Semi-automatic Program Refinement from Specifications Using Library Modules

Semi-automatic Program Refinement from Specifications Using Library Modules

Yoneharu FUJITA*, Fujio NISHIDA* and Osamu SHIONOIRI*

(Received June 15, 1985)

This paper describes an approach to a semi-automatic program refinement from specifications.

Specifications can be written in a kind of procedural expressions or input-output expressions. The system called SAPRE checks feasibility of the specifications, refines them semi-automatically by interacting with the user or referring to library modules which are constructed on a hierarchical logical basis and finally generates the corresponding program in a compiler language specified by the user. An experimental system is working now.

1. Introduction

In recent years, program specification techniques have been actively studied. In some approaches of them, specifications are described in a restricted natural language like PSL, analyzed semi-automatically from various aspects and summarized into a program document[3].

On the other hand program synthesis has been also studied for the past two decades and automatic construction and modification of programs can be made when the inference steps required for construction of programs are considerably small.

Different kinds of jobs are usually processed by different programs constructed on their job basis except the use of subroutines expressed by a programming language concretely, and a vast number of programs have been made so far. However if programs are classified from the view point of their algorithms, many parts of programs of different kinds can be found to consist of the same fundamental procedures and functions.

Hence if the program specifications and refinement are made in a top-down manner of data processing from planning to coding by using library modules, library modules applicable to the program specifications will be found, combined with each other and expanded at each level to a program according to the details of the specifications without serious difficulty[4)5)6)7)].

Along this line the present paper describes an approach to semi-automatic program refinement using library modules. The specifications are given by procedural expressions and IN-OUT expressions. These are described formally at the present stage of experiments but will also be written in an informal form such as a restricted natural language like PSL in the near future.

The system called the Semi-Automatic Program Refinement and Expansion system abbreviated to SAPRE searches library modules appropriate to given specifications and refines the specification by applying the library modules. Each library module has an operation part described in the same form as the specifications and involves query items about user's options for the refinement. The results of the refinement are shown in

* Department of Electrical Engineering

kind of diagrams of HIPO and the interaction is done between the user and the machine through a display.

The refinement is repeated until the specification is finally expanded into a target programming language.

In applying a library-module to refinement of specification or one of the descendants, unification in the second order logic is applied in a limited version with reasonable efficiency.

2. Program Specification

There are various fields in which specifications are needed. Therefore, the relevant expression form of specification can not be restricted to one form. From this point of view three kinds of program specifications can be used in this paper. They are a state-expression, a procedural expression and a formal Pascal-like expression.

2.1. Procedural expressions

The procedural expression used here is similar to those used in various programming languages. The procedural expression used here usually has case labels prefixed to the arguments for readability and convenience of conversion to a restricted natural language expression like PSL.

The main case labels and the definitions are shown as follows:

OBJect: the object entity of operation.
SOURce: the location of the data base from which the object is brought.
GOal: the location of the data base or the value to which the processed object is brought.
KEY: the key parameter used for processing the object.

Example 1

$$\text{sort}\ (OBJ: t\ (1\ ..\ 5000, 1\ ..\ 9),\ KEY: t\ (\$, 9),\ GO: t\ (1\ ..\ 5000, 1\ ..\ 9)) \tag{1a}$$

$$\text{search}\ (OBJ: t\ (\$, 1\ ..\ m),\ COND: P\ (t\ (\$, 1\ ..\ m), r\ (1\ ..\ m)),\ SO: t\ (1\ ..\ n, 1\ ..\ m),\ GO: (1\ ..\ l, 1\ ..\ m)), \tag{1b}$$

where $t\ (1\ ..\ 5000, 1\ ..\ 9)$ denotes a set consisting of 5000 tuples from $t\ (1, 1\ ..\ 9)$ to $t\ (5000, 1\ ..\ 9)$ and the symbol "$" denotes an arbitrary value in the range specified in a given set.

