Semi-Automatic Program Construction From Specifications Using Library Modules

Fujio Nishida, Shinobu Takamatsu, Yoneharu Fujita, and Tadaaki Tani

Abstract— **This paper describes a method of semi-automatic specification refinement and program generation using library modules. Users write their specifications, modify and rearrange them so that the specifications can be refined with the aid of the library modules. When a specification is given, a refinement system called MAPS searches for library modules applicable to the given specification, replaces the specification with a more detailed description written in the operation part of the modules, and converts the refined specification into a program written in a programming language designated by the user. Case-like expressions or pseudo-natural language expressions are used for describing user's specifications and specifications for library modules.**

Index Terms—**Automatic documentation, multilingual translation, natural language, reusable module, semi-automatic linking, semi-automatic refinement, unification.**

I. Introduction

As the needs for improving software productivity and quality have become hot topics, the demand for automatic program generation has increased.

As for techniques concerned with program generation, various software engineering methodologies have been proposed [1]–[19]:

1) The techniques proposed in [1]–[3] to help construct large software systems provide methodologies for dividing and abstracting systems.
2) The study on module reusability, including that for requirements and program design [9]–[11], [16]–[18], suggested to us not only methods that were efficient for developing reliable software, but also ways feasible for automation.
3) Program transformation techniques suggested possibilities of generating programs from specifications which are written in very high-level languages such as limited natural languages [8].
4) Problem solving techniques [6], [7], [19], [20] have given impacts to automatic programming.

Manuscript received January 6, 1987; revised March 5, 1991.

F. Nishida was with the Department of Electrical Engineering, Faculty of Engineering, University of Osaka Prefecture, Mozu-Umemachi, Sakai, Osaka, Japan. He is now with the Department of Industrial Management, Fukui Institute of Technology, Gakuen, Fukui, Japan.

S. Takamatsu and T. Tani are with the Department of Electrical Engineering, Faculty of Engineering, University of Osaka Prefecture, Mozu-Umemachi, Sakai, Osaka, Japan.

Y. Fujita is with the Department of Computer Science and Intelligent Systems, Faculty of Engineering, Oita University, Tannoharu, Oita, Japan.

IEEE Log Number 9101137.

The authors studied a method to construct programs semi-automatically using library modules, and developed an experimental system called the library-Module-Aided Program construction System (MAPS). Compared with other studies such as [7]–[11], MAPS has the following unique features:

1) It transforms specifications written in a pseudo-natural language to specifications which are written using a class of case-grammar [27]. The latter specifications are hereafter called case-like specifications.
2) It has a powerful unification capability which links reusable modules and constructs target programs.
3) When a problem is given by a case-like specification which includes inputs and outputs, it automatically infers a linked set of library modules of which sum result satisfies the constraints between inputs and outputs.

The system configuration of MAPS is outlined in the next section. Case-like specifications and library modules are described in Section III. Specification refinement based on library modules and program generation are explained in Section IV. Transformation between case-like expressions and pseudo-natural language expressions is described in Section V, and experimental results are shown in Section VI.

II. System Configuration

Fig. 1 shows a schematic diagram of MAPS. MAPS consists of three primary subsystems: a pseudo-natural language specification analyzer, a refinement subsystem, and a paraphraser. The specification analyzer transforms specifications (written in a pseudo-natural language) to case-like specifications. Each case-like specification is formulated in such manner that each "case" associates a term with a procedure or a predicate.

The refinement subsystem searches for the library modules applicable to the given specification, replaces the specification with a more detailed description which is stored in the operation part of respective modules, or converts the case-like specification into a linked set of library modules. If MAPS cannot identify applicable modules, it refines the specification with the help of user's inputs. A concrete model of this interactive process is explained in Section VI.

The module library is a collection of fundamental program modules. The structure of the module library is hierarchical. For example, assume that "READ-IN-ARRAY" is the module procedure name, and both "READ-CHARACTER" and "INDEX-INCREMENT", . . . are the procedure names of modules used to construct the operation part of "READ-IN-

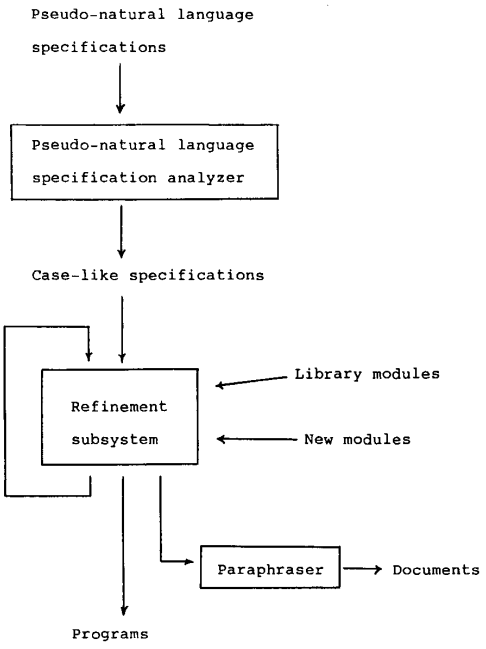


Fig. 1. System configuration.

ARRAY". Then "READ-IN-ARRAY" is hierarchically upper than "READ-CHARACTER" and "INDEX-INCREMENT".

In order to help programmers understand case-like specifications, the paraphraser generates a document written in a pseudo-natural language from these specifications. The refinement subsystem finally transforms the specification into a program written in a programming language such as Lisp or C, which is chosen by the user.

III. Case-Like Expressions of Specifications and Library Modules

The context of expressions used to describe case-like specifications is based on a case-grammar [30]. The case-grammar representation is similar to the tree structure representation [21] where each heading symbol corresponds to a tree-root, while each case label $C_i (i = 1, 2, \ldots, n)$ corresponds to a tree branch. Each case term $t_i (i = 1, 2, \ldots, n)$, which is the new heading symbol, corresponds to a tree-leaf or a root to a subsidiary tree. The tree structure is shown in Fig. 2. The tree structure of Fig. 2 is concisely represented with the following form:

$$\text{heading-symbol}(C_1 : t_1, \ldots, C_n : t_n). \quad (1)$$

The heading symbol corresponds to the structure name. Each structure name may represent a procedure name or a predicate symbol. The argument part consists of several pairs. Each pair consists of a case label and a term. Each term may represent a constant, a variable, an array, a function, a logical formula, or a tree expression. The case label describes the semantic role

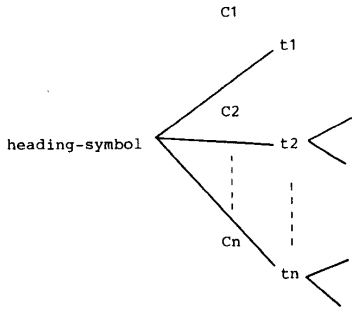


Fig. 2. The tree representation of a case-like expression.

of the corresponding term [27]. Some of the fundamental case labels used in this paper are as follows:

- OBJECT: the object to be processed. Case label OBJECT is sometimes omitted in printed representations for simplicity, especially when no other case is contained in the structure.
- SOURCE: the set or collection of data, including objects to be processed.
- PARTICIPANT: the auxiliary or complementary object.
- CONDITION: the processing condition.
- GOAL: the memory location for storing processed results.
- LOCATION: the location of the object.
- COMPARISON: the item to be compared with the object.
- KEY: the key item used for data processing.
- FORMAT: the format for presenting the object.
- MODE: the processing mode, usually an algorithm name used for processing.

Users can add new case labels to the arguments in order to describe various attributes and relations more precisely.

Mathematical formulas such as functions and arrays are described in the form of Pascal-like prefixed expressions (e.g., MULTIPLY(OBJECT:x,PARTICIPANT:y),TABLE(ROW:1..M,COLUMN:1..N). These case labels can be omitted for simplicity if no confusion arises).

The case-like expressions are classified into three types of expressions: procedure expressions; input–output expressions; and programming language-like expressions.

A. Procedure Expressions

The procedure expression takes the form shown in (1). The heading symbol represents a procedure name. The procedure name may be a verb like "retrieve." The argument part usually includes the GOAL case which stores the retrieved result. In an expression without a GOAL case, the result will be stored in the location having the same name as the heading symbol.

A string consisting of lowercase letters denotes a variable; a string consisting of uppercase letters denotes a constant. Each input variable and output variable in the user's specifications are written in uppercase letters and are processed as a constant in the unification process.

Example 1: Let us consider the following pseudo-English specification: "Retrieve from TABLE(1..N,1..M) all tuples

TABLE(I,1..M) that satisfy the condition TABLE(I,1)=R(1), and store the result in ANS(1..N,1..M)."

The above specification in pseudo-English can be transformed into the following procedure expression applying the method mentioned in Section V-A:

```
RETRIEVE(SOURCE:TABLE(1..N,1..M),
    OBJECT:TABLE(I,1..M)
        (PREDICATE:satisfy,OBJECT:*,
    CONDITION:TABLE(I,1)=R(1)),
    GOAL:ANS(1..N,1..M),
    MODE:FORALL-I)                    (1a)
```

The part which follows TABLE(I,1..M) in the case OBJECT is called a *modifying part.* This part corresponds to the relative clause "that satisfy the condition TABLE(I,1)=R(1)" in the specification. The star symbol "*" denotes the term TABLE(I,1..M) prefixed to the modifying part.

B. Input–Output Expressions

An input–output expression must include an expression on the input and an expression on the required output. The form is useful when the specifications are refined.

Let us consider an example:

```
IN : GIVEN(OBJECT:x,LOCATION:loc),
     P(OBJECT:x),
OUT: GIVEN(OBJECT:q(x),LOCATION:z)    (2a)
```

The form shown in (2a) is called an input–output expression. The above expression means that when any input x is GIVEN at a location loc and has a property P, there exists an output z which satisfies the relation $\forall x \exists z$ Q(OBJECT:z,PARTICIPANT:x) for which the Skohlem function q(x) is used. The input expression generally consists of a logical product of several literals, while the output expression consists of a logical product of several clauses. A clause is the logical sum of several literals. A literal is an atomic formula such as (1) or its negative form. In the above example, the input expression represents the logical product of literals of GIVEN(OBJECT:x, LOCATION:loc) and P(OBJECT:x).

How to use input–output expressions is explained in Section IV-C.

Variable types and data types which appear in input–output expressions must be defined in the type part of type expressions before they appear. The type expression has the same form as (1). An example of the type expression of a record structure is shown in (2b). Here, the heading symbol denotes the data name. The argument part describes various data attributes such as REGION, STRUCTURE, and VARIABLE-ROLE. Each attribute consists of an attribute name and an attribute value:

```
data-name(STRUCTURE:RECORD(
            ATTRNAME-1:value-1,
            ..., ATTRNAME-n:value-n)  (2b)
```

where the value-i stands for a value, a data name, or a recursive nest of (2b).

