A Translation Method from Natural Language Specifications of Communication Protocols into Executable Algebraic Specifications

Yasunori Ishihara

January 1995

A Translation Method from Natural Language Specifications of Communication Protocols into Executable Algebraic Specifications

by

Yasunori Ishihara

January 1995

Dissertation submitted to Graduate School of Engineering Science of
Osaka University in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in Engineering

Abstract

In a software development process, informal requirements and/or specifications are often written in a natural language since they become readable and the intuitive meanings understandable. However, natural language specifications are apt to be incomplete. To verify whether a program satisfies a specification, the specification needs to be written formally.

In this dissertation, we propose a translation method from natural language specifications into algebraic specifications such that (1) incompleteness of a natural language specification is detected or reduced in a translation process, (2) information necessary for translation (e.g., lexical items) can be obtained as automatically as possible, and (3) derived algebraic specifications are executable. An input of the translation method is a part of a protocol specification which specifies action sequences performed by the protocol machine (program).

In Chapter 1, related topics on translation methods are briefly summarized from the above viewpoints (1)–(3).

Chapter 2 summarizes algebraic specification language ASL, which is adopted as a formal specification language in this dissertation. An algebraic specification in ASL is a pair of a context-free grammar (cfg) and a set of axioms. A cfg specifies a set of expressions and their syntax, and a set of axioms specifies their semantics.

Usually, a sentence in an input natural language specification implicitly specifies the state of the protocol machine at which the described actions must be performed. In Chapter 3, we propose a method of analyzing the implicitly specified states of the protocol machine taking the OSI session protocol specification (265 sentences) as an example. Implicitly specified states are determined by analyzing dependency, called S-dependency, among constituents (phrases, clauses, or sentences) in the natural language specification. The method uses the following properties and information: (a) syntactic properties of a natural language (English in this dissertation), (b) type information assigned to words in a natural language specification, and (c) properties specific to the target domain, e.g., properties of data types. In this method, these properties and information are assumed to be given and stored in a dictionary. An input natural language specification is analyzed by this method and translated into algebraic axioms in the form of logical formulas. The result of applying this method to the main part of the OSI session protocol specification (29 paragraphs, 98 sentences) is also presented. For 95 sentences, the S-dependency was uniquely determined by using only (a) and (b) described above. By using (c) in addition, the S-dependency of the remaining three sentences was uniquely determined. Thus, this method is effective in reducing incompleteness of natural language specifications.

In Chapter 3, an assignment of data types to words in a natural language specification

is assumed to be given. However, it is desirable that data types be assigned systematically. Chapter 4 proposes a method of constructing a cfg representing an assignment of data types to words in a natural language specification. The resulting cfg becomes a part of the cfg specifying the syntax of logical formulas in an algebraic specification. In our method, a cfg is mechanically constructed from sample sentences in a natural language specification, where the cfg represents type declarations of expressions and type hierarchy. Then, the cfg is appropriately modified by adding nonterminals/production rules that represent type inclusion relation. Finally, the cfg is simplified based on structural equivalence. The result of applying this method to a part of the OSI session protocol specification (39 sentences) is also presented. By using the cfg obtained by our method, we reduced an ambiguity in the natural language specification, which was not reduced by a cfg given manually.

Chapter 5 proposes a method of translating logical formulas, which are derived by the method in Chapter 3, into executable algebraic specifications called BE programs. A BE program specifies action sequences performed by a machine, called the BE interpreter, which has a finite number of registers and I/O buffers. In this method, for each predicate p in logical formulas, the meaning of p is given as a BE subprogram and stored as a lexical item of p. Then a BE program for the logical formulas is constructed in a bottom-up manner. The result of applying this method to the logical formulas derived from a part of the OSI session protocol specification (18 paragraphs, 45 sentences) is also presented. The behavior of the obtained BE programs was just as the human translator intended.

In this way a natural language specification of a communication protocol can be translated into an executable specification within a single framework of algebraic specification methods. Chapter 6 summarizes our research.

Acknowledgments

I am deeply indebted to many people for the advice, feedback and support they gave to me in the course of this work. I would especially like to thank Professor emeritus Tadao Kasami, currently Professor of Nara Institute of Science and Technology for his invaluable support, discussions and encouragement throughout the work.

I am grateful to my supervisor Professor Kenichi Taniguchi for his invaluable suggestions and discussions on the work. I am also obliged to Professor Mamoru Fujii and Professor Akihiro Hashimoto for their helpful comments and suggestions. I would like to thank Professor Hideo Miyahara, Professor Tohru Kikuno, and Associate Professor Teruo Higashino for their valuable comments. I am also thankful to Professor Minoru Ito of Nara Institute of Science and Technology for his invaluable comments and continuous encouragement.

I would like to thank Dr. Kazuhito Ohmaki of Electrotechnical Laboratory, Associate Professor Motoshi Saeki of Tokyo Institute of Technology, and Mr. Hisayuki Horai of Fujitsu Laboratories Ltd. for their valuable comments.

I am extremely thankful to Associate Professor Hiroyuki Seki of Nara Institute of Science and Technology for his invaluable discussions and great support throughout the work. I also thank to Mr. Jun Shimabukuro, Mr. Tetsuya Yagi, and Mr. Atsushi Ohsaki for their helpful discussions.

I would like to thank Associate Professor Toru Fujiwara, Associate Professor Toyoo Takata of Nara Institute of Science and Technology, Dr. Robert H. Morelos-Zaragoza, Research Associates Masahiro Higuchi, Ryuichi Nakanishi and Kozo Okano for their kind and helpful support. I am thankful to Research Associates Yuichi Kaji and Hajime Watanabe of Nara Institute of Science and Technology for their valuable support. I am also grateful to Ms. Machiko Uehara for her kind support.

Lastly, I would like to thank all the members of Kasami Laboratory of Osaka University and Ito Laboratory of Nara Institute of Science and Technology.

List of Publications

Journal Papers

[1] 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 Transactions on Information and Systems, Vol. E76-D, No. 12, pp. 1479–1489, Dec. 1993.

[2] Ishihara, Y., Seki, H. and Kasami, T., "Implementation of Natural Language Specifications of Communication Protocols by Executable Specifications," Transactions of Information Processing Society of Japan (to appear).

[3] Ishihara, Y., Ohsaki, A., Seki, H. and Kasami, T., "Assignment of Data Types to Words in a Natural Language Specification," IEICE Transactions on Information and Systems (submitted).

International Conferences

[4] Ishihara, Y., Seki, H. and Kasami, T., "A Translation Method from Natural Language Specifications into Formal Specifications Using Contextual Dependencies," Proceedings of the IEEE International Symposium on Requirements Engineering '93 (San Diego, California), pp. 232–239, Jan. 1993.

Workshops

[5] Ishihara, Y., Seki, H. and Kasami, T., "On a Translation from Natural Language Specifications of Communication Protocols into Algebraic Specifications in the Form of an Abstract Sequential Machine," IPSJ SIG Notes, SE-82-11, Dec. 1991 (in Japanese).

[6] Ohsaki, A., Ishihara, Y., Seki, H. and Kasami, T., "Generation of Signatures of Functions in Translation from Natural Language Specifications into Algebraic Specifications," IPSJ SIG Notes, SE-96-10, Jan. 1994 (in Japanese).

[7] Ishihara, Y., Seki, H. and Kasami, T., "An Algebraic Definition of a LOTOS-Like Language and Its Application," IPSJ SIG Notes, SE-99-1, July 1994.

Contents

Chapter 1

Introduction

In a software development process, informal requirements and/or specifications are often written in a natural language since they become readable and the intuitive meanings understandable. However, natural language specifications are apt to be incomplete. In order to verify whether a program satisfies a specification, the specification needs to be written formally. Therefore, it is desirable for a natural language specification to be translated into a formal one. Moreover, if derived formal specifications are executable, rapid prototyping techniques can be easily applied to natural language specifications.

Various researches on semi-automatic translation methods from natural language specifications into formal specifications are carried out [5], [22]. From the point of view of a human translator, a translation method should satisfy the following requirements:

(1) In a translation process, incompleteness of a natural language specification is detected or reduced;

(2) Information necessary for translation (e.g., lexical items) can be obtained as automatically as possible; and

(3) A derived formal specification is executable, or can be easily translated into an executable program.

This dissertation aims at developing, for specifications describing dynamic behavior of systems, a translation method which satisfies the above three requirements (1)–(3). It is important to develop a translation method for such specifications since many of practical specifications such as protocol specifications describe dynamic behavior of systems. However, few systematic translation method for such specifications have been proposed.

In this dissertation, we adopt natural language specifications of communication protocols as input specifications. In most of protocol specifications, protocol machines are modelled as sequential machines. Therefore, most of protocol specifications contain only sequential or conditional descriptions, i.e., "complex" execution control such as repetition, recursion, etc. is not used. Because of this simplicity, we can develop a translation method satisfying above (1)–(3) by solving only the essential problems in (1)–(3).

As a formal specification language, algebraic specification language ASL [13] is adopted 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 with arbitrary structure and arbitrary degree of abstraction.

An algebraic specification in ASL is a pair $SPEC = (G, AX)$ of a context-free grammar (cfg) G and a set AX of axioms. G specifies the set of expressions and their syntax, and AX specifies their semantics. Chapter 2 gives the formal definition of ASL.

Among the above three requirements, main difficulty in the translation lies in many kinds of incompleteness of natural language specifications. Balzer et al. [1] classify the incompleteness of natural language specifications into partial sequencing, missing operand, incomplete reference, scope of conditionals, and so on. They claim that the incompleteness in three example specifications (27 sentences all together) are successfully handled by their prototype system. To reduce the incompleteness, they use some criteria for program well-formedness. For example, to reduce the ambiguity in scope of conditionals, it uses a criterion that, for each case of conditionals, there exists an input data which makes the case true. Although five criteria are presented in Ref. [1], how the incompleteness is systematically handled by their approach is not described in detail.

In Chapter 3, a systematic method of reducing incompleteness of a natural language specification of a communication protocol is proposed. In a protocol specification, a sentence often specifies an action which a protocol machine (program) has to perform, and as illustrated in the following example, it often specifies implicitly the state of the protocol machine at which the described action has to be performed.

Example 1.1: Consider the following consecutive sentences in Ref. [10]:

(E1) A valid incoming MAJOR SYNC POINT SPDU (with ...) results in an S-SYNC-MAJOR indication.

(E2) 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 (E2) immediately after it performs the actions specified by (E1). However, sentence (E2) does not specify explicitly when the actions has to be performed. □

In this dissertation, a state of the protocol machine 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. When the pre-situation of X_2 and the post-situation of X_1 indicate the same situation, we say "X_2 S-depends on X_1." For example, (E2) S-depends on (E1) since the pre-situation of (E2) is equal to the post-situation of (E1). The pre-/post-situations are also defined for a sequence of sentences.

In general, a constituent X does not necessarily S-depend on the constituent appearing immediately before X. S-dependency among constituents is analyzed based on the following properties and information:

(a) properties of the syntax of natural languages,

(b) type information assigned to words in the input natural language specification, and

(c) properties derived from the axioms on data types.

These properties and information are assumed to be given and stored in a dictionary. After the analysis of S-dependency, algebraic axioms in the form of logical formulas are generated.

According to the proposed analysis method, we implemented a prototype system. By using this system, the S-dependency in a part of the OSI session protocol specification (29 paragraphs, 98 sentences) was analyzed. For 95 sentences, the system uniquely determined the S-dependency using only (a) and (b) described above. By using (c) in addition, the S-dependency of the remaining three sentences was uniquely determined. Thus, this method is effective in reducing incompleteness of natural language specifications of communication protocols.

As for the above requirement (2), it is time-consuming to construct lexical items (dictionary) manually. Saeki et al. [17] developed a pseudo natural language called TELL/NSL. By a method called lexical decomposition, they analyze sentences written in TELL/NSL in which the meanings of words are defined. In an input specification, each word has to be defined by built-in words of TELL/NSL, but they do not consider the change of the vocabulary. The vocabulary used for writing specifications varies according to the problem domain. Hence, it is unrealistic to fix the vocabulary or built-in words. In this dissertation, we let the lexical items and the syntax of a natural language be definitely separated from the translation method. Then, if necessary, a human translator can expand the vocabulary and the syntax of a natural language.

In the method proposed by Chapter 3, type information assigned to words is assumed to be given and stored in a dictionary. In Chapter 4, a method of constructing a cfg representing an assignment of data types to words in a natural language specification is proposed. When a natural language specification is translated into algebraic axioms, the cfg representing an assignment becomes a part of the cfg specifying the syntax of expressions in the algebraic axioms.

In ASL, each data type is represented by a nonterminal, and subtype relation is defined based on the derivation relation between nonterminals. In our construction method, each of sample sentences in a natural language specification is parsed based on a given grammar of the natural language. Next, from each parse tree, a set of "expressions" is generated, where, roughly speaking, each verb corresponds to a prefix function symbol and each noun corresponds to a variable of axioms. A set of nonterminals and a set of production rules to generate those "expressions" are mechanically constructed. Then a syntactic analysis of the union *EXP* of the sets of "expressions" is performed. That is, for each function symbol f, each argument position i of f, and each sub-expression *exp* of an expression in *EXP*, it is examined whether *exp* appears as the i-th argument of f in *EXP*. The result of the syntactic analysis is represented as subtype relation (i.e., derivation relation between nonterminals) to be satisfied by the cfg to be derived. Next, in a semantic analysis, a human analyzer considers the meanings of the words in the natural language specification, and then

- groups some data types (i.e., nonterminals) together and introduces a new supertype of them, and
- augments the subtype relation to be satisfied.

Finally, a cfg which meets the result of the analyses is constructed, and then simplified based on structural equivalence [20].

According to the proposed construction method, we implemented a prototype system. By using this system, a part of the OSI session protocol specification (39 sentences) was analyzed. By using the cfg obtained by our method, we reduced an ambiguity in the natural language specification, which was not reduced by a cfg given manually.

As for the above requirement (3), it is also difficult to translate a natural language specification into a program directly since a natural language specification does not necessarily define the operational semantics of all words appearing in it. To fill the gap, a human translator has to give the operational semantics of all words in the natural language specification. Ichikawa et al. [9] proposed a translation method from TELL/NSL specifications into Prolog programs. In their method, an input specification is translated into Horn clauses. Then, the order of the literals and clauses are semi-automatically modified so that the obtained Horn clauses become an executable Prolog program. That is, a human translator gives the operational semantics of words by modifying the order of the literals and clauses. However, they do not consider translating specifications that describe dynamic behavior of systems. On the other hand, the method of Seki et al. [18] translates a natural language specification into an algebraic specification [4], [13] whose axioms are in the form of logical formulas, but they do not consider translating such logical formulas into executable programs.

In Chapter 5, a method of translating logical formulas, which are derived by the method in Chapter 3, into executable algebraic specifications is proposed. In the method proposed in

Chapter 3, each word specifying actions in a natural language specification is translated into a predicate. Then the valid sequences of actions defined by the specification are represented by axioms in the form of logical formulas. To develop a translation method from such logical formulas into an executable specification, a model of protocol machines must be defined so that a human translator can give the operational semantics of the logical formulas.

A natural language specification of a communication protocol, such as Ref. [10], often assumes that a protocol machine has registers. In Chapter 5, we define an interpreter (machine), called the BE interpreter, as a model of protocol machines. The BE interpreter has a finite number of registers and unbounded I/O buffers, and performs three kinds of atomic actions: (1) input from a buffer, (2) output to a buffer, and (3) calculation using its registers. An input program for the BE interpreter, called a BE program, specifies the order of actions by means of operators such as action-prefix, choice, conditional, and so on. The syntax of BE programs is defined within the framework of algebraic specifications. The semantics of BE programs, i.e., the behavior of the BE interpreter, is defined by axioms based on a state transition model. Therefore, the BE interpreter specification with a given BE program can be easily compiled into an executable program.

Adopting the BE interpreter as a model of protocol machines, Chapter 5 proposes a method of translating logical formulas into BE programs. The operational semantics of each predicate is given as a BE subprogram. Then, a BE program for logical formulas is constructed in a bottom-up manner from BE subprograms for the predicates.

According to the translation method, we implemented a prototype system. By using this system, the logical formulas derived from a part of the OSI session protocol specification (18 paragraphs, 45 sentences) are translated into BE programs. We also implemented a simulator which executes a given BE program. For the above BE programs obtained by the system, the simulator behaved just as the user intended.

In this way a natural language specification of a communication protocol can be translated into an executable specification within a single framework of algebraic specification methods.

Chapter 2

Algebraic Specification Language ASL

As stated in Chapter 1, this dissertation adopts algebraic specification language ASL [13] as a formal specification language. A specification in ASL is a pair $SPEC = (G, AX)$ of a context-free grammar (cfg) G and a set AX of axioms. G specifies a 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 production rules respectively. Let $A \underset{G}{\Rightarrow} \alpha$ denote that $A \in N$ generates $\alpha \in (N \cup T)^*$ by one-step derivation in G. Let $\overset{*}{\underset{G}{\Rightarrow}}$ be the reflexive and transitive closure of $\underset{G}{\Rightarrow}$. Moreover, let $L_G[A] = \{w \in T^* \mid A \overset{*}{\underset{G}{\Rightarrow}} w\}$, and let $L_G = \bigcup_{A \in N} L_G[A]$. An element in L_G is called an expression (in the specification $SPEC$). N corresponds to the set of sorts (data types); A nonterminal A is sometimes called "data type A" and an expression in $L_G[A]$ may be called "an expression of type A." For $A, A' \in N$, we say that A is a subtype of A' (or A' is a supertype of A) if $A' \overset{*}{\underset{G}{\Rightarrow}} A$. T corresponds to the set of function symbols. P corresponds to the set of signatures of functions and definitions of subtype relation.

An axiom is a pair $exp == exp'$ 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 $A_{\bar{x}}$ is associated (declared by "$\bar{x} : A_{\bar{x}}$" in $SPEC$), and an arbitrary expression in $L_G[A_{\bar{x}}]$ can be substituted into $\bar{x}$. The least congruence relation that satisfies all the axioms in AX is denoted by $\equiv_{SPEC}$. See Ref. [13] for details.

Throughout this dissertation, 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) is assumed. It is also assumed that $SPEC_0$ supports the following data types:

1. Boolean; Let Bool be a nonterminal which generates Boolean expressions.

2. Sequence; Let Seq_ be a constructor on data types to support sequences of a given data type, i.e., for any data type A, Seq_A generates sequences of expressions of type A. Formally, $SPEC_0$ has the production schemata and axioms shown in Table 2.1, where $\lambda, \cdot, \mathsf{head}, \mathsf{tail}, \mathsf{member} \in T_0$. Constant function λ denotes the empty sequence and function "$\cdot$" denotes the concatenation operation. For a given sequence, head returns the first element and tail returns the sequence obtained by eliminating the first element. Predicate member is true if and only if the first parameter is an element of the second parameter.

Table 2.1 Specification of sequences.

- Production schemata:

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

- Axioms:

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

Chapter 3

Analysis of Contextual Dependencies in Natural Language Specifications of Communication Protocols

3.1 Introduction

In Chapter 1, the problem of reducing incompleteness of a natural language specification is considered as the one of analyzing S-dependency of the specification. In this chapter, the part which defines valid sequences of operations of the protocol machine in the OSI session protocol specification [10] (265 sentences), denoted $SPEC_{\mathrm{OSI}}$, is adopted as a translation example, and a method of analyzing S-dependency in a natural language specification is proposed.

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 the paragraph). 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 the form of logical formulas.

The proposed analysis method consists of the following two phases:

(i) Sentence analysis; Each sentence S is analyzed independently and the pre-situation of each constituent of S is represented as a value relative to the pre-situation of the whole sentence S.