These expressions correspond to the following imperative sentences respectively:

"sort $t\ (1\ ..\ 5000, 1\ ..\ 9)$ by taking $t\ (\$, 9)$ as the key!" (2a)

"search $t\ (\$, 1\ ..\ m)$ under the condition $P\ (t\ (\$, 1\ ..\ m), r\ (1\ ..\ m))$ from $t\ (1\ ..\ n, 1\ ..\ m)$ and store it in ans $(1\ ..\ l, 1\ ..\ m)$!" (2b)

Expr. (1) and (2) can be converted to each other by a kind of simple rewriting rules.

A procedure $p\ (C_1: t_1, ..., C_n: t_n, GO: t)$, which has the GOal case can be also rewritten as follows:

$$t := p\ (C_1: t_1, ..., C_n: t_n), \tag{3}$$

where C_i and t_i $(i = 1, ..., n)$ are a case label and an argument respectively.

2.2. State expressions

A state expression consists of a pair of an input condition and an output condition. The formal expression of the condition usually takes a predicate form as follows:

$$(IN\colon P(x),\ OUT\colon Q(x)), \tag{4}$$

where $P(x)$ takes a conjunctive normal form and $Q(x)$ takes a disjunctive normal form or a conjunctive implication form;

$$\bigwedge_i \left(\bigwedge_j P_{ij} \rightarrow Q_i\right).$$

The case labels are also prefixed to the arguments optionally in the same way as the procedural expressions.

If p denotes a procedure which leads to the output state, the output condition can be also expressed as $\{p\}$. This is a convenient expression if it is defficult to express the output condition briefly.

Example 2

$$(IN\colon GIV\,(OBJ\colon t\,(1\,..\,5000, 1\,..\,8)),$$
$$OUT\colon {}^{\forall}ie\,(1\,..\,5000)\,(t\,(k, 9) = \mathrm{sum}\,(t\,(i, 4\,..\,8)))), \tag{5a}$$

where $GIV\,(OBJ\colon t)$ means that t is given as a datum.

The expression (5*a*) can be converted to the following declarative sentences and vice versa.

IN: "$t\,(1\,..\,5000, 1\,..\,8)$ is given."
OUT: "for any $ie\,(1\,..\,5000)$ $t\,(i, 9)$ is equal to the sum of entities from $t\,(i, 4)$ to $t\,(i, 8)$." (5b)

2.3. Pascal-like expressions

The usage of a compiler language like Pascal in specifications is also desirable and moreover, necessary for increase in flexibility of expressions such as iterative statements. However, it is desirable that the programming language expressions can be immediately converted to the other languages like *C* and *LISP* in expansion from specifications to these programming languages. For this purpose the formal expressions (6.1*a*) and (6.2*a*) corresponding to Pascal-like statements (6.1*b*) and (6.2*b*) are implemented as follows:

$$\mathrm{do}\,(FOR\colon i\colon 1, TO\colon n, OP\colon (IN\colon i, OUT\colon o)) \tag{6.1a}$$

$$\Leftrightarrow \mathrm{for}\ i\colon = 1\ \mathrm{to}\ n\ \mathrm{do}\ (IN\colon i, OUT\colon o), \tag{6.1b}$$

$$\mathrm{do}\,(WHILE\colon P, OP\colon (IN\colon i, OUT\colon o)) \tag{6.2a}$$

$$\Leftrightarrow \mathrm{while}\ P\ \mathrm{do}\ (IN\colon i, OUT\colon o). \tag{6.2b}$$

From the above formal expressions, the expansion to the other language expressions can be easily obtained by a conversion table.

In this paper, specifications are given by a sequence of the above three kinds of expressions in a mixed form. In order that the sequence of specifications is executable, the input conditions of state expressions and the arguments of procedural expressions except the arguments of GOal must be predetermined by the input data of the specifications or by the intermediate results using the input data. SAPRE checks whether or not the necessary condition holds.

Example 3

(a) Suppose that the data of 5000 applicants are given in an array *mark* (1 . . 5000, 1 . . 9) and the columns from the 5th to the 8th are marks in an examination. Then the specifications which store the sum of the marks into the 9th column of the table and sorts the 5000 tuples in the descending order of the sum are expressed by a sequence of Expr. (5a) followed by Expr. (1a).

