. 3 Execution of each of $\beta_{P_1}, \beta_{P_2}, \ldots, \beta_{P_l}$.

**Step 1:** For each paragraph $P \in SPEC_{\mathrm{NL}}$, a set $\beta_P$ of behavior definitions is constructed by Steps 2 and 3. The behavior of a BE interpreter which executes $\pi_{\bar{s}_0}$ ($\bar{s}_0$ is the pre-situation of $P$) defined by $\beta_P$ is as follows:

(a) the BE interpreter looks ahead the first element $d$ of an input buffer,

(b) it examines whether paragraph $P$ specifies actions for $d$, and

(c) it performs the actions specified by $P$ if $P$ passes the examination (b).

See Fig. 3. Each of the black circles represents some state $s$ such that $\mathsf{bexp}(\pi_{\bar{s}_0}, s) \equiv_{SPEC}$ $\mathsf{true}$, and each of the white circles represents the state at which the BE interpreter performs the examination (b). Each of the lines from the black circles to the white ones represents a sequence of actions to perform (a). And, each of the triangles represents sequences of actions which are specified by the paragraph. For each paragraph $P$, the behavior expression to perform (a) is the same, and the behavior expression to perform (b) and (c) is in the form of $[p_P]$ −> $B_P$, where $p_P$ corresponds to the examination (b) and $B_P$ corresponds to (c) (see Steps 2.1 and 2.2 below).

Let $\beta = \bigcup_{P \in SPEC_{\mathrm{NL}}} \beta_P$. Since the pre-situation of any paragraph is denoted by $\bar{s}_0$, $\beta$ has in general more than one behavior definitions of $\pi_{\bar{s}_0}$. Let $s$ be the state immediately after (a) is performed. As stated at the end of section 4, $\mathsf{bexp}([p_P]$ −> $B_P, s) \equiv_{SPEC}$ $\mathsf{true}$ for each paragraph $P \in SPEC_{\mathrm{NL}}$. Therefore, at state $s$, only the actions specified by a paragraph $P$

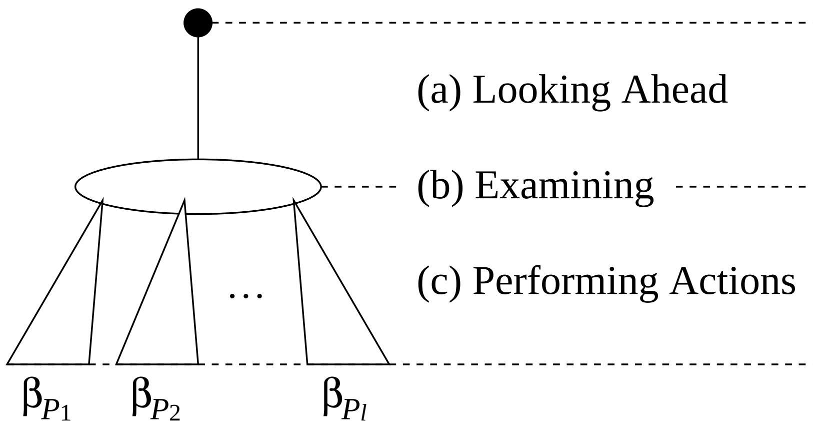

Fig. 4 Execution of $\beta$.

such that $\mathsf{val}(p_P, s) \equiv_{SPEC} \mathsf{true}$ are performed (see Fig. 4; the oval corresponds to $s$). Thus, $\beta = \bigcup_{P \in SPEC_{\mathrm{NL}}} \beta_P$ is the implementation of $SPEC_{\mathrm{NL}}$.

**Step 2:** For each sentence $S \in P$, $(R_1 \wedge \cdots \wedge R_m) \supset pred_S$ is translated into behavior definitions (denoted by $behavior[\langle R_1, \ldots, R_m \rangle, pred_S]$) as follows:

**Step 2.1:** The "subroutines" for "variable bindings" are defined as a set of behavior definitions (denoted by $bind[\langle R_1, \ldots, R_m \rangle, pred_S]$). Let $\alpha\{\gamma'_1/\gamma_1, \ldots, \gamma'_m/\gamma_m\}$ denote the expression obtained by replacing each subexpression $\gamma_j$ $(1 \leq j \leq m)$ of expression $\alpha$ by expression $\gamma'_j$. Let $pre[pred_S]$ denote the actual parameter of $pred_S$ which represents the pre-situation of $pred_S$, and let $post[pred_S]$ denote the actual parameter of $pred_S$ which represents the post-situation of $pred_S$. For simplicity, define $\tau_{\bar{s}_0}$, where $\bar{s}_0$ denotes the pre-situation of the paragraph, as the identity function on type $\mathsf{Situation}$. Suppose that $pre[pred_S] = \tau_k(\bar{s}_0, \bar{x}_1, \ldots, \bar{x}_j)$ $(0 \leq j \leq m)$ and $post[pred_S] = \tau_{k'}(\bar{s}_0, \bar{x}_1, \ldots, \bar{x}_{j'})$ $(j \leq j' \leq m)$. Then, during the "execution" of $pred_S$, the input data represented by $\bar{x}_{j+1}, \ldots, \bar{x}_{j'}$ are received and each of $\bar{x}_{j+1}, \ldots, \bar{x}_{j'}$ is bound to some value. A behavior identifier which simulates these variable bindings is denoted by $\rho_{pre[pred_S], post[pred_S]}$. The behavior definition of $\rho_{pre[pred_S], post[pred_S]}$ is in the following form:

$$\begin{aligned}\rho_{pre[pred_S], post[pred_S]} \ := \ & \mathsf{set}(var_{j+1} \leftarrow \hat{t}_{j+1}); \cdots ; \mathsf{set}(var_{j'} \leftarrow \hat{t}_{j'}); \\ & [(R_{j+1} \wedge \cdots \wedge R_{j'})\{var_1/\bar{x}_1, \ldots, var_{j'}/\bar{x}_{j'}\}] \mathrel{-\!\!>} \\ & \quad \mathsf{set}(tmp_{\mathrm{bind}} \leftarrow tmp_{\mathrm{bind}}); \mathsf{stop}.\end{aligned}$$

Here, $\hat{t}_{j''}$ ($j + 1 \le j'' \le j'$) is an expression which indicates how the value of $\mathit{var}_{j''}$ is obtained, and is specified by a human implementor. Action $\mathsf{set}(\mathit{tmp}_{\mathrm{bind}} \leftarrow \mathit{tmp}_{\mathrm{bind}})$ is performed as a “signal” which denotes successful completion of the variable bindings.

**Step 2.2:** The “main routine” of $\mathit{pred}_S$ is defined as a set of behavior definitions (denoted by $\mathit{dic}[\mathit{pred}_S]$). As shown in the following examples, a behavior definition of $\pi_{\mathit{pre}[\mathit{pred}_S]}$ has to be in $\mathit{dic}[\mathit{pred}_S]$, and $\pi_{\mathit{post}[\mathit{pred}_S]}$ has to appear in some behavior definitions in $\mathit{dic}[\mathit{pred}_S]$.

If $\mathit{dic}[\mathit{pred}_S]$ is already defined and stored as a “lexical item” of $\mathit{pred}_S$, then a human implementor has only to define $\mathit{bind}[\langle R_1, \ldots, R_m \rangle, \mathit{pred}_S]$.

**Example 3:** $\mathit{dic}[\mathsf{receive}(\hat{t}, \hat{s}, \hat{s}')]$ is shown in Table 7 (a), where $\mathit{type}[\mathit{tmp}_{\mathrm{in}}] = D$ such that

$$\mathsf{Bool} \rightarrow \mathsf{receive}(D, \mathsf{Situation}, \mathsf{Situation}) \in P_{\mathrm{LF}}.$$

The meaning of the behavior definition is as follows. When $\mathsf{buf}_{\mathsf{inSPDU}}$ is not empty, then look ahead the first element $d$ of $\mathsf{buf}_{\mathsf{inSPDU}}$, copy $d$ to a temporary register $\mathit{tmp}_{\mathrm{in}}$, and perform variable bindings $\rho_{\hat{s},\hat{s}'}$. During the execution of $\rho_{\hat{s},\hat{s}'}$, $\mathsf{set}(\mathit{tmp}_{\mathrm{bind}} \leftarrow \mathit{tmp}_{\mathrm{bind}})$ is performed if the variable bindings are completed successfully. Then, move the first element $d$ of $\mathsf{buf}_{\mathsf{inSPDU}}$ to $\mathit{tmp}_{\mathrm{in}}$, and execute $\pi_{\hat{s}'}$.

