Semi-Automatic Program Generation with the aids of Reusable Modules

Semi-Automatic Program Generation with the aids of Reusable Modules

Fujio NISHIDA*, Shinobu TAKAMATSU* and Yoneharu FUJITA**

(Received June 16, 1988)

This paper describes a method of semi-automatic specification refinement and program construction using library modules. Users write their specifications, modifies and rearranges them so that the specifications can be refined with the aids of the library modules. Given specifications, a refinement system called MAPS searches for library modules applicable to the given specifications, refines them by linking several modules or by replacing them by a more detailed description in the operation part of the module and expands the refined specifications into a programming language designated in the user's option.

1. Introduction

In recent years high productivity and flexibility of computerization have been recognized in every field and the need of program generation has been rapidly increasing.

Correspondingly, various production methods of software have been proposed by many researchers[1-12]. The research of the requirement techniques[1-3] proposed a powerful means of costructing a big system by division and abstraction of the system as well as the various specification languages. The study of reusability of program modules[9-11] suggests not only an efficient method of reliable system development but also feasibility of automation in software engineering. The reuse of software components allows the software developer to write fewer total symbols in less time with fewer mistakes in the development of a system.

Furthermore, recent development of problem solving techniques seems to drive the study of the automatic programming into practical application[6-8,12,13].

Along these trends, the authors also have been studying a semi-automatic program construction by using library modules and established an experimental system called the library-Module Aided Program construction System abbreviated to MAPS. Compared with the associated studies[7-11], the focus of this paper is placed on 1) the introduction of a powerful unification technique into application of the knowledge of the library modules and 2) refinement by linking modules with the aids of the input-output assertion.

The module library consists of fundamental program modules constructed hierarchically in some specific fields such as symbol manipulations and language processing as well as in the general fields.

In MAPS each user describes program specifications by referring to a heading list of available library modules. The expressions of user's specifications are restricted to the

* Department of Electrical Engineering, College of Engineering
** Faculty of Engineering, Oita University

same form as those of library modules. They take a limited form of a natural language like Japanese or the corresponding formal expression except for arithmetic expressions and type expressions. The system searches for library modules applicable to the given specification using the heading expressions for every block of specifications.

If MAPS succeeds in search, it begins to refine the specifications. MAPS replaces the unifiable part of the specifications by the instantiated operation part of the module which provides a more detailed procedure, or it calls the instantiated operation part as an auxiliary program. If a specification block is described in an input-output assertion, it links several library modules to meet the assertion. If MAPS fails to find out available modules, the user provides the system with a more refined specification or constructs a new module to meet the specification block.

By repeating these operations, specifications can be refined to be completely resolved into primitive operations. Subsequently the refined specifications are transformed into some programming languages such as Lisp and C language according to the user's designation.

2. Formal Expressions of Specifications and Library Modules

The formal expressions are classified into three kinds of expressions, namely, procedure expressions, input-output predicate expressions and control expressions. They can be represented in a function form with a role indicator prefixed to each parameter as follows:

$$\textit{heading-symbol}\ (C_1 : t_1, \ldots, C_n : t_n) \qquad (1)$$

The heading symbol takes a structure name such as a procedure name and a predicate symbol in this paper. The argument part consists of a sequence of several pairs of a case label C_i $(i = 1, 2, \ldots, n)$ and a term t_i. Some of case labels are shown as follows:

OBJECT: the object to be processed
SOURCE: a set or a collection of data including the processed object
PARTICIPANT: an auxiliary or complementary object
CONDITION: a condition of the processing
GOAL: a memory location storing the processed results
LOCATION: a location of the object
COMPARISON: a thing compared with the object
KEY: a key item used for data processing
FORMAT: a representation format of the object
MODE: a processing mode designated by an algorithm name

In the following, a string consisting of small letters, in many cases, denotes a variable for which a constant or another variable can be substituted while a string consisting of large letters denotes a constant. The input or the output variable in specifications is processed as a constant in unilateral unification with library modules (or substitution for them) and is written in a string consisting of large letters.

2.1 Procedure expressions

The procedure expression is useful for designating various macro operations and takes a form of Expr.(1). The heading symbol represents a procedure name and forms the principal part of a key available for retrieving modules in the library. It consists of a verb word or a noun word such as 'retrieve' and 'sum'. In the argument, the GOAL case denotes the location or the variable in which the result is stored. It is assumed that an expression without the GOAL case stands for a function or represents the result itself.