(b) Suppose a reservation system of a hotel which receives a request and retrieves rooms satisfying a request condition of the form $r(1 \ldots m)$ from a room table to an answer table. If the request is a reservation the retrieved room is reserved using a room table.

The above specifications can be represented as follows:

request = "inquiry" ∨ request = "*reservation*"
→ search (*OBJ*: *room* ($, *j*1 . . *j*2), *COND*: ∧ (*FOR*: *j*: = *j*1, *TO*: *j*2, *OBJ*: *EQ* (*room* ($, *j*), req (*j*))), *SO*: *room* (*i*1 . . *i*2, *j*1, *j*2), *GO*: ans (*i*1 . . *i*2, *j*1 . . *j*2)) ;
request = "reservation" ∧ *k* ∈ ans (*i*1 . . *i*2, *j*1 . . *j*2)
→ update (*OBJ*: *room* (*i*, *reservation*), *COND*: *room* (*i*, key) = *k*,
SO: *room* (*i*1 . . *i*2, *j*1 . . *j*2), *GO*: 1), (7)

where the last expression means updating of the value of *room* (*i, reservation*) to unity and ∧ (*FOR*: *j*: = *j*1, *TO*: *j*2, *OBJ*: *Q*(*j*)) means $\overset{j2}{\underset{j=j1}{K}} Q(j)$.

3. Module Libraries and Unification

3.1. Module libraries

In order to refine given specifications semi-automatically, SAPRE has procedural modules in a library. These modules are classified into arithmetic-operation, table-operation, input-output operation and so on.

Each module consists of a heading part comprising a procedural expression and an input-output expression, a type description part, an operation part and a supplementary part as shown in Table 1. The two former parts are used for examining the applicabilities of modules to given specifications.

The types are classified into a scalar type and a composite type. The former consists of integer, real, character and others, while the latter consists of array records and others. A two-dimentional array consisting of tuples of *a* (*n*, *m*1 . . *m*2) (*n* ∈ an ordered set of (*n*1 . . *n*2)) is denoted by *a* (*n*1 . . *n*2, *m*1 . . *m*2) and the (*n*, *m*) element is denoted by *a* (*n*, *m*).

The operation part describes the procedures that bring the input state to the output

state. The procedures are also described in a form used for specifications.

The inquiry item to the user is implemented in the following form:

$$? \text{ select } (t1, t2, ..., tn), \tag{8}$$

which urges the user to specify one of the given arguments.

The supplementary part, which is omitted here, describes a subdivision of the operation method such as linear search, and also a rough measure of required steps.

The procedure symbol $f.*$ in the arithmetic operation denotes a repeated application of a binary operation f such as repeated addition or multiplication over n numbers. The repeated application can be defined if the binary operation f is associative and commutative.

3.2. Unification

Every library module takes a general form to cover a wide class of specifications. It often involves variable function symbols and variable predicate symbols. Hence, refinement of specifications requires unification of a library module with given specifications in the second order ligic. It is well known that there is no most general unifier in the second order logic and the unification tree is not closed generally. However, if the unification is limited to the unilateral direction from a library module to given specifications or one of the descendants, the unification tree is closed and the unification procedures can be carried out with reasonable efficiency.

Let us suppose two objects α and β, α is a literal consisting of symbols $a_1 \ldots a_k \ldots a_n$ and is included in a procedural expression in a library module or in its input or output predicate as follows:

$$(IN: P(X), OUT: Q(x, z), OP: op(x, z)). \tag{9}$$

Similarly, β is a literal consisting of symbols $b_1 \ldots b_k \ldots b_m$ and is included in given specifications or one of the descendants.

Assume that a_k and b_k are the leftmost symbols of disagreement in the objects α and β respectively. Then the simplified version of the unification procedure of α and β used here is described as follows:

(1) In the case of a_k, being an individual variable;
The unification procedure in the first order logic is applied by the following substitution;

$$\theta = \{a_k \leftarrow b_k\ b_{k+1} \ldots b_{k+l}\}, \tag{10}$$

where $b_k\ b_{k+1} \ldots b_{k+l}$ $(0 \leqslant l \leqslant m-k)$ is a subobject of β.