C. Programming Language-Like Expressions

Since programming language-like specifications have a feature that they can be refined and converted into efficient target programs, they are adopted when the specifications describe procedures.

Programming language-like expressions are classified into arithmetic expressions, data-type expressions, and control expressions. The arithmetic expressions are described in a prefixed form of Pascal notation and converted into a program written in the language assigned by the user.

Data-type expressions meet three aspects of data types—namely, region (e.g., real number or character), structure (e.g., scalar or struct), and variable role (e.g., input or output). The declaration part of the objective program in the assigned programming language is derived from data-type expressions.

Control expressions are classified into conditional branching and iteration.

1) Conditional Branching: Conditional branching consists of a procedure expression and an input–output expression. The procedure expression takes an IF-THEN-type expression as shown in the following:

```
PROC:IF-THEN(CONDITION:t(x),OBJECT:q(x),GOAL:z)
                                          (3)
```

which means: "if t(x) then store q(x) into z."

Similarly, the binary branch expression is given as follows:

```
PROC:IF-THEN-ELSE(
        CONDITION1:t(x), OBJECT1:q1(x),
        CONDITION2: ¬t(x), OBJECT2:
            q2(x), GOAL:z)            (4)
```

If q1(x) does not affect t(x), (4) is equivalent to the following expression:

```
IF-THEN(CONDITION:t(x), OBJECT:q1(x),
  GOAL:z);
IF-THEN(CONDITION: ¬t(x), OBJECT:q2(x),
  GOAL:z);
```

where the semicolon denotes a sequential composition of procedures.

2) Iteration: Iteration is classified into parallel type and serial type. In parallel type, the result of each iteration stage does not affect other stages. In serial type, the result of the preceding iteration stage is used at the next iteration stage.

a) Parallel-type iteration: Parallel-type iteration is used when an operation is applied to every element of an object repeatedly. In Example 1, the same matching operation is applied to the first column of each row. The procedure expression of the parallel-type iteration applied to such examples is shown in the following:

```
PROC:FOR(INDEX:i,FROM:m,TO:n,
        OBJECT:compute(OBJECT:x(i),GOAL:z(i)))
                                                    (5)
```

where "compute" stands for a concrete procedure name.

b) Serial-type iteration: The procedure expression of the serial type iteration is the following:

```
PROC:FOR(INDEX:i,FROM:m,TO;n,
         OBJECT:compute(OBJECT:x(i),
                PARTICIPANT:y, GOAL:y),
                GOAL:z)                             (6)
```

Example 2: A library module which gives the sum of n entity values of an array is described:

```
PROC:SUM(OBJECT:x(1..n), GOAL:z)
IN :GIVEN(OBJECT:x(1..n))
OUT :GIVEN(OBJECT:SUM(OBJECT:x(1..n)),
           LOCATION:z)
OP :ASSIGN(OBJECT:0, GOAL:y); __initialize
                                  y with 0
   FOR(INDEX:i, FROM:1, TO:n,
       OBJECT:PLUS(OBJECT:x(i),
       PARTICIPANT:y,
       GOAL:y));
                          __y <- x(i)+y
ASSIGN(OBJECT:y, GOAL:z);
```

Each of the procedure expressions (noted as (3)–(6)) is written using input–output expressions of (3a)–(6a), respectively:

```
IN :GIVEN(x)
OUT :¬t(x)∨GIVEN(OBJECT:q(x),                       (3a)
LOCATION:z)
IN :GIVEN(x)
OUT : ¬t(x)∨ GIVEN(OBJECT:q1(x),
LOCATION:z),
        t(x)∨GIVEN(OBJECT:q2(x),
LOCATION:z)                                         (4a)
```

where q1(x) does not affect t(x).

```
IN : GIVEN(OBJECT:x(m..n))
OUT : FORALL i (¬m ≤ i ≤ n
        GIVEN(OBJECT:compute(OBJECT:x(i)),
              LOCATION:z(i)))                       (5a)
IN :GIVEN(OBJECT:x(m..n))
OUT :GIVEN(OBJECT:FOR(INDEX:i,FROM:m,
      TO:n,OBJECT:compute(OBJECT:x(i),
      PARTICIPANT:y,
      GOAL:y),LOCATION:z))                          (6a)
```

where x(m..n) represents an array which has indices m, (m+1), . . . , n.

D. Library Modules

The specification for each library module is a set consisting of a PROC expression, an IN expression, an OUT expression, a TYPE expression, and an OP expression. The operation part or OP expression describes the detail of the procedure shown in the PROC expression. Tables I and II show examples of library modules. In the first example the rows of a two-dimensional array are sorted using jth column values of the array as the primary keys. The second example specifies a program to produce a graph that plots the points assigned by $f(x(m)) \ldots f(x(n))$, where the expression $f(x(m)) \ldots f(x(n))$ stands for an ordered set $\langle f(x(i)) | i = m, m+1, \ldots, n-1, n \rangle$. In this expression f is a variable function-symbol which can be instantiated to a given real valued function by applying the second-order unification mentioned in Section IV-A. In both examples a string consisting of lowercase letters stands for a variable which can be unified by the corresponding term of the given specification.

The OUT expression usually involves an expression similar to the PROC expression in the case of OBJECT following the predicate GIVEN. This expression is required to be refined by the module linking mentioned in Section IV-C. Let us take a program which consists of a sequence of n modules each of which is represented by input–output expressions:

```
1st module
IN:GIVEN(OBJECT:x) OUT:GIVEN(OBJECT:q1(x))
2nd module
IN:GIVEN(OBJECT:q1(x)) OUT:
        GIVEN(OBJECT:q2(x))
.....
n-th module
IN:GIVEN(OBJECT:qn-1(x)) OUT:
        GIVEN(OBJECT:qn(x))
```

In each OUT expression, GIVEN(OBJECT:qn(x)), instead of GIVEN(OBJECT:qn(qn-1(... q2(q1(x)) ...))), is included.

The basic idea of library module construction in MAPS is similar to that of the Draco system [9]. The library modules are classified into refinement modules and conversion modules. The refinement modules are further classified into fundamental modules common to various application fields, and application modules to be used for some specific application fields. Conversion modules are used to transform refined specifications to practical programs written in a programming language designated by the user.

E. User Specifications

Users of MAPS write specifications using case-like expressions or pseudo-natural language expressions (described in Section V) by referring the table which lists the headings of the stored modules. The table is constructed in advance and each heading in the table consists of heading symbols and argument parts of the PROC, IN, and OUT expressions.

The specifications are written using a mixture of procedure expressions, input–output expressions, and programming language-like expressions. The input–output expressions represent specifications more concisely than a sequence of procedure expressions. MAPS searches for a library module

TABLE I
AN EXAMPLE OF THE "SORT" MODULE

```
PROC:SORT(OBJECT:array(m1 .. n1,m2 .. n2),
          KEY_COLUMN:j,MODE:DESCENDING_ORDER),
IN:GIVEN(OBJECT:array(m1 .. n1,m2 .. n2),KEY_COLUMN:j),
OUT:GIVEN(OBJECT:SORT(OBJECT:array(m1 .. n1,m2 .. n2),
                      KEY_COLUMN:j,MODE:DESCENDING_ORDER)),
TYPE: array(REGION:REAL,STRUCTURE:ARRAY(m1 .. n1,m2 .. n2),
            VARIABLEROLE:INOUT),
      m1,n1,m2,n2,j(REGION:INTEGER,STRUCTURE:SCALAR,
                    VARIABLEROLE:INPUT)
        ....................................
OP:FOR(INDEX:i,FROM: m1,TO:n1,
       OBJECT:FOR(INDEX:k,FROM:+(i,1),TO:n1,
                  OBJECT:IF-THEN(CONDITION:GREATER_THAN(
                                           OBJECT:array(k,j),
                                           COMPARISON:array(i,j)),
                                 OBJECT:EXCHANGE_VECTOR(
                                OBJECT:array(i,m2 .. n2),
                                PARTICIPANT:array(k,m2 .. n2)))));
```

which has an output expression unifiable with the expressions included in the user's specification. If any unifiable module is found, MAPS then tries to refine the specification and produce a linked set of modules which satisfies the input and output relation. If the specification written in the input–output expression is not refinable, the user divides the specification manually and rewrites the divided part using the procedure expressions or the programming language-like expressions which can be unifiable with the modules in the library.

Example 3: Let us consider the specification for a program which sorts a table containing examinees' grades. The following specification is written in pseudo-English:

```
TYPE:MARK-TABLE:ARRAY[1..1000,0..6]
(1) FOR I FROM 1 TO 1000
        READ MARK-FILE IN ARRAY
        FORM AND
        STORE THE RESULT IN MARK-
            TABLE[I,1 .. 5];
(2) FOR I FROM 1 TO 1000
        FIND THE SUM OF MARK-TABLE[I,1
        .. 5] AND
        STORE THE RESULT IN MARK-
        TABLE[I,6];
(3) SORT MARK-TABLE[1 .. 1000,0 .. 6]
        IN DESCENDING ORDER
        BY TAKING THE 6-TH COLUMN AS
        THE KEY;
(4) FOR I FROM 1 TO 1000
        STORE I IN MARK-TABLE[I,0];
(5) WRITE MARK-TABLE[1 .. 1000,0 .. 6]
        ON MARK-FILE IN ARRAY FORM;
```

The case-like expressions for the same specification is the following:

```
(1) FOR(INDEX:I,FROM:1,TO:1000,
        OBJECT:READ-IN-ARRAY(OBJECT:MARK-
    FILE,
    GOAL:MARK-TABLE(I,1 .. 5)));
(2) FOR(INDEX:I,FROM:1,TO:1000,
      OBJECT:SUM(OBJECT:MARK-TABLE
            (I,1 .. 5),
      GOAL:MARK-TABLE(I,6)));
(3) SORT(OBJECT:MARK-TABLE(1 .. 1000,
            0 .. 6),
         KEY_COLUMN:6,MODE:DESCENDING_ORDER);
(4) FOR(INDEX:I,FROM:1,TO:1000,OBJECT:
          :=(MARK-TABLE(I,0),I));
(5) WRITE-IN-ARRAY(OBJECT:
          MARK-TABLE(1 .. 1000,0 .. 6),
          GOAL:MARK-FILE);
```

