UNDERSTANDING OF JAPANESE IN AN INTERACTIVE PROGRAMMING SYSTEM

Kenji Sugiyama[1], Masayuki Kameda, Kouji Akiyama, Akifumi Makinouchi
Software Laboratory
Fujitsu Laboratories Ltd.
1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, JAPAN

ABSTRACT

KIPS is an automatic programming system which generates standardized business application programs through interactive natural language dialogue. KIPS models the program under discussion and the content of the user's statements as organizations of dynamic objects in the object-oriented programming sense. This paper describes the statement-model and the program-model, their use in understanding Japanese program specifications, and how they are shaped by the linguistic singularities of Japanese input sentences.

I INTRODUCTION

KIPS, an interactive natural language programming system, that generates standardized business application programs through interactive natural language dialogue, is under development at Fujitsu (Sugiyama, 1984). Research on natural language programming systems ("NLPS") (Heidorn, 1976, McCune, 1979) has been pursued in America since the late 1960's and some results of prototype systems are emerging (Biermann, 1983). But in Japan, although Japanese-like programming languages (Ueda, 1983) have recently appeared, there is no natural language programming system.

Generally, for a NLPS to understand natural language specifications, modeling of both the program under discussion and of the content of the user's statements is required. In conventional systems (Heidorn, 1976, McCune, 1979), programs and rules encoding linguistic knowledge first govern parsing procedures which extract from the user's input a statement-model; then "program model building rules" direct procedures which update or modify the program-model in light of what the user has stated. There are thus two separate models and two separate procedural components.

However, we believe that knowledge about semantic parsing and program model building should be incorporated into the statement-model and the program-model, respectively. In the NLPS we are working on, these two models are organizations of objects (in the object-oriented programming sense (Bobrow, 1981)), each possessing local knowledge and procedures. The user's input is first parsed by a syntactic analysis procedure which communicates subtrees to the statement-model objects for semantic judgments and annotations, such that the completed parse tree is trivially transformable into the statement model. In the second stage, the statement model is sent to an object in the program model (#PROGRAM) which sends messages to other program-model objects corresponding to components of the user's statement; it is these objects which perform the updating and modification operations.

This paper describes the statement-model and the program-model, their use in understanding Japanese program specifications, and how they have been shaped by the linguistic singularities of the Japanese input sentences dealt with so far.

[1]Sugiyama's current address is Advanced Computer Systems Department, SRI International, Menlo Park, CA 94025.

II MODELS

A. Program Model

To get a better understanding of the way users describe programs, we asked programmers to specify programs in a short paragraph, and sampled illustrative descriptions of simple programs from a Hyper COBOL user's manual (Fujitsu, 1981) (Hyper COBOL is the target programming language of KIPS). This resulted in a corpus of 60 program descriptions, comprising about 300 sentences.

The program model we built to deal with this corpus is divided into a model of files and a model of processes (Figure 1).