(ii) Context analysis; S-dependency among constituents of different sentences is analyzed based on the following properties and information:

  (a) properties of the syntax of natural languages,

(b) type information assigned to words in the input natural language specification, and

(c) properties derived from the axioms on data types.

The pre-situation of each constituent is represented as a value relative to the pre-situation of the paragraph.

As an example for the analysis system which we implemented, we concentrate on the main part of the OSI session protocol specification, denoted $SPEC_{\mathrm{MAIN}}$ in the rest of this chapter. $SPEC_{\mathrm{MAIN}}$ covers kernel, half-duplex, duplex, minor synchronize and major synchronize functional units, and it defines valid sequences of actions performed by the session protocol machine. In $SPEC_{\mathrm{MAIN}}$, there are 98 sentences, which form 29 paragraphs; The number of the different words is 251 and each sentence consists of 5 to 50 words (the mean value is about 15 words). The lexical items for $SPEC_{\mathrm{MAIN}}$ were constructed by analyzing the part of Ref. [10] which defines the data types or the meanings of the terms used in $SPEC_{\mathrm{MAIN}}$.

Table 3.1 shows two paragraphs Γ_1: (S1)–(S7) and Γ_2: (S8)–(S10) in $SPEC_{\mathrm{MAIN}}$. A noun phrase between brackets is treated as a single word in this paper since it is a term defined within the OSI specifications to denote a protocol data unit, a service primitive, etc. We will use abbreviations such as "MAP" for "MAJOR SYNC POINT SPDU" or "SSYN-Mind" for "S-SYNC-MAJOR indication," according to the abbreviations defined in the OSI specifications.

This chapter is organized as follows. Section 3.2 presents the framework of analysis of S-dependency. In Sections 3.3 and 3.4, the sentence analysis (phase (i) above) and the context analysis (phase (ii) above) are described respectively. Section 3.5 describes an analysis system implemented according to the proposed method. Section 3.6 summarizes this chapter.

3.2 Framework of S-Dependency Analysis

As mentioned in Chapter 1, pre-/post-situations are often specified implicitly in a protocol specification. To express pre-/post-situations formally in an algebraic specification, data types Event, Seq_Event and Situation are introduced. An expression of type Event denotes an atomic action of a protocol machine such as a transmission of data or an update of a particular register. An expression e of type Event is simply called "event e." Seq_Event is a type of sequence of events which is generated by type Event with the constructor Seq_ (see Chapter 2). An expression of type Situation is either a constant function of type Situation or $\Delta(\sigma, e)$, where σ and e are expressions of type Situation and Event respectively. $\Delta(\sigma, e)$ denotes the situation immediately after event e occurs at situation σ. Each pre-/post-situation is expressed by an expression of type Situation.

Table 3.1 Some Paragraphs of $SPEC_{MAIN}$.

Paragraph Γ_1:

(S1) An [S-CONNECT (reject) response] results in a [REFUSE SPDU].

(S2) This SPDU is sent on the [transport normal flow].

(S3) No [session connection] is established.

(S4) If the [Transport Disconnect parameter] indicates that the [transport connection] can be reused, the SPM waits for a [CONNECT SPDU].

(S5) Otherwise the SPM starts the timer, TIM, and waits for a [T-DISCONNECT indication].

(S6) If the timer expires before receipt of a [T-DISCONNECT indication], the SPM requests [transport disconnection] with a [T-DISCONNECT request].

(S7) The timer is cancelled on receipt of a [T-DISCONNECT indication].

Paragraph Γ_2:

(S8) A valid incoming [MAJOR SYNC POINT SPDU] (with received [serial number] equal to V(M)) results in an [S-SYNC-MAJOR indication].

(S9) If Vsc is false, V(A) is set equal to V(M).

(S10) V(M) is incremented by one.

The relation between expressions of type Situation is represented by a predicate $\doteq$ which takes two arguments of type Situation. If $\sigma \doteq \sigma'$ is true, σ and σ' denote the same situation. We extend Δ to apply to a situation and a sequence of events as follows:

$$\begin{aligned} \Delta(\sigma, \lambda) &\doteq \sigma, \\ \Delta(\sigma, e_{seq} \cdot e) &\doteq \Delta(\Delta(\sigma, e_{seq}), e). \end{aligned}$$

Example 3.1: By using types Event, Seq_Event, Situation and other primitive types, the two sentences in Example 1.1 are translated into an axiom of F == true, where F is the following formula in a first-order predicate logic (Existential quantifiers are defined in the framework of ASL by transforming them into functions similar to Skolem functions [19]):

$\bar{\sigma}_1, \bar{\sigma}'_1, \bar{\sigma}''_1, \bar{\sigma}_2, \bar{\sigma}'_2, \bar{\sigma}''_2, \bar{\sigma}'''_2$: Situation
$\bar{x}_1$: SPDU $\bar{x}_2$: SSprm
$\forall\bar{\sigma}_1\forall\bar{x}_1\exists\bar{\sigma}''_1\exists\bar{x}_2\exists\bar{\sigma}'_1\exists\bar{\sigma}_2\exists\bar{\sigma}'_2\exists\bar{\sigma}''_2\exists\bar{\sigma}'''_2$
(T1) [valid($\bar{x}_1$) $\wedge$ incoming($\bar{x}_1$) $\wedge$ MAP($\bar{x}_1$) $\supset$
  [SSYNMind($\bar{x}_2$) $\wedge$
  [receive($\bar{x}_1, \bar{\sigma}_1, \bar{\sigma}''_1$) $\wedge$ send($\bar{x}_2, \bar{\sigma}''_1, \bar{\sigma}'_1$)] $\wedge$
  [$\bar{\sigma}'_1 \doteq \Delta(\bar{\sigma}_1$, in($\bar{x}_1$) $\cdot$ out($\bar{x}_2, \bar{x}_3$))] $\wedge$
(R1–2) [$\bar{\sigma}_2 \doteq \bar{\sigma}'_1$] $\wedge$
(T2) [if_then(Vsc = false,
    set_equal_to(Va, Vm, $\bar{\sigma}''_2, \bar{\sigma}'''_2$),
    $\bar{\sigma}_2, \bar{\sigma}'_2$)]]].

We use $\bar{\sigma}_1$, $\bar{\sigma}_2$,... as variables to denote pre-situations and $\bar{\sigma}'_1$, $\bar{\sigma}'_2$,... as variables to denote post-situations unless otherwise stated.

In the above formula, SPDU is a data type which denotes data units transmitted and received by protocol machines, and SSprm is a data type which denotes service primitives provided by a protocol machine to its user. Sub-formulas (T1) and (T2) correspond to sentences (E1) and (E2) respectively. (R1–2) states that the pre-situation of (E2) is equal to the post-situation of (E1). Intuitive meanings of subexpressions in the formula are shown in Table 3.2. □

3.3 Analysis of S-Dependency among Constituents of a Sentence

In Refs. [18] and [21], an I-structure which represents local (not contextual) information on a constituent was introduced. S-dependency among constituents of a sentence is analyzed by constructing the I-structure of the sentence.

Table 3.2 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($\bar{x}_2$): $\bar{x}_2$ is a service primitive S-SYNC-MAJOR indication.
- receive($\bar{x}_1, \bar{\sigma}_1, \bar{\sigma}_1''$): At situation $\bar{\sigma}_1$, the event "receipt of $\bar{x}_1$" is allowed to occur and the situation immediately after the event is $\bar{\sigma}_1''$.
- send($\bar{x}_2, \bar{\sigma}_1'', \bar{\sigma}_1'$): At situation $\bar{\sigma}_1''$, the event "sending $\bar{x}_2$" has to occur and the situation immediately after the event is $\bar{\sigma}_1'$.
- set_equal_to(Va, Vm($\bar{\sigma}_2$), $\bar{\sigma}_2'', \bar{\sigma}_2'''$): At situation $\bar{\sigma}_2''$, 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 $\bar{\sigma}_2'''$.
- if_then(q, *pred*, $\bar{\sigma}_2, \bar{\sigma}_2'$): At situation $\bar{\sigma}_2$, the events specified by *pred* occur if q is true, and no events occur otherwise. The situation immediately after these events is $\bar{\sigma}_2'$.

An I-structure is a parse tree each node of which is labeled with a structure called C-structure, which is resembling a category of HPSG [16]. A C-structure is a finite set of pairs of a feature and its value such that for any two pairs $\langle f_1, v_1 \rangle$ and $\langle f_2, v_2 \rangle$ in the set, if $f_1 = f_2$ then $v_1 = v_2$. For a C-structure C and a feature f, Let $C.f$ be v if $\langle f, v \rangle \in C$, and undefined otherwise. If v is a sequence, $v\#i$ represents the i-th value of v. For a constituent X, let $CR[X]$ denote the C-structure which is the label of the root of the I-structure of X. The I-structure of a word is defined and stored as (a part of) the lexical item in the dictionary, and the I-structure of a constituent is constructed in a bottom-up manner.

In this dissertation, we introduce new features **pre**, **post** and **event** defined as follows. For a constituent X, $CR[X]$.**pre** and $CR[X]$.**post** denote the pre- and post-situations of X respectively. Actually, the value is an index $\sigma\uparrow$ of type Situation in a C-structure. The same indices of type A denote the same expression of type A. When the axiom is generated, every index of type Situation is replaced by a variable of type Situation. $CR[X]$.**event** is a set of expressions of type Seq_Event such that

$$\bigvee_{e_{\mathrm{seq}} \in CR[X].\mathbf{event}} (CR[X].\mathbf{post} \doteq \Delta(CR[X].\mathbf{pre}, e_{\mathrm{seq}})) \equiv_{SPEC_{\mathrm{LF}}} \mathsf{true},$$

where $SPEC_{\mathrm{LF}}$ is the algebraic specification into which the natural language specification is translated. When the I-structure of a sentence S is constructed, the pre-situation of each constituent of S denoting an event is represented as a value relative to the pre-situation of the whole sentence S.

Example 3.2: We show how the S-dependency among the constituents of a sentence S = "X_1 results in X_2" is analyzed. For a constituent X, let $pre[X]$ denote the pre-situation of X, and $post[X]$ denote the post-situation of X.

To analyze the sentence S, the I-structure of "results" for phrase "results in" must be defined in the dictionary (see Fig. 3.1). By analyzing the function of "results in" in $SPEC_{\mathrm{OSI}}$, it is found out that S always means "on receipt of data X_1, the SPM has to transmit data X_2" in $SPEC_{\mathrm{OSI}}$ [23]. Therefore, for example, phrases "a valid incoming MAP" and "an SSYNMind" as constituents of (E1) specify events, although they just stand for the names of data by themselves.

Let C denote the C-structure in Fig. 3.1. C.**args**#1 and C.**args**#2 (and also C.**subcat**#1 and C.**subcat**#2) correspond to X_1 and X_2 respectively. By C.**subcat**, indices $x_1\uparrow$ and $x_2\uparrow$ are supposed to denote the expressions representing the meanings of X_1 and X_2 by themselves respectively. The values of **pre**, **post** and **event** in C, C.**args**#1 and C.**args**#2 claim that

- $pre[X_1] \doteq pre[S]$,
- $pre[X_2] \doteq post[X_1] \doteq \Delta(pre[X_1], \mathsf{in}(x_1\uparrow)) \doteq \Delta(pre[S], \mathsf{in}(x_1\uparrow))$, and

- $post[S] \doteq post[X_2] \doteq \Delta(pre[X_2], \mathsf{out}(x_2\uparrow, x_3\uparrow))$
  $\doteq \Delta(pre[S], \mathsf{in}(x_1\uparrow) \cdot \mathsf{out}(x_2\uparrow, x_3\uparrow))$.

Thus, we can analyze S-dependency in any sentence in the form of "X_1 results in X_2" in $SPEC_{\mathrm{OSI}}$. □

The value of feature **trans** is an expression representing the meaning of the constituent in terms of ASL. Let $C = CR[X]$ for some constituent X specifying events. Then, C.**pre**, C.**post**, C.**event** and C.**trans** are related as follows (see also the explanation of (T1) and (T2) in Example 3.1):

1. If C.**event** is a singleton and the element e is an incoming event (e.g., receipt of data or expiration of a timer), C.**trans** represents that e is allowed to occur at situation C.**pre** and the situation immediately after it occurs is C.**post**.

2. Similarly, if C.**event** is a singleton and the element e is an outgoing event (e.g., transmission of data or cancellation of a timer), C.**trans** represents that e has to occur at situation C.**pre** and the situation immediately after it occurs is C.**post**.

3. If C.**event** is a singleton and the element is a sequence of events $e_1 \cdots e_n$ with $n \geq 2$ (e.g., sentence (E1) in Example 1.1), then C.**trans** = $p_1 \wedge \cdots \wedge p_n$, where p_i $(1 \leq i \leq n)$ represents that e_i is allowed to (or has to) occur at situation σ_{i-1} and the situation immediately after it occurs is σ_i, $\sigma_0 = C$.**pre** and $\sigma_n = C$.**post**.

4. If C.**event** is not a singleton, that is, C.**event** = $\{e_{\mathrm{seq}1}, \ldots, e_{\mathrm{seq}m}\}$ for some $m \geq 2$, then C.**trans** = $q_1 \vee \cdots \vee q_m$, where q_i $(1 \leq i \leq m)$ is the predicate determined by applying the above 3 to $e_{\mathrm{seq}i}$.

Let X be a constituent of a sentence S. Let $CS_S[X]$ denote

$$CR[S].\mathbf{args}\#i_1.\mathbf{args}\#i_2.\ldots.\mathbf{args}\#i_n$$

which corresponds to X. If S is obvious from the context, we write just $CS[X]$. As illustrated in Example 3.2, $CS_S[X]$ represents information on the actually given X as a constituent of S.

3.4 Analysis of S-Dependency among Constituents of Different Sentences

3.4.1 Definitions

For a paragraph, we define the S-dependency graph of the paragraph, which is a digraph to represent the relation among the pre-/post-situations of the constituents in the paragraph.

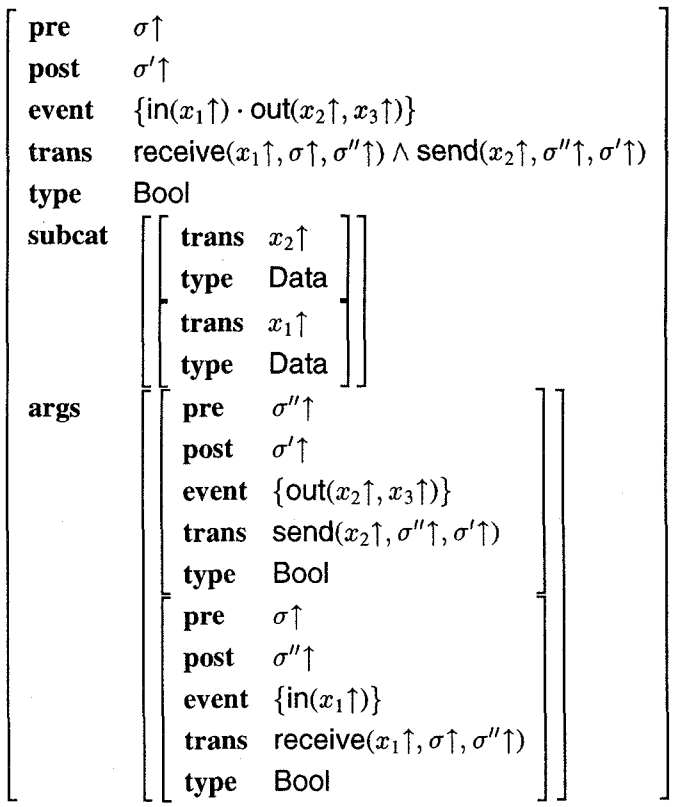

Fig. 3.1 *CR*["results"] for phrase "results in".

Definition 3.1: The S-dependency graph of a paragraph P is a digraph $G = (V, E)$ such that

1. V has a one-to-one correspondence to $\{CS[X].\textbf{pre}, CS[X].\textbf{post} \mid X$ is a constituent which appears in $P\}$; and
2. $E = \{v_1 \xrightarrow{e} v_2 \mid$ There exists a constituent X in P such that (a) v_1 and v_2 are nodes in V which correspond to $CS[X].\textbf{pre}$ and $CS[X].\textbf{post}$ respectively, and (b) $CS[X].\textbf{event} = \{e\}$ where e is a single event (hence, neither a null sequence nor a sequence of events of length ≥ 2)$\}$.

The node which corresponds to the pre-situation of P is called the initial node. □

Let $G = (V, E)$ be an S-dependency graph. For $v \in V$, let $\bar{\sigma}_v$ be a variable of type Situation which expresses the pre-/post-situation corresponding to v. The quantifier of $\bar{\sigma}_v$ is universal if v has no incoming arcs, and existential otherwise. An arc $v \xrightarrow{e} v'$ means that

$$(\bar{\sigma}_{v'} \doteq \Delta(\bar{\sigma}_v, e)) \equiv_{SPEC_{\mathrm{LF}}} \mathsf{true},$$

where $SPEC_{\mathrm{LF}}$ is the algebraic specification into which the natural language specification is translated. If v' has n incoming arcs $v_1 \xrightarrow{e_1} v', \ldots, v_n \xrightarrow{e_n} v'$, then the arcs claim that

$$\bigvee_{i=1}^{n} (\bar{\sigma}_{v'} \doteq \Delta(\bar{\sigma}_{v_i}, e_i)) \equiv_{SPEC_{\mathrm{LF}}} \mathsf{true}.$$

A paragraph P is translated into an axiom in the form of

$$\begin{aligned} &\bar{x}_1 : A_1, \ldots, \bar{x}_m : A_m \\ &\quad Q_1 \bar{x}_1 \cdots Q_m \bar{x}_m [p_1 \gg [\cdots p_m \gg [F_0] \cdots]] \;==\; \mathsf{true}, \end{aligned} \tag{3.1}$$

where F_0 is a logical formula without quantifiers, $\bar{x}_1, \ldots, \bar{x}_m$ are all the distinct variables appearing in F_0, and A_j $(1 \leq j \leq m)$ is the data type of $\bar{x}_j$. We mean by $p_j \gg F$ that $p_j(\bar{x}_1, \ldots, \bar{x}_j) \wedge F$ if $Q_j = \exists$, and $p_j(\bar{x}_1, \ldots, \bar{x}_j) \supset F$ if $Q_j = \forall$. Each p_j and A_j in Eq. (3.1) are determined by the values of features **restriction** (whose value represents conditions to be satisfied by variables) and **type** respectively [18].

Let $G = (V, E)$ be the S-dependency graph of $P = S_1 \cdots S_l$. The formula F_0 is constructed as follows:

$$F_0 \triangleq TRANS(G) \wedge \left(\bigwedge_{k=1}^{l} CR[S_k].\textbf{trans} \right).$$

Now we define $TRANS(G)$. If $v \in V$ has no incoming arcs, then

$$TRANS\_V(v) \triangleq \mathsf{true}.$$

Suppose that $v \in V$ has n_v incoming arcs $v_1 \xrightarrow{e_1} v, \ldots, v_{n_v} \xrightarrow{e_{n_v}} v$. Let $\bar{\sigma}_v$ and $\bar{\sigma}_{v_i}$ $(1 \leq i \leq n_v)$ be a variable of type Situation corresponding to v and v_i respectively. Then,

$$TRANS\_V(v) \triangleq \bigvee_{i=1}^{n_v} (\bar{\sigma}_v \doteq \Delta(\bar{\sigma}_{v_i}, e_i)).$$

TRANS(G) is defined as follows:

$$TRANS(G) \triangleq \bigwedge_{v \in V} TRANS\_V(v).$$

3.4.2 Analysis Based on Syntax of a Natural Language

The S-dependency is

1. explicitly specified by special words and phrases (we call them S-dependency controllers), and

2. implicitly specified by the order of sentences in the paragraph.