Also,

$$\mathit{bind}[\langle \mathsf{valid}(\bar{x}_1) \wedge \mathsf{incoming}(\bar{x}_1) \wedge \mathsf{MAP}(\bar{x}_1) \rangle, \mathsf{receive}(\bar{x}_1, \hat{\sigma}_0, \hat{\sigma}_1)]$$

can be defined as

$$\begin{aligned} \rho_{\hat{\sigma}_0,\hat{\sigma}_1} \;:=\; & \mathsf{set}(\mathit{var}_1 \leftarrow \mathit{tmp}_{\mathrm{in}}); \\ & [\mathsf{valid}(\mathit{var}_1) \wedge \mathsf{incoming}(\mathit{var}_1) \wedge \mathsf{MAP}(\mathit{var}_1)] \text{ -> } \\ & \quad \mathsf{set}(\mathit{tmp}_{\mathrm{bind}} \leftarrow \mathit{tmp}_{\mathrm{bind}}); \mathsf{stop}. \end{aligned}$$

Here, it is specified that the value of $\bar{x}_1$ is equal to the value of $\mathit{tmp}_{\mathrm{in}}$, i.e., the first element of $\mathsf{buf}_{\mathsf{inSPDU}}$. □

As shown in Example 3, a human implementor should specify $\mathit{dic}$ so that the behavior definitions for any predicate which involves an input action (such as $\mathsf{receive}$) use the same temporary register $\mathit{tmp}_{\mathrm{in}}$ to store an input data. Then, he/she can specify $\mathit{tmp}_{\mathrm{in}}$ as $\hat{t}$ appearing in $\mathit{bind}$.

Table 7 “Lexical items” for predicates.

(a) $dic[\mathsf{receive}(\hat{t}, \hat{s}, \hat{s}')]$.

$$\begin{aligned}\pi_{\hat{s}} := {} & ([\mathsf{buf}_{\mathsf{inSPDU}} \neq \lambda] \mathbin{-\!\!>} \mathsf{set}(tmp_{\mathrm{in}} \leftarrow \mathsf{head}(\mathsf{buf}_{\mathsf{inSPDU}})); \rho_{\hat{s},\hat{s}'} \\ & \gg \lambda \cdot \mathsf{set}(tmp_{\mathrm{bind}} \leftarrow tmp_{\mathrm{bind}}) \gg \\ & [(tmp_{\mathrm{in}} = \hat{t}\{var_1/\bar{x}_1, \ldots, var_m/\bar{x}_m\})] \mathbin{-\!\!>} \mathsf{in}(\mathsf{buf}_{\mathsf{inSPDU}}, tmp_{\mathrm{in}}); \pi_{\hat{s}'}).\end{aligned}$$

(b) $dic[\mathsf{send}(\hat{t}, \hat{s}, \hat{s}')]$.

$$\begin{aligned}\pi_{\hat{s}} := {} & (\rho_{\hat{s},\hat{s}'} \\ & \gg \lambda \cdot \mathsf{set}(tmp_{\mathrm{bind}} \leftarrow tmp_{\mathrm{bind}}) \gg \\ & \mathsf{set}(tmp_{\mathrm{out}} \leftarrow \hat{t}\{var_1/\bar{x}_1, \ldots, var_m/\bar{x}_m\}); \mathsf{out}(\mathsf{buf}_{\mathsf{outSSprm}}, tmp_{\mathrm{out}}); \pi_{\hat{s}'}).\end{aligned}$$

(c) $dic[\mathsf{set\_equal\_to}(\hat{t}_1, \hat{t}_2, \hat{s}, \hat{s}')]$.

$$\begin{aligned}\pi_{\hat{s}} := {} & (\rho_{\hat{s},\hat{s}'} \\ & \gg \lambda \cdot \mathsf{set}(tmp_{\mathrm{bind}} \leftarrow tmp_{\mathrm{bind}}) \gg \\ & \mathsf{set}(\hat{t}_1 \leftarrow \hat{t}_2\{var_1/\bar{x}_1, \ldots, var_m/\bar{x}_m\}); \pi_{\hat{s}'}).\end{aligned}$$

**Example 4:** $dic[\mathsf{send}(\hat{t}, \hat{s}, \hat{s}')]$ is defined as shown in Table 7 (b), where $type[tmp_{\mathrm{out}}] = D$ such that

$$\mathsf{Bool} \to \mathsf{send}(D, \mathsf{Situation}, \mathsf{Situation}) \in P_{\mathrm{LF}}.$$

First, perform the variable bindings $\rho_{\hat{s},\hat{s}'}$. When this is completed successfully, calculate $\hat{t}$, assign the result to a temporary register $tmp_{\mathrm{out}}$, and output it to $\mathsf{buf}_{\mathsf{outSSprm}}$.