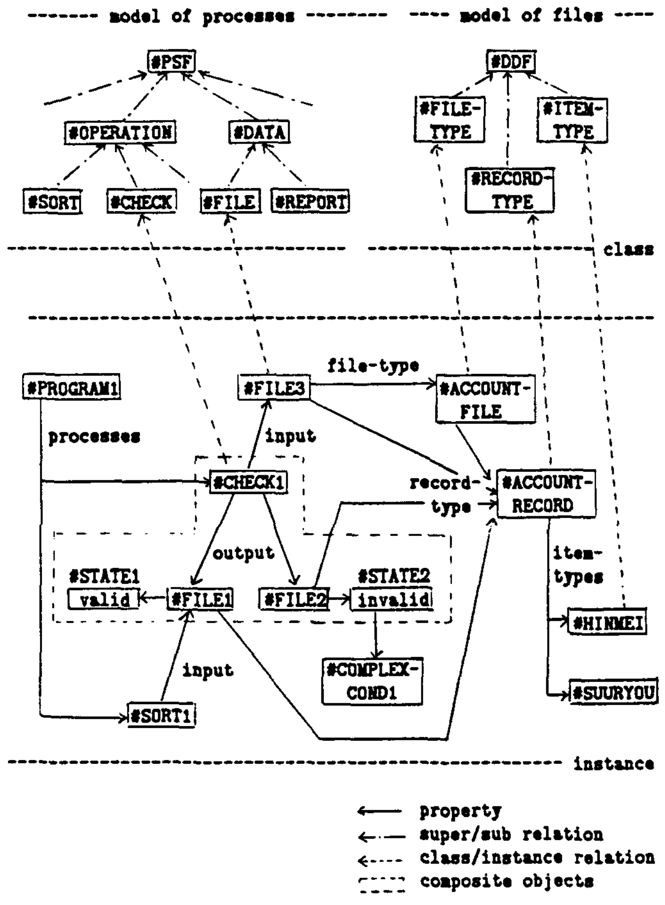


Figure 1. The program model

The model of files comprises in turn several sub-models, objects containing knowledge about file types, record types and item types. A particular file is represented by an object which is an instance of all three of these. Class-level objects have such properties as bearing a certain relation to other class-level objects, having a name, and so forth. For example, the object #RECORD-TYPE has ITEM-TYPES relations with the #ITEM-TYPE object, and DATA-LENGTH and CHARACTER-CLASS properties. Objects on the instance level have such properties as a specific data length and a specific name.

The model of processes is a taxonomy of objects bearing super/subset relations to one another. On the highest level we find such objects as #OPERATION, #DATA, #PROGRAM, #CONDITION, and #STATE.

The specific program-model, which is built up through a dialogue with the user, is a set of instance-level objects belonging to both file and process classes.

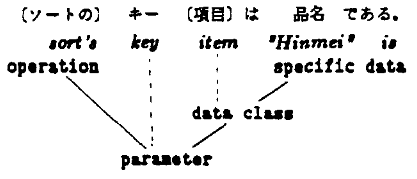
B. Statement Model

In a NLPS system, it is necessary to represent the content of the user's input sentences in an intermediary form, rather than incorporating it directly into the program model, because the user's statements may either contradict what was said previously, or omit some essential information. The statement model provides this intermediary representation, whose content must be checked for consistency, and sometimes augmented, before it is assimilated and acted upon.

The sentences in the corpus can, for the purpose of statement-model building, be classified into operations sentences, parameter sentences, and item-condition sentences (Figure 2). Their semantic components can be divided into nominal phrases and relations – names or descriptions of operations, parameters, data classes, and specific pieces of data (e.g. the item "*Hinmei*"), and relations between these[2] (Figure 3). Naming these elements, identifying subclasses of operations, and categorizing the dependencies yields the statement model (Figure 4): subcomponents of the sentence correspond to class-level objects organised in a super/sub hierarchy, and the content of the sentence as a whole corresponds to a system of instance-level objects, descendants from those classes.

operation sentence	大阪売上ファイルを，品名をキーにならべかえ，売上ファイル1へ出力する。 *Sort Osaka account file with a key "Hinmei", then output it to the account file 1.*
parameter sentence	ソートのキー項目は品名である。 *The key item of sort is "Hinmei".*
item-condition sentence	数量が0または99より大なら，～。 *If "Suuyou" is 0 or greater than 99, ...*

Figure 2. Three sentence types



〔ソートの〕 キー 〔項目〕は 品名 である。
sort's *key* *item* *"Hinmei"* *is*
operation
specific data
data class
parameter

Figure 3. The semantic elements

[2]Subordinating sentential conjunctions are treated as relations between states or operations, seen as described by sentential clauses.

III Understanding of Japanese

KIPS understands Japanese program specifications in two phases. The *sentence analysis phase* analyzes an input and generates an instance of a statement model. The *specification acquisition phase* builds an instance of the program model from the extracted semantics.

A. Implementing the Models

To realize a natural language understanding system using the models we are developing, objects in the models have to be dynamic as well as static, in the sense that the objects should express, for instance, how to instantiate themselves as well as static relations such as super/sub relations. Object-oriented and data-oriented program structures (Bobrow, 1981) are good ways to express dynamic objects of this sort. KIPS uses FRL (Roberts, 1977) extended by message passing functions to realize these programming styles.

B. Sentence Analysis

The sentence analysis phase performs both syntactic and sematic analysis. As described above, the semantics is represented in the statement model. Syntax in KIPS is expressed by rules of TEC (Sugiyama, 1982) which is an enhancement of PARSIFAL (Marcus, 1980). The fundamental difference is that TEC has look-back buffers whereas PARSIFAL has an attention shift mechanism. This change was made in order to cope with two important aspects of Japanese, viz., (1) the predicate comes last in a sentence, and (2) *bunsetsu*[3] sequences are otherwise relatively arbitrary.

The basic idea of TEC is as follows. To determine the relationship between a noun *bunsetsu*, which comes early in the sentence, and the predicate, the predicate *bunsetsu* has to be parsed. Since it comes last in the sentence, the noun *bunsetsu* has to be stored for later use to form an upper grammatical constituent. The arbitrary number of noun *bunsetsus* are stored in look-back buffers, and are later used one by one in a relatively sequence-independent way.