2.2 Input-output predicate expressions

The input-output predicate expression or assertion is a kind of problem-oriented descriptions or requirement-defining descriptions and useful for finding an applicable module sequence to a given specification as mentioned in section 3.2. It is also useful for checking the validity of specifications by examining whether or not the input condition is satisfied at every stage.

An input-output assertion consists of a pair of an input predicate and an output predicate. Both expressions consist of a conjunction of clauses and each clause is a disjunction of literals. Each literal has a form of Expr. (1) or the negation form where the heading symbol stands for a predicate.

Let us consider the following input-output expression:

$$\begin{aligned}&IN\colon GIVEN\,(OBJECT\colon x,\ LOCATION\colon loc),\ P\,(OBJECT\colon x),\\ &OUT\colon \exists\, z\ Q\,(OBJECT\colon z,\ PARTICIPANT\colon x)\end{aligned} \tag{2}$$

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 a relation $Q(z, x)$ to the input x.

The output part can be also written in a form

$$GIVEN\ (OBJECT\colon q\,(x)) \tag{2a}$$

using the Skohlem function for $\forall\ x\ \exists\ z\ Q\,(z, x)$. The input-output expressions (2) and (2a) are useful for finding a sequence of modules which satisfies a given specification.

2.3. Control expressions

The control expressions are classified into those of branching and iteration. They have a procedure expression and an input-output expression as described in the following.

(1) Conditional branching

The fundamental branch type is the IF-THEN type. The procedure expression and the input-output expression are described respectively as follows:

$$PROC\colon IF\text{-}THEN\ (CONDITION\colon t(x),\ OBJECT\colon q\,(x),\ GOAL\colon z) \tag{3}$$

$$\begin{aligned}&IN\colon GIVEN\,(x)\\ &OUT\colon \neg\, t(x) \vee GIVEN\,(OBJECT\colon q\,(x),\ LOCATION\colon z)\end{aligned} \tag{3a}$$

Similarly, the binary branch expressions are given as follows:

$$PROC: IF\text{-}THEN\text{-}ELSE\ (CONDITION1: t1(x), OBJECT1: q1(x), \quad CONDITION2: t2(x), OBJECT2: q2(x), GOAL: z) \tag{4}$$

$$IN: GIVEN(x)$$
$$OUT: \neg\, t1(x) \vee GIVEN\ (OBJECT: q1(x), LOCATION: z), \quad \neg\, t2(x) \vee GIVEN\ (OBJECT: q2(x), LOCATION: z) \tag{4a}$$

where $t1(x)$ and $t2(x)$ are complementary to each other.

If the procedure $q1(x)$ has no effect on $t2(x)$, it can be shown that the procedure of IF-THEN-ELSE is equivalent to the following procedures:

$IF\text{-}THEN\ (CONDITION: t1(x), OBJECT: q1(x), GOAL: z);$

$IF\text{-}THEN\ (CONDITION: t2(x), OBJECT: q2(x), GOAL: z);$

(2) Iteration

Iteration is classified into the parallel type and the serial type.

(2.1) Parallel type

Parallel type iteration repeatedly applies the same operation to each element of an object as seen in retrieving of elements with a specified attribute value. In this type, the result of processing in the i-th stage of iteration does not affect that in the $j(\neq i)$-th one. The procedure expression and the input-output expression are given respectively as follows:

$$PROC: FOR\ (INDEX: i, FROM: m, TO: n, \quad OBJECT: proc\ (OBJECT: x1(i), \quad PARTICIPANT: x2(i), GOAL: z(i))) \tag{5}$$

$$IN: GIVEN\ (OBJECT: x1(m..n), x2(m..n))$$
$$OUT: FORALL\ i\ (\neg\, m \leqq i \leqq n \quad \vee GIVEN\ (OBJECT: proc\ (OBJECT: x1(i), PARTICIPANT: x2(i), \quad GOAL: z(i)))) \tag{5a}$$

where the output expression means that the relation in the OBJECT case holds for all the INDEX value i of $m \leqq i \leqq n$.

(2.2) Serial type

The serial type iteration processes data for every stage of iteration based on the results of data processing in the preceding stage.