The behavior definition

$$bind[\langle \mathsf{valid}(\bar{x}_1) \wedge \mathsf{incoming}(\bar{x}_1) \wedge \mathsf{MAP}(\bar{x}_1)\rangle,$$

$$\mathsf{send}(\mathsf{SSYNMind}(\bar{x}_1), \hat{\sigma}_1, \hat{\sigma}_2)]$$

is simply defined as $\rho_{\hat{\sigma}_1,\hat{\sigma}_2} := \mathsf{set}(tmp_{\mathrm{bind}} \leftarrow tmp_{\mathrm{bind}}); \mathsf{stop}$, since there are no variables to be bound. □

**Example 5:** $dic[\mathsf{set\_equal\_to}(\hat{t}_1, \hat{t}_2, \hat{s}, \hat{s}')]$ is defined as shown in Table 7 (c). Also,

$$bind[\langle \mathsf{valid}(\bar{x}_1) \wedge \mathsf{incoming}(\bar{x}_1) \wedge \mathsf{MAP}(\bar{x}_1)\rangle,$$

$$\mathsf{set\_equal\_to}(\mathsf{Va}, \mathsf{Vm}, \hat{\sigma}_3, \hat{\sigma}_4)]$$

can be defined as $\rho_{\hat{\sigma}_3,\hat{\sigma}_4} := \mathsf{set}(tmp_{\mathrm{bind}} \leftarrow tmp_{\mathrm{bind}}); \mathsf{stop}$. □

As shown in the following example, one can construct behavior definitions for a predicate which takes other predicates as its parameters.

**Example 6:** We define

$$\mathit{behavior}[\langle R_1, \ldots, R_m \rangle, \mathsf{if\_then}(\hat{q}, \hat{p}, \hat{s}, \hat{s}')]$$

as

$$\begin{aligned}&\mathit{behavior}[\langle R_1, \ldots, R_m \rangle, \hat{p}],\\ &\mathit{dic}[\mathsf{if\_then}(\hat{q}, \hat{p}, \hat{s}, \hat{s}')].\end{aligned}$$

And, $\mathit{dic}[\mathsf{if\_then}(\hat{q}, \hat{p}, \hat{s}, \hat{s}')]$ is the set of the following behavior definitions:

$$\begin{aligned}\pi_{\hat{s}} \quad &:= \quad ([\hat{q}\{\mathit{var}_1/\bar{x}_1, \ldots, \mathit{var}_m/\bar{x}_m\}] \mathrel{-\!\!>} \pi_{\mathit{pre}[\hat{p}]}\\ &\qquad\quad \diamond\\ &\qquad [\neg\hat{q}\{\mathit{var}_1/\bar{x}_1, \ldots, \mathit{var}_m/\bar{x}_m\}] \mathrel{-\!\!>} \pi_{\hat{s}'}),\\ \pi_{\mathit{post}[\hat{p}]} \quad &:= \quad \pi_{\hat{s}'}.\end{aligned}$$

□

**Step 3:** Let $\beta'_P = \bigcup_{S \in P} \mathit{behavior}[\langle R_1, \ldots, R_m \rangle, \mathit{pred}_S]$. Let $\tau_l(\bar{s}_0, \ldots)$ be the post-situation of paragraph $P$. Then, $\pi_{\tau_l(\bar{s}_0,\ldots)} := \pi_{\bar{s}_0}$ is added to $\beta'_P$. That is, after the completion of performing the actions specified by $P$, a BE interpreter executes $\pi_{\bar{s}_0}$, i.e., it looks ahead the next input (recall that the pre-situation of any paragraph $P'$ is denoted by $\bar{s}_0$). The resultant set of behavior definitions is $\beta_P$.