1. Overview

The syntactic characteristics of the sample sentences, which were found to be useful in designing the sentence analysis, are that (1) the semantic elements, which are stated above, correspond closely to *bunsetsu*, (2) parameter sentences and item-condition sentences can be embeded in operation sentences and tend to be expressed in noun sentences (sentences like "A is B"), and (3) operation sentences tend to be expressed in verb sentences (sentences like "do A"). Guided by these observations, parsing rules are divided into three phases; *bunsetsu* parsing, operand parsing, and

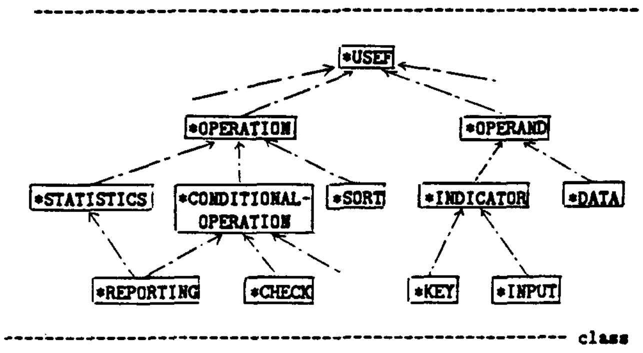


Figure 4. The statement model

[3]A linguistic constituent which approximately corresponds to "phrase" in English.

operation parsing. *Bunsetsu* parsing identifies from the input word sequence a set of *bunsetsu* structures, each of which contains at most one semantic element. Operand parsing makes up such operands as parameter and item-condition specifications that may be governed directly by operations. Operation parsing determines the relations between an operation and various operands that have been found in the input sentence. Each of these phases sends messages to the statement model, so that it can add to a parse tree information necessary for building the semantic structure of an input or can determine the relationship between the partial trees built so far. An

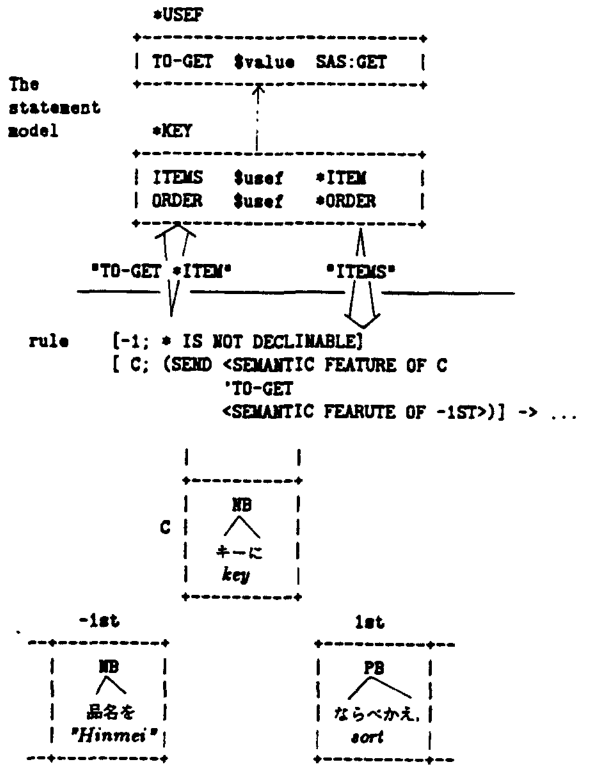


Figure 5. Syntax and Semantic interaction

instance of the statement model is extracted from the semantic information attached to the final parse tree.

2. Syntax and Semantics Interaction

Figure 5 shows how message passing between the syntactic component (rules) and the semantic component (model) occurs in order to determine the semantic relationship between the *bunsetsus* ("*Hinmei*" and *key*). The boxes denoted by -1st, C, 1st are grammatical constituent storages called look-back buffer, look-up stack, and look-ahead buffer in TEC (Sugiyama, 1982), respectively. One portion of the rule's patterns (viz. [-1;...]) checks if the constituent in the -1st buffer is not declinable. Another portion (viz. [C;...]) sends the message "TO-GET *ITEM" to the semantic component (*KEY) asking it to perform semantic analysis.