IV. REFINEMENT OF SPECIFICATIONS AND CONVERSION TO PROGRAMS

If specifications are given in pseudo-natural language expressions, MAPS first transforms them to the case-like expressions. If the heading included in the case-like expression is a procedure expression, MAPS searches for a library module which has the procedure name unifiable with that heading name and checks that its argument part also is unifiable. What is unification is explained in Section IV-A. MAPS refines the heading by substituting the operation part of the unified module. If the heading is an input–output expression, MAPS searches for a linked set, which is the sequence of several

TABLE II
EXAMPLES OF GRAPH DRAWING MODULES

```
PROC:PRINTGRAPH(OBJECT:ORDERED_SET(OBJECT:f(x(m)) .. f(x(n))),
                PARTICIPANT:x(m .. n),FORMAT:form
                GRAPH_RANGE:X_RANGE(gx_min,gx_max),
                            Y_RANGE(gy_min,gy_max)),
IN:GIVEN(OBJECT:x(m .. n)),
   GIVEN(OBJECT:ORDERED_SET(f(x(m))  .. f(x(n))),
         LOCATION:y(m  .. n)),
   GIVEN(OBJECT:MAX(OBJECT: y(m .. n)),LOCATION:maxy),
   GIVEN(OBJECT:MIN(OBJECT: y(m .. n)),LOCATION:miny),
   GIVEN(OBJECT:x(n),LOCATION:maxx),
   GIVEN (OBJECT:x(m),LOCATION:minx),
OUT:GIVEN(OBJECT:PRINTGRAPH(OBJECT:
                      ORDERED_SET(f(x(m)) .. f(x(n))),
                      PARTICIPANT:x(m .. n), FORMAT:form,
                      GRAPH_RANGE:X_RANGE(gx_min,gx_max),
                                  Y_RANGE(gy_min,gy_max))),
TYPE: 'gx_min,gx_max,gy_min,gy_max'(
           REGION:INTEGER,STRUCTURE:SCALAR,VARIABLEROLE:INPUT),
       x(REGION:REAL,STRUCTURE:ARRAY(m .. n),VARIABLEROLE:INPUT),
        ........................................,
OP:DRAW_XY_AXIS(FORMAT:STRIPE,
                VALUE_RANGE:X_RANGE(minx,maxx),
                            Y_RANGE(miny,maxy),
                GRAPH_RANGE:X_RANGE(gx_min,gx_max),
                            Y_RANGE(gy_min,gy_max));
 FOR(INDEX:i,FROM:1,TO:n,
     OBJECT:PRINTGRAPHPT(OBJECT:y(i),x(i),FORMAT:STRIPE,
                VALUE_RANGE:X_RANGE(minx,maxx),
                            Y_RANGE(miny,maxy),
                GRAPH_RANGE:X_RANGE(gx_min,gx_max),
                            Y_RANGE(gy_min,gy_max)));

PROC: COMPUTE_FUNCTION(OBJECT:ORDERED_SET(f(x(m)) .. f(x(n))),
IN: GIVEN(OBJECT:x(m .. n)),
OUT: GIVEN(OBJECT:ORDERED_SET(f(x(m)) .. f(x(n))),
          LOCATION:y(m .. n)),
TYPE:
    x(REGION:REAL,STRUCTURE:ARRAY(m .. n),VARIABLEROLE:INPUT),
    y(REGION:REAL,STRUCTURE:ARRAY(m .. n),VARIABLEROLE:OUTPUT),
OP: FOR(INDEX:i,FROM:m,TO:n,
        OBJECT: :=(y(i),f(x(i))) );
```

where ' :=(y,x)' is a programming language-like expression equivalent to ' ASSIGN(OBJECT:x,GOAL:y)'.

library modules satisfying the given input–output expression. If any sequence is searched, MAPS replaces the input–output expression with the procedure expressions of the modules involved in the linked set.

A. Unification

The refinement of a case-like specification is performed using library modules. The refinement is made by looking at the heading of every library module, which consists of a PROC expression, an IN expression, an OUT expression, and a TYPE expression. The unification can be carried out systematically over a wide variety of input–output specifications by using the unification algorithm of the first-order predicate calculus, as well as the second-order predicate calculus. The second-order unification is used to unify such a predicate as $t(x)$ of (3) and functions associating an indefinite number of arguments.

The first-order unification stated here is performed in the following way. Suppose that expression A in a case-like specification has the form: $p(C_1 : t_1, \ldots, C_n : t_n)$, where p is a heading symbol, and expression B in module specification has the form: $p(C_1 : x_1, \ldots, C_n : x_n)$. The unification between A and B is performed so that x_i is substituted by t_i (for all t_i, where $i = 1, 2, \ldots, n$) using the algorithm of [4].

The second-order unification is performed in the following way. Suppose that an expression in a case-like specification has the form, $p(C_1 : t_1, \ldots, C_n : t_n)$, and an expression in module specification has the form, $q(C_1 : x_1, \ldots, C_m : x_m)$, where q is a variable. The heading symbol q is substituted by the imitation [5] which will be the expression in (10). When the imitation is applied to q, the following substitution will be performed:

$$q \leftarrow \text{LAMBDA } u_1 \ldots u_m \cdot p(C_1 : f_1(u_1, \ldots, u_m), \ldots, C_n : f_n(u_1, \ldots, u_m)). \quad (7a)$$

In the above, "$\leftarrow$" denotes the substitution of the right-hand side for the left-hand side, and the expression interposed between "$\cdot$"s denotes the body of the lambda notation.

Then the projection [5] shown in the following is applied:

$$f_i \leftarrow \text{LAMBDA } u_1 \ldots u_m \cdot u_i \cdot \quad (7b)$$

where i=1, 2, . . . , n. As a result the module specification expression will be transformed to an expression with p as the heading symbol. Then x_i is substituted by t_i by the first-order unification.

The simplified algorithms for the first-order unification[4] and the second-order unification[5] are shown in the following.

Let us suppose entities α and β, where entities mean terms (individual terms and functional terms) or well-formed formulas defined in predicate logic.

α is an entity consisting of symbols

$$a_1 \ldots a_k \ldots a_n \quad (8a)$$

and is included either in a procedure expression of a library module or in its input or output predicate.

β is an entity consisting of symbols

$$b_1 \ldots b_k \ldots b_m \quad (8b)$$

and is included in a given set of specifications where $k < m$, $k < n$, and $m \geq n$.

Assume that a_k and b_k are not identical or that $a_k \neq b_k$, where $a_i = b_i$ (i = 1, 2, . . . , (k-1)).

Then the unification between β and α is made in the following ways:

1) In the case that a_k is an individual variable where k=1, 2, . . . , n, the first-order unification procedure is applied to a_k in the following way:

$$a_k \leftarrow b_k b_{k+1} \ldots b_{k+r} \ (0 \leq r \leq m - k) \quad (9)$$

where the sequence $b_k b_{k+1} \ldots b_{k+r}$ is a subentity of β. The subentity is a constituent of an entity. The subentity of a function may be a set of symbols consisting of a function symbol and succeeding arguments. If, for example, an argument of function $f1$ is function $f2$, then $f2$ is the subentity of $f1$.

2) In the case that a_k is the head of a subentity and is the symbol of a variable function or a variable predicate which has p parameters—$a_{k+1}, a_{k+2}, \ldots, a_{k+p}$—both imitation and projection are tried.

1) Imitation: Let b_k be the head of a subentity representing a function symbol, where the function has q parameters. Symbol a_k is substituted with the subentity, $b_k b_{k+1} \ldots b_{k+q}$, where b_k is the first symbol:

$$a_k \leftarrow \text{LAMBDA} u_1 \ldots u_p \cdot b_k h_1 \ldots h_q \cdot \quad (10)$$

where $h_i = f_i u_1 \ldots u_p (1 \leq i \leq q)$. f_i is a function symbol which does not appear in α and β.

2) Projection: If the type of some a_{k+i}, where $1 \leq i \leq p$, is identical to the type of $a_k a_{k+1} \ldots a_{k+p}$, then the subentity $a_k a_{k+1} \ldots a_{k+p}$ is projected to a_{k+i}:

$$a_k \leftarrow \text{LAMBDA} u_1 \ldots u_p \cdot u_i \cdot \quad (11)$$

where $1 \leq i \leq p$.

The notion of the type referred to is explained in the following.

The set T of types [5], which are different from data type, is defined inductively as follows from the set T_0 of basic types:

1) If $t \in T_0$, then $t \in T$, where T_0 is the set consisting of CLASS and BOOLEAN. CLASS is the type of individual element and BOOLEAN is the type of truth value.
2) If $t_i \in T (1 \leq i < n)$ and $t_n \in T_0$, then $(t_1, \ldots, t_n) \in T$.

For example, $(\underbrace{\text{CLASS}, \ldots, \text{CLASS}}_{n}, \text{CLASS})$ is the type of each function symbol which has n parameters of individual elements, and $(\underbrace{\text{CLASS}, \ldots, \text{CLASS}}_{m}, \text{BOOLEAN})$ is the type of each predicate symbol which has m parameters of individual elements.

The types of functional expression and lambda expression are defined recursively as follows, where $\tau(x)$ represents the type of symbol x or expression x:

1) If $\tau(f) = (t_1, \ldots, t_n)$ and $\tau(x_i) = t_i (1 \leq i \leq m, 0 \leq m \leq n)$ then

$$\tau(fx_1 \ldots x_m) = \begin{cases} t_n, & \text{in the case } m = n-1 \\ (t_{m+1}, \ldots, t_n), & \text{in the case } m < n-1 \end{cases}$$

2) If $\tau(u_i) = t_i (1 \leq i \leq m)$ and $\tau(x) = t$ then $\tau(\mathrm{LAMBDAu_1 \ldots u_m \cdot x\cdot}) = (\mathrm{t_1, \ldots, t_m, t})$.

The LAMBDA expression applied in (11) maps $u_1 \ldots u_p$ to u_i. The sequence $a_k a_{k+1} \ldots a_{k+p}$ is replaced by a_{k+i} in α to form a new term sequence α', and the unification process is applied to α' and β. All substitutions generated by this recursive unification process form the complete substitution necessary to effect the unification.

Example 4: (a) Two arithmetic entities $f(x, y)$ as α and $-(+(x, y), /(x, y))$ as β can be unified by the following projection substitution:

$$f \leftarrow \mathrm{LAMBDA}uv \cdot u \cdot .$$

This projection substitution results in $f(x, y)$ being replaced by $\mathrm{LAMBDAuv \cdot u \cdot (x, y)}$, which is transformed into x by the lambda conversion. Next we use (9) in the following to replace the individual variable x:

$$x \leftarrow -(+(x, y), /(x, y)).$$

Now the replaced matches the given literal in β.
Similarly,

$$f \leftarrow \mathrm{LAMBDA}uv \cdot v\cdot$$

and

$$y \leftarrow -(+(x, y), /(x, y))$$

completes other unification.