**Example 7:** For the logical formula in Example 2, the behavior definitions in Table 8 are obtained. Here, we write $\pi_k$ instead of $\pi_{\hat{\sigma}_k}$ ($0 \leq k \leq 5$). By Step 3, the behavior definition of $\pi_5$ is added. □

## 7. Implementation System

We have implemented a prototype system which implements logical formulae derived from natural language specifications by BE programs. This system is written in Prolog (100

Table 8  Implementation of the logical formula in Example 2.

$$
\begin{aligned}
\pi_0 &:= ([\mathsf{buf}_{\mathsf{inSPDU}} \neq \lambda] \mathrel{-\!>} \mathsf{set}(tmp_{\mathrm{in}} \leftarrow \mathsf{head}(\mathsf{buf}_{\mathsf{inSPDU}})); \rho_{\hat{\sigma}_0,\hat{\sigma}_1}\\
&\qquad \gg \lambda \cdot \mathsf{set}(tmp_{\mathrm{bind}} \leftarrow tmp_{\mathrm{bind}}) \gg\\
&\quad [(tmp_{\mathrm{in}} = var_1)] \mathrel{-\!>} \mathsf{in}(\mathsf{buf}_{\mathsf{inSPDU}}, tmp_{\mathrm{in}}); \pi_1),\\
\pi_1 &:= (\rho_{\hat{\sigma}_1,\hat{\sigma}_2}\\
&\qquad \gg \lambda \cdot \mathsf{set}(tmp_{\mathrm{bind}} \leftarrow tmp_{\mathrm{bind}}) \gg\\
&\quad \mathsf{set}(tmp_{\mathrm{out}} \leftarrow \mathsf{SSYNMind}(var_1));\\
&\quad \mathsf{out}(\mathsf{buf}_{\mathsf{outSSprm}}, tmp_{\mathrm{out}}); \pi_2),\\
\pi_2 &:= ([\mathsf{Vsc} = \mathsf{false}] \mathrel{-\!>} \pi_3 \diamond [\neg(\mathsf{Vsc} = \mathsf{false})] \mathrel{-\!>} \pi_5),\\
\pi_3 &:= (\rho_{\hat{\sigma}_3,\hat{\sigma}_4}\\
&\qquad \gg \lambda\cdot\mathsf{set}(tmp_{\mathrm{bind}} \leftarrow tmp_{\mathrm{bind}}) \gg\\
&\quad \mathsf{set}(\mathsf{Va} \leftarrow \mathsf{Vm}); \pi_4),\\
\pi_4 &:= \pi_5,\\
\pi_5 &:= \pi_0,\\
\rho_{\hat{\sigma}_0,\hat{\sigma}_1} &:= \mathsf{set}(var_1 \leftarrow tmp_{\mathrm{in}});\\
&\quad [\mathsf{valid}(var_1) \wedge \mathsf{incoming}(var_1) \wedge \mathsf{MAP}(var_1)] \mathrel{-\!>}\\
&\qquad \mathsf{set}(tmp_{\mathrm{bind}} \leftarrow tmp_{\mathrm{bind}}); \mathsf{stop},\\
\rho_{\hat{\sigma}_1,\hat{\sigma}_2} &:= \mathsf{set}(tmp_{\mathrm{bind}} \leftarrow tmp_{\mathrm{bind}}); \mathsf{stop},\\
\rho_{\hat{\sigma}_3,\hat{\sigma}_4} &:= \mathsf{set}(tmp_{\mathrm{bind}} \leftarrow tmp_{\mathrm{bind}}); \mathsf{stop}.
\end{aligned}
$$

clauses). Using this system, we have implemented the logical formulae derived from a part of the OSI session protocol specification [5] (18 paragraphs, 45 sentences). The number of the "lexical items" ($dic[pred_S]$) specified by the human implementor is 27, and the output of the system is 189 behavior definitions.

We have also implemented a simulator which executes a given BE program. This simulator is written in C, lex, and yacc (1954 lines). The simulator executing the behavior definitions obtained by the implementation system behaved just as the human implementor intended.