On receiving the message from the syntax rule, *KEY determines the semantic relation with *ITEM, and returns the answer "ITEMS". The process is as follows. The message activates a method corresponding to the first argument of the message (viz. TO-GET). Since the corresponding method is not defined in *KEY itself, it inherits the method SAS:GET from the upper frame *USEF. This method searches for the slot names that have the facet $usef with *ITEM, and finds the semantic relation ITEMS.

As illustrated in the example, the syntax and semantics interaction results in a syntactic component free from semantics, and a semantic component free from syntax. Knowledge of semantic analysis can be localized, and duplication of the same knowledge can be avoided through the use of an inheritance mechanism. Introducing a new semantic element is easy, because a new semantic frame can be defined on the basis of semantic characteristics shared with other semantic elements.

C. Specification Acquisition

Filling the slots which represent a user's program specification is considered as a set of subgoals and completing a frame as a goal. Program models are built through message passing among program model objects in a goal-oriented manner.

1. Subgoaling

[Strucure of subgoaling knowledge]

The input semantic structure to the acquisition (1) is fragmentary, (2) varies in specifying the same program, and (3) the sequence of specifying program functions is relatively arbitrary. To deal these phenomena, several subgoaling methods, each of which corresponds to a different way of specifing a piece of program information, are defined in different facets under a same slot. For example, a program model object #CHECK in Figure 6 has $file and $acquire facets under the slot INPUT.

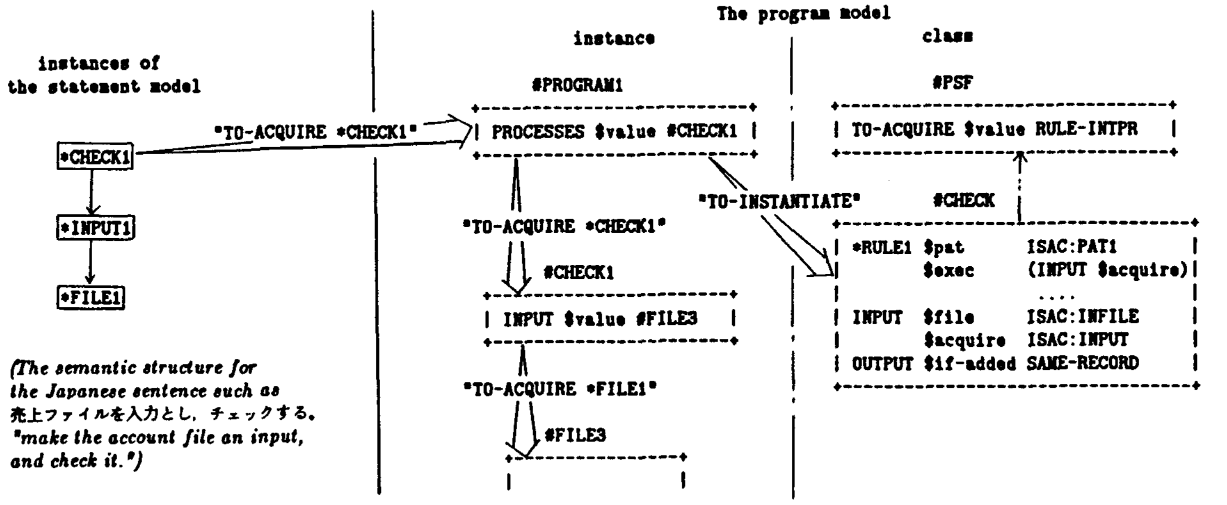


Figure 6. Subgoaling

In order to select one of the different subgoaling methods, depending on the input semantic structure, a rule-like structure is introduced. A pattern for a rule (e.g. *RULE1 in #CHECK) is defined under $pat which tests the input semantic structure, and an action part of a rule is defined under $exec which shows the subgoal's names (slots) to be filled and the subgoaling methods (facets) to do the job. The message "TO-ACQUIRE ~" triggers a rule interpreter. The interpreter is physically defined in the highest frame of the process model (#PSF), since it expresses overall common knowledge.

#PROGRAM1 has a discourse model in order to acquire information provided relatively arbitrarily. The current model depends on the kind of operations and the sequence in which they are defined. Usually, the most currently defined or referred to operation gets first attention.

[Process of subgoaling]

The example of acquisition of the semantic structure in Figure 6 begins with sending the message "TO-ACQUIRE *CHECK1" to #PROGRAM1. On receiving the message, #PROGRAM1 eventually instantiates the #CHECK operation, makes the instance (#CHECK1) one of the processes, and then send it another message "TO-ACQUIRE *CHECK1" which specifies what semantic structure it must acquire (viz. the structure under *CHECK1).