(b) If we apply imitation substitution to the same problem we take the steps shown in the following. The imitation substitution gives the expression:

$$f \leftarrow \mathrm{LAMBDA}uv \cdot -(f1(u, v), f2(u, v)) \cdot .$$

Then entity α is replaced by $-(f1(x, y), f2(x, y))$. We apply imitation substitutions to $f1$ and $f2$:

$$f1 \leftarrow \mathrm{LAMBDA}uv \cdot +(f3(u, v), f4(u, v))\cdot$$
$$f2 \leftarrow \mathrm{LAMBDA}uv \cdot /(f5(u, v), f6(u, v)) \cdot .$$

Then entity $-(f1(x, y), f2(x, y))$ is replaced by

$$-(+(f3(x, y), f4(x, y)), /(f5(x, y), f6(x, y))).$$

We apply the four projection substitutions as follows:

$$f3 \leftarrow \mathrm{LAMBDA}uv \cdot u\cdot, \quad f4 \leftarrow \mathrm{LAMBDA}uv \cdot v\cdot$$
$$f5 \leftarrow \mathrm{LAMBDA}uv \cdot u\cdot, f6 \leftarrow \mathrm{LAMBDA}uv \cdot v \cdot .$$

These substitutions change the entity α to $-(+(x, y), /(x, y))$.

B. Refinement of Procedure Expressions

When MAPS finds a procedure expression in the library modules which can be matched to a procedure expression included in the given specification, it refines the procedure expression of the latter by using the operation parts of the unified procedure expression of the former. The substitution in the unification will be made for all variables appearing in the library module.

There are two methods in the refinement. In the direct replacement method the original specification part is replaced by the operation part which has been unified. In the procedure call method the original specification part is replaced by the form calling the operation part which has been refined as the subprogram. Draco system has also applied these two methods. MAPS allows the user to chose one of these methods.

In the procedure call method, MAPS constructs the procedure name, parameters, and declaration statements referring to the procedure expressions and the type expressions in the headings of the library modules. The user can chose "call by value" or "call by reference," referring to the calling type of target programming language. Then MAPS constructs the body part by using the unified operation part of the module.

Example 5: The case-like specification given in Example 3 is first refined by applying the direct replacement method. The vector-sort module shown in Table I and some library modules are used. The replaced specification shown in (1)–(3) and (5) of the following have resulted by the replacement using READ-IN-ARRAY, SUM, SORT, and WRITE-IN-ARRAY, respectively:

```
(1)FOR(INDEX:I,FROM:1,TO:1000,
    OBJECT:FOR(INDEX:I1,FROM:1,TO:5,
      OBJECT:READ(OBJECT:MARK-FILE,
        GOAL:MARK-TABLE(I,I1))));
(2)FOR(INDEX:I,FROM:1,TO:1000,
    OBJECT: :=(MARK-TABLE(I,6),0);
      FOR(INDEX:I4,FROM:1,TO:5,
        OBJECT: :=(MARK-TABLE(I,6),
          +(MARK-TABLE(I,6),
          MARK-TABLE(I,I4)))));
(3)FOR(INDEX:I5,FROM:1,TO:1000,
    OBJECT:FOR(INDEX:K5,FROM:+(I5,1),TO:1000,
      OBJECT:IF-THEN(
        CONDITION:GREATER_THAN(
          OBJECT:MARK-TABLE(K5,6),
          COMPARISON:MARK-TABLE(I5,6)),
        OBJECT:EXCHANGE_VECTOR(
          OBJECT :MARK-TABLE(K5,0 .. 6),
          PARTICIPANT:MARK-TABLE
            (I5,0 .. 6)))));
(4)FOR(INDEX:I,FROM:1,TO:1000,OBJECT:
    :=(MARK-TABLE(I,0),I));
(5)FOR(INDEX:I18,FROM:1,TO:1000,
  OBJECT:FOR(INDEX:I28,FROM:0,TO:6,
    OBJECT:WRITE (OBJECT:MARK-TABLE
      (I18,I28),
        GOAL:MARK-FILE)));
```

C. Refinement of Input–Output Expressions

The linking of library modules is necessary in order to replace an input–output expression with a linked set of library modules. The linking of modules is made using expressions called *clause forms.* The two expressions of (2a) are put together to create the clause form shown in the following:

$$\text{GIVEN}(q(x)) \vee \neg\ \text{GIVEN}(x) \vee \neg P(x). \tag{12}$$

IN and OUT expressions of library modules are converted to a clause form, and OP expression are in the form shown below:

$$\text{OP} : z.q := q0(x) \tag{13}$$

where $q0(x)$ stands for a certain operation specified in the library module.

The linking of modules is made in the following steps. Let us assume that the given specification has the following input–output expression in clause forms:

$$\begin{aligned}&\text{GIVEN}\ (Q(A))\vee\neg\text{GIVEN}(Q1(A))\vee\neg P1(Q1(A))\\ &\quad\vee\neg\text{GIVEN}(Q2(A))\vee\neg P2(Q2(A))\\ &\quad\cdots\cdots\cdots\cdots\cdots\cdots\\ &\quad\vee\neg\text{GIVEN}(Qm(A))\vee\neg Pm(Qm(A)).\end{aligned} \tag{14}$$

Expressions (15a) and (15b) describe the assumption that the above specification is not feasible or does not hold:

$$\neg\text{GIVEN}(Q(A)) \tag{15a}$$

$$\begin{aligned}&\text{GIVEN}(Q1(A)), P1(Q1(A)),\\ &\text{GIVEN}(Q2(A)), P2(Q2(A)),\\ &\cdots\cdots\cdots\cdots\cdots\cdots\\ &\text{GIVEN}(Qm(A)), Pm(Qm(A)).\end{aligned} \tag{15b}$$

If there is a refutation from (15a) and (15b) and a set of clause forms in library modules that support (14), specification refinement is permitted.

We classify the refinement step into two classes: *cascade connection* and *controlled connection.*

1) Cascade Connection: Linking Modules: A cascade connection of library modules is performed using a refutation-proof method driven by goal. As a result of the connection, we find a set of clauses in the form of (12), where each of the clauses represents the specification of a library module. Fig. 3 shows a refutation proof tree— $C0$ corresponds to the clause of (15a), which we call a *top clause*, and $Bi(i = 0, 1, \ldots, n-1)$ corresponds to clauses in (15b) or some clause which describes a library module. $Ci(i = 1, 2, \ldots, n)$ is a resolvent produced by $Ci - 1$ and $Bi - 1$ through modus ponens, and Cn is a null resolvent.

The input–output expression of each library module is converted to the clause form of (12). Then the clause form is coded in the form of the prolog statement as follows:

$$\text{GIVEN}(q(x)) :- \text{GIVEN}(x), P(x).$$

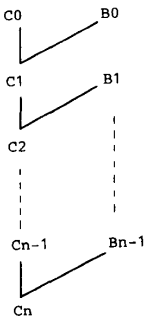


Fig. 3. A refutation proof tree.

The output expression of a user specification is converted to a prolog statement which has only one body part and lacks the head part shown in (15a). The input expression of the same user specification is converted to another prolog statement which has only the head part shown in (15b).

By executing these prolog statements, the prolog processor carries out the refutation proof. When the goal—i.e., the output expression—is proven, the linked set of library modules is produced.

The procedural sequence which is performed to find a cascade connection is as follows:

1) Convert the given specification to a clause form.
2) Perform the refutation proof by putting the negation of the output predicate of the specification clause as the top clause, giving the input clauses as premise, and identifying modules needed to carry out the proof.
3) Instantiate the procedure expression and operation parts of the library modules which are found.
4) Arrange the instantiated procedure expressions of the modules so that the corresponding input and output data satisfy a parent–child order, which is defined by the ordered pair of a datum A and a datum B, where A is needed to evaluate B. Then refine each instantiated procedure expression according to the direct replacement or the procedure call method mentioned in Section IV-B.

The normal execution of MAPS produces a linked set consisting of an average of ten modules in about 1 s.

Example 6: Suppose that the specification is given in the following pseudo-English:

'GIVEN A, DISPERSION(A)IS OBTAINED
AT Z',

This same specification is converted to the following clause form:

GIVEN(OBJECT:DISPERSION(A), LOCATION:Z)
$\vee\neg$GIVEN(OBJECT:A).

The negation of the above clause is given in the following clauses ⑤ and ⑥

¬GIVEN(OBJECT:DISPERSION(A),LOCATION:Z) ....⑤

GIVEN(OBJECT : A). ....⑥

Four library modules of the clause forms are shown in ①, ②, ③, ④ (The procedure expression for each module is also shown.)

GIVEN(OBJECT:NUMBER(x), LOCATION:y)
∨¬GIVEN(OBJECT:x) ....①

PROC:FIND-NUMBER(OBJECT:x, GOAL:y)

GIVEN(OBJECT:SUM(x), LOCATION:y)
∨¬GIVEN(OBJECT:x) ....②

PROC:SUM-UP(OBJECT:x, GOAL:y)

GIVEN(OBJECT:AVERAGE(x), LOCATION:y3)
∨¬GIVEN(OBJECT:SUM(x),LOCATION:y1)
∨¬GIVEN(OBJECT:NUMBER(x),LOCATION:y2)
....③

PROC:AVERAGE(OBJECT:{y1,y2}, GOAL:y3)

GIVEN(OBJECT:DISPERSION(x),LOCATION:y3)
∨¬GIVEN(OBJECT:AVERAGE(x),LOCATION:y1)
∨¬GIVEN(OBJECT:NUMBER(x),LOCATION:y2)
∨¬GIVEN(OBJECT:x) ....④

PROC:FIND-DISPERSION(OBJECT:{y1,y2,x},
GOAL:y3)

Fig. 4 shows the refutation tree of which clause ⑤ is the top clause. Rearranging the procedure expressions according to the parent–child order, the refined specification shown in the following is derived:

FIND-NUMBER(OBJECT:A, GOAL:y2);
SUM-UP(OBJECT:A, GOAL:y1′);
AVERAGE(OBJECT:{y1′,y2}, GOAL:y1);
FIND-DISPERSION(OBJECT:{y1,y2,A},
GOAL:Z);

The procedure expressions will be further refined.