Fig. 5 shows a part of the execution of the simulator when the behavior definitions in Table 8 are given as its input program. When the simulator executes a behavior expression, it computes actions to be performed by using the axioms on exec (Table 3). Since, in general, there may be behavior definitions which cause infinite applications of the axioms (such as $\pi := (\pi \diamond a; \pi'))$, the simulator tries only $n$ applications of the axioms for a given constant $n$ (10 by default, but the user can change $n$ to a greater value). Then, the user selects an action to be performed. The user can also request the simulator to show the contents of all registers and buffers.

## 8. Conclusion

This paper has described a method of implementing natural language specifications of communication protocols by executable specifications. By using this implementation method and the simulator stated in section 7, one can apply rapid prototyping techniques to such a natural language specification. Then, he/she can detect and correct errors, if any, in the natural language specification easily.

```
.....
*** recursion depth = 10 ***
bexp:       \pi_{s1}
--- 1 ---
action:     \set ( \tmpbind \leftarrow \tmpbind )
next bexp: \set ( \tmpout \leftarrow \ssynmmind ( \var_{1} ) ) ; \out
( \bufoutssprm , \tmpout ) ; \pi_{s2}
which ? 1
*** recursion depth = 10 ***
bexp:       \set ( \tmpout \leftarrow \ssynmmind ( \var_{1} ) ) ; \out
( \bufoutssprm , \tmpout ) ; \pi_{s2}
--- 1 ---
action:     \set ( \tmpout \leftarrow \ssynmmind ( \var_{1} ) )
next bexp: \out ( \bufoutssprm , \tmpout ) ; \pi_{s2}
which ? 1
*** recursion depth = 10 ***
bexp:       \out ( \bufoutssprm , \tmpout ) ; \pi_{s2}
--- 1 ---
action:     \out ( \bufoutssprm , \tmpout )
next bexp: \pi_{s2}
which ? 1
*** recursion depth = 10 ***
bexp:       \pi_{s2}
--- 1 ---
action:     \set ( \tmpbind \leftarrow \tmpbind )
next bexp: \set ( \regva \leftarrow \regvm ) ; \pi_{s4}
which ? s
\bufinspdu: \lambda \cdot 2001
\tmpin: 2000
\tmpbind: \true
\var_{1}: 2000
\tmpout: 22000
\bufoutssprm: \lambda \cdot 22000
\regvsc: \false
\regva: 0
\regvm: 0
which ? 1
*** recursion depth = 10 ***
.....
```

Fig. 5    Execution of the simulator.

## References

[1] Cheng, Z., Takahashi, K., Shiratori, N. and Noguchi, S.: An Automatic Implementation Method of Protocol Specifications in LOTOS, IEICE Trans. Inf. & Syst., Vol. E75-D, No. 4, pp. 543–556 (1992).

[2] Goguen, J. A., Thatcher, J. W. and Wagner, E. G.: An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, IBM Research Report, RC 6487 (1976), also in ed, Yeh, R., Current Trends in Programming Methodology IV: Data Structuring, Prentice Hall, pp. 80–144 (1978).

[3] Ishihara, Y., Seki, H., Kasami, T., Shimabukuro, J. and Okawa, K.: A Translation Method from Natural Language Specifications of Communication Protocols into Algebraic Specifications Using Contextual Dependencies, IEICE Trans. Inf. & Syst., Vol. E76-D, No. 12, pp. 1479–1489 (1993).

[4] Ishihara, Y., Seki, H. and Kasami, T.: An Algebraic Definition of a LOTOS-Like Language and Its Application, Preprint of WG on Software Engineering of IPS Japan, SE-99-1 (1994).

[5] ISO: Basic Connection Oriented Session Protocol Specification, ISO 8327 (1987).

[6] ISO: Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, ISO 8807 (1989).

[7] Karjoth, G.: Implementing Process Algebra Specifications by State Machines, Testing and Verification VIII, pp. 47–60 (1988).

[8] Kasami, T., Taniguchi, K., Sugiyama, Y. and Seki, H.: Principles of Algebraic Language ASL/$*$, Trans. IECE Japan, Vol. J69-D, No. 7, pp. 1066–1074, Jul. 1986 (in Japanese), also in Systems and Computers in Japan, Vol. 18, No. 7, pp. 11–20 (1987).

[9] Seki, H., Kasami, T., Nabika, E. and Matsumura, T.: A Method for Translating Natural Language Program Specifications into Algebraic Specifications, Trans. IEICE Japan, Vol. J74-D-I, No. 4, pp. 283–295 (1991) (in Japanese).