The procedure expression and the input-output expression are given as follows:

$PROC: FOR\ (INDEX: i, FROM: m, TO: n,$

$\quad OBJECT: proc\ (OBJECT: x(i),$

PARTICIPANT: *y*,
GOAL: *y*), *GOAL*: *z*) (6)

IN: *GIVEN* (*OBJECT*: *x* (*m*..*n*), *y*)
OUT: *GIVEN* (*OBJECT*: *FOR* (*INDEX*: *i*, *FROM*: *m*, *TO*: *n*,
OBJECT: *proc* (*OBJECT*: *x* (*i*),
PARTICIPANT: *y*,
GOAL: *y*), *GOAL*: *z*)) (6a)

where the initial value of *y* requested in the input expression depends on the property of the repeated procedure.

2.4. Library modules

Each module has a heading consisting of both the PROCedure and the INput-OUTput expressions followed by data TYPE part of entities and an OPeration part. The operation part describes the details of a procedure by using the headings of hierarchically lower library modules. Table 1 shows an example of library modules.

Table 1 An example of "SORT" module

```
PROC:SORT(OBJECT:array(m1 .. n1,m2 .. n2),
          KEY_COLUMN:j,MODE:DESCENDING_ORDER),
IN:GIVEN(OBJECT:array,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),
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)
                                                            ))))
```

2.5. Specifications

Users of the system MAPS describe specifications in a formal expression mentioned above or in a natural language-like expressions by referring to program modules contained in the library. The specifications can be written using a mixed form of procedure expressions input-output expressions and programming language-like expressions in a similar way to descriptions in an operation part of a library module.

Example 1

Let us consider a sorting program for a table of examinees' grades. The specifications written in a certain kind of limited English are given as follows:

```
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 corresponding formal specifications are given as follows:

```
(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)
```

3. Refinement of Specifications and Conversion to Programs

The given specifications are refined for every block by MAPS using applicable program modules. If they are given by informal expressions like limited natural language expressions, MAPS first transforms them to the formal expressions[18].

The formal specification contains various heading expressions of library modules. If the heading expression is a procedure expression, MAPS searches for a library module which has a unifiable procedure name with that of the specification block and checks whether or not the argument part is also unifiable. Subsequently, if unifiable, it refines the heading expression by using the instantiated operation part of the unified module. If the heading expression is an input-output expression, MAPS searches for a link of several library modules which satifies the input-output assertion as a whole and then refines the procedures of the link. At each refinement, if an operation part has several selections, MAPS asks user's selection and makes refinement according to user's option. When nonprimitive expressions such as heading expressions of the modules can not be found any more after refinement, MAPS converts the refined specification to a program written in a programming language designated by the user.

3.1. Refinement of procedure expressions

When MAPS finds a unifiable procedure expression of a library module for a given procedure block contained in a specification, it refines the procedure expression by using the instantiated operation part of the unifiable procedure expression.

There are two methods of realizing the refined expression. One of them is the direct replacement of the original specification by the instantiated operation part. The other is modularization of the refined expression and call of the modularized expression as a subroutine from the block of the specification. MAPS asks the user the selection of the two methods or semi-automatically selects one of them according to the set conditions of switches at the beginning of refinement.

(1) Direct replacement method

Though the method is liable to generate a larger and complicated program, the generated program has high running speed. In many cases, the body part in a repetition can be refined by direct replacement by the operation part of the unifiable procedure of a module.

(2) Procedure call method

This method has the advantage of bringing compactness and readability of the constructed procedure. If the procedure has not been registered yet as a program module in a module table, MAPS constructs the procedure module as follows:

(1) MAPS constructs the heading of the program module and the declaration of variables by referring to both the procedure expression of the heading part and the type part of the library module. They are generated in a programming language designated by the user's option. Then MAPS constructs the body part by putting the instantiated operation part of the module and then records the procedure name in a program module table.

(2) If MAPS finds some non-primitive procedure expressions in the body part of the program module, it refines them by applying the above mentioned refining method respectively.

Example 2

The formal specification given in Example 1 can be first refined by using library modules like a vector-sort module shown in Table 1 as follows:

```
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) ) ) );
```

```
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) ) ) ) );
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) ) ) ) ) );
FOR (INDEX: I, FROM: 1, TO: 1000, OBJECT: : = (MARK-TABLE (I, 0), I) );
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) ) )
```

The above expressions are further refined and finally transformed into a program written in C language as shown in appendix 1 by using transformation to programming languages mentioned in section 3.4.

3.2. Refinement of input-output expressions

Linking of modules is performed based on clause forms of the input-output predicate expressions. The two expressions in Expr. (2) are put together to form a clause form as follows:

$$GIVEN\,(q(x)) \vee \neg GIVEN\,(x) \vee \neg P(x) \qquad (7)$$

where $q(x)$ is a Skohlem function for z defined by $\forall x\ \exists\ zQ\,(x, z)$.

The operation which leads to the above input-output relation is written as follows:

$$OP{:}\ z.q{:} = q_0\,(x) \qquad (8)$$

where $q_0\,(x)$ stands for a concrete operation part corresponding to the Skohlem function $q(x)$.

The refinement of specifications by linking modules is based on the following principle. Let us assume that the given specification has the following input-output relation in a clause form:

$$GIVEN\,(Q(A)) \vee \neg\, GIVEN\,(Q_1\,(A)) \vee \ldots \vee \neg GIVEN\,(Q_m\,(A)) \qquad (9)$$

```
The expansion result in C:

main( )
 (int l121; int l111; int k51;
  int i41; int i; int i51; int l111;
  ..............................
  int mark-table(1001) (7);
  int wk51(2) (7)
  ...........................
  for(i = 1; i <= 1000; ++ i)
   (
    mark-table(i) (6) = 0;
    for(i41 = 1; i41 <= 5; ++ i41)
     (
      mark-table(i) (6) =
          mark-table(i) (6) + mark-table(i) (i41));
     );
   );
  for(i51 = 1; i51 <= 1000; ++ i51)
   (
    for(k51 = (i51 + 1); k51 <= 1000; ++ k51)
     (
      if(mark-table(k51) (6) > mark-table(i51) (6))
       (
        for(l101 = 0; l101 <= 6; ++ l101)
         (
          wk51(1) (l101) = mark-table(i51) (l101);
         );

        for(l111 = 0; l111 <= 6; ++ l111)
         (
          mark-table(i51) (l111) = mark-table(k51) (l111);
         );
        for(l121 = 0; l121 <= 6; ++ l121)
         (
          mark-table(k51) (l121) = wk51(1) (l121);
         );
       );
     );
  ................................
)
```

Appendix 1

Assumption that the above specification is not feasible or cannot hold leads to the negation of the above,

$$\neg\, GIVEN\ (Q(A)), GIVEN\,(Q_1\,(A)), \ldots, GIVEN\,(Q_m(A)) \qquad (10)$$

If there is a refutation from Expr. (10) and a clause set of library modules that support Expr.(9), refinement of the specification can be constructed by calling procedures that are made of the instantiated procedure expressions of the library modules that contribute to the refutation or by direct replacement by the instantiated operation parts of the library modules.

(1) Cascade connection

Cascade connection of library modules is a fundamental method of specification refinement obtained by linking several library modules. The refined specification is constructed by serial or parallel linking of the instantiated operation parts of the modules that contribute to the refutation.

$$\begin{tikzcd}
 & Q_0(A) \arrow[dl] \arrow[dr] \arrow[rrddd, bend left] & & \\
Q_1(A) \arrow[d] & & Q_3(A) \arrow[d] & \\
Q_2(A) \arrow[dr] & & Q_4(A) \arrow[dl] & \\
 & Q_5(A) \arrow[dr] & & Q_6(A) \arrow[dl] \\
 & & Q_7(A) &
\end{tikzcd}$$

Fig. 1 A data-flow of a sequential link

In Fig. 1, suppose that a specification requires an output data $Q_7(A)$ for a given input data $(Q_0(A)=)A$, where Q_i $(i = 0, 1, 2, ...)$ denotes a function symbol. For the specification, the input refutation can be obtained in the choice of the negation of GIVEN $(Q_7(A))$ as the top clause by using Prolog. From the refutation a refined specification can be constructed by the aids of library modules used in the refutation.

When a datum A is needed to evaluate a datum B, the ordered pair of A and B is called a parent-child order. By using this definition, the procedure of the cascade connection is described as follows:

(1) Transform into clause forms the input-output expressions of both the specification and library modules associated with the refutation.
(2) Carry out the input refutation. Choose as the top clause of refutation the negation of the output predicate of the specification.
(3) Instantiate the procedure expression or the operation part of the module used for refutation by the unification substitution corresponding to the procedure call or the direct replacement by the instantiated operation part.
(4) Arrange the instantiated operation parts of the modules that contribute to refutation so that the corresponding unified input and output data satisfy a parent-child order.
(5) If some functions are involved in the argument part of a function, replace them by intermediate variables to improve efficiency. Be careful no collision of variable names occurs in the program block by the aids of the list of variables.

Example 3

Suppose that there are two modules both of which have the clause forms of the input-output expressions and the operation parts as follows:

GIVEN (OBJECT: F(x), LOCATION: y)$\vee\neg$GIVEN (OBJECT: x)

OP: y: = F_0(x) ....①

GIVEN (OBJECT: H(x, y2), LOCATION: y1)

$\vee\neg$GIVEN (OBJECT: F(x), LOCATION: y2)

OP: y1: = H_0(x, y2) ....②

The specification is given in English as follows:

'GIVEN U, H(U, F(U)) IS OBTAINED AT Z',

or in a clause form as follows:

GIVEN (OBJECT: H(U, F(U)), LOCATION: Z) $\vee \neg$ GIVEN (OBJECT: U)

The negation of the above is

$\neg$GIVEN (OBJECT: H(U, F(U)), LOCATION: Z) ....③
and GIVEN (OBJECT: U) ....④