2) Controlled Connection: This subsection describes a method of refining the OBJECT case included in the control expression (described in Section III) by applying conditional branching or iteration.

a) Conditional branching: If conditional branching is specified using the procedure of expression of (3) or the input–output expression of (3a), the refinement can be reduced to the refinement of the input–output expression embedded in the OBJECT case, as shown in (16a):

IF-THEN(CONDITION:t(x),OBJECT:
(IN:GIVEN(x),t(x), OUT:GIVEN(q(x))),
GOAL:z) (16a)

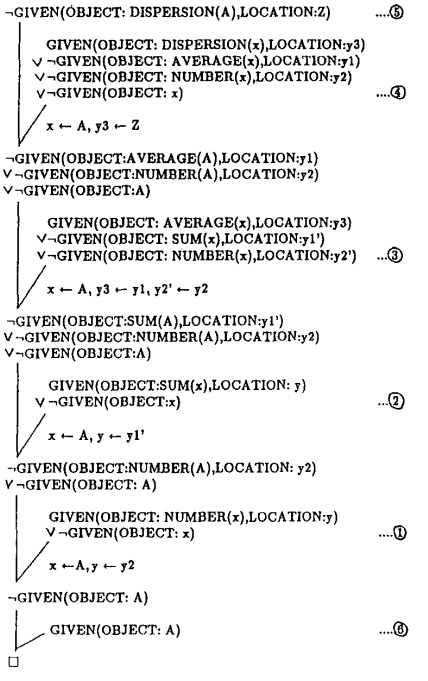


Fig. 4. A refutation process.

If conditional branching is specified using (4), the specification will be refined to the form of (16b):

IF-THEN-ELSE(CONDITION1:t(x),
OBJECT1:(IN:GIVEN(x),t(x),
OUT:GIVEN(q1(x)))),
CONDITION2:¬t(x),
OBJECT2:(IN:GIVEN(x),¬t(x),
OUT:GIVEN(q2(x)))),
GOAL:z) (16b)

The refinement of the input–output expressions embedded in expressions such as (16a) and (16b) will be performed using cascade connection.

b) Iteration: Similar to the conditional branching, the refinement of iteration expressions can be reduced to the refinement of the input–output expressions embedded in the iteration expressions.

The operation part of the parallel iteration module of (5) is refined as follows:

FOR(INDEX:i, FROM:m, TO:n,
OBJECT:(IN:GIVEN(x(i)) m≤i≤n,
OUT:GIVEN(compute(OBJECT:x(i),
GOAL:z(i))))) (17a)

Similarly, the operation part of the serial iteration module of (6) is refined as follows:

Again, the cascade connection is used to refine the IN–OUT expression.

TABLE III
THE CONTENT OF THE "FOR" MODULE

```
PROC:FOR(INDEX:i,FROM:m,TO:n,OBJECT:s),
TYPE:'i,m,n'( REGION:INTEGER,VARIABLEROLE:INPUT),
OP:(LISP  (SETQ i m)                    __When user chooses LISP,
          (LOOP ( ) s                   __this part is used for
               (SETQ i (ADD1 i))        __refinement
               (COND ((GREATERP i n)
                     (EXIT-LOOP)))))
   (C     FOR(i=m;i≤ n;++i){s})         __When user chooses C,
     ..............................     __this part is used
     ..............................     __for refinement
```

D. Fusion of Iterative Loops

The specification refinement using the library modules generally produces a large collection of the instantiated modules. The iterations included in this module collection usually have the same control structures. In this subsection we shall consider optimization of iterations by what is called *fusion*. MAPS can perform the fusion for FOR-type iteration shown in (5) and (6), where iterations to be fused have the same initial index values, the same step index values, and the same final index values.

Let us suppose that two FOR-type iterative expressions are generated in the refinement process. In the first iteration, output Z1, initialized with Z10, is computed using input X1 and function Q1. In the second iteration, output Z2, initialized with Z20, is computed using input X2 and function Q2:

```
Z1:=Z10;
FOR (INDEX:I, FROM:M, TO:N),
      OBJECT: Z:= Q1(X1(I), Z1));
Z2:=Z20;
FOR (INDEX:J, FROM: M, TO: N
      OBJECT: Z:= Q2(X2(J), Z2));
```

Under these conditions, these iterations are put together to form a single iterative expression as follows:

```
Z1:=Z10; Z2:=Z20;
FOR (INDEX:I, FROM:M, TO:N,
     OBJECT: (Z1:=Q1(X1(I), Z1);
              Z2:=Q2(X2(I), Z2)));
```

E. Transformation to Programming Languages

When the refined specifications only consist of primitives that are transformable to a text in the objective programming language, the refinement is completed. MAPS then transforms the refined specification into the program written in a programming language (C or LISP), and transforms the type part into the declaration statements. If LISP is chosen, for example, MAPS transforms the record-type specifications into a LISP program by using an associative list and an associative function.

```
main( )
 {int 1121; int 1111; int k51;
  int 141; int 1; int 151; int 1111;
  ..............................
  int mark-table(1001)(7);
  int wk51(2)(7)
  ..............................
  for(1 = 1; 1 <= 1000; ++ 1)
   {
    mark-table(1)(6) = 0;
    for(141 = 1; 141 <= 5; ++ 141)
     {
      mark-table(1)(6) =
          mark-table(1)(6) + mark-table(1)(141));
     };
   };
  for(151 = 1; 151 <= 1000; ++ 151)
   {
    for(k51 = (151 + 1); k51 <= 1000; ++ k51)
     {
      if(mark-table(k51)(6) > mark-table(151)(6))
       {
        for(1101 = 0; 1101 <= 6; ++ 1101)
         {
          wk51(1)(1101) = mark-table(151)(1101);
         };
        for(1111 = 0; 1111 <= 6; ++ 1111)
         {
          mark-table(151)(1111) = mark-table(k51)(1111);
         };
        for(1121 = 0; 1121 <= 6; ++ 1121)
         {
          mark-table(k51)(1121) = wk51(1)(1121);
         };
       };
     };
  ..............................
 }
```

Fig. 5. The produced C program.

MAPS transforms the specified case-like control expressions into a control statement by using a module such as is shown in Table III.

The specification in Example 5 is further refined and transformed into the program shown in Figs. 5 and 6. In Fig. 6, "store," which is a procedure assigning a value to an array, is provided by the target LISP system.

V. TRANSFORMATION BETWEEN CASE-LIKE EXPRESSIONS AND PSEUDO-NATURAL EXPRESSIONS

The pseudo-natural language specification analyzer transforms given specifications written in pseudo-natural expressions into case-like expressions, as shown in Fig. 1. The

```
(array mark-table t 1001 7)
(array wk51 t 7)
(defun exam nil
 (prog (i11 i51 i i41 k51 i101 i121 ...)
  ....................................
  (setq i 1)
  (loop nil
   (store (mark-table i 6) 0)
   (setq i41 1)
   (loop nil
    (store (mark-table i 6)
           (plus (mark-table i 6) (mark-table i i41)))
    (setq i41 (add1 i41))

    (cond ((greaterp i41 5) (exit-loop))))
   (setq i (add1 i))
   (cond ((greaterp i 1000) (exit-loop))))
  (setq i51 1)
  (loop nil
   (setq k51 (plus i51 1))
   (loop nil
    (cond ((greaterp (mark-table k51 6) (mark-table i11 6))
    (setq i101 0)
    (loop nil
     (store (wk51 1 i101) (mark-table i51 i101))
     (setq i101 (add1 i101))
     (cond ((greaterp i101 6) (exit-loop))))
    (setq i111 0)
    (loop nil
     (store (mark-table i51 i111) (mark-table k51 i111))
     (setq i111 (add1 i111))
     (cond ((greaterp i111 6) (exit-loop))))
    (setq i121 0)
    (loop nil
     (store (mark-table k51 i121) (wk51 1 i121))
     (setq i121 (add1 i121))
     (cond ((greaterp i121 6) (exit-loop))))))
    (setq k51 (add1 k51))
    (cond ((greaterp k51 1000) (exit-loop))))
   (setq i51 (add1 i11))
   (cond ((greaterp i51 1000) (exit-loop)))
  (setq i61 1)
  (loop nil
   (store (mark-table i61 0) i61)

   (cond ((greaterp i51 1000) (exit-loop))))
 ....................................
```

Fig. 6. The produced LISP program.

paraphraser transforms refined case-like expressions into documents written in pseudo-natural expressions.

The pseudo-natural expressions consist of pseudo-natural language expressions, arithmetic expressions of the infixed form, and data-like records. The grammar of pseudo-natural language expressions is specified in Fig. 7. Users write pseudo-natural language expressions according to the syntax shown in the figure, referring to the list of pseudo-natural language expressions which represent the headings of library modules. This section describes the outline of the transformation performed in the pseudo-natural language specification analyzer and the paraphraser. The transformation is based on the case grammar.

A. Transformation of Pseudo-Natural Language Expressions into Case-Like Expressions

The transformation is performed via an intermediate expression. In the first step, the pseudo-natural language expressions are transformed into the intermediate expressions. In the second step, the produced intermediate expressions are transformed to the case-like expressions.

1) The First-Step Transformation: In the first step transformation, pseudo-natural specifications written in a pseudo-natural language are parsed by using syntactic rules and the

```
(1)<procedure statement>
   ::= <procedure sentence>|<conditional branch sentence>
       |<repeat sentence>|<assignment statement>

(2)<procedure sentence>
   ::= <procedure predicate verb><term>
              [(to|from|with)<term>]*
              [in<format>][by<procedure name>]
         |<procedure sentence>, and
             store the result in <term>

(3)<conditional branch sentence>
   ::= if<property-relational sentence>
              <procedure statement>*
           [else <procedure statement>*]

(4)<repeat sentence>
   ::= while <property-relational sentence>
                  <procedure statement>*
       |for <variable> from <variable|constant>
        to <variable constant>[by <variable|constant>]
           <procedure statement>*

(5)<input-output statement>
   ::= when <GIVEN sentence|property-relational sentence>*,
       <OBTAINED sentence|property-relational sentence>*

(6)<GIVEN sentence>
   ::= <term> is given [at <variable>]

(7)<OBTAINED sentence>
   ::= <term> is obtained [at <variable>]

(8)<property-relational sentence>
   ::= <term><property-relational predicate>
              [<preposition><term>]

(9)<term>
   ::= <constant>|<variable>|<function>
      |<variable><property-relational clause>
```

Fig. 7. The rewriting rules.

vocabulary. Then the intermediate expressions are generated. The intermediate expression which corresponds to a sentence has the form of:

$$(C_1 : t_1, C_2 : t_2, \ldots, C_n : t_n) \tag{18a}$$

where C_i and $t_i (i = 1, 2, \ldots, n)$ stand for a case label and its

term, respectively. A case label indicates the semantic role of a term involved in the expression.

The intermediate expression for a noun phrase modified with terms $t_1, t_2, \ldots, t_n$ is constructed in the form of:

$$t_0(\text{OBJECT:}*, C_1 : t_1, C_2 : t_2, \ldots, C_n : t_n). \quad (18b)$$

In this expression t_0 denotes the main noun word in the noun phrase. An asterisk denotes that t_0 is also the term which has the role of OBJECT.