The message sent to #CHECK1 then activates the rule interpreter defined in #PSF. The interpreter finds *RULE1 as appropriate, and executes the subgoaling methods specified as (INPUT $acquire) and so forth. One of the methods (ISAC:INPUT) creates #FILE3, makes it INPUT of the current frame (#CHECK1), and asks it to acquire the remaining semantic structure (*FILE1).

2. Internal Subgoaling

As explained before, some inputs lack the information necessary to complete the program model. This information is considered to be in subgoals internal to the system and supplemented by either defaults, demons (Roberts, 1977) or composite objects (Bobrow, 1981). For example, the default is used to supplement the sorting order unless stated otherwise explicitly.

Demons are used to build a record type automatically. The input sentence seldom specifies the record types. This is because output record type is automatically calculable from the input record type depending on the operation employed. However, the program model needs explicit record type descriptions. This is accomplished by the demons defined under the OUTPUT slot in the operation frames. For example, when a output file is created for the operation #CHECK in Figure 6, the $if-added demon (viz. SAME-RECORD) is activated to find a record type for the output file. As shown in Figure 1, this results in finding the same record type (#ACCOUNT-RECORD) for the output files (#FILE1, #FILE2) as that of the input file (#FILE3).

Specification of output files is implicit in many cases. For example, the CHECK operation assumes that it creates a valid file which satisfies the constraints, and an invalid file which does not. As a natural way of implementation, composite objects are employed, and the output files as well as the files' states are also instantiated as a part of #CHECK's instantiation (Figure 1).

3. Discussion

Program specification acquisition is realized using the program model, which is a natural representation of the user's program image. This is accomplished through message passing, default usage, demon activation and composite objects instantiation. Knowledge in an object in the model is localized and hence easy to update. Inheritance makes it possible to eliminate duplicate representation of the same knowledge, and adding a new object is easy because of the knowledge localization.

IV CONCLUSION

This paper discussed the problems encountered when implementing a Japanese understanding subsystem in an interactive programming system, KIPS, and proposed an "object-centered" approach. The subsystem consists of sentence analysis and specification acquisition, and the task domain of each is modeled using dynamic objects. The "object-centered" approach is shown to be useful for making the system flexible. A prototype system is now operational on M-series machines and has successfully produced several dozens of programs from the Japanese specification. Our next research will be directed toward understanding Japanese sentences that contain other than the process specifications.

V ACKNOWLEDGEMENTS

The authors would like to express their thanks to Tatsuya Hayashi, Manager of Software Laboratory, for providing a stimulating place in which to work. We would also like to thank Dr. Don Walker, Dr. Robert Amsler and Mr. Armar Archbold of SRI International, who have provided valuable help in preparing this paper.

VI REFERENCES

Biermann,A.W.; Ballard,B.W.; Sigmon,A.H. An Experimental Study of Natural Language Programming. *Int. J. Man-Machine Studies*, 1983, *(18)*, 71-87.

Bobrow,D.G; Stefik,M. *The LOOPS Manual.* Technical Report, Xerox PARC, 1981. KB-VLSI-81-13.

Fujitsu Ltd. Hyper COBOL Programming Manual V01. , 1981. [in Japanese].

Heidorn,G.E. Automatic Programming Through Natural Language Dialogue: A Survey. *IBM J. Res. & Develop.*, 1976, *20(4)*, 302-313.

Marcus,M.P. *A Theory of Syntactic Recognition for Natural Language.* : MIT Press 1980.

McCune,B.P. *Building Program Model Incrementally from Informal Descriptions.* PhD thesis, Stanford Univ., 1979. AIM-333.

Roberts,R.B.; Goldstein,I.P. *The FRL Manual.* Technical Report, MIT, AI Lab., 1977. memo 409.

Sugiyama,K.; Yachida,M.; Makinouchi,A. A Tool for Natural Language Analysis: TEC. *25th Annual Convention, Information Processing Society of Japan*, 1982, , 1033-1034. [in Japanese].

Sugiyama,K.; Akiyama,K.; Kameda,M.; Makinouchi,A. An Experimental Interactive Natural Language Programming System. *The Transactions of the Institute of Electronics and Communication Engineerings of Japan*, 1984, *J67-D(3)*, 297-304. [in Japanese, and is being translated into English by USC Information Sciences Institute].

Ueda; Kanno; Honda. Development of Japanese Programming Language on Personal Computer. *Nikkei Computer*, 1983, *(34)*, 116-131. [in Japanese].