An S-dependency controller is a conditional, an anaphoric phrase, a conjunctive adverb, or one of their equivalents. For example, conditionals associated with each other (see Example 3.3 and 3.4) specify that the pre-situations of the sentences including one of the conditionals are the same. Such an anaphoric phrase as "the former" or "the latter" restricts the set of candidates for the constituent on which the sentence including the anaphoric phrase S-depends. Such a conjunctive adverb as "then" specifies that the sentence including the conjunctive adverb S-depends on the previous sentence. In general, S-dependency controllers can introduce discontinuous structure into the S-dependency. On the other hand, the order of the sentences gives continuous structure to the S-dependency as far as it is consistent with the structure specified by S-dependency controllers. This section summarizes these properties formally as the following properties of an S-dependency graph.

First, we examine conditionals.

Example 3.3: Consider the following paragraph:

(E3) If the Transport Disconnect parameter indicates that the transport connection can be reused, the SPM waits for a CONNECT SPDU.

(E4) V(M) is incremented by one.

(E5) Otherwise the SPM starts the timer, TIM, and waits for a T-DISCONNECT indication.

The word "otherwise" in (E5) is associated with the word "if" in (E3) since the "if" precedes the "otherwise" and is the closest to it. Therefore (E3) and (E5) are conditionals under the same situation. □

Example 3.4: Consider the following paragraph:

(E6) If Vsc is false, V(A) is set equal to V(M).

(E7) If the Transport Disconnect parameter indicates that the transport connection can be reused, the SPM waits for a CONNECT SPDU.

(E8) Otherwise the SPM starts the timer, TIM, and waits for a T-DISCONNECT indication.

The word "otherwise" in (E8) is associated with the word "if" in (E7) as mentioned in Example 3.3. However, the syntax of natural languages can not determine whether the "if" in (E7) is associated with the "if" in (E6) or not. □

Property 3.1: Let $S_1 \cdots S_l$ be a sequence of sentences including conditionals associated with each other in the above sense, where S_i precedes S_j in the paragraph if $i < j$. Then, for i $(2 \leq i \leq l)$, $pre[S_i] \doteq pre[S_1]$ holds. The pre-situation of S_1 will be determined by another S-dependency controller or the order of sentences. □

Next, we examine anaphoric phrases.

Example 3.5: Consider sentence (S6) in paragraph Γ_1 of Table 3.1. The anaphoric noun phrase "the timer" refers to "the timer, TIM" in (S5), and can not refer to any constituents in (S1)–(S4). Therefore (S6) does not S-depend on any constituents in (S1)–(S4). □

Property 3.2: Let S be a sentence with an anaphoric phrase PH in it, and v be the node of the S-dependency graph which corresponds to $pre[S]$. Then, on every path from the initial node to v, there exists at least one node which corresponds to the pre-situation of the constituent including the antecedent of PH and specifying a single event. □

Lastly, we consider the order of the sentences in a paragraph.

Example 3.6: Consider the following paragraph:

(E9) If the Transport Disconnect parameter indicates that the transport connection can be reused, the SPM waits for a CONNECT SPDU.

(E10) Otherwise the SPM starts the timer, TIM, and waits for a T-DISCONNECT indication.

(E11) V(M) is incremented by one.

It is ambiguous whether V(M) is incremented by one only when the condition stated in if-clause of (E9) does not hold, or it is incremented independently of the condition. However, it is probable that this paragraph does not specify that V(M) is incremented only when the condition in (E9) holds. □

Now we introduce a reduced S-dependency graph for concise representation of the continuous structure.

Definition 3.2: For an S-dependency graph $G = (V, E)$, the reduced S-dependency graph (of G) is a digraph $G' = (V', E')$ satisfying the following conditions:

1. V' consists of every node in V that corresponds to $pre[S]$ for some sentence S;
2. For any $v_1, v_2 \in V'$, $v_1 \rightarrow v_2 \in E'$ if and only if there exists a path from v_1 to v_2 in G and no nodes on the path except v_1 and v_2 belong to V'. □

Property 3.3: Let $G' = (V', E')$ be the reduced S-dependency graph of a paragraph. Suppose that a sentence S satisfies the following conditions:

(a) S is not the first sentence in the paragraph; and

(b) S includes no S-dependency controllers or includes conditionals but they are not associated with any other conditionals preceding them.

Let v be the node in G' corresponding to the pre-situation of S. Then, v has n incoming arcs $v_1 \rightarrow v, \ldots, v_n \rightarrow v$ for some $n \geq 1$, and the following conditions hold:

1. Let S_i $(1 \leq i \leq n)$ be the sentence whose pre-situation corresponds to v_i. For any i, S_i precedes S in the paragraph and v_i has no outgoing arcs except $v_i \rightarrow v$.
2. Moreover, there exists i such that S immediately follows S_i in the paragraph. □

3.4.3 Analysis Based on Data Types Assigned to Words

For expressions including variables, unification [8] (denoted by $\bar{\wedge}$) is useful in checking the possibility that the expressions denote the identical object. In analysis of S-dependency, unification is used for detecting those constituents which indicate the same event.

Example 3.7: Consider sentences (S1) and (S2) in paragraph Γ_1 of Table 3.1. In these sentences, constituents specifying single events are

X_1 = "an S-CONNECT (reject) response,"
X_2 = "a REFUSE SPDU," and
X_3 = "this SPDU is sent on the transport normal flow,"

where "this SPDU" in X_3 refers to X_2. That is, X_3 specifies the flow ("the transport normal flow") to refine the event "output of a REFUSE SPDU" indicated by X_2. Therefore $pre[X_3]$ is not $post[X_2]$ but $pre[X_2]$.

Let $x_2\uparrow$ be an index of type SPDU denoting "a REFUSE SPDU," TNF be an expression of type Flow denoting "the transport normal flow," and out(d, f) be an expression of type Event denoting the event "sending a data unit d on the flow f." Then, the expression e_2 which denotes the event indicated by X_2 is

$$e_2 = \mathsf{out}(x_2\uparrow, x_3\uparrow), \tag{3.2}$$

where $x_3\uparrow$ is an index of type Flow representing that the second parameter of out is unknown. On the other hand, the expression e_3 which denotes the event indicated by X_3 is

$$e_3 = \mathsf{out}(x_2\uparrow, \mathsf{TNF}), \tag{3.3}$$

since "this SPDU" refers to "a REFUSE SPDU." From Eqs. (3.2) and (3.3), the value of $e_2 \barwedge e_3$ is defined. Hence it is possible that X_2 and X_3 specify the same event. □

Suppose that sentence S_2 S-depends on some constituent of sentence S_1 and that the properties of the syntax of natural languages stated above can not decide which constituent S_2 S-depends on. If there are constituents X_1 in S_1 and X_2 in S_2 which specify events and satisfy Condition 1 stated below, then it is possible that the event specified by X_2 is equal to that specified by X_1. Otherwise (i.e., no constituents satisfy Condition 1), $post[S_1]$ is selected as the first candidate for $pre[S_2]$. We analyzed $SPEC_{\mathrm{MAIN}}$ using the translation system based on this method, and all of the first candidates were correct.

Condition 3.1: The value of C_1.**event** $\barwedge$ C_2.**event** is defined, where $C_1 = CS_{S_1}[X_1]$ and $C_2 = CS_{S_2}[X_2]$. □

The translation system lists every pair of constituents X_1 and X_2 satisfying Condition 1, and asks a human translator whether the event specified by X_1 is equal to that specified by X_2. If the human translator answers in the affirmative, both C_1.**event** and C_2.**event** are replaced by C_1.**event** $\barwedge$ C_2.**event**.

3.4.4 Analysis Based on Axioms Specifying Data Types

Let $SPEC'_{\mathrm{LF}} = (G'_{\mathrm{LF}}, AX'_{\mathrm{LF}})$ be an algebraic specification obtained by analyzing S-dependency in a natural language specification. $SPEC'_{\mathrm{LF}}$ is incomplete since, usually, it does not specify properties specific to the problem domain of the specification. To obtain a complete algebraic specification $SPEC_{\mathrm{LF}} = (G_{\mathrm{LF}}, AX_{\mathrm{LF}})$, an algebraic specification $SPEC_{\mathrm{DOM}} = (G_{\mathrm{DOM}}, AX_{\mathrm{DOM}})$ specifying such properties must be added to $SPEC'_{\mathrm{LF}}$, where $G_{\mathrm{LF}} = G'_{\mathrm{LF}} \cup G_{\mathrm{DOM}}$ and $AX_{\mathrm{LF}} = AX'_{\mathrm{LF}} \cup AX_{\mathrm{DOM}}$.

Suppose that AX'_{LF} includes axioms which represent ambiguity of S-dependency, e.g., $(\sigma_i \doteq \sigma_j) \vee (\sigma_i \doteq \sigma_k) == \mathsf{true}$. It is expected that this ambiguity is reduced in $SPEC_{\mathrm{LF}}$, e.g., $(\sigma_i \doteq \sigma_j) \equiv_{SPEC_{\mathrm{LF}}} \mathsf{true}$ and $(\sigma_i \doteq \sigma_k) \equiv_{SPEC_{\mathrm{LF}}} \mathsf{false}$. However, since it is undecidable, in

general, whether $exp_1 \equiv_{SPEC_{LF}} exp_2$ holds for given expressions exp_1, exp_2 and specification $SPEC_{LF}$, this does not always work.

In $SPEC_{MAIN}$, there are three sentences the pre-situations of which can not be determined uniquely by using only the properties stated so far. For these three sentences, the reason why the S-dependency can not be uniquely determined is the same. In the rest of this section, we show that this ambiguity is resolved by considering properties on data types.

Example 3.8: Consider the S-dependency in paragraph Γ_1 of Table 3.1. By using only the properties stated so far, it can not be determined whether *pre*[(S7)] is equal to *pre*[(S6)] or *post*[(S6)] (See Fig. 3.2). The reason is that it can not be determined whether the conditional "on" in (S7) is associated with "if" in (S6) (*pre*[(S7)] $\doteq$ *pre*[(S6)]) or not (*pre*[(S7)] $\doteq$ *post*[(S6)]). □

Consider a property of timer: "Cancellation of a timer makes sense only if the timer is on." Then, we can conclude that (S6) and (S7) are associated with each other and *pre*[(S7)] $\doteq$ *pre*[(S6)]. In Example 3.9 stated below, a predicate VALID(s) which denotes the requirement for any situation s to meet is introduced. VALID(s) reflects properties of the domain (see Ref. [6] for detailed explanation of VALID). These properties are represented by axioms. These axioms are added to the set of axioms derived from the natural language specification.

Example 3.9: Let σ_6, σ_6', σ_6'', σ_7, σ_7' and σ_7'' be the situations indicated in Fig. 3.2. Precisely, these situations are denoted by variables with existential quantifiers or Skolem functions. However, for notational convenience, parameters of these Skolem functions are omitted in this example. Table 3.3 illustrates the axioms obtained from (S6) and (S7), where expire and cancel denote the events "expiration of the timer" and "cancellation of the timer" respectively, and out(TDISreq) and in(TDISind) denote the events "output of a T-DISCONNECT request" and "input of a T-DISCONNECT indication" respectively. Table 3.4 shows the axioms claiming that every situation has to be "valid." Table 3.5 shows axioms on properties of timer, where $\bar{\sigma}$ is a variable of type Situation and timer_on($\bar{\sigma}$) is a predicate which is true if and only if the timer is on at situation $\bar{\sigma}$. Axiom (AX9) claims "the timer is off immediately after the timer expires," (AX11) and (AX12) claim "neither in(TDISind) nor out(TDISreq) affects the state of the timer," and (AX14) implies "the situation immediately after the timer is cancelled is valid only if the timer is on at the situation immediately before the timer is cancelled." Table 3.6 illustrates axioms on primitive data types. The ambiguity of S-dependency illustrated in Example 3.8 is resolved by these axioms.

First, from (AX4), (AX18) and (AX19),

$$\mathsf{VALID}(\sigma_7') \Leftrightarrow \mathsf{VALID}(\Delta(\sigma_7'', \mathsf{cancel})) \equiv \mathsf{true},$$

and from (AX7), (AX15) and (AX14),

$$\mathsf{VALID}(\sigma_7'') \wedge \mathsf{timer\_on}(\sigma_7'') \equiv \mathsf{true},$$

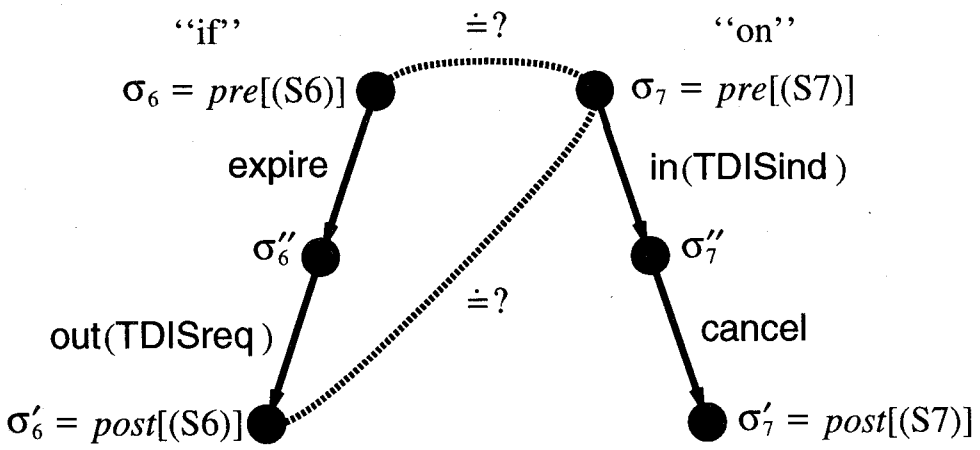

Fig. 3.2 Ambiguity of S-dependency.

Table 3.3 Axioms Obtained from Sentences (S6) and (S7) in Table 3.1.

$$\sigma''_6 \doteq \Delta(\sigma_6, \text{expire}) \;==\; \text{true} \qquad \text{(AX1)}$$

$$\sigma'_6 \doteq \Delta(\sigma''_6, \text{out(TDISreq)}) \;==\; \text{true} \qquad \text{(AX2)}$$

$$\sigma''_7 \doteq \Delta(\sigma_7, \text{in(TDISind)}) \;==\; \text{true} \qquad \text{(AX3)}$$

$$\sigma'_7 \doteq \Delta(\sigma''_7, \text{cancel}) \;==\; \text{true} \qquad \text{(AX4)}$$

$$(\sigma_7 \doteq \sigma_6) \vee (\sigma_7 \doteq \sigma'_6) \;==\; \text{true} \qquad \text{(AX5)}$$

Table 3.4 Axioms on Validity of Every Situation.

$$
\begin{aligned}
\mathsf{VALID}(\sigma_1) &== \mathsf{true} \\
&\vdots \\
\mathsf{VALID}(\sigma_7) &== \mathsf{true} && \text{(AX6)} \\
\mathsf{VALID}(\sigma_1') &== \mathsf{true} \\
&\vdots \\
\mathsf{VALID}(\sigma_7') &== \mathsf{true} && \text{(AX7)} \\
&\vdots \\
\mathsf{VALID}(\sigma_7'') &== \mathsf{true} && \text{(AX8)} \\
&\vdots
\end{aligned}
$$

Table 3.5 Axioms on Properties of Timer.

$$
\begin{aligned}
\mathsf{timer\_on}(\Delta(\bar{\sigma}, \mathsf{expire})) &== \mathsf{false} && \text{(AX9)} \\
\mathsf{timer\_on}(\Delta(\bar{\sigma}, \mathsf{cancel})) &== \mathsf{false} && \text{(AX10)} \\
\mathsf{timer\_on}(\Delta(\bar{\sigma}, \mathsf{in(TDISind)})) &== \mathsf{timer\_on}(\bar{\sigma}) && \text{(AX11)} \\
\mathsf{timer\_on}(\Delta(\bar{\sigma}, \mathsf{out(TDISreq)})) &== \mathsf{timer\_on}(\bar{\sigma}) && \text{(AX12)} \\
\mathsf{VALID}(\Delta(\bar{\sigma}, \mathsf{expire})) &== \mathsf{VALID}(\bar{\sigma}) \wedge \mathsf{timer\_on}(\bar{\sigma}) && \text{(AX13)} \\
\mathsf{VALID}(\Delta(\bar{\sigma}, \mathsf{cancel})) &== \mathsf{VALID}(\bar{\sigma}) \wedge \mathsf{timer\_on}(\bar{\sigma}) && \text{(AX14)} \\
&\vdots
\end{aligned}
$$

Table 3.6 Axioms on Primitive Data Types.

$$\mathsf{true} \Leftrightarrow \bar{x} \;==\; \bar{x} \qquad \text{(AX15)}$$

$$\bar{x} \Leftrightarrow \mathsf{false} \;==\; \neg\bar{x} \qquad \text{(AX16)}$$

$$\bar{x} \Leftrightarrow \bar{y} \;==\; \bar{y} \Leftrightarrow \bar{x} \qquad \text{(AX17)}$$

$$(\bar{\sigma} \doteq \bar{\sigma}') \supset (f(\bar{\sigma}) \Leftrightarrow f(\bar{\sigma}')) \;==\; \mathsf{true} \qquad \text{(AX18)}$$

$$\mathsf{true} \supset \bar{x} \;==\; \bar{x} \qquad \text{(AX19)}$$

$$\bar{x} \supset \mathsf{false} \;==\; \neg\bar{x} \qquad \text{(AX20)}$$

$$\bar{x} \vee \mathsf{false} \;==\; \bar{x} \qquad \text{(AX21)}$$

$$\mathsf{true} \wedge \bar{x} \;==\; \bar{x} \qquad \text{(AX22)}$$

$$\neg\neg\bar{x} \;==\; \bar{x} \qquad \text{(AX23)}$$

$$\neg\mathsf{true} \;==\; \mathsf{false} \qquad \text{(AX24)}$$

$$\neg\mathsf{false} \;==\; \mathsf{true} \qquad \text{(AX25)}$$

and hence from (AX8) and (AX22),

$$\mathsf{timer\_on}(\sigma_7'') \equiv \mathsf{true}. \tag{3.4}$$

Moreover, from (AX3) and (AX18) and (AX19),

$$\mathsf{timer\_on}(\sigma_7'') \Leftrightarrow \mathsf{timer\_on}(\Delta(\sigma_7, \mathsf{in(TDISind)})) \equiv \mathsf{true},$$

and hence from (3.4), (AX15) and (AX11),

$$\mathsf{timer\_on}(\sigma_7) \equiv \mathsf{true}. \tag{3.5}$$

On the other hand, from (AX1), (AX18) and (AX19),

$$\mathsf{timer\_on}(\sigma_6'') \Leftrightarrow \mathsf{timer\_on}(\Delta(\sigma_6, \mathsf{expire})) \equiv \mathsf{true},$$

and from (AX9),

$$\mathsf{timer\_on}(\sigma_6'') \Leftrightarrow \mathsf{false} \equiv \mathsf{true},$$

and hence from (AX16), (AX23) and (AX24),

$$\mathsf{timer\_on}(\sigma_6'') \equiv \mathsf{false}. \tag{3.6}$$

Moreover, from (AX2), (AX18) and (AX19),

$$\mathsf{timer\_on}(\sigma_6') \Leftrightarrow \mathsf{timer\_on}(\Delta(\sigma_6'', \mathsf{out(TDISreq)})) \equiv \mathsf{true},$$

and from (AX12) and (3.6),

$$\mathsf{timer\_on}(\sigma_6') \Leftrightarrow \mathsf{false} \equiv \mathsf{true},$$

and hence from (AX16), (AX23) and (AX24),

$$\mathsf{timer\_on}(\sigma_6') \equiv \mathsf{false}. \tag{3.7}$$

Hence, from (AX15), (3.5) and (3.7),

$$\mathsf{timer\_on}(\sigma_7) \Leftrightarrow \mathsf{timer\_on}(\sigma_6') \equiv \mathsf{false}. \tag{3.8}$$

From (3.8) and (AX18),

$$(\sigma_7 \doteq \sigma_6') \supset \mathsf{false} \equiv \mathsf{true},$$

and from (AX20), (AX23) and (AX24),

$$(\sigma_7 \doteq \sigma_6') \equiv \mathsf{false}. \tag{3.9}$$

From (AX5), (3.9) and (AX21),

$$(\sigma_7 \doteq \sigma_6) \equiv \mathsf{true}.$$

Thus the ambiguity of S-dependency is resolved. □

3.5 Analysis System

According to the proposed method, an analysis system was implemented on DECstation 3100 and incorporated into a translation system based on the method in Ref. [18] (see Fig. 3.3). The syntax rules of English are written in GPSG [3]. Parser was implemented by translating the rules in GPSG into the rules in Definite Clause Grammar (DCG) [15]. It consists of about 140 DCG rules. I-structure Constructor was implemented in Prolog and has about 240 clauses. Context Analyzer was implemented in C. It consists of Anaphoric Binding Analyzer (400 lines), Quantifier Scoping Analyzer (100 lines), S-dependency Analyzer (900 lines) and some libraries (600 lines). S-dependency Analyzer uses the properties of the syntax of natural languages and type constraints stated in Section 3.4. Axiom Generator was implemented in C and has about 500 lines. At present, Dictionary consists of about 320 lexical items.