```
③   ¬GIVEN(OBJECT:H(U,F(U)),LOCATION:Z)
②        |   GIVEN(OBJECT:H(x,y2),LOCATION:y1)
         |   v¬GIVEN(OBJECT:F(x),LOCATION:y2)
         | /
         |/   x ← U, y1 ← Z, y2 ← F(U)
    ¬GIVEN(OBJECT:F(U),LOCATION:F(U))
①        |   GIVEN(OBJECT:F(x),LOCATION:y)
         |   v¬GIVEN(OBJECT:x)
         | /
         |/   x ← U, y ← F(U)
    ¬GIVEN(OBJECT:U)
④        |
         |_GIVEN(OBJECT:U)
       [ ]
```

Fig. 2 A refutation process

Fig. 2 shows a refutation by Prolog in which clause ③ is taken as the top clause. By rearranging the operation parts of the modules in the parent-child order of data, the refined specification is obtained as follows:

y2: = F(U); Z: = H(U, y2);

(2) Linking modules which include control expressions

It is not so difficult to link some modules by the above mentioned cascade connection so that the given specification of the input-output expression is satisfied. Each module in the cascade connection can be dealt with as a procedure from a global point of view and it does not matter whether some of the modules contain control expressions or not. For example, the maximum finding function can be dealt with a procedure and can be easily linked to another modules by a cascade connection. On the other hand, it is difficult to refine a given specification or to construct a program at a constituent level of control statements flexibly.

This section describes a method of reducing a refining problem of a control specification mentioned in chapter 2 to that in the OBJECT case in the control expression.

(2.1) Linking in a conditional branching

If a conditional branch is specified by a form

IF-THEN (*CONDITION*: $t(x)$, *OBJECT*: $q(x)$, *GOAL*: z) (3)

or by the input-output predicate form, the refinement problem of the above can be reduced to a link problem of the input-output expression within the OBJECT case as follows:

IF-THEN (*CONDITION*: $t(x)$,
OBJECT: (*IN*: *GIVEN* (x), $t(x)$,
OUT: *GIVEN* $(q(x))$), *GOAL*: z) (11)

Expr. (4) can be also reduced to the same type problem as the above by replacing OBJECTi case (i = 1, 2) by

(*IN*: *GIVEN* (x), *ti* (x), *OUT*: *GIVEN* (*qi* (x))) (11a)

In other words, the refinement of a specification of a conditional branching like Expr. (4) can be replaced by refining the OBJECT case of each branch through an input-output expression like Expr. (11a) reconstructed from the specification.

(2.2) Linking in an iteration frame

In a similar manner to the conditional branching, refinement of iterative specifications can be reduced to a link problem within the OBJECT case of an iterative procedure.

The operation part in a parallel iteration module of heading Expr. (5) is given as follows:

OP: *FOR* (*INDEX*: i, *FROM*: m, *TO*: n,
OBJECT: (*IN*: *GIVEN* $(x1(i), x2(i))$,
OUT: *GIVEN* (*proc* (*OBJECT*: $x1(i)$,
PARTICIPANT: $x2(i)$, *GOAL*: $z(i)$))))) (12)

Similarly, the operation part in a serial iteration module of Expr. (6) is given as follows:

OP: *FOR* (*INDEX*: i, *FROM*: m, *TO*: n,
OBJECT: (*IN*: *GIVEN* $(x(i), y)$,
OUT: *GIVEN* (*proc* (*OBJECT*: $x(i)$,
PARTICIPANT: y, *GOAL*: y))), *GOAL*: z) (12a)

The operation part of a WHILE type module is the same as that of the above serial FOR type except for the conditional case of $t(x(i), y)$.

3.3. Fusion of iterative loops

Fundamental library modules are generally constructed so as to have a single output. Accordingly, specification refinement performed by these library modules often brings a concatenation of several loops which have the same iteration number as each other. Hence it is desired to improve the efficiency of the generated program by fusing these loops globally.

Let us suppose that two FOR-type iterative expressions are generated after specification refinement by using library modules as follows:

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

In the above, each index variable of the iterative expression has the same range of repetition as the other and each iteration does not require the output of the other. The former can be checked by a unification technique and the latter can be examined through the input-output expressions corresponding to the respective procedure expressions.

Under these conditions, the above two 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)));
```