The term $t_i (i = 1, \ldots, n)$ in (18a) or (18b) represents a word; it also represents the form of (18a) or (18b) recursively.

The transformation from a pseudo-natural language expression to the intermediate expression is performed using a rewriting rule of the form shown as follows:

$$\begin{aligned}&\langle A_0(C_{01} : t_{01}, \ldots, C_{0n0} : t_{0n0})\rangle \\ &::= \langle A_1(C_{11} : t_{11}, \ldots, C_{1n1} : t_{1n1})\rangle \ldots \\ &\langle A_m(C_{m1} : t_{m1}, \ldots, C_{mnm} : t_{mnm})\rangle. \end{aligned} \quad (19a)$$

The symbol $A_i (i = 0, 1, 2, \ldots, m)$ stands for the syntactic name of the subsequent intermediate expression $(C_{i1} : t_{i1}, \ldots, C_{ini} : t_{ini})$. Expression (19a) defines the rule to produce intermediate expressions $(C_{11} : t_{11}, \ldots, C_{1n1} : t_{1n1}), \ldots$, and $(C_{m1} : t_{m1}, \ldots, C_{mnm} : t_{mnm})$ from intermediate expression $(C_{01} : t_{01}, \ldots, C_{0n0} : t_{0n0})$, or to conversely reduce the former intermediate expressions to the latter one. Typical examples of rewriting rules for a kind of pseudo-English are shown in the following:

<procedure sentence(PREDICATE:*t*0,OBJECT:*t*1,
[SOURCE:*t*2],[GOAL:*t*3])>
::= <procedure predicate verb(*t*0)>
<term(*t*1)> [from <term(*t*2)>]
[to <term(*t*3)>] (19b)

<procedure sentence(PREDICATE:*t*0,*C*1:*t*1,GOAL:*t*2)>
::=<procedure sentence(PREDICATE:*t*0,*C*1:*t*1)>,
and store the result in <term(*t*2)> (19c)

<term(*t*(PREDICATE:*t*0,OBJECT:*,*C*1:*t*1))>
::= <variable(*t*)>
<property-relational clause(PREDICATE:*t*0,
OBJECT:which, *C*1:*t*1)>
(19d)

In these rules, the parts enclosed by square brackets can be omitted. Expressions (19b) and (19c) stand for the rewriting rules for a simple sentence and for a compound sentence, respectively. A simple sentence is an expression which does not include any conjunctive word. A compound sentence is an expression which includes at least one conjunctive word. Expression (19d) stands for the rewriting rule for a noun phrase modified by a relative clause.

The major rewriting rules used to transform pseudo-English expressions into intermediate expressions are described in Fig. 7.

Syntactic names, semantic names, and case-frames for each word are stored in the word dictionary. In the parsing stage, the parser [22] is called and the information for each word included in pseudo-English expressions is first matched with the words in the dictionary. Then the parser searches for the rewriting rule in which the right-hand side matches the current parts of the input expression. The input expression is either the pseudo-English expression or the intermediate expression which is partially constructed hitherto for a phrase of the pseudo-English expression. The parser identifies the case labels of the dependents by referring to the semantic names of the dependents and the case-frame of the main predicate. Then the parser applies the rewriting rule and produces the intermediate expression corresponding to the left-hand side of the applied rewriting rule. The parser repeats the above process in a parallel bottom–up manner and thereby constructs the intermediate expression.

Example 7: Suppose that a pseudo-English expression is given as follows: "For any I, retreive from TABLE(1..N,1..M) TABLE(I,1..M) which satisfies the condition TABLE(I,1)=R(1), and store the result in ANS(1..N,1..M)."

The intermediate expression is produced through the following steps:

(a) The noun phrase "TABLE(I,1..M) which satisfies the condition TABLE(I,1)=R(1)," is transformed into the intermediate expression:

<variable(TABLE(I,1..M)>
<property-relational clause
(PREDICATE:satisfy,OBJECT:which,
CONDITION:TABLE(I,1)=R(1))>.

The above intermediate expression is transformed into the next intermediate expression:

<term
TABLE(I,1..M)(PREDICATE:satisfy,OBJECT:*,
CONDITION:TABLE(I,1)=R(1))>

by applying the rewriting rule (19d).

(b) The simple sentence "retrieve from TABLE(1 . . N,1 . . M) TABLE (I,1..M) . . . " is transformed into the intermediate expression:

<procedure predicate verb(retrieve)>
<term(TABLE(I,1..M)(.....))>
from <term(TABLE(1..N,1..M))>.

The above intermediate expression is transformed into the next intermediate expression:

<procedure sentence
(PREDICATE:retrieve,SOURCE:TABLE
(1..N,1..M),
OBJECT:TABLE(I,1..M),
.....)>

by applying the rewriting rule (19b).

(c) The compound sentence "retrieve . . . and store the result in ANS(1..N,1..M)" is transformed into the next intermediate expression:

```
<procedure sentence
  (PREDICATE:retrieve,
    SOURCE:TABLE(1..N,1..M),
    OBJECT:TABLE(I,1..M)(PREDICATE:satisfy,
      OBJECT:*,
        CONDITION:TABLE(I,1)=R(1)),
    GOAL:ANS(1..N,1..M),MODE:
      FORANY-I)>
```

by applying the rewriting rule (19c).

2) The Second-Step Transformation: The intermediate expressions produced in the first-step transformation are transformed into case-like expressions in the second step. Looking at the syntactic name of each intermediate expression, the system identifies each intermediate expression to see if it is a procedure expression, a control expression, or an input–output expression. A transformation is then performed by applying the transformation rule.

a) Procedure expressions: The intermediate expression of a procedure expression sometimes consists of only a single predicate verb which has several meanings. Such a predicate verb should be transformed into some appropriate unique heading-symbol defined in advance. The heading-symbol consists of a predicate verb and its objective word, or it consists of a predicate verb and adverbial word for it.

For example, the predicate verb "find" includes multiple meanings. Hence the intermediate expression:

```
(PREDICATE:FIND, OBJECT:
  DISAGREEMENTSET(OBJECT:e1,
                  PARTICIPANT:e2))        (20a)
```

is transformed into the case-like expression shown in (20b) by composing the predicate verb "FIND" and the noun word "DISAGREEMENTSET," which is in the OBJECT case of the expression:

```
FIND-DISAGREEMENTSET(OBJECT:e1,
                     PARTICIPANT:e2).     (20b)
```

This expression must be defined in advance and is stored in the module library of the refinement subsystem. To make this transformation, we provide the transformation rule shown as follows:

```
(PREDICATE:FIND,OBJECT:function(OBJECT:x))
  → FIND-function(OBJECT:x).              (20c)
```

The above transformation is applied only in the case where the word "function" is the name of the function included in the word dictionary.

b) Control expressions: The intermediate expression for a conditional control expression (IF-THEN-ELSE type) is described as follows:

```
(CONDITION:(PREDICATE:t,OBJECT:x),
THEN:(PREDICATE:q1, OBJ:x),
ELSE:(PREDICATE:q2, OBJ:x)).              (21)
```

The intermediate expression is transformed to the form of (4).

Similarly, the intermediate expressions of the iterative control expressions are transformed into the form of (5) and (6).

B. Transformation of Case-Like Expressions into Pseudo-Natural-Language Expressions

The transformation of case-like expressions into pseudo-natural-language expressions is performed so that the produced pseudo-natural-language specification serves as the user's documents. The composite word in the case-like procedure expression is first dissolved, and the case-like expression such as (20b) is transformed to the intermediate expression (20a), and then the pseudo-natural-language expression is generated by the transformation.

Example 8: The case-like expressions of Example 5 are transformed into the following pseudo-English expressions by using the rewriting rules shown in Fig. 7. For example, the repeat sentence (1) is generated by applying rewriting rules (1), (2), and (4):

```
(1)FOR I FROM 1 TO 1000
     FOR I1 FROM 1 TO 5
       READ MARK-FILE AND
       STORE THE RESULT IN
         MARK-TABLE(I,I1) ;
(2)FOR I FROM 1 TO 1000
     MARK-TABLE(I,6) := 0 ;
     FOR I4 FROM 1 TO 5
       MARK-TABLE(I,6) := (MARK-
         TABLE(I,6) + MARK-TABLE(I,I4)) ;
(3)FOR I5 FROM 1 TO 1000
     FOR K5 FROM (I5 + 1) TO 1000
       IF MARK-TABLE(K5,6) IS GREATER
         THAN MARK-TABLE(I5,6)
       EXCHANGE MARK-TABLE(K5,0 .. 6)
       FOR MARK-TABLE(I5,0 .. 6) ;
(4)FOR I FROM 1 TO 1000
     MARK-TABLE(I,0) := I ;
(5)FOR I18 FROM 1 TO 1000
     FOR I28 FROM 0 TO 6
       WRITE MARK-TABLE(I18,I28)
         ON MARK-FILE;
```

VI. The Experimental System

An experimental MAPS written in LISP was constructed on the time-sharing system of our university computer center (ACOS 850), and also on a workstation (CPU MC68000, clock 10-MHz memory 3.5 megabyte).

Fig. 8 shows the functional flow in the refinement subsystem. This subsystem scans the case-like specifications and notifies the user of errors. If the heading symbol of a user specification is found unifiable with a heading symbol in the library modules, but the argument parts of the unifiable module do not match, MAPS displays this library module to the user as a candidate module, which is then corrected to

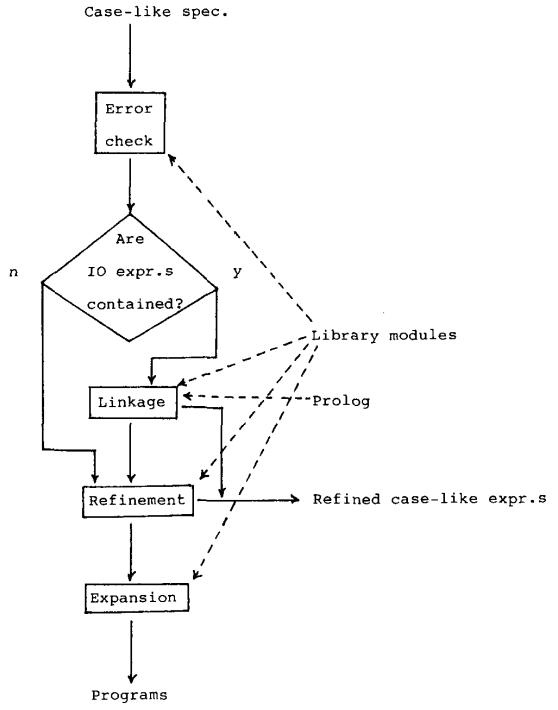


Fig. 8. Functional flow in the refinement subsystem.

the form applicable to the specification by the user. If the heading symbol of the user specification is not unifiable with any heading symbol in the library modules, MAPS returns the message "REUSABLE_MODULES NOT AVAILABLE." If the user specification includes an input–output expression, MAPS tries to replace it with a set of library modules, as was explained in Section IV-C.