In $SPEC_{\mathrm{MAIN}}$ (29 paragraphs, 98 sentences), there are 120 constituents which specify single actions. Table 3.7 shows the result of the analysis of S-dependency in $SPEC_{\mathrm{MAIN}}$. The CPU time needed for the translation was about 750 seconds.

3.6 Conclusions

In this chapter, a method of analyzing S-dependency in a natural language specification was proposed. The result of applying this analysis method to the main part of the OSI session protocol specification was also presented.

As mentioned in Chapter 1, we assume that there are many kinds of incompleteness in natural language specifications and we have to reduce the incompleteness to translate them into formal ones. Ref. [1] is a pioneer work whose aim is similar to ours. In the following, we make a comparison between Balzer's method proposed in Ref. [1] and our method.

For the analysis of S-dependency among constituents of a sentence, our method uses the relation between pre- and post-situations of each word. The relation is formally represented as the I-structure of the word in the lexical item. On the other hand, in Ref. [1], it is not described how each word in the natural language specification has been analyzed and how systematically the result of the analysis is used in their method.

For the analysis of S-dependency among constituents of different sentences, our method assumes that the order of sentences implicitly specifies the S-dependency unless a discontinuity is explicitly expressed, while Balzer's method just assumes a partial description of the sequence of operations. This difference is important when we analyze, or translate, a natural language specification which describes actions to be performed, in order of execution, e.g., a protocol specification. As shown in Table 3.7, our method can determine the pre-situations of 26 constituents by using the property of the order of sentences. On the other hand, Balzer's

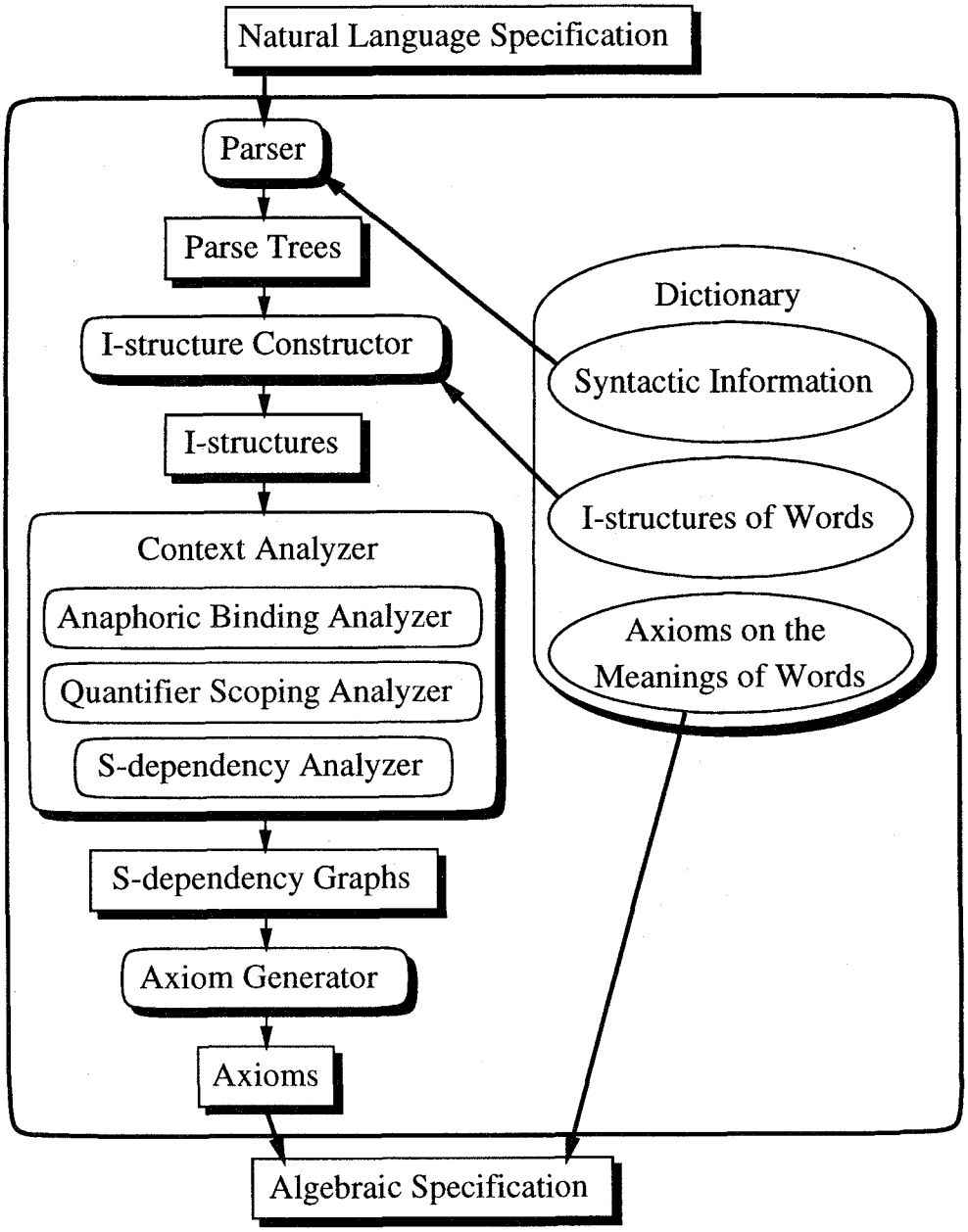

Fig. 3.3 Translation System.

Table 3.7 Analysis of S-dependency in $SPEC_{MAIN}$.

the pre-situations of the paragraphs			29
determined by sentence analysis			43
determined by context analysis			48
	using property (a)		32
		conditional	4
		anaphoric phrase	2
		order of sentences	26
	using property (b)		13
	using property (c)		3
total			120

property (a): syntax of natural languages
property (b): type constraints
property (b): axioms on data types

method reduces the incompleteness by using only properties of program well-formedness. Therefore, Balzer's method can not probably resolve the ambiguity of S-dependency among constituents of different sentences unless it introduces stronger criteria based on the syntactic properties of a natural language (property (a) stated in Chapter 1 and Section 3.1). Besides property (a), our method uses properties of the translated algebraic specification such as type constraints (property (b)) or axioms on data types (property (c)). They could be formalized in Balzer's method as criteria for program well-formedness mentioned above. However, it was not discussed how such properties are represented and handled.

Consequently, our method based on formalized properties of both source and object languages in a combined way is more systematic and effective when it is applied to protocol specifications.

Chapter 4

Construction of a Context-Free Grammar for Logical Formulas from a Natural Language Specification

4.1 Introduction

This chapter presents a method of constructing a context-free grammar (cfg) for logical formulas to be derived from a natural language specification. The resulting cfg is stored as a part of lexical items used by the translation method in Ref. [18] and Chapter 3.

Most of formal specification languages have a concept of data type. When a natural language specification is translated into a formal one, it is important for objects and operations appearing in the natural language specification to be appropriately classified according to the framework of data types. The reasons are as follows:

- The formal specification becomes simple and concise. Hence, the refinement step will be easily done; and
- Ambiguity or incorrectness of the informal specification can be detected or reduced by means of type checking of the derived formal specification.

However, it is difficult to classify them manually, since in general a large number of objects and operations appear in an informal specification.

Let $SPEC_{NL}$ be a specification written in a natural language (English in this dissertation). Suppose that $SPEC_{NL}$ is translated into $SPEC_{LF} = (G_{LF}, AX_{LF})$ by the method in Chapter 3. Each function or predicate appearing in AX_{LF} corresponds to a word or phrase in $SPEC_{NL}$, and its data type is specified by G_{LF} (see Chapter 2). However, in Chapter 3, G_{LF} is assumed to be given and only AX_{LF} is generated. It is desirable for G_{LF} to be constructed mechanically and systematically.

Let *EXP* be the finite set of all the expressions appearing in AX_{LF}. *EXP* can be obtained from $SPEC_{NL}$ based on a grammar G_{NL} of the natural language. Our method of constructing G_{LF} for *EXP* consists of the following three phases (see Fig. 4.1):

1. A set of nonterminals and a set of production rules to generate *EXP* are mechanically constructed. Then subtype relation (see Chapter 2) to be satisfied by G_{LF} is obtained by

analyzing such an expression *exp* in *EXP* that *exp* occurs as a subexpression in another expression in *EXP*;

2. A human translator adds some sentences which are semantically correct but not in $SPEC_{NL}$ to the set of expressions in the following way:

   - groups some nonterminals (i.e., data types) together and introduces a new supertype of them, and
   - augments the subtype relation to be satisfied by G_{LF};

   and

3. From the set of nonterminals, the set of productions, and the subtype relation, G_{LF} is constructed. G_{LF} is simplified based on structural equivalence [20] of cfg's.

Phases 2–3 are repeated until G_{LF} becomes appropriate.

This chapter is organized as follows. Sections 4.2–4.4 explain the phases 1–3 above, respectively. Section 4.5 shows that ambiguity of a natural language specification can be reduced by a cfg obtained by our method. Section 4.6 describes a construction system implemented according to the proposed method. Section 4.7 summarizes this chapter.

4.2 Naive Construction of a Grammar for Logical Formulas

Table 4.1 shows six sentences containing phrase "results in" in Ref. [10]. In the rest of this chapter, these sentences are used for explaining our construction method.

In the first phase of our method, a cfg G for the set *EXP* of expressions in the logical formulas from a natural language specification is naively constructed. *EXP* is generated similarly to the method in Ref. [18] and Chapter 3. In the method, each noun with no parameters is translated into a variable and a prefix predicate symbol called the restriction of the noun. On the other hand, each noun with parameters (e.g., gerunds) is translated into a prefix function symbol. Each verb or modifier is translated into a prefix predicate symbol.

Example 4.1: Consider sentence (S16) in Table 4.1. This sentence is translated into the following axiom, where "implicitly specified parameters" introduced in Chapter 3 are omitted for simplicity:

$$\bar{x}_1 : \mathsf{In\_SPDU} \quad \bar{x}_2 : \mathsf{Out\_SSprm}$$
$$\forall \bar{x}_1 \exists \bar{x}_2 (\mathsf{valid}(\bar{x}_1) \wedge \mathsf{incoming}(\bar{x}_1) \wedge \mathsf{MAP}(\bar{x}_1) \supset$$
$$(\mathsf{SSYNMind}(\bar{x}_2) \wedge \mathsf{result\_in}(\bar{x}_1, \bar{x}_2))) == \mathsf{true}$$

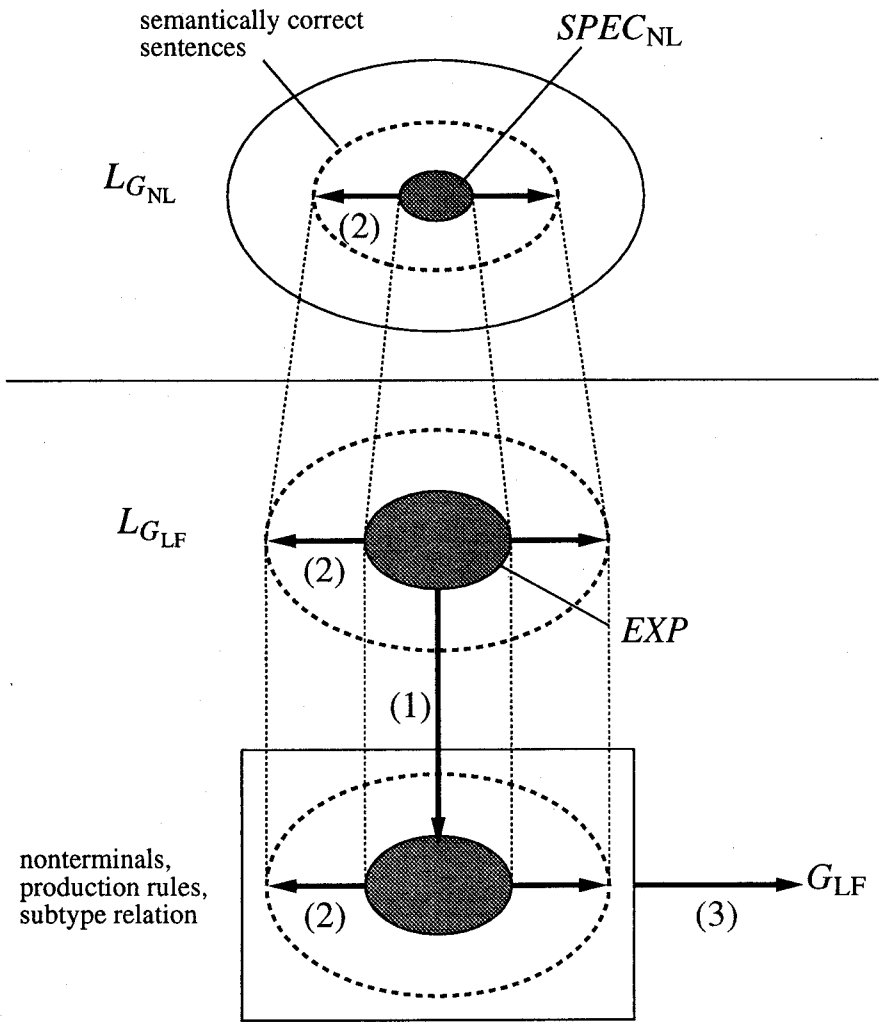

Fig. 4.1 Overview of the construction method.

Table 4.1 Sentences containing "results in" in Ref. [10].

(S11) An S-CONNECT request results in the assignment of a transport connection.

(S12) An S-CONNECT accept response results in an ACCEPT SPDU.

(S13) A valid incoming ACCEPT SPDU results in an S-CONNECT accept confirm.

(S14) A valid incoming ABORT SPDU results in sending an ABORT ACCEPT SPDU.

(S15) An S-SYNC MAJOR request results in a MAJOR SYNC POINT SPDU.

(S16) A valid incoming MAJOR SYNC POINT SPDU results in an S-SYNC MAJOR indication.

The set of expressions generated from the sentence (S16) is:

$$\{\mathsf{result\_in}(\bar{x}_1, \bar{x}_2), \bar{x}_1, \bar{x}_2, \mathsf{MAP}(\bar{x}_1), \mathsf{valid}(\bar{x}_1), \mathsf{incoming}(\bar{x}_1), \mathsf{SSYNMind}(\bar{x}_2)\}.$$

□

Then, (1) a cfg $G = (N, T, P)$ which generates *EXP* and (2) a subtype relation R to be satisfied by G_{LF} are mechanically constructed. G_{LF} is obtained by modifying G in the second and third phases.

The initial values of N, T, and P are the following sets:

$$\begin{aligned}
T &= \{\text{“,”}, \text{“(”}, \text{“)”}\} \cup \{f \mid f \text{ is a function (predicate) symbol appearing in } EXP\} \\
N &= \{A_f \mid f \in T \text{ is a function (predicate) symbol}\} \\
&\quad \cup \{A_{(f,1)}, \ldots, A_{(f,n)} \mid f \in T \text{ is an } n\text{-ary function (predicate) symbol}\} \\
&\quad \cup \{A_{\bar{x}} \mid \bar{x} \text{ is a variable appearing in } EXP\} \\
P &= \{A_f \rightarrow f(A_{(f,1)}, \ldots, A_{(f,n)}) \mid f \in T \text{ is an } n\text{-ary function (predicate) symbol}\} \\
&\quad \cup \{A_c \rightarrow c \mid c \in T \text{ is a constant (0-ary function symbol)}\}
\end{aligned}$$

The intuitive meaning of each nonterminal is as follows: A_f denotes the type of the return value of f. $A_{(f,i)}$ denotes the type of the i-th parameter of f. $A_{\bar{x}}$ denotes the type of $\bar{x}$.

Example 4.2: From the six sentences shown in Table 4.1, 15 production rules shown in Table 4.2 are constructed. □

Next, a subtype relation R to be satisfied by G_{LF} is obtained as follows:

- For each variable $\bar{x}$ and function symbol f appearing in *EXP*, if $f(\ldots, \underset{i}{\bar{x}}, \ldots)$ appears in *EXP*, then $A_{(f,i)} \overset{*}{\Rightarrow} A_{\bar{x}}$ is in R. Intuitively, the type of $\bar{x}$ must be a subtype of the one of the i-th parameter of f.

- For each function symbol g and function symbol f appearing in *EXP*, if $f(\ldots, \underset{i}{g(\cdots)}, \ldots)$ appears in *EXP*, then $A_{(f,i)} \overset{*}{\Rightarrow} A_g$ is in R. Intuitively, the type of the return value of g must be a subtype of the one of the i-th parameter of f.

Example 4.3: From the six sentences shown in Table 4.1, a subtype relation shown in Table 4.3 is constructed. The phrase to which each variable appearing in Table 4.3 corresponds is shown in Table 4.4. □

Table 4.2 Production rules constructed from the sentences in Table 4.1.

```
A[abort_accept_spdu] ---> abort_accept_spdu(A[abort_accept_spdu,1])
A[abort_spdu] ---> abort_spdu(A[abort_spdu,1])
A[accept_spdu] ---> accept_spdu(A[accept_spdu,1])
A[assignment] ---> assignment(A[assignment,1]).
A[incoming] ---> incoming(A[incoming,1])
A[major_sync_point_spdu]
    ---> major_sync_point_spdu(A[major_sync_point_spdu,1])
A[results_in] ---> results_in(A[results_in,1],A[results_in,2])
A[s_connect_accept_confirm]
    ---> s_connect_accept_confirm(A[s_connect_accept_confirm,1])
A[s_connect_accept_response]
    ---> s_connect_accept_response(A[s_connect_accept_response,1])
A[s_connect_request]
    ---> s_connect_request(A[s_connect_request,1])
A[s_sync_major_indication]
    ---> s_sync_major_indication(A[s_sync_major_indication,1])
A[s_sync_major_request]
    ---> s_sync_major_request(A[s_sync_major_request,1])
A[sending] ---> sending(A[sending,1],A[sending,2])
A[transport_connection]
    ---> transport_connection(A[transport_connection,1])
A[valid] ---> valid(A[valid,1])
```

Table 4.3 Subtype relation constructed from the sentences in Table 4.1.

```
A[abort_accept_spdu,1] ===> A[_x7]
A[abort_spdu,1] ===> A[_x9]
A[accept_spdu,1] ===> A[_x3]
A[accept_spdu,1] ===> A[_x6]
A[assignment,1] ===> A[_x1]
A[incoming,1] ===> A[_x6]
A[incoming,1] ===> A[_x9]
A[incoming,1] ===> A[_x13]
A[major_sync_point_spdu,1] ===> A[_x10]
A[major_sync_point_spdu,1] ===> A[_x13]
A[results_in,1] ===> A[_x2]
A[results_in,1] ===> A[_x4]
A[results_in,1] ===> A[_x6]
A[results_in,1] ===> A[_x9]
A[results_in,1] ===> A[_x11]
A[results_in,1] ===> A[_x13]
A[results_in,2] ===> A[_x3]
A[results_in,2] ===> A[_x5]
A[results_in,2] ===> A[_x10]
A[results_in,2] ===> A[_x12]
A[results_in,2] ===> A[assignment]
A[results_in,2] ===> A[sending]
A[s_connect_accept_confirm,1] ===> A[_x5]
A[s_connect_accept_response,1] ===> A[_x4]
A[s_connect_request,1] ===> A[_x2]
A[s_sync_major_indication,1] ===> A[_x12]
A[s_sync_major_request,1] ===> A[_x11]
A[sending,1] ===> A[_x8]
A[sending,2] ===> A[_x7]
A[transport_connection,1] ===> A[_x1]
A[valid,1] ===> A[_x6]
A[valid,1] ===> A[_x9]
A[valid,1] ===> A[_x13]
```

Table 4.4 Variables appearing in Table 4.3.