Thereby the running time and the memory for the program module can be reduced.

Similarly several WHILE-type iterative expressions can be fused to a single WHILE-type expression if they have the same test conditions of iteration and do not require the output of each other as the input data.

3.4. Transformation to programming languages

When non-primitive expressions cannot be found any more after refinement, MAPS transforms the refined specification into a programming language like C or LISP chosen in the user's option. The chosen programming language, however, must satisfy various conditions in order to realize the given refined specifications. For example, it needs to be able to process the data of some types given in a specification. After confirming the fulfilment of these conditions by the aids of a knowledge base of programming languages or a user's suggestions, MAPS transforms the type part into declaration statements of the chosen language. For example, MAPS transforms the record-type data processing specifications into a LISP program by using an associative list and an associative function according to the user's option.

MAPS also transforms the formal control expressions into a control statement of a programming language by using a table such as Table 2 which has a structure similar to that of a library module.

```
PROC:FOR(INDEX:i,FROM:m,TO:n,OBJECT:s),
TYPE:'i,m,n'( REGION:INTEGER,VARIABLEROLE:INPUT),
OP:(LISP (SETQ i m)
          (LOOP ( ) s
                    (SETQ i (ADD1 i))
                    (COND ((GREATERP i n)
                           (EXIT-LOOP)))))
   (C FOR(i=m;i<=n;++i){s})
     ................ ............
     ..........................
```

Table 2 The content of the "FOR" module

4. The Experimental System

An experimental system MAPS written in LISP has been constructed on the TSS of our university computer center (ACOS 850) and on a workstation (CPU MC68000, clock 10 MHz. memory 3.5 MB).

Fig. 3 shows an overview of the total system of MAPS. If specifications are written in a limited natural language like Japanese, MAPS transforms them to the formal specification. Then MAPS scans the formal specifications and notifies the users of various errors and defaults or specification blocks which cannot be unifiable to any heading of library module by referring to library modules. The user corrects the errors, complements the defaults by giving some newly built modules or giving some details to the specification block and inputs the validated specifications. If the specifications involve an input-output expression, MAPS tries to meet the specification block by linking some library modules. Then MAPS refines every block of the specifications by using operation

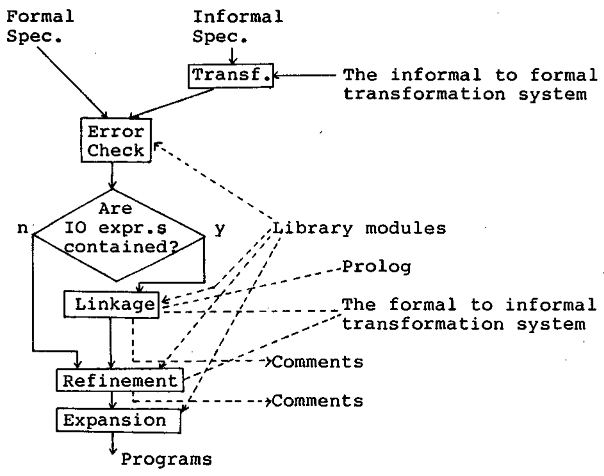

Fig. 3 An overview of the total system

parts of the unifiable modules, tries simple optimization such as fusion, and outputs a natural language like expressions corresponding to the refined specifications by the user's option. After refinement, MAPS transforms the specifications to a programming language expressions designated by the user.