The average time required to transform pseudo-English expressions to the case-like expressions is about 0.2-s per word by ACOS 850. The refinement of a single case-like expression requires about 0.6 s.

Example 9 demonstrates how a specification is analyzed and transformed to the case-like expressions, pseudo-natural-language specifications, and the program text.

Example 9: A program to draw a bar-type graph representing function $F(x)$ for horizontal axis x is required. MAPS transforms a pseudo-English input–output specification into a case-like expression, and then constructs procedure expressions by linking the procedure expressions of several library modules. The pseudo-English specification given is:

```
IN: X(0..20) IS GIVEN;
OUT: PRINT OF THE GRAPH
     OF ORDERED_SET(F(X(0))..F(X(20)))
     FOR X(0..20) IN THE GRAPH RANGE
     X_RANGE(100, 400),
     Y_RANGE(100, 400) IN THE FORMAT
     OF STRIPE IS OBTAINED;
```

The pseudo-natural language specification analyzer transforms the previous specification to the case-like specification as follows:

```
IN:GIVEN(OBJECT: X(0..20));
OUT:GIVEN(OBJECT:PRINTGRAPH(OBJECT:
    ORDERED_SET(F(X(0))..F(X(20))),
    PARTICIPANT: X(0..20), FORMAT: STRIPE,
    GRAPH_RANGE: X_RANGE(100,400),
                 Y_RANGE(100,400))).
```

As the result of refinement, the procedure expressions (22) are produced. In this production the library modules COMPUTE_FUNCTION, MAX, MIN, and PRINTGRAPH are linked sequentially:

```
COMPUTE_FUNCTION(OBJECT:
          ORDERED_SET(F(X(0)) ..
          F(X(20))),
             GOAL: Y(0..20));
MAX(OBJECT: Y(0..20), GOAL: MAXY);
MIN(OBJECT: Y(0..20), GOAL: MINY);
:=(MAXX, X(20));
:=(MINX, X(0));
PRINTGRAPH(OBJECT:
        ORDERED_SET(F(X(0))..F(X(20))),
        PARTICIPANT:X(0..20), FORMAT:
STRIPE,
        VALUE_RANGE:X_RANGE(MINX,MAXX),
          Y_RANGE(MINY,MAXY),
        GRAPH_RANGE:X_RANGE(100,400),
          Y_RANGE(100,400));                (22)
```

The paraphraser generates the following pseudo-English documents from the procedure expressions of (22):

```
COMPUTE ORDERED_SET(F(X(0))
  ..F(X(20))) AND
STORE THE RESULT IN Y(0..20);
COMPUTE THE MAXIMUM OF Y(0..20)
AND STORE THE RESULT IN MAXY ;
COMPUTE THE MINIMUM OF Y(0..20)
AND STORE THE RESULT IN MINY ;
MAXX := X(20);
MINX := X(0);
PRINT THE GRAPH OF ORDERED_SET(F(X(0))
..F(X(20))) FOR X(0..20) FOR THE VALUE
RANGE X_RANGE(MINX,MAXX),
Y_RANGE(MINY,MAXY) IN THE GRAPH
RANGE X_RANGE(100,400),
Y_RANGE(100,400) IN THE FORMAT OF STRIPE;
```

The case-like expression of (22) includes nonprimitive expressions: COMPUTE_FUNCTION, MAX, MIN, and PRINTGRAPH. These nonprimitive expressions are then refined. The following procedure expressions are produced as a result: s

```
FOR(INDEX: i1, FROM: 0, TO:20,
      OBJECT: :=(Y(i1),F(X(i1))));
:=(MAXY,Y(0));
FOR(INDEX: i2, FROM: +(0,1), TO: 20,
     OBJECT:IF-THEN(CONDITION:
        GREATER_THAN(OBJECT: Y(i2),
          COMPARISON: MAXY),
        OBJECT: :=(MAXY,Y(i2))));
:=(MINY,Y(0));
FOR(INDEX: i3, FROM: +(0,1), TO: 20,
     OBJECT:IF-THEN(CONDITION:
        LESS_THAN(OBJECT: Y(i3),
          COMPARISON: MINY),
        OBJECT: :=(MINY,Y(i3))));
:=(MAXX,X(20));
:=(MINX,X(0));
DRAW_XY_AXIS(FORMAT:STRIPE,
     VALUE_RANGE:X_RANGE(MINX,MAXX),
        Y_RANGE(MINY,MAXY),
     GRAPH_RANGE:X_RANGE(100,400),
        Y_RANGE(100,400));
FOR(INDEX: i6, FROM: 0, TO: 20,
     OBJECT: PRINTGRAPHPT(OBJECT:
          Y(i6),X(i6),
        FORMAT: STRIPE,
        VALUE_RANGE:X_RANGE(MINX,MAXX),
          Y_RANGE(MINY,MAXY),
        GRAPH_RANGE:X_RANGE(100,400),
          Y_RANGE(100,400)));                (23)
```

The paraphraser generates the following pseudo-English documents from procedure expressions of (23):

```
FOR i1 FROM 0 TO 20
   Y(i1) := F(X(i1));
MAXY := Y(0);
FOR i2 FROM (0 + 1) TO 20
   IF Y(i2) IS GREATER THAN MAXY
     MAXY := Y(i2);
MINY := Y(0);
FOR i3 FROM (0 + 1) TO 20
   IF Y(i3) IS LESS THAN MINY
     MINY : = Y(i3);
MAXX := X(20);
MINX := X(0);
DRAW THE XY AXIS FOR THE VALUE
RANGE X_RANGE(MINX,MAXX),
Y_RANGE(MINY,MAXY) IN THE GRAPH
RANGE X_RANGE(100,400),
Y_RANGE(100,400) IN THE FORMAT OF STRIPE ;
FOR i6 FROM 0 TO 20
   PRINT THE GRAPH OF (Y( i6),X(i6))
   FOR THE VALUE RANGE
   X_RANGE(MINX,MAXX), Y_RANGE
     (MINY,MAXY)
   IN THE GRAPH RANGE
   X_RANGE(100,400), Y_RANGE(100,400)
   IN THE FORMAT OF STRIPE;
```

```
(array x t 21)
(array y t 21)
(defun printgraph (x)
 (prog (maxy miny maxx minx i41 i111 i201)
  (setq i11 0)
  (loop nil
   (store (y i11) (f (x i11)))
   (setq i11 (add1 i11))
   (cond ((greaterp i11 20) (exit-loop))))
  (setq maxy (y 0))
  (setq i41 (plus 0 1))
  (loop nil
   (cond ((greaterp (y i41) maxy) (setq maxy (y i41))))
   (setq i41 (add1 i41))
   (cond ((greaterp i41 20) (exit-loop))))
  (setq miny (y 0))
  (setq i111 (plus 0 1))
  (loop nil
   (cond ((lessp (y i111) miny) (setq miny (y i111))))
   (setq i111 (add1 i111))
   (cond ((greaterp i111 20) (exit-loop))))
  (setq maxx (x 20))
  (setq minx (x 0))
  (draw_xy_axis 'stripe minx maxx miny maxy 100 400 100 400)
  (setq i201 0)
  (loop nil
   (printgraphpt (y i201) (x i201) 'stripe
                 minx maxx miny maxy 100 400 100 400)
   (setq i201 (add1 i201))
   (cond ((greaterp i201 20) (exit-loop))))))

(defun printgraphpt (yl xl format x_min x_max y_min y_max)
 (prog (xy_pos1 x_pos1 y_pos1 xy_pos2 x_pos2 y_pos2)

  (setq xy_pos1
   (find_xy_gposition yl xl format
                      x_min x_max y_min y_max
                      gx_min gx_max gy_min gy_max))
.....
.....
```

Fig. 9. The produced LISP program.

The programs produced from (23) are shown in Figs. 9 and 10.

VII. Discussion

There are several program generation systems which use reusable software modules: the Draco system [9], the SAFE/TI system [8], and the MODEL system [18]. The comparisons between MAPS and these systems from the aspects of user specification, reusable module specification, program synthesis, and program documentation are shown in Table IV.

A. The Aspect of User Specification

In the Draco system the user specification is described as using a domain language, the grammar of which the user specifies. The described specification is parsed and transformed into the Draco internal form (a tree whose nodes are the names of the components of the domain). In the SAFE/TI system, a kind of English specification is processed and is converted to a specification in GIST language. In MAPS, user specifications are specified using pseudo-English, which has a fixed grammar. The pseudo-English specification is parsed based on the case grammar and transformed into an input–output expression or a procedure expression. Because the grammar of pseudo-English is similar to that of natural English, ease of writing and reading specification is improved.