Variable	Sentence	Phrase
A[_x1]	(S11)	"a transport connection"
A[_x2]	(S11)	"an S-CONNECT request"
A[_x3]	(S12)	"an ACCEPT SPDU"
A[_x4]	(S12)	"an S-CONNECT accept response"
A[_x5]	(S13)	"an S-CONNECT accept confirm"
A[_x6]	(S13)	"a (valid incoming) ACCEPT SPDU"
A[_x7]	(S14)	"an ABORT ACCEPT SPDU"
A[_x8]	(S14)	the subject of "sending"
A[_x9]	(S14)	"a (valid incoming) ABORT SPDU"
A[_x10]	(S15)	"a MAJOR SYNC POINT SPDU"
A[_x11]	(S15)	"an S-SYNC MAJOR request"
A[_x12]	(S16)	"an S-SYNC MAJOR indication"
A[_x13]	(S16)	"a (valid incoming) MAJOR SYNC POINT SPDU"

4.3 Augmentation of the Grammar

In the second phase, the set N of nonterminals and the subtype relation R are augmented appropriately by a human translator. This augmentation method consists of the following three steps (a)–(c):

Step (a): From the semantics of a natural language adopted in this dissertation, the data type A of the return value of each function corresponding to a verb or a modifier, or the restriction of each noun must be Bool. For such a data type A, Bool $\stackrel{*}{\Rightarrow} A$ and $A \stackrel{*}{\Rightarrow}$ Bool are added to R.

Example 4.4: To the subtype relation obtained in the previous section, the subtype relation shown in Table 4.5 is added, where "`A <==> B`" means "`A` $\stackrel{*}{\Rightarrow}$ `B` and `B` $\stackrel{*}{\Rightarrow}$ `A`." □

Step (b): A natural language specification often defines and uses some new noun phrases (such as "S-CONNECT request" in Ref. [10]). Such definitions of new noun phrases should be translated into a part of cfg as well as axioms.

Let $w_1, \ldots, w_n$ be new noun phrases which have a common property. Let $\bar{x}_{i1}, \ldots, \bar{x}_{im_i}$ $(1 \leq i \leq n)$ be variables corresponding to w_i (there are more than one such variables in general since a variable is introduced for each occurrence of w_i). Then, a data type A denoting the common property is introduced and added into N, and, for each $\bar{x}_{ij}$ $(1 \leq i \leq n, 1 \leq j \leq m_i)$, subtype relation $A \stackrel{*}{\Rightarrow} A_{\bar{x}_{ij}}$ is added to R.

Example 4.5: In Ref. [10], there is a description of SS primitives and SPDUs as follows:

> Information is transferred to and from the SS-user using the session service primitives listed in table 1. Table 1 also defines the SPDUs associated with each of the service primitives.

Table 1 of Ref. [10], whose caption is "session service primitives," has three columns: The first one is "service," the second one is "primitives," and the third one is "associated SPDUs." According to the table, the subtype relation shown in Table 4.6 is added, where "`A ===> B1|...|Bn`" means "`A` $\stackrel{*}{\Rightarrow}$ `B1`,..., `A` $\stackrel{*}{\Rightarrow}$ `Bn`." In this example, a supertype `SPDU` of the types of the variables corresponding to "ABORT ACCEPT SPDU," "ABORT SPDU," or "MAJOR SYNC POINT SPDU" is introduced. Similarly, for noun phrases denoting SS primitives, a supertype `SSprm` is introduced.

In the case of Ref. [10], Step (b) can be semi-automated by considering the meanings of verbs such as "list" and "define," and fixing the semantics of tables. For example, consider sentence "Xs are listed in table Y." Suppose that noun phrases $w_1, \ldots, w_n$ are listed in table Y. Let $\bar{x}_{i1}, \ldots, \bar{x}_{im_i}$ $(1 \leq i \leq n)$ be variables corresponding to w_i. Then, the sentence is translated into a data type A_X and subtype relation $A \stackrel{*}{\Rightarrow} A_{\bar{x}_{ij}}$ $(1 \leq i \leq n, 1 \leq j \leq m_i)$. □

Table 4.5 Subtype relation added by Step (a).

```
Bool <==> A[abort_accept_spdu]
Bool <==> A[abort_spdu]
Bool <==> A[accept_spdu]
Bool <==> A[incoming]
Bool <==> A[major_sync_point_spdu]
Bool <==> A[results_in]
Bool <==> A[s_connect_accept_confirm]
Bool <==> A[s_connect_accept_response]
Bool <==> A[s_connect_request]
Bool <==> A[s_sync_major_indication]
Bool <==> A[s_sync_major_request]
Bool <==> A[transport_connection]
Bool <==> A[valid]
```

Table 4.6 Subtype relation added by Step (b).

```
SPDU ===> A[_x3]|A[_x6]|A[_x7]|A[_x9]|A[_x10]|A[_x13]
SSprm ===> A[_x2]|A[_x4]|A[_x5]|A[_x11]|A[_x12]
```

Step (c): To make R appropriate, natural language phrases which are semantically correct but not in $SPEC_{NL}$ are taken into account. The syntax of the restrictions of nouns is also considered. However, this is difficult when the size of the input natural language specification is large.

We use the following heuristics:

Heuristics 4.1: Let A and B be nonterminals. Suppose that for any nonterminal C, $B \stackrel{*}{\Rightarrow} C$ whenever $A \stackrel{*}{\Rightarrow} C$. Then, A may be a subtype of B. □

Heuristics 4.2: Let A and B be nonterminals. Suppose that for any nonterminal C, $C \stackrel{*}{\Rightarrow} A$ if and only if $C \stackrel{*}{\Rightarrow} B$. Then, $A \stackrel{*}{\Rightarrow} B$ or $B \stackrel{*}{\Rightarrow} A$ or both of them may hold. □

All the tuples of nonterminals satisfying the conditions in the above heuristics can be mechanically generated. A human translator examines each tuple by considering the meanings of words which have type A or B, and decides whether the tuple should be added to R.

Example 4.6: Suppose that a human translator decides that the the data types of the parameters of the restrictions on SPDUs are the same, and she/he also decides similarly for SS primitives. Then, the subtype relation shown in Table 4.7 is added to R.

Next, according to Heuristics 4.1, the set of candidates shown in Table 4.8 is obtained. `nt1` and `nt2` are the equivalence classes of $\stackrel{*}{\Leftrightarrow}$, and the elements in these equivalence classes are also shown in Table 4.8. A human translator examines each candidate. For example, for the first candidate in Table 4.8, she/he examines whether for any natural language word or phrase w, if "incoming w" is semantically correct phrase, then the type of w is `SPDU`. For the eighth candidate, she/he decides whether `abort_accept_spdu`, `abort_spdu`, etc. take expressions of type `SPDU` as their first parameter. According to such examination or decision, each candidate is added to R or discarded. □

In general, there is a subtype relation which should be satisfied by G_{LF} but is not necessarily obtained by the above heuristics. A human translator must find such a subtype relation. If G and R are simplified by the method stated in the next section, finding such a subtype relation becomes easier.

4.4 Simplification of the Grammar

In the last phase, $G = (N, T, P)$ and R are simplified based on structural equivalence [20] of cfg's. For simplicity, in the following definitions on structural equivalence, each element $A \stackrel{*}{\Rightarrow} B$ in R is regarded as a production rule $A \rightarrow B$ to be in P.

First we define the notion of structural equivalence.

Table 4.7 Subtype relation for restrictions.

```
A[abort_accept_spdu,1] <==> A[abort_spdu,1]
A[abort_spdu,1] <==> A[accept_spdu,1]
A[accept_spdu,1] <==> A[major_sync_point_spdu,1]
A[s_connect_accept_confirm,1] <==> A[s_connect_accept_response,1]
A[s_connect_accept_response,1] <==> A[s_connect_request,1]
A[s_connect_request,1] <==> A[s_sync_major_indication,1]
A[s_sync_major_indication,1] <==> A[s_sync_major_request,1]
```

Table 4.8 Subtype relation obtained by Heuristics 4.1.

```
SPDU ===> A[incoming,1]
SPDU ===> nt1
SPDU ===> A[sending,2]
SPDU ===> A[valid,1]
SSprm ===> nt2
A[assignment,1] ===> [transport_connection,1]
A[incoming,1] ===> [valid,1]
nt1 ===> SPDU
nt1 ===> A[incoming,1]
nt1 ===> A[sending,2]
nt1 ===> A[valid,1]
nt2 ===> SSprm
A[results_in,1] ===> A[incoming,1]
A[results_in,1] ===> A[valid,1]
A[transport_connection,1] ===> A[assignment,1]
A[valid,1] ===> A[incoming,1]

nt1 = {A[abort_accept_spdu,1], A[abort_spdu,1],
       A[accept_spdu,1], A[major_sync_point_spdu,1]}
nt2 = {A[s_connect_accept_confirm,1], A[s_connect_accept_response,1],
       A[s_connect_request,1], A[s_sync_major_indication,1],
       A[s_sync_major_request,1]}
```

Definition 4.1: Let $TREE_G$ denote the set of all parse trees generated by cfg G. For any $tree = (V, E) \in TREE_G$, define two mappings $h_V : V \to V$ and $h_E : E \to E$ as follows:

$$h_V(v) = \begin{cases} h_V(v') & \text{(if } v \text{ has exactly one child node } v' \text{ and} \\ & \quad v' \text{ is an internal node)} \\ v & \text{(otherwise)} \end{cases}$$

$$h_E((v_1, v_2)) = \begin{cases} (h_V(v_1), h_V(v_2)) & (\text{if } h_V(v_1) \neq h_V(v_2)) \\ \text{undefined} & (\text{if } h_V(v_1) = h_V(v_2)) \end{cases}$$

In addition, let

$$h(tree) = (\{h_V(v) \mid v \in V\}, \{h_E(e) \mid e \in E\}),$$

and

$$h(TREE_G) = \{h(tree) \mid tree \in TREE_G\}.$$

Two cfg's G_1 and G_2 are said to be structurally equivalent, if $h(TREE_{G_1}) = h(TREE_{G_2})$ when the labels of the internal nodes are ignored. □

Intuitively, structural equivalence of G_1 and G_2 means that the set of all parse trees generated by G_1 is equal to the one generated by G_2 when the labels of the internal nodes and applications of unit productions ($A \to A'$) are ignored.

Ref. [14] presents an algorithm which decides whether two arbitrary parenthesis grammars are weakly equivalent. For a given cfg G, the equivalence classes of the nonterminals based on structural equivalence (defined below) are obtained by means of this algorithm. G is simplified by replacing each nonterminal with its equivalence class.

Definition 4.2: Let $TREE_G[A]$ denote the set of parse trees with internal nodes labelled A. Let $TREE_G[A \leftarrow \Omega] = \{tree' \mid tree'$ is obtained from some $tree \in TREE_G[A]$ by replacing one subtree of *tree* whose root is A with a single node labelled $A\}$. Define h in the same way as defined in Definition 4.1.

Two nonterminals A and B of G are said to be structurally equivalent, if

$$h(TREE_G[A \leftarrow \Omega]) = h(TREE_G[B \leftarrow \Omega])$$

when the labels of the internal nodes of $h(TREE_G[A \leftarrow \Omega])$ and $h(TREE_G[B \leftarrow \Omega])$ are ignored. □

The algorithm proposed in Ref. [14] takes $O(2^{2^r})$ time for a cfg with r nonterminals. However, in a grammar to be constructed by our method, each function symbol is a prefix one. This improves the time complexity of the algorithm. For example, we assume that G satisfies the following condition:

For any function symbol f, there is at most one nonterminal C such that

$$C \underset{G}{\Rightarrow} f(A_{(f,1)}, A_{(f,2)}, \ldots, A_{(f,n)}).$$

Then, by Definition 4.2, the following property holds:

> Nonterminals A and B are structurally equivalent if and only if for any function symbol f and its parameter position i,
>
> $$A_{(f,i)} \overset{*}{\underset{G}{\Rightarrow}} A \text{ iff } A_{(f,i)} \overset{*}{\underset{G}{\Rightarrow}} B. \tag{4.1}$$

This method can simplify G in polynomial time of the description size of G.

Augmentation of G and R follows this simplification if necessary. As stated in the end of the previous section, and as also illustrated in the following example, the augmentation becomes easier for a human translator.

Example 4.7: Suppose that every subtype relation in Table 4.8 is added to R by a human translator. The result of simplification of R is shown in Fig. 4.2, where an arc from A' to A means that A is a subtype of A'. Each `nt*` is an equivalence class of nonterminals based on structural equivalence, and its members are shown in Table 4.9.

The graph shown in Fig. 4.2 is symmetric when the arc from `nt8` to `nt12` is ignored. Consider the meaning of `nt12`. It is a subtype of `nt8`, which contains `SPDU`, and it contains `A[sending,2]`, i.e., the data type of objects of "sending." It follows that `nt12` is a data type denoting SPDUs to be transmitted. On the other hand, `nt3` is a subtype of both `nt8` and `nt10`. Since `nt10` contains `A[results_in,2]`, i.e., the data type of objects of "results in," `nt8` is also a data type denoting SPDUs to be transmitted. Therefore, it seems appropriate for the following subtype relation

```
nt3 <==> nt12
```

to be added to R.

As a result, the subtype relation shown in Fig. 4.3 is obtained, where the name of each nonterminal is replaced with a mnemonic one (see Table 4.10). □

When a human translator considers G and R to be appropriate, each $A \overset{*}{\Rightarrow} B$ in R is transformed into a production $A \rightarrow B$ of G. Then, G_{LF} is constructed as the component-wise union of G and G_0, where G_0 is the specification of primitive data types introduced in Chapter 2.

Example 4.8: From the six sentences in Table 4.1, the grammar shown in Table 4.11 is constructed as an appropriate one. □

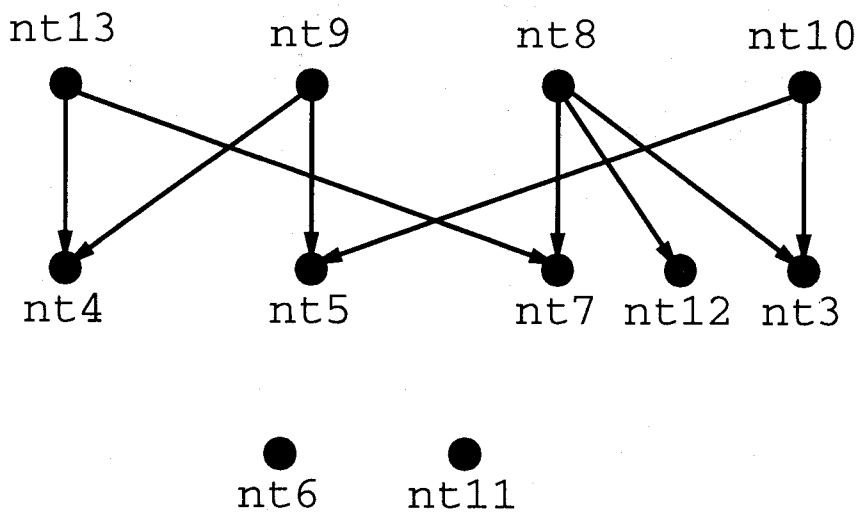

Fig. 4.2 Result of simplification of the subtype relation.

Table 4.9 Equivalence classes in Fig. 4.2.

```
nt3  = {A[_x3], A[_x10]}
nt4  = {A[_x2], A[_x4], A[_x11]}
nt5  = {A[_x5], A[_x12]}
nt6  = {A[_x1], A[assignment,1], A[transport_connection,1]}
nt7  = {A[_x6], A[_x9], A[_x13], A[incoming,1], A[valid,1]}
nt8  = {SPDU, A[abort_accept_spdu,1], A[abort_spdu,1],
        A[accept_spdu,1], A[major_sync_point_spdu,1]}
nt9  = {SSprm, A[s_connect_accept_confirm,1],
        A[s_connect_accept_response,1], A[s_connect_request,1],
        A[s_sync_major_indication,1], A[s_sync_major_request,1]}
nt10 = {A[assignment], A[results_in,2], A[sending]}
nt11 = {A[_x8], A[sending,1]}
nt12 = {A[_x7], A[sending,2]}
nt13 = {A[results_in,1]}
```

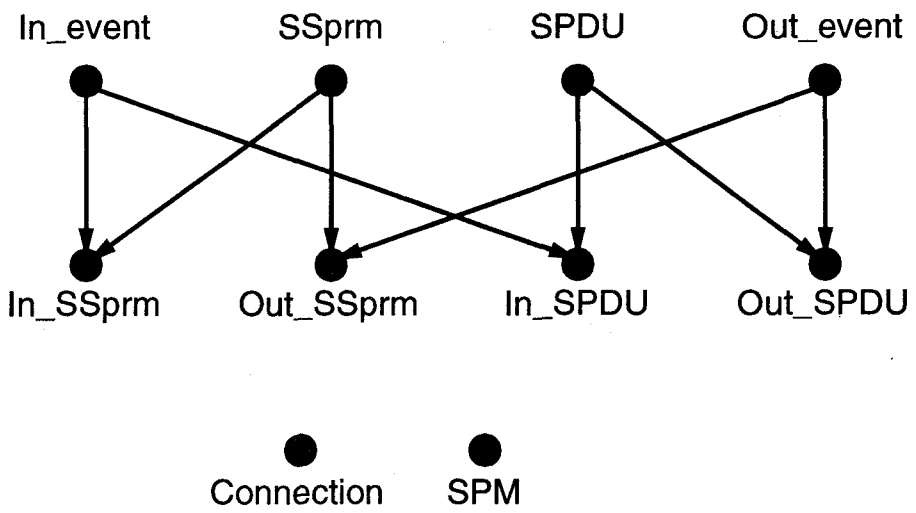

Fig. 4.3 Appropriate subtype relation.

Table 4.10 Data types in Fig. 4.3.

- In_event: Input events.
- Out_event: Output events.
- SPDU: Session protocol data units.
- SSprm: Session service primitives.
- In_SPDU: Received session protocol data units.
- In_SSprm: Received session service primitives.
- Out_SPDU: Transmitted session protocol data units.
- Out_SSprm: Transmitted session service primitives.
- Connection: Connections.
- SPM: Session protocol machines.