Some experiments of refinement and transformation have been carried out. The average transformation time between Japanese-like expressions and the formal ones is about 0.2 sec. per word on ACOS 850. The linking and refinement of a specification block requires about 0.6 sec. on the average.

As for refining specifications in a somewhat specific field, a machine translation system written in C and LISP was reconstructed on a library module basis by MAPS[16]. The library modules were made by referring to the original machine translation system written in LISP. At present, a new MAPS itself also is under reconstruction by referring to original one.

Example 4 shows some simple experimental results[15]. The specification can be given in a certain kind of limited Japanese or English. It is transformed into a formal specification and refined. Then the limited Japanese or the limited English sentences corresponding to the refined specification are generated as a more readable expression for every refinement.

Example 4

A user gives a program specification of graph representation for a given mathematical function F(x). MAPS translates a Japanese input-output specification into a formal expression, constructs a refined specification by linking the operation parts of several instantiated modules by the aids of Prolog in order to obtain the maximum value of F(x) as follows:

The English specification:

```
    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 formal specification:

```
    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)))
```

The refinement by linking of modules:

```
    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,
               GRAPH_RANGE: X_RANGE (100, 400);
                            Y_RANGE (100, 400))              (13)
```

The formal expression Expr.(13) is further refined by replacing non-primitive expressions by the instantiated operation parts of their unifiable modules as follows:

```
  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))),

The expansion results in Lisp are shown in appendix 2. These programs works together with the function F(x) which is added to these programs as a function subroutine.

```
The program in Lisp:

(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 (y1 x1 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 y1 x1 format
                      x_min x_max y_min y_max
                      gx_min gx_max gy_min gy_max))
..........................................
..........................................
```

Appendix 2

5. Conclusion

The research has been done for several years. For practical use there remains several problems to be refined. One of them is to develop library modules which are more flexible more modifiable and more applicable to various specific fields. The other is to extend applicable natural-language like expressions and to introduce various mnemonic expressions similar to the conventional notations. However, it is expected that the method of refining specifications by using abstract program modules will bring fruitful results to various software development.

References

1) D.L. Parnas, Commun. ACM, **15**, (12), 1053 (1972).
2) D. TeiBchroew and E.A. Hershey, IEEE Trans. Software Eng., **SE-3-1**, 41 (1977).
3) N. Wirth, Commun. ACM, **14**, (4), 221 (1971).
4) C.L. Chang and R.C. Lee, "Symbolic Logic and Mechanical Theorem Proving", Academic Press (1973).
5) T. Pietrzykowski, J. Ass. Comput. Mach., **20**, (2), 333 (1973).
6) J.L. Darlington, "Automatic Synthesis of SNOBOL Programs," in Computer Oriented Learning Process, J.C. Simon Ed. Nordhoff-Leyden, pp. 443–453 (1976).
7) D.R. Barstow, "Knowledge-Based Program Construction", North Holland, (1979).
8) H. Partsch and R. Steinbruggen, Computing Surveys, **15**, (3), 199 (1983).
9) J.M. Neighbors, IEEE Trans. Software Eng., **SE-10**, (5), 564 (1984).
10) E. Horowitz and J.B. Munson, IEEE Trans. Software Eng., **SE-10**, (5), 477 (1984).
11) Y. Matsumoto, IEEE Trans. Software Eng., **SE-10**, (5), 502 (1984).
12) Z. Manna and R. Waldinger, "Studies in Automatic Programming Logic", North-Holland, New York (1977).
13) P.R. Cohen and E.A. Feigenbaum ed. "The handbook of artificial intelligence", vol. 3, W. Kaufmann (1982).
14) F. Nishida and Y. Fujita, Trans. of Inf. Proc. Soc. Japan, **25**, (5), 785 (1984).
15) F. Nishida, Y. Fujita and S. Takamatsu, Simposium on Prototyping and Requirements Specification, Inf. Proc. Soc. Japan, 111 (1986).
16) F. Nishida, Y. Fujita and S. Takamatsu, Proc. of 11th Inter. Conf. Comput. Linguist., 649 (1986).
17) F. Nishida, Y. Fujita and S. Takamatsu, Trans. of Inf. Proc. Soc. Japan, **28**, (5), 489 (1987).
18) F. Nishida, S. Takamatsu and T. Tani, Trans. of Inf. Proc. Soc. Japan, **29**, (4), 368 (1988).