B. The Aspect of Reusable Module Specification

In the Draco system, reusable software components are specified for each problem domain using the algorithmic

TABLE IV
COMPARISON BETWEEN MAPS AND OTHER PROGRAM-GENERATION SYSTEMS

	Draco	MODEL	SAFE/TI	MAPS
user specification	specified by domain language; user can specify the grammar	specified by MODEL nonprocedual language	specified by (pre-parsed) English	specified by pseudo-English or Japanese
reusable module specification	specified by algorithmic language SIMAL		specified by forms of program transformation rules	specified by input-output expressions and procedure expressions
synthesis	Draco internal form generated from domain language parser is refined using internal form of program components	procedual programming language is synthesized using dataflow graph generated from MODEL language parser	operational specification generated from (pre-parsed) English specification is refined by program transformations	case-like expression transformed from pseudo-English or Japanese is refined by replacing and linking library modules

```
printgraph(x) double x(21);
 (double maxy; double miny; double maxx; double minx;
  int i21; int i41; int i111; int i201;
  double y(21);
  for(i11 = 0; i11 <= 20; ++ i11)
   (
    y(i11) = f(x(i11));
   );
  maxy = y(0);
  for(i41 = (0 + 1); i41 <= 20; ++ i41)
   (
    if(y(i41) > maxy)
     (
      maxy = y(i41);
     );
   );
  miny = y(0);
  for(i111 = (0 + 1); i111 <= 20; ++ i111)
   (
    if(y(i111) < miny)
     (
      miny = y(i111);
     );
   );
  maxx = x(20);
  minx = x(0);
  draw_xy_axis('stripe',minx,maxx,miny,maxy,100,400,100,400);

  for(i201 = 0; i201 <= 20; ++ i201)
   (
    printgraphpt(y(i201),x(i201),
                 minx,maxx,miny,maxy,100,400,100,400);
   );
 )

printgraphpt(yl,xl,format,
             x_min,x_max,y_min,y_max,
             gx_min,gx_max,gy_min,gy_max)
 double yl; double xl; *char format;
 double x_min; double x_max; double y_min; double y_max;
 int gx_min; int gx_max; int gy_min; int gy_max;
 (int x_pos11; int y_pos11; int x_pos21; int y_pos21;
  find_xy_gposition(yl,xl,format,
                    x_min,x_max,y_min,y_max,
                    gx_min,gx_max,gy_min,gy_max,x_pos11,y_pos11);
 .....
 .....
```

Fig. 10. The produced C program.

```
[1] Fundamental modules

(a) Input-output processing
     READ-IN-ARRAY, WRITE-IN-ARRAY, READ-RECORD, WRITE-RECORD,
     OPEN-FILE, CLOSE-FILE, PRINTGRAPH,.....
                                          about 15 modules
(b) Arithmetic processing
     SUM-UP, AVERAGE, COMPUTE-FUNCTION, FING-MAXIMUM,
     FIND-MINIMUM, FIND-DISPERSION, FIND-FACTORIAL,.....
                                          about 20 modules
(c) String processing and table(array) manipulation
     COPY-STRING, CONCATENATION, COUNT-CHARACTERS, COUNT-WORDS,
     EXCHANGE-VECTOR, SORT, RETRIEVE, .....
                                          about 15 modules

[2] Application modules

(a) File processing
     SORT-FILE, MERGE-FILE, UPDATE-FILE, GROUP-RECORD,
     RETRIEVE-RECORD, .....
                                          about 15 modules
(b) Logical inference processing
     UNIFICATION, INSTANTIATION, RENAMING, FIND-DISAGREEMENTSET,
     PATTERN-TRANSFER, .....
                                          about 20 modules
(c) Language processing
     LEXICAL-ANALYSIS, PARSING, SEARCH-HANDLE, RULE-APPLICATION,
     STRUCTURE-TRANSFER, SYNTACTIC-GENERATION,
     LEXICAL-GENERATION, .....
                                          about 35 modules
```

Fig. 11. The library modules.

language SIMAL. Each problem-domain component is associated with a set of implementation modules. The description written using SIMAL is transformed into Draco internal form. In MAPS, reusable modules are specified by procedure and input–output expressions.

C. The Aspect of Program Synthesis

In the Draco system, Draco internal forms transformed from domain specifications (user specifications) are optimized by program transformation, and then are refined using the internal form of reusable software components. The refinement is performed by replacing a tree node with one of the possible implementation modules. In the SAFE/TI system, the GIST specification generated from the user specification is transformed into a program text by correctness-preserving transformations. In MAPS, the case-like expression of a pseudo-English specification is refined by linking specifications of library modules (in the form of the procedure expression or input–output expression) using the first- and second-order unification.

D. The Aspect of Program Documentation

In the SAFE/TI system, the GIST paraphraser produces English documents from GIST specifications. In the Draco system, the refined Draco internal form can be pretty-printed. In MAPS, refined case-like expressions can be paraphrased into pseudo-English documents.

VIII. Conclusion

Through this research the authors tried to realize an automatic program generation system. The way how to specify reusable modules, how to refine, and how to produce programs using reusable modules are studied. A system to perform transformation between case-like specifications and readable pseudo-natural-language specifications was developed.

Presently, the library of MAPS includes about 50 fundamental modules, and about 70 application modules for table manipulation, string processing, file processing, and language processing. The procedure names of major modules in the library are listed in Fig. 11. MAPS is mainly used for trials to transform specifications of such applications as scientific computation and utility subprograms. As a result of applying MAPS, the refinement subsystem of MAPS itself has been generated using MAPS.

References

[1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[2] D. Teichroew and E. A. Hershey, "PSL/PSA-A computer-aided technique for structured documentation and analysis of information processing systems," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 41–48, Jan. 1977.

[3] N. Wirth, "Program development by stepwise refinement," *Commun. ACM*, vol. 14, no. 4, pp. 221–227, Apr. 1971.

[4] J. A. Robinson, "A machined-oriented logic based on the resolution principle," *J. Assoc. Comput. Mach.*, vol. 12, no. 1, pp. 23–41, Jan. 1965.

[5] T. Pietrzykowski, "A complete mechanization of second-order-type theory," *J. Assoc. Comput. Mach.*, vol. 20, no. 2, pp. 333–365, Apr. 1973.

[6] J. L. Darlington, "Automatic synthesis of SNOBOL programs," in *Computer-Oriented Learning Process*, J. C. Simon, Ed. Nordhoff-Leyden, 1976, pp. 443–453.

[7] D. R. Barstow, *Knowledge-Based Program Construction*. Amsterdam: North-Holland, 1979.

[8] H. Partsch and R. Steinbruggen, "Program transformation systems," *Comput. Surveys*, vol. 15, no. 3, pp. 199–236, Sept. 1983.

[9] J. M. Neighbors, "The Draco approach to constructing software from reusable components," *IEEE Trans. Software Eng.*, vol. SE-10, pp.56 4–573, Sept. 1984.

[10] E. Horowitz and J. B. Munson, "An expansive view of reusable software," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 477–487, Sept. 1984.

[11] Y. Matsumoto, "Some experience in promoting reusable software: presentation in higher abstract levels," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 502–512, Sept. 1984.

[12] D. L. Parnas, P. C. Clements, and D. M. Weiss, "The modular structure of complex systems," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 259–266, Mar. 1985.

[13] D. R. Barstow, "Domain-specific automatic programming," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 1321–1336, Nov. 1985.

[14] I. J. Hayes, "Specification-directed module testing," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 124–133, Jan. 1986.

[15] Z. L. Lichtman, "Generation and consistency checking of design and program structures," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 172–181, Jan. 1986.

[16] R. T. Mittermeir and W. Rossak, "Software bases and software archives, " in *Exploring Technology Today and Tomorrow*, pp. 21–28, 1987.

[17] G. E. Kaiser, "Composing software systems from reusable building blocks," in *Proc. 20th Hawaii Int. Conf. Syst. Sci.*, 1987, vol. 2, pp. 536–545.

[18] N. S. Prywes and A. Pnueli, "Compilation of nonprocedural specifications into computer programs," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 267–279, May 1983.

[19] Z. Manna and R. Waldinger, *Studies in Automatic Programming Logic*. New York: North-Holland, 1977.

[20] P. R. Cohen and E. A. Feigenbaum, Eds., *The Handbook of Artificial Intelligence*, vol. 3. Los Altos, CA: Kaufmann, 1982, chap. XV, pp. 513–562.

[21] B. Vauquois, "Aspect of mechanical translation in 1979," GETA, Univ. Grenoble, 1979.

[22] F. Nishida, S. Takamatsu, and H. Kuroki, "English–Japanese translation through case structure conversion," in *Proc. 8th Int. Conf. Comput. Linguist.*, 1980, pp. 447–454.

[23] F. Nishida and S. Takamatsu, "Japanese–English translation through internal expressions," in *Proc. 9th Int. Conf. Comput. Linguist.*, 1982, pp. 271–276.

[24] F. Nishida and Y. Fujita, "Semi-automatic program refinement from specification using library modules," *Trans. Inf. Soc. Japan*, vol. 25, no. 5, pp. 785–793, Sept. 1984 (in Japanese).

[25] F. Nishida, Y. Fujita, and S. Takamatsu, "Semi-automatic program generation from Japanese specifications by the aids of library modules" in *Proc. Symp. on Prototyping and Requirements Specification, Inf. Proc. Soc. Japan*, Apr. 1986, pp. 111–120 (in Japanese).

[26] F. Nishida, Y. Fujita, and S. Takamatsu, "Construction of a modular and portable translation system," in *Proc. 11th Int. Conf. Comput. Linguist.*, Aug. 1986, pp. 649–651.

[27] E. Charniak and Y. Wilks, Eds., "Linguistics," in *Computational Semantics* (Fundamental Studies in Comput. Sci., vol. 4). Amsterdam: North-Holland, 1976.

[28] F. Nishida, Y. Fujita, and S. Takamatsu, "Refinement and error detection of program specifications by a linking technique of library modules," *Trans. Inf. Proc. Soc. Japan*, vol. 28, no. 5, pp. 489–498, May 1987 (in Japanese).

[29] F. Nishida, S. Takamatsu, and T. Tani, "Transformation between Japanese expressions and formal expressions in program specifications," *Trans. Inf. Proc. Soc. Japan*, vol. 29, no. 4, pp. 368–377, Apr. 1988 (in Japanese).

[30] C. J. Fillmore, "The case for case," in *Universals in Linguistic Theory*, E. Bach and R. Harms, Eds. New York: Holt, Rinehart and Winston, 1968.

Fujio Nishida received the B.E. and Ph.D. degrees in electrical engineering from Kyoto University, Kyoto, Japan, in 1950 and 1959, respectively.

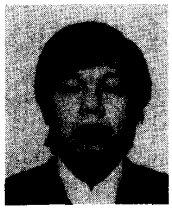
He was a Professor in the Department of Electrical Engineering of the University of Osaka Prefecture through 1990, and is now a Professor in the Department of Industrial Management, Fukui Institute of Technology. His current work is concerned with natural language processing, refinement of program specifications, and knowledge engineering.

Shinobu Takamatsu received the B.E., M.E., and Ph.D. degrees in electrical engineering from the University of Osaka Prefecture, Osaka, Japan, in 1971, 1973, and 1978, respectively.

Presently, he is a Lecturer in the Department of Electrical Engineering of the University of Osaka Prefecture, where he is engaged in research concerning natural language processing and knowledge information processing and their applications to software engineering.

Yoneharu Fujita received the B.E., M.E., and Ph.D. degrees in electrical engineering from Osaka University, Osaka, Japan, in 1968, 1970, and 1970, respectively.

From 1973 through 1986 he was with the University of Osaka Prefecture, and is presently with Oita University, where he is a Professor in the Department of Computer Science and Intelligent Systems. His research interests include cognitive science concerning mechanisms of emotion, intelligent human–machine interfaces, and applications of artificial intelligence to software engineering.



Tadaaki Tani received the B.E. degree in literature from Osaka City University, Osaka, Japan, in 1982.

He is presently a Technical Official in the Department of Electrical Engineering of the University of Osaka Prefecture, where he is engaged in research concerning natural language processing.