Table 4.11 Appropriate grammar.

```
Bool ---> abort_accept_spdu(SPDU)
Bool ---> abort_spdu(SPDU)
Bool ---> accept_spdu(SPDU)
Bool ---> incoming(In_SPDU)
Bool ---> major_sync_point_spdu(SPDU)
Bool ---> results_in(In_event,Out_event)
Bool ---> s_connect_accept_confirm(SSprm)
Bool ---> s_connect_accept_response(SSprm)
Bool ---> s_connect_request(SSprm)
Bool ---> s_sync_major_indication(SSprm)
Bool ---> s_sync_major_request(SSprm)
Bool ---> transport_connection(Connection)
Bool ---> valid(In_SPDU)
Out_event ---> assignment(Connection)
Out_event ---> sending(SPM,Out_SPDU)
In_event ---> In_SPDU
In_event ---> In_SSprm
Out_event ---> Out_SPDU
Out_event ---> Out_SSprm
SPDU ---> In_SPDU
SPDU ---> Out_SPDU
SSprm ---> In_SSprm
SSprm ---> Out_SSprm
```

4.5 Evaluation of the Construction Method

Ref. [18] proposes a method of analyzing anaphoric binding in a natural language specification. In the method, type information is used for selecting candidates for the antecedent of an anaphoric phrase. For the method to work well, words and phrases must be appropriately classified. In this section, we show an example in which an ambiguity of anaphoric binding is reduced by G_{LF} constructed by our method.

Example 4.9: Consider the following sentences in Ref. [10]:

(E12) A valid incoming ABORT SPDU results in sending an ABORT ACCEPT SPDU.

(E13) This SPDU is sent on the transport normal flow.

From these two sentences, the following expressions are obtained:

$$\mathsf{valid}(\bar{x}_1), \mathsf{incoming}(\bar{x}_1), \mathsf{result\_in}(\bar{x}_1, \mathsf{sending}(\bar{x}_2)), \mathsf{be\_sent\_on}(\bar{x}_?, \bar{x}_3),$$

where $\bar{x}_1$ corresponds to "ABORT SPDU," $\bar{x}_2$ does to "ABORT ACCEPT SPDU," $\bar{x}_?$ does to "this SPDU," and $\bar{x}_3$ does to "the transport normal flow."

Suppose that $\bar{x}_1$, $\bar{x}_2$ and $\bar{x}_?$ are of type SPDU. By only using this type information, it can not be determined whether "this SPDU" refers to "a valid incoming ABORT SPDU" or "an ABORT ACCEPT SPDU."

Now suppose that G_{LF} contains the production rules shown in Table 4.11 and Bool $\rightarrow$ be_sent_on(Out_event, Flow). Also suppose that L[In_event] $\cap$ L[Out_event] = $\emptyset$. Then,

- the data type of $\bar{x}_1$ is In_SPDU,
- the data type of $\bar{x}_2$ is Out_SPDU, and
- the data type of $\bar{x}_?$ is Out_SPDU.

Assume that "this SPDU" refers to "ABORT SPDU." Then the variables corresponding to these two phrases are identical, i.e., $\bar{x}_? = \bar{x}_1$, and the type of $\bar{x}_1$ is the greatest type among subtypes of both In_SPDU and Out_SPDU. On the other hand, L[In_SPDU]$\cap$$L$[Out_SPDU] = $\emptyset$ since L[In_event] $\cap$ L[Out_event] = $\emptyset$, and hence such a data type does not exist. Thus, it can be determined that "this SPDU" can not refer "ABORT SPDU." □

4.6 Construction System

According to the method proposed in this chapter, we implemented a prototype system (see Fig. 4.4) on SPARCstation IPX. Production Generator consists of about 200 DCG [15]

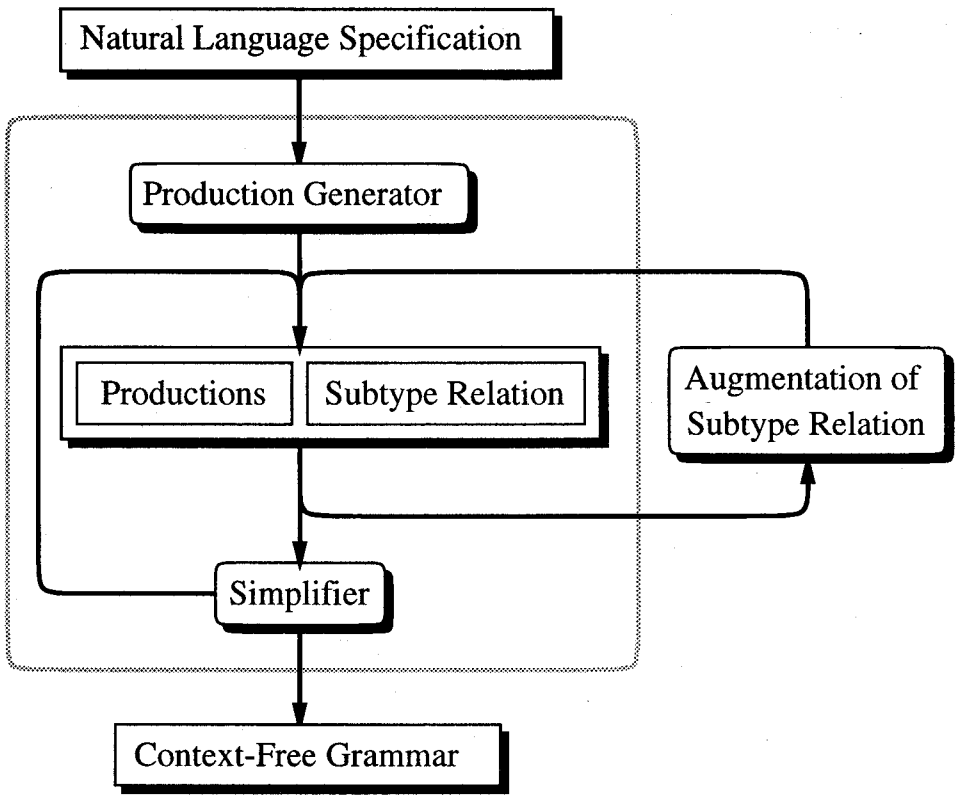

Fig. 4.4 Construction system.

rules and about 660 Prolog rules. Simplifier is written in C and the size is about 230 lines. We also implemented a program in Tcl/Tk which shows a given subtype relation graphically.

By using the prototype system, a part of the OSI session protocol specification [10] was analyzed. The part consists of 39 sentences including words "result" or "wait," or describing operations on the registers of a session protocol machine. The result of applying the method to the 39 sentences is shown in Table 4.12, where a couple of subtype relations $A \stackrel{*}{\Rightarrow} B$ and $B \stackrel{*}{\Rightarrow} A$ added simultaneously are counted as one subtype relation. Since the prototype system automatically substitutes `Bool` for the type of the return value of the function denoting the restriction of each noun, subtype relations corresponding to such replacements are not counted in Table 4.12.

4.7 Conclusions

This chapter proposed a method of constructing a cfg for logical formulas to be derived from a natural language specification. The result of applying this method to a part of the OSI session protocol specification was also presented.

In recent years, various object-oriented analysis methods have been proposed [7]. In many of them, candidates of objects and methods are extracted by considering the part of speech of each word in a natural language specification. Then, the candidates are refined and classified by considering the meanings of the words. As stated in Section 4.1, classifying objects and methods is difficult when an input natural language specification is large. Moreover, an appropriate classification often seems to be obtained after a human analyzer understands the natural language specification by trial and error. Therefore, it is desirable that the classification process be semi-automated. Our semi-automated method will contribute toward improving those object-oriented analysis methods.

Table 4.12 Result of the construction.

	$\|N\|$	$\|R\|$
Generated at Phase 1	169	180
Added based on		
Boolean	1	15
Grouping	11	111
Meanings of words	0	14
Obtained by removing $\overset{*}{\Leftrightarrow}$	127	248
Added based on		
Heuristics 4.1†	0	72
Obtained by removing $\overset{*}{\Leftrightarrow}$	99	121
Added based on		
Heuristics 4.2†	0	47
Obtained by removing $\overset{*}{\Leftrightarrow}$	44	37
Obtained by simplification	27	23
Added based on		
Meanings of words	0	3
Obtained by removing $\overset{*}{\Leftrightarrow}$	24	20

†The numbers of candidates by Heuristics 4.1 and 4.2 were 84 and 49, respectively.

Chapter 5

Translation from Logical Formulas into Executable Specifications

5.1 Introduction

Chapter 3 proposed a translation method from natural language specifications into algebraic axioms in the form of logical formulas. However, such logical formulas are too abstract to be compiled into an executable program directly. This chapter defines a model of protocol machines called the BE interpreter, and proposes a method of translating such logical formulas into executable algebraic specifications, called BE programs, which are also introduced in this chapter.

This chapter is organized as follows. Section 5.2 explains logical formulas derived from a natural language specification. In Sections 5.3 and 5.4, BE programs and the BE interpreter specification are defined respectively. Section 5.5 proposes a method of translating logical formulas by BE programs. Section 5.6 describes a prototype system based on the translation method. Section 5.7 summarizes this chapter.

5.2 Logical Formulas from a Natural Language Specification

This chapter assumes that the format of logical formulas derived from a natural language specification is modified as follows. First, every expression denoting the relation between the pre- and post-situation of a constituent (e.g., $\bar{\sigma}'_1 \doteq \Delta(\bar{\sigma}_1, \mathsf{in}(\bar{x}_1) \cdot \mathsf{out}(\bar{x}_2, \bar{x}_3))$ in Example 3.1) is eliminated, since the information of such an expression can be considered as a part of meanings of the predicate corresponding to the constituent. Next, every expression denoting S-dependency among constituents (e.g., $\bar{\sigma}_2 \doteq \bar{\sigma}'_1$ in Example 3.1) is eliminated by using the same variable name. Finally, each existentially quantified variable is replaced with a Skolem function introduced by a human translator. Consequently, it is assumed that a paragraph P of a natural language specification is translated into an axiom in the form of

$$\bar{\sigma}_0 : \mathsf{Situation},\ \bar{x}_1 : A_1,\ \ldots,\ \bar{x}_m : A_m$$
$$[R_1 \wedge \cdots \wedge R_m] \supset \bigwedge_{S \in P} pred_S \ == \ \mathsf{true},$$

where $pred_S$ 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 $\bigwedge_{S \in P} pred_S$, 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 $\bar{x}_j$. 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 5.1: The paragraph which consists of the two sentences in Example 1.1 is translated into an axiom of F == true, where F is the following logical formula:

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

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-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. SSYNMind$(\bar{x}_1)$, which is introduced by a human translator, denotes the service primitive S-SYNC-MAJOR indication to be sent when a protocol machine receives $\bar{x}_1$. Intuitive meanings of the other subexpressions in the formula are the same as Table 3.2. □

The semantics of τ_k is defined by the semantics of the predicate which includes $\tau_k(\bar{\sigma}_0, \ldots)$. Consider the logical formula in Example 5.1. 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 3.2). In the translation method proposed in this chapter, the semantics of predicates denoting actions is defined in terms of BE programs.

5.3 A Subclass of Executable Specifications — BE Programs —

As stated in Chapter 1, the 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) Arbitrary names can be used for registers and I/O buffers of the 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 the 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. Intuitively, AX_{PRG} corresponds to a program text, and G_{PRG} does to the syntax of the program text.

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

1. Reg $\in N_{\mathrm{PRG}}$ generates names of registers of the BE interpreter.
2. Buf $\in N_{\mathrm{PRG}}$ generates names of I/O buffers.

Define *REG* and *BUF* as $L_{G_{\mathrm{PRG}}}$[Reg] and $L_{G_{\mathrm{PRG}}}$[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 $D_{reg} \rightarrow reg \in P_{\mathrm{PRG}}$. 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 Seq_D_{buf} $\rightarrow buf \in P_{\mathrm{PRG}}$. D_{buf} is denoted by $type[buf]$.

Thirdly, for the requirement (c), the following data type, Action, is introduced:

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

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

where in, out, set, $\leftarrow \in T_{\mathrm{PRG}}$, $type[buf] = type[reg]$, and $D_{reg} = type[reg]$. in(buf, reg) denotes that the BE interpreter receives a data from buffer *buf* and the data is stored in register *reg*. out(buf, reg) denotes that the BE interpreter transmits a data stored in register *reg* to buffer *buf*. set$(reg \leftarrow t)$ denotes an assignment of the value of an expression t to register *reg* (the value of an expression is formally defined in Section 5.4).

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. B_id $\in N_{\mathrm{PRG}}$ generates behavior identifiers. There are productions of the following form:

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

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

8. B_exp $\in N_{\mathrm{PRG}}$ generates behavior expressions. The following productions are in P_{PRG}:

$$\begin{aligned}
\mathsf{B_exp} &\rightarrow \mathsf{stop}, \\
\mathsf{B_exp} &\rightarrow \mathsf{B_id}, \\
\mathsf{B_exp} &\rightarrow \mathsf{Action; B_exp}, \\
\mathsf{B_exp} &\rightarrow (\mathsf{B_exp} \diamond \mathsf{B_exp}), \\
\mathsf{B_exp} &\rightarrow (\mathsf{B_exp} | \mathsf{Seq_Action} | \mathsf{B_exp}), \\
\mathsf{B_exp} &\rightarrow [\mathsf{Bool}] \text{->} \mathsf{B_exp}, \\
\mathsf{B_exp} &\rightarrow (\mathsf{B_exp} \gg \mathsf{Seq_Action} \gg \mathsf{B_exp}), \\
\mathsf{B_exp} &\rightarrow (\mathsf{B_exp} [\!> \mathsf{Seq_Action} [\!> \mathsf{B_exp}),
\end{aligned}$$

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

Table 5.1 shows the intuitive meanings of the operators used in behavior expressions. The formal semantics is defined in Section 5.4 as the behavior of the BE interpreter.

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

9. There is a production

$$\mathsf{Bool} \rightarrow \mathsf{B\_id} := \mathsf{B\_exp}$$

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

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

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

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 the BE interpreter.

Table 5.1 Meanings of operators.

- stop means that no actions are performed, i.e., the BE interpreter which executes it goes into a dead state.
- Execution of a behavior identifier π is equivalent to execution of the behavior expression which is associated with π.
- Action-prefix: $a; B$ specifies that the BE interpreter performs action a, then executes behavior expression B.
- Choice: $(B_1 \diamond B_2)$ specifies that the BE interpreter executes either B_1 or B_2 nondeterministically. If the BE interpreter performs an action performable in common with B_1 and B_2, it is considered that the BE interpreter is executing "both" of B_1 and B_2 (see the end of Section 5.4 for detail).
- Parallel composition: $(B_1|\lambda \cdot a_1 \cdots a_n|B_2)$ specifies that the 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: $[p] \text{->} B$ specifies that the 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 the 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 BE interpreter begins to execute behavior expression B_2.
- Disabling: $(B_1 [\text{>} \lambda \cdot a_1 \cdots a_n [\text{>} B_2)$ specifies that the BE interpreter executes behavior expression B_1, and the BE 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.

10. There is a production

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

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

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

in AX_{PRG}, where $\mathsf{main} \in T_{\mathrm{PRG}}$ and $\pi \in L_{G_{\mathrm{PRG}}}[\mathsf{B\_id}]$. $\mathsf{main}(\pi) \equiv_{SPEC_{\mathrm{PRG}}} \mathsf{true}$ means that π 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 $y \in REG \cup BUF$, the following production is in P_{PRG}:

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

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

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

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

5.4 The BE Interpreter

5.4.1 Definition of the BE Interpreter

The semantics of BE programs is defined in terms of the behavior of the BE interpreter. A BE program can use arbitrary terminal symbols as names of registers and buffers. The BE interpreter should have a function of interpreting declarations of names of registers and buffers in a given BE program $SPEC_{\mathrm{PRG}} = (G_{\mathrm{PRG}}, AX_{\mathrm{PRG}})$, and it is possible to define the BE interpreter specification $SPEC_{\mathrm{INT}} = (G_{\mathrm{INT}}, AX_{\mathrm{INT}})$ ($G_{\mathrm{INT}} = (N_{\mathrm{INT}}, T_{\mathrm{INT}}, P_{\mathrm{INT}})$) so that the BE interpreter has such a function. However, the main role of the BE interpreter in this section is to define the semantics of behavior expressions. Therefore, we simply assume that $G_{\mathrm{INT}} \supset G_{\mathrm{PRG}}$ (component-wise containment), and define $SPEC_{\mathrm{INT}}$ so that $SPEC_{\mathrm{INT}} \cup SPEC_{\mathrm{PRG}}$ (component-wise union) specifies the behavior of the BE interpreter when $SPEC_{\mathrm{PRG}}$ is given as its input program (see Fig. 5.1). Define REG and BUF as $L_{G_{\mathrm{PRG}}}[\mathsf{Reg}]$ and $L_{G_{\mathrm{PRG}}}[\mathsf{Buf}]$, respectively.

To define the semantics of the operators used in behavior expressions, a quadruple relation $EXEC$ is introduced. Let $B, B' \in L_{G_{\mathrm{INT}}}[\mathsf{B\_exp}]$, $p \in L_{G_{\mathrm{INT}}}[\mathsf{Bool}]$, and $a \in L_{G_{\mathrm{INT}}}[\mathsf{Action}]$. $\langle B, p, a, B' \rangle \in EXEC$ means that "if the BE interpreter is about to execute

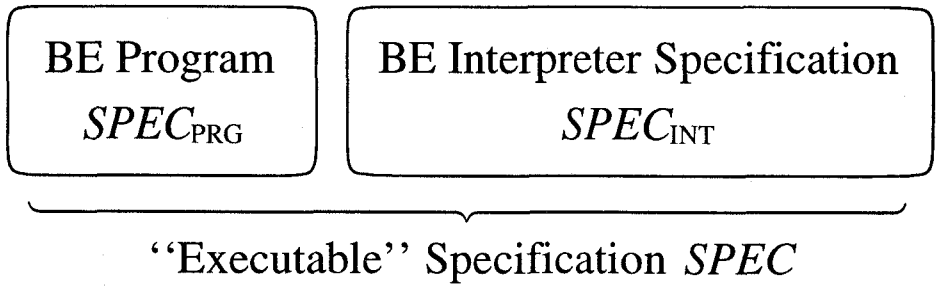

Fig. 5.1 Executable specification *SPEC*.

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 $SPEC_{\mathrm{INT}}$, relation $EXEC$ is represented by a predicate exec. A production

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

is included in P_{INT}, and the axioms shown in Table 5.2 are included in AX_{INT}.

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

$$\begin{aligned} \mathsf{State} &\rightarrow \mathsf{s_{init}}, \\ \mathsf{State} &\rightarrow \delta(\mathsf{State}, \mathsf{Action}), \end{aligned}$$

where $\mathsf{s_{init}}, \delta \in T_{\mathrm{INT}}$. $\mathsf{s_{init}}$ 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 the 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 $SPEC_{\mathrm{INT}}$, for each $D \in N_0$, a production

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

is introduced into P_{INT}, where $\mathsf{val} \in T_{\mathrm{INT}}$. For any expression t which includes some of 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 5.3.

Lastly, a production

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

is introduced into P, where $\mathsf{bexp} \in T_{\mathrm{INT}}$, and the axioms shown in Table 5.4 into AX_{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 bexp, define the behavior of the BE interpreter as follows:

Definition 5.1: The BE interpreter performs, at state s, an action a such that

$$\mathsf{bexp}(B, \delta(s, a)) \equiv_{SPEC} \mathsf{true}$$

for some $B \in L_G[\mathsf{B\_exp}]$. If such an action does not exist, the BE interpreter goes into a dead state. □

Table 5.2 Axioms for exec.

$\bar{B}, \bar{B}', \bar{B}_1, \bar{B}_1', \bar{B}_2, \bar{B}_2'$: B_exp, $\bar{a}$: Action,
$\bar{p}, \bar{p}', \bar{p}_1, \bar{p}_2$: Bool, $\bar{\pi}$: B_id, $\bar{A}$: Seq_Action

- Action-prefix:

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

- Choice:

$$\mathsf{exec}(\bar{B}_1, \bar{p}_1, \bar{a}, \bar{B}_1') \supset \mathsf{exec}((\bar{B}_1 \diamond \bar{B}_2), \bar{p}_1, \bar{a}, \bar{B}_1') \;==\; \mathsf{true},$$
$$\mathsf{exec}(\bar{B}_2, \bar{p}_2, \bar{a}, \bar{B}_2') \supset \mathsf{exec}((\bar{B}_1 \diamond \bar{B}_2), \bar{p}_2, \bar{a}, \bar{B}_2') \;==\; \mathsf{true}.$$

- Behavior identifier:

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

- Parallel composition:

$$(\mathsf{exec}(\bar{B}_1, \bar{p}_1, \bar{a}, \bar{B}_1') \wedge \neg\mathsf{member}(\bar{a}, \bar{A})) \supset$$
$$\mathsf{exec}((\bar{B}_1|\bar{A}|\bar{B}_2), \bar{p}_1, \bar{a}, (\bar{B}_1'|\bar{A}|\bar{B}_2)) \;==\; \mathsf{true},$$
$$(\mathsf{exec}(\bar{B}_2, \bar{p}_2, \bar{a}, \bar{B}_2') \wedge \neg\mathsf{member}(\bar{a}, \bar{A})) \supset$$
$$\mathsf{exec}((\bar{B}_1|\bar{A}|\bar{B}_2), \bar{p}_2, \bar{a}, (\bar{B}_1|\bar{A}|\bar{B}_2')) \;==\; \mathsf{true},$$
$$(\mathsf{exec}(\bar{B}_1, \bar{p}_1, \bar{a}, \bar{B}_1') \wedge \mathsf{exec}(\bar{B}_2, \bar{p}_2, \bar{a}, \bar{B}_2') \wedge \mathsf{member}(\bar{a}, \bar{A})) \supset$$
$$\mathsf{exec}((\bar{B}_1|\bar{A}|\bar{B}_2), \bar{p}_1 \wedge \bar{p}_2, \bar{a}, (\bar{B}_1'|\bar{A}|\bar{B}_2')) \;==\; \mathsf{true}.$$

- Conditional:

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

Table 5.2 Axioms for exec (continued).

- Enabling:

$$\begin{aligned}
&(\mathsf{exec}(\bar{B}_1, \bar{p}_1, \bar{a}, \bar{B}_1') \wedge \neg\mathsf{member}(\bar{a}, \bar{A})) \supset \\
&\qquad \mathsf{exec}((\bar{B}_1 \gg \bar{A} \gg \bar{B}_2), \bar{p}_1, \bar{a}, (\bar{B}_1' \gg \bar{A} \gg \bar{B}_2)) \;==\; \mathsf{true}, \\
&(\mathsf{exec}(\bar{B}_1, \bar{p}_1, \bar{a}, \bar{B}_1') \wedge \mathsf{member}(\bar{a}, \bar{A})) \supset \\
&\qquad \mathsf{exec}((\bar{B}_1 \gg \bar{A} \gg \bar{B}_2), \bar{p}_1, \bar{a}, \bar{B}_2) \;==\; \mathsf{true}.
\end{aligned}$$

- Disabling:

$$\begin{aligned}
&(\mathsf{exec}(\bar{B}_1, \bar{p}_1, \bar{a}, \bar{B}_1') \wedge \neg\mathsf{member}(\bar{a}, \bar{A})) \supset \\
&\qquad \mathsf{exec}((\bar{B}_1 \,[\!> \bar{A} \,[\!> \bar{B}_2), \bar{p}_1, \bar{a}, (\bar{B}_1' \,[\!> \bar{A} \,[\!> \bar{B}_2)) \;==\; \mathsf{true}, \\
&(\mathsf{exec}(\bar{B}_1, \bar{p}_1, \bar{a}, \bar{B}_1') \wedge \mathsf{member}(\bar{a}, \bar{A})) \supset \\
&\qquad \mathsf{exec}((\bar{B}_1 \,[\!> \bar{A} \,[\!> \bar{B}_2), \bar{p}_1, \bar{a}, \bar{B}_1') \;==\; \mathsf{true}, \\
&\mathsf{exec}(\bar{B}_2, \bar{p}_2, \bar{a}, \bar{B}_2') \supset \mathsf{exec}((\bar{B}_1 \,[\!> \bar{A} \,[\!> \bar{B}_2), \bar{p}_2, \bar{a}, \bar{B}_2') \;==\; \mathsf{true}.
\end{aligned}$$

Table 5.3 Axioms for val.

$\bar{s}$: State

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

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

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

$$\begin{aligned}
&\bar{t}_1 : A_1, \ \ldots, \ \bar{t}_n : A_n \\
&\quad \mathsf{val}(f(\bar{t}_1, \ldots, \bar{t}_n), \bar{s}) \;==\; f(\mathsf{val}(\bar{t}_1, \bar{s}), \ldots, \mathsf{val}(\bar{t}_n, \bar{s})).
\end{aligned}$$

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

$$\begin{aligned}
\mathsf{val}(reg, \mathsf{s}_{\mathsf{init}}) &== \mathsf{initial}(reg), \\
\mathsf{val}(buf, \mathsf{s}_{\mathsf{init}}) &== \mathsf{initial}(buf).
\end{aligned}$$

Table 5.3 Axioms for val (continued).

- in(buf, reg): For each $reg, reg' \in REG$ such that $reg \neq reg'$, and for each $buf, buf' \in BUF$ such that $buf \neq buf'$,

$$\begin{aligned}
\mathsf{val}(reg, \delta(\bar{s}, \mathsf{in}(buf, reg))) &== \mathsf{head}(\mathsf{val}(buf, \bar{s})), \\
\mathsf{val}(reg', \delta(\bar{s}, \mathsf{in}(buf, reg))) &== \mathsf{val}(reg', \bar{s}), \\
\mathsf{val}(buf, \delta(\bar{s}, \mathsf{in}(buf, reg))) &== \mathsf{tail}(\mathsf{val}(buf, \bar{s})), \\
\mathsf{val}(buf', \delta(\bar{s}, \mathsf{in}(buf, reg))) &== \mathsf{val}(buf', \bar{s}).
\end{aligned}$$

- out(buf, reg): For each $reg, reg' \in REG$, and for each $buf, buf' \in BUF$ such that $buf \neq buf'$,

$$\begin{aligned}
\mathsf{val}(reg', \delta(\bar{s}, \mathsf{out}(buf, reg))) &== \mathsf{val}(reg', \bar{s}), \\
\mathsf{val}(buf, \delta(\bar{s}, \mathsf{out}(buf, reg))) &== \mathsf{val}(buf, \bar{s}) \cdot \mathsf{val}(reg, \bar{s}), \\
\mathsf{val}(buf', \delta(\bar{s}, \mathsf{out}(buf, reg))) &== \mathsf{val}(buf', \bar{s}).
\end{aligned}$$

- set($reg \leftarrow t$): For each $reg, reg' \in REG$ such that $reg \neq reg'$, and for each $buf' \in BUF$,

$$\begin{aligned}
&\bar{t} : type[reg] \\
&\quad \mathsf{val}(reg, \delta(\bar{s}, \mathsf{set}(reg \leftarrow \bar{t}))) == \mathsf{val}(\bar{t}, \bar{s}), \\
&\quad \mathsf{val}(reg', \delta(\bar{s}, \mathsf{set}(reg \leftarrow \bar{t}))) == \mathsf{val}(reg', \bar{s}), \\
&\quad \mathsf{val}(buf', \delta(\bar{s}, \mathsf{set}(reg \leftarrow \bar{t}))) == \mathsf{val}(buf', \bar{s}).
\end{aligned}$$

Table 5.4 Axioms for bexp.

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

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

5.4.2 Properties of the BE Interpreter

This section presents some properties of the BE interpreter. Let $SPEC_{\text{PRG}}$ be a BE program, and let $SPEC$ (= (G, AX)) be $SPEC_{\text{PRG}} \cup SPEC_{\text{INT}}$.

First, $SPEC$ is consistent for any $SPEC_{\text{PRG}}$ since, intuitively, $SPEC_{\text{PRG}}$ corresponds to a program text and its syntax, and $SPEC_{\text{INT}}$ does to the semantics of the program text. A more formal reason is as follows. The left-hand side of each axiom for exec or bexp is in the form of a monotonic inference rule $p_1, p_2, \ldots, p_n \supset q$. Hence, in the case of bexp for example, $\mathsf{bexp}(B, s) \equiv_{SPEC} \mathsf{false}$ never holds for any B and s. Also, each axiom for val reduces the "size" of its parameter expressions in a unique way. Therefore, $SPEC$ is consistent for an arbitrary $SPEC_{\text{PRG}}$.

Next, we state a property of the choice operator ("$\Diamond$"). Suppose that

$$\mathsf{bexp}((B_1 \Diamond B_2), s) \equiv_{SPEC} \mathsf{true},$$

where $s \in L_G[\mathsf{State}]$ and $B_1, B_2 \in L_G[\mathsf{B\_exp}]$ such that

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

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

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

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, the BE interpreter chooses between $[p_1] \text{->} B_1'$ and $[p_2] \text{->} B_2'$ at state $\delta(\cdots \delta(s, a_1), \cdots, a_n)$. Similarly, this property holds in the case that $\mathsf{bexp}(\pi, s) \equiv_{SPEC}$ true, where $\pi \in L_G[\mathsf{B\_id}]$ has more than one behavior definitions. In Section 5.5, we propose a translation method which uses this property.

5.5 Translation from Logical Formulas into BE Programs

5.5.1 Overview

Let $SPEC_{\mathrm{NL}}$ be a natural language specification, i.e., a set of paragraphs. Let $SPEC_{\mathrm{LF}}$ be the algebraic specification derived from $SPEC_{\mathrm{NL}}$ by the method proposed in Chapter 3. In this section, we consider translating $SPEC_{\mathrm{LF}}$ into a BE program $SPEC_{\mathrm{PRG}}$.

The input of the translation method is as follows:

- An algebraic specification $SPEC_{\mathrm{LF}} = (G_{\mathrm{LF}}, AX_{\mathrm{LF}})$ $(G_{\mathrm{LF}} = (N_{\mathrm{LF}}, T_{\mathrm{LF}}, P_{\mathrm{LF}}))$ derived from $SPEC_{\mathrm{NL}}$, where AX_{LF} consists of:
  - axioms on primitive data types; and
  - axioms in the following form:

$$\bar{\sigma}_0 : \mathsf{Situation},\ \bar{x}_1 : A_1,\ \ldots,\ \bar{x}_m : A_m \qquad \bigwedge_{S \in P} \left[(R_1 \wedge \cdots \wedge R_m) \supset pred_S \right] \ ==\ \mathsf{true}, \tag{5.1}$$

  where $P \in SPEC_{\mathrm{NL}}$ 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 $P \in SPEC_{\mathrm{NL}}$, the pre-situation of P is denoted by $\bar{\sigma}_0$, and an input data to be received at the pre-situation of P is denoted by $\bar{x}_1$.

- The "lexical items" *dic* for predicates denoting actions in $SPEC_{\mathrm{LF}}$. Each item $dic[p(\ldots, \hat{\sigma}, \hat{\sigma}')]$ (p is a predicate, and $\hat{\sigma}$, $\hat{\sigma}'$ are the pre- and post-situations respectively) is a set of behavior definitions. The set of behavior definitions contains at least two behavior identifiers $\pi_{\hat{\sigma}}$ and $\pi_{\hat{\sigma}'}$; $\pi_{\hat{\sigma}}$ corresponds to the pre-situation $\hat{\sigma}$ of p, and $\pi_{\hat{\sigma}'}$ does to the post-situation $\hat{\sigma}'$. Moreover, there is at least one behavior definition of $\pi_{\hat{\sigma}}$ in the set. For example, if a predicate $p(\ldots, \hat{\sigma}, \hat{\sigma}')$ denotes a sequence $a_1, \ldots, a_k$ of actions, $dic[p(\ldots, \hat{\sigma}, \hat{\sigma}')]$ will be $\{\pi_{\hat{\sigma}} := a_1; \cdots; a_k; \pi_{\hat{\sigma}'}\}$. Other examples of *dic* are shown in Section 5.5.2.

Let $SPEC_{\mathrm{PRG}}$ be a BE program, and let $SPEC$ $(= (G, AX))$ be $SPEC_{\mathrm{PRG}} \cup SPEC_{\mathrm{INT}}$. We say that $SPEC_{\mathrm{PRG}}$ is a correct implementation of $SPEC_{\mathrm{LF}}$ with respect to *dic* if the following condition holds:

Condition 5.1: There is a mapping $\theta : L_{G_{\mathrm{LF}}}[\mathsf{Situation}] \rightarrow L_G[\mathsf{State}]$ such that, for any paragraph $P \in SPEC_{\mathrm{NL}}$, there are registers $var_1, \ldots, var_m$ in $SPEC$ such that (note that $\pi_{\hat{\sigma}}$ appears in *dic*):

(a) Let σ_{init} be an expression of $SPEC_{\mathrm{LF}}$ denoting the initial situation. Then, $\theta[\sigma_{\mathsf{init}}] = \mathsf{s}_{\mathsf{init}}$ and $\mathsf{bexp}(\pi_{\bar{\sigma}_0}, \mathsf{s}_{\mathsf{init}}) \equiv_{SPEC} \mathsf{true}$; and

(b) Let $\alpha\{\gamma'_1/\gamma_1, \ldots, \gamma'_m/\gamma_m\}$ denote the expression obtained by replacing each subexpression γ_j $(1 \leq j \leq m)$ of expression α by expression γ'_j. Suppose that, for some σ_0 and $x_1, \ldots, x_m$,

$$\Big(\bigwedge_{S \in P} pred_S\Big)\{\sigma_0/\bar{\sigma}_0, x_1/\bar{x}_1, \ldots, x_m/\bar{x}_m\} \equiv_{SPEC_{\mathrm{LF}}} \mathsf{true}$$

holds by Axiom (5.1). Also suppose that

- $\mathsf{bexp}(\pi_{\bar{\sigma}_0}, \theta[\sigma_0]) \equiv_{SPEC} \mathsf{true}$, or
- $\mathsf{bexp}(\pi_0, \theta[\sigma_0]) \equiv_{SPEC} \mathsf{true}$ for some π_0 such that

$$(\pi_0 := \pi_1) \wedge (\pi_1 := \pi_2) \wedge \cdots \wedge (\pi_{i-1} := \pi_i) \wedge (\pi_i := \pi_{\bar{\sigma}_0}) \equiv_{SPEC} \mathsf{true}$$

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

For each pre-/post-situation $\hat{\sigma}_k$ appearing in Axiom (5.1) $(1 \leq k \leq l$, and $\hat{\sigma}_l$ is the post-situation of $P)$, let σ_k be $\hat{\sigma}_k\{\sigma_0/\bar{\sigma}_0, x_1/\bar{x}_1, \ldots, x_m/\bar{x}_m\}$. Then,

(i) for each k $(1 \leq k \leq l)$, $\theta[\sigma_0]$ is a subexpression of $\theta[\sigma_k]$,

(ii) for each k $(1 \leq k \leq l)$, $\mathsf{bexp}(\pi_{\hat{\sigma}_k}, \theta[\sigma_k]) \equiv_{SPEC} \mathsf{true}$, and

(iii) for each j $(1 \leq j \leq m)$, $\mathsf{val}(var_j, \theta[\sigma_l]) \equiv_{SPEC} x_j$.

□

5.5.2 Translation Method

In this section, a method of translating $SPEC_{\mathrm{LF}}$ into a BE program $SPEC_{\mathrm{PRG}}$ is presented. Let $SPEC = SPEC_{\mathrm{PRG}} \cup SPEC_{\mathrm{INT}}$ (see Fig. 5.2).

In our translation method, each variable of a primitive data type in Axiom (5.1) corresponds to a register. To do this, the following registers are introduced into $SPEC_{\mathrm{PRG}}$:

1. $REG_{\mathrm{VAR}} = \{var_1, \ldots, var_m\}$: Each register var_j is used for storing the value of variable $\bar{x}_j$ in Axiom (5.1);

2. $REG_{\mathrm{PRED}} = \{reg_1, \ldots, reg_n\}$: Each register reg_i has been defined in $SPEC_{\mathrm{LF}}$ (e.g., Vsc, Va, and Vm in Example 5.1); and

3. REG_{TMP}: A register in REG_{TMP} is a temporary or dummy one, and denoted by *tmp* with some subscripts.

The translation method consists of the following three steps:

Step 1: For each paragraph $P \in SPEC_{\mathrm{NL}}$, a set β_P of behavior definitions is constructed by Steps 2 and 3. Then, $\beta = \bigcup_{P \in SPEC_{\mathrm{NL}}} \beta_P$ is the implementation of $SPEC_{\mathrm{NL}}$.

For each P, the behavior of the BE interpreter which executes $\pi_{\bar{\sigma}_0}$ ($\bar{\sigma}_0$ is the pre-situation of P) defined by β_P is as follows:

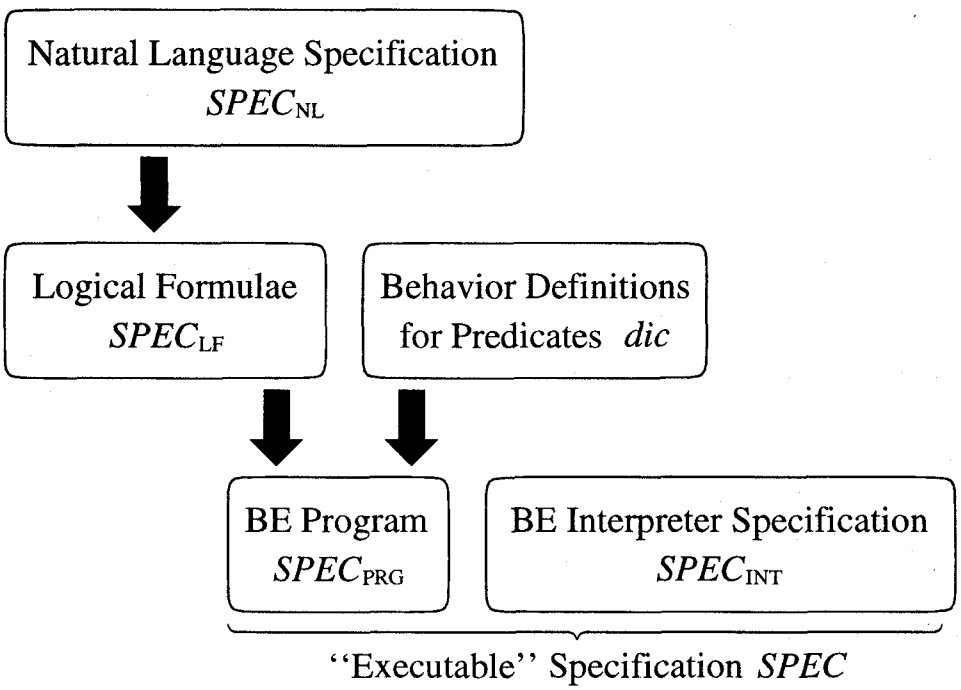

Fig. 5.2 Implementation method.

(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. 5.3. Each of the black circles represents some state s such that $\mathsf{bexp}(\pi_{\bar{\sigma}_0}, s) \equiv_{SPEC} \mathsf{true}$, 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 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 $[p_P] \rightarrow B_P$, where p_P corresponds to the examination (b) and B_P corresponds to (c). In order to satisfy this condition, we assume that *dic* of any predicate which involves an input action (such as receive) has common behavior expressions for (a).

Since the pre-situation of any paragraph is denoted by $\bar{\sigma}_0$, β ($= \bigcup_{P \in SPEC_{\mathrm{NL}}} \beta_P$) has in general more than one behavior definitions of $\pi_{\bar{\sigma}_0}$. Let s' be the state immediately after (a) is performed. As stated at the end of Section 5.4.2, $\mathsf{bexp}([p_P] \rightarrow B_P, s') \equiv_{SPEC} \mathsf{true}$ for each paragraph $P \in SPEC_{\mathrm{NL}}$. Therefore, at state s', only the actions specified by a paragraph P such that $\mathsf{val}(p_P, s') \equiv_{SPEC} \mathsf{true}$ are performed (see Fig. 5.4; the oval corresponds to s'). Thus, β is the implementation of $SPEC_{\mathrm{NL}}$.

Step 2: For each sentence $S \in P$, logical formula $(R_1 \wedge \cdots \wedge R_m) \supset pred_S$ is translated into a set $behavior[\langle R_1, \ldots, R_m \rangle, pred_S]$ of behavior definitions. $behavior[\langle R_1, \ldots, R_m \rangle, pred_S]$ is the union of $dic[pred_S]$ and $\{bind[\langle R_1, \ldots, R_m \rangle, pred_S]\}$ stated below.

Behavior definition $bind[\langle R_1, \ldots, R_m \rangle, pred_S]$ represents the "subroutine" for "variable bindings," and is defined by a human translator. 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 τ_0 as the identity function on type Situation. Suppose that $pre[pred_S] = \tau_k(\bar{\sigma}_0, \bar{x}_1, \ldots, \bar{x}_j)$ $(0 \leq j \leq m)$ and $post[pred_S] = \tau_{k'}(\bar{\sigma}_0, \bar{x}_1, \ldots, \bar{x}_{j'})$ $(j \leq j' \leq m)$. 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:

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

Here, $\hat{t}_{j''}$ $(j+1 \leq j'' \leq j')$ is an expression which indicates how the value of $var_{j''}$ is obtained, and is specified by a human translator. Action $\mathsf{set}(tmp_{\mathrm{bind}} \leftarrow tmp_{\mathrm{bind}})$ is performed as a "signal" which denotes successful completion of the variable bindings.

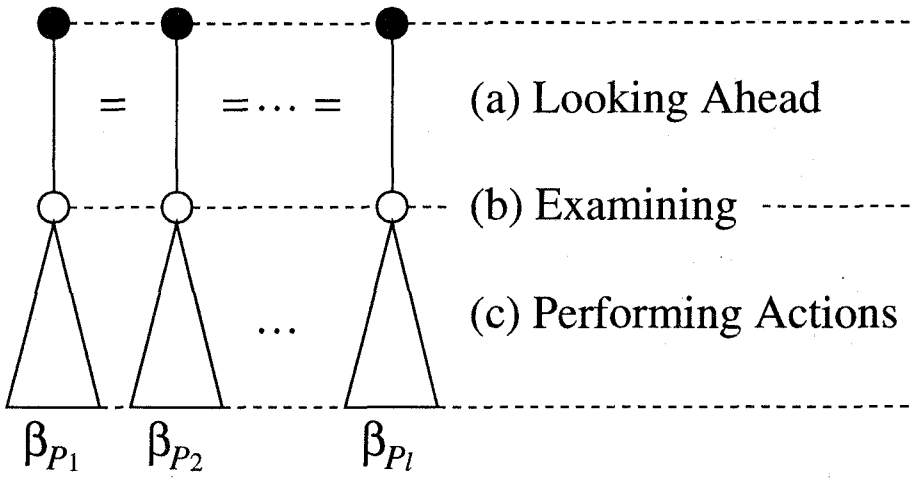

Fig. 5.3 Execution of each of $\beta_{P_1}, \beta_{P_2}, \ldots, \beta_{P_l}$.

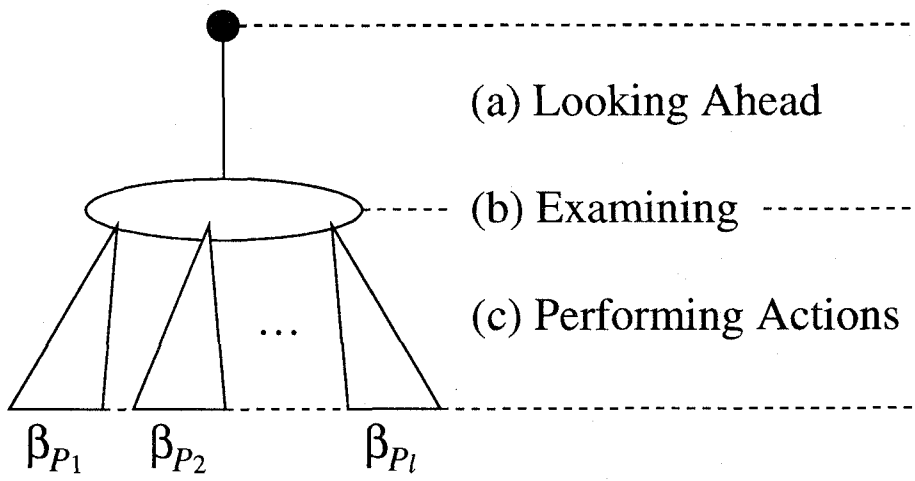

Fig. 5.4 Execution of β.

In what follows, some examples of *dic* and *bind* are presented.

Example 5.2: *dic*[receive($\hat{t}, \hat{\sigma}, \hat{\sigma}'$)] is defined as shown in Table 5.5(a), where $type[tmp_{\text{in}}] = D$ such that

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

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

In addition,

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

can be defined as

$$\begin{aligned} \rho_{\hat{\sigma}_0,\hat{\sigma}_1} \quad := \quad & \text{set}(var_1 \leftarrow tmp_{\text{in}}); \\ & [\text{valid}(var_1) \wedge \text{incoming}(var_1) \wedge \text{MAP}(var_1)] \rightarrow \\ & \quad \text{set}(tmp_{\text{bind}} \leftarrow tmp_{\text{bind}}); \text{stop}. \end{aligned}$$

Here it is specified that the value of $\bar{x}_1$ be equal to the value of tmp_{in}, i.e., the first element of $\text{buf}_{\text{inSPDU}}$. If a common register, say tmp_{in}, is used to store the input data in *dic* of all predicates which involve input actions, a human translator can specify tmp_{in} as $\hat{t}_{j''}$ appearing in *bind*. □

Example 5.3: *dic*[send($\hat{t}, \hat{\sigma}, \hat{\sigma}'$)] is shown in Table 5.5(b), where $type[tmp_{\text{out}}] = D$ such that

$$\text{Bool} \rightarrow \text{send}(D, \text{Situation}, \text{Situation}) \in P_{\text{LF}}.$$

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

The behavior definition

$$\begin{aligned} & bind[\langle \text{valid}(\bar{x}_1) \wedge \text{incoming}(\bar{x}_1) \wedge \text{MAP}(\bar{x}_1) \rangle, \\ & \quad \text{send}(\text{SSYNMind}(\bar{x}_1), \hat{\sigma}_1, \hat{\sigma}_2)] \end{aligned}$$

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

Table 5.5 "Lexical items" for predicates.

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

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

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

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

(c) dic[set_equal_to($\hat{t}_1, \hat{t}_2, \hat{\sigma}, \hat{\sigma}'$)].

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

Example 5.4: dic[set_equal_to($\hat{t}_1, \hat{t}_2, \hat{\sigma}, \hat{\sigma}'$)] is defined as shown in Table 5.5(c). In addition,

$$bind[\langle \mathsf{valid}(\bar{x}_1) \wedge \mathsf{incoming}(\bar{x}_1) \wedge \mathsf{MAP}(\bar{x}_1) \rangle, \\ \mathsf{set\_equal\_to}(\mathsf{Va}, \mathsf{Vm}, \hat{\sigma}_3, \hat{\sigma}_4)]$$

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

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

Example 5.5: We define

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

as

$$behavior[\langle R_1, \ldots, R_m \rangle, \hat{p}] \cup dic[\mathsf{if\_then}(\hat{q}, \hat{p}, \hat{\sigma}, \hat{\sigma}')].$$

And, dic[if_then($\hat{q}, \hat{p}, \hat{\sigma}, \hat{\sigma}'$)] is the set of the following behavior definitions:

$$\begin{aligned} \pi_{\hat{\sigma}} \quad &:= \quad ([\hat{q}\{var_1/\bar{x}_1, \ldots, var_m/\bar{x}_m\}] \text{ -> } \pi_{pre[\hat{p}]} \\ &\qquad \diamond \\ &\qquad [\neg\hat{q}\{var_1/\bar{x}_1, \ldots, var_m/\bar{x}_m\}] \text{ -> } \pi_{\hat{\sigma}'}), \\ \pi_{post[\hat{p}]} \quad &:= \quad \pi_{\hat{\sigma}'}. \end{aligned}$$

□

Step 3: Let $\beta'_P = \bigcup_{S \in P} behavior[\langle R_1, \ldots, R_m \rangle, pred_S]$. Let $\hat{\sigma}_l$ be the post-situation of paragraph P. Then, $\pi_{\hat{\sigma}_l} := \pi_{\bar{\sigma}_0}$ is added to β'_P. That is, after the actions specified by P have been completed, the BE interpreter executes $\pi_{\bar{\sigma}_0}$, i.e., it looks ahead the next input (recall that the pre-situation of any paragraph P' is denoted by $\bar{\sigma}_0$). The resulting set of behavior definitions is β_P.

Example 5.6: For the logical formula in Example 5.1, the behavior definitions in Table 5.6 are obtained. Here, we write π_k instead of $\pi_{\hat{\sigma}_k}$ ($0 \leq k \leq 5$). By Step 3, the behavior definition of π_5 is added. □

Table 5.6 Implementation of the logical formula in Example 5.1.

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

5.5.3 Correctness

We briefly describe a proof of the correctness of this translation method. Suppose that $\mathsf{main}(\pi_{\tilde{\sigma}_0}) == \mathsf{true}$ in AX_{PRG}. Define θ for Condition 5.1 as follows:

1. $\theta[\sigma_{\mathrm{init}}] = \mathsf{s}_{\mathrm{init}}$.
2. Suppose that some σ_0 and $x_1, \ldots, x_m$ satisfy the hypothesis of (b) in Condition 5.1. Define $\hat{\sigma}_k$ and σ_k $(1 \leq k \leq l)$ in the same way as defined in Condition 5.1. Let the BE interpreter start to execute $\pi_{\tilde{\sigma}_0}$ at state $\theta[\sigma_0]$ step by step. Define $\theta[\sigma_k]$ as the first s_k such that $\mathsf{bexp}(\hat{\sigma}_k, s_k) \equiv_{SPEC} \mathsf{true}$.

It is easy to show that θ satisfies (a) and (b) in Condition 5.1 if:

- *bind* is properly constructed by a human translator, that is, it is correctly specified where each input data comes from; and
- for each "lexical item" $dic[p(\ldots, \hat{\sigma}, \hat{\sigma}')]$, the BE interpreter eventually reaches a state s' such that $\mathsf{bexp}(\pi_{\hat{\sigma}'}, s') \equiv_{SPEC} \mathsf{true}$, if it starts the execution of $\pi_{\hat{\sigma}}$ at any state s and the variable bindings are successfully completed.

Thus, the correctness of the translation method depends on how *dic* and *bind* are defined. A human translator must know not only the entire of $SPEC_{\mathrm{INT}}$ (since $SPEC_{\mathrm{INT}}$ is the semantics of BE programs) but also how *dic* are defined and how to define *bind* correctly.

5.6 Translation System from Logical Formulas into BE Programs

We have implemented a prototype system which translates logical formulas derived from natural language specifications into BE programs. This system is written in Prolog (100 clauses). Using this system, we translated the logical formulas derived from a part of the OSI session protocol specification [10] (18 paragraphs, 45 sentences). The number of the "lexical items" *dic* 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 translation system behaved just as the human implementor intended.

Fig. 5.5 shows a part of the execution of the simulator when the behavior definitions in Table 5.6 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 5.2). 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.

```
.....
*** 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. 5.5 Execution of the simulator.

5.7 Conclusions

This paper has described a method of translating logical formulas derived from natural language specifications of communication protocols into executable specifications. By using this translation method and the simulator stated in section 5.6, 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.

The syntax of BE programs is modeled on the syntax of LOTOS [11]. A major difference between BE programs and LOTOS is that the BE interpreter has registers while the concept of processes (machines) in LOTOS does not have any. Since it is possible to specify the behavior of registers and buffers by means of processes, LOTOS specifications can be substituted for BE programs. On the other hand, to implement rendezvous (synchronous communication) of LOTOS, Ref. [2] uses shared memories, and Ref. [12] uses registers. We can conclude that LOTOS specifications can be translated into BE programs, and therefore, the expressive power of BE programs is equal to that of LOTOS.

One of the advantages of introducing BE programs is that the whole translation from natural language specifications into executable programs is handled in the same framework. Another advantage is that the BE interpreter is more appropriate for a model of protocol machines than the concept of processes in LOTOS, since a natural language specification of a communication protocol such as Ref. [10] often assumes that a protocol machine has registers as stated above. Because of these two reasons, the whole translation becomes simple and concise.

On the other hand, it may be possible that one translates a natural language specification into a LOTOS specification, and then implements the LOTOS specification by using the methods in Ref. [2] or [12]. However, there are no published papers on translation from natural language specifications into LOTOS specifications as far as the authors know. Such a translation method will be complicated since a protocol machine assumed in a natural language specification has registers while the concept of processes in LOTOS does not have any. Moreover, as stated above, shared memories or registers are introduced when a LOTOS specification is implemented. This is the reason why we do not use LOTOS for translation from natural language specifications into executable specifications.

Chapter 6

Conclusions

In this dissertation, three important sub-methods in translation from natural language specifications of communication protocols into algebraic specifications were proposed. In Chapter 3, a method of analyzing incompleteness of a natural language specification was proposed. In Chapter 4, for the lexical items to be constructed more easily, a method of constructing a context-free grammar for logical formulas to be derived from a natural language specification was proposed. In Chapter 5, a method of translating logical formulas derived from natural language specifications into executable algebraic specifications was proposed. We implemented a prototype system according to the proposed method. By using this system, a part of the OSI session protocol specification was successfully translated.

Throughout this dissertation, we used a part of the OSI session protocol specification as a translation example. Although it is a well-written specification, it still has incompleteness and ambiguity which must be resolved when it is translated into a formal one. Therefore, most of such incompleteness and ambiguity should be considered inherent in natural language specifications describing dynamic behavior of systems. Our method will be useful in reducing or detecting incompleteness and ambiguity of such natural language specifications. Moreover, by modifying the BE interpreter and BE programs appropriately, such natural language specifications will be translated into executable specifications.

In Chapter 1, we assumed an input natural language specification to contain only sequential or conditional descriptions. When a specification specifies "complex" execution control such as repetition or recursion, special words and phrases (called S-dependency controllers in Chapter 3) must be used. The meanings of such words and phrases can be easily incorporated into our method so that our method can handle repetition, recursion, and so on.

One of the major advantages of our method is that a natural language specification can be translated into an executable specification within a single framework of algebraic specification methods. Hence, the whole translation became simple and concise. Another advantage is that our method uses the properties both the source language (natural language) and the target language (ASL). For example, in Chapter 3, use of these properties made our analysis of S-dependency more refined. Moreover, in Chapter 4, syntactic information on the target language was used for classifying words of a natural language specification, and hence, ambiguity in the specification was reduced. Our method is superior to others which use only the properties of either the source language or the target language.

References

[1] Balzer, R., Goldman, N. and Wile, D., "Informality in Program Specifications," IEEE Trans. Software Eng., Vol. 4, No. 2, pp. 94–102, Mar. 1978.

[2] 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, July 1992.

[3] Gazdar, G., Klein, E., Pullum, G. and Sag, I., "Generalized Phrase Structure Grammar," Basil Blackwell, 1985.

[4] 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 Yeh, R. (ed.), "Current Trends in Programming Methodology IV: Data Structuring," Prentice Hall, pp. 80–144, 1978.

[5] Heidorn, G. E., "Automatic Programming Through Natural Language Dialogue: A Survey," Readings in Artificial Intelligence and Software Engineering, pp. 203–214, 1986.

[6] Higashino, T., Mori, M., Sugiyama, Y., Taniguchi, K. and Kasami, T., "An Algebraic Specification of HDLC Procedures and Its Verification," IEEE Trans. Software Eng., Vol. 10, No. 6, pp. 825–836, Nov. 1984.

[7] Honiden, S. and Yamashiro, A., "Object-Oriented Analysis and Design," Journal of IPS Japan, Vol. 35, No. 5, pp. 392–401, May 1994 (in Japanese).

[8] Huet, G. and Oppen, D. C., "Equations and Rewrite Rules: A Survey," Book, R. (ed.), Formal Languages: Perspectives and Open Problems, Academic Press, pp. 349–393, 1980.

[9] Ichikawa, I., Horai, H., Saeki, M., Yonezaki, N. and Enomoto, H., "The Method of Transformation from Natural Language Based Functional Specifications into Prototype Programs," Trans. of IPS Japan, Vol. 27, No. 11, pp. 1112–1127, Nov. 1986 (in Japanese).

[10] ISO, "Basic Connection Oriented Session Protocol Specification," ISO 8327, 1987.

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

[12] Karjoth, G., "Implementing Process Algebra Specifications by State Machines," Testing and Verification VIII, pp. 47–60, 1988.

[13] 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, July 1986 (in Japanese), also in Systems and Computers in Japan, Vol. 18, No. 7, pp. 11–20, July 1987.

[14] McNaughton, R., "Parenthesis Grammar," Journal of ACM, Vol. 14, No. 3, pp. 490–500, July 1967.

[15] Pereira, F. C. N. and Warren, D. H. D., "Definite Clause Grammars for Language Analysis," Artificial Intelligence, Vol. 13, pp. 231–278, 1980.

[16] Pollard, C. J., "Lecture on HPSG," unpublished manuscript, Stanford University, Feb. 1985.

[17] Saeki, M., Yonezaki, N. and Enomoto, H., "Formal Specification Method Based on Lexical Decomposition of Natural Language," Trans. of IPS Japan, Vol. 25, No. 2, pp. 204–215, Mar. 1984 (in Japanese).

[18] 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, Apr. 1991 (in Japanese).

[19] Seki, H., Nabika, E., Matsumura, T., Sugiyama, Y., Fujii, M., Torii, K. and Kasami, T., "A Processing System for Program Specifications in a Natural Language," Proc. 21th Annual Hawaii International Conference on System Sciences, pp. 754–763, Jan. 1988.

[20] Taniguchi, K. and Kasami, T., "Reduction of Context-Free Grammars," Trans. IECE Japan, Vol. 52-C, No. 12, pp. 827–834, Dec. 1969 (in Japanese).

[21] Taniguchi, K., Seki, H. and Kasami, T., "Translation from Specifications in a Natural Language into Algebraic Specifications and their Stepwise Refinement," Linguistic Engineering '91 (Versailles, France), Vol. 3, Jan. 1991.

[22] Tsujii, J. and Uehara, K., "Software Engineering and Natural Language Processing," Journal of IPS Japan, Vol. 28, No. 7, pp. 913–921, July 1987 (in Japanese).

[23] Yagi, T., Seki, H. and Kasami, T., "Translation from Natural Language Specifications into Algebraic Specifications —Extension of Translation Based on Word Meanings—," IEICE Technical Report, SS91-34, Mar. 1992 (in Japanese).