(2) In the case of a_k being the head of a subobject of a p-place function symbol or a p-place predicate symbol;

(a) Projection

If the type of the i-th argument of the function or the predicate is equal to that of a_k, the subobject is projected to the i-th argument by the following substitution;

$$\theta = \{a_k \leftarrow \lambda u_1 \ldots u_p \cdot u_i \cdot\}, \tag{11}$$

where $1 \leqslant i \leqslant p$.

(b) Imitation

Let b_k be a q-place function symbol or a q-place predicate symbol. Then the subobject in α is instantiated by the following substitution so that it imitates the subobject which has b_k as the first symbol;

$$\theta = \{a_k \leftarrow \lambda u_1 \ldots u_p \cdot b_k h_1 \ldots h_q \cdot\},$$

where $h_i = f_i u_1 \ldots u_p$, $(1 \leqslant i \leqslant q)$ and f_i is a p-place predicate symbol or a p-place function symbol which does not appear in α nor β.

In the above unification, the substitution is done for all the variable a_k appearing in Expr. (9).

Now the next theorem describes that the unilateral unification described here ends in finite steps.

Theorem 1

If unification is restricted to one direction from one object to another, the unification tree is closed.

Proof

Let us assume that a_k and b_k are the leftmost symbols and unification is restricted to one direction from α to β which is kept constant.

If a_k is a constant, this unification branch is pruned here because the execution of unification is impossible here.

If a_k is a variable, the unification procedure (1) or (2) is tried. In this case, if the type of a_k does not coincide with that of the subobject of β, the unification branch is also pruned. Otherwise a_k is unified with $b_k \ldots b_{k+l}$. Though several different branches of unification are generated for each symbol of disagreement in general, the length of the remaining disagreement part of β is always reduced by at least one every unification. Hence the unification tree is closed.

Example 4

(a) Two arithmetic objects α: $f(x, y)$ and β: $-(+(x, y), /(x, y))$ are unified by the substitution

$$f \leftarrow \lambda uv \cdot -(+(u, v), /(u, v)).$$

(b) Two logical objects α: $P(x(\$, l1 \ldots l2), r(l1 \ldots l2))$ and β: $\bigwedge_{j=j1}^{j2} EQ(room(\$, j), req(j))$ are unified by applying the following substitutions sequentially:

$P \leftarrow \lambda ul\,(v1\,..\,v2)\,u2\,(v1\,..\,v2)\cdot \bigwedge_{j=v1}^{v2} EQ\,(u1\,(j), u2\,(j))$;

$l1 \leftarrow j1$,

$l2 \leftarrow j2$,

$x \leftarrow \lambda u1u2 \cdot room\,(u1, u2)\cdot$,

$r \leftarrow \lambda u \cdot req\,(u)\cdot$

where $u1\,(v1\,..\,v2)\,u2\,(v1\,..\,v2)$ is the abbreviation of $u1\,(v1)\,..\,u1\,(v2)\,u2\,(v1)\,..\,u2\,(v2)$ and $\bigwedge_{j=j1}^{j2} Q\,(j)$ is written as $\wedge\,(FOR: j:=j1, TO: j2, OBJ: Q(j))$ in SAPRE as shown in Example 3(b).

4. Refinement and Modification

Specifications are modified and refined by using a module library. SAPRE searches modueles applicable to given specifications or to its parts by referring to the operation kind, the procedures or the input-output expressions, and the types of the involved subobjects of library modules.

4.1. Refinement

Let us suppose that SAPRE finds a library module that involves given specifications as an instance. If there are such library modules more than one which are different from each other in detailed operations, a relevant module is selected from them by the user or SAPRE under the support of the supplementary parts. The scheme of refinement is replaced by the unified result of the OP part of the module together with the type of the involved objects and the result can be also shown by a diagram like HIPO. Then selection is urged to be made by the user.

If the user designates one of the arguments of selection, SAPRE adopts the selected string. One kind of selection is that of block structure or nonblock one. The selection will be made reasonably if the user knows in advance whether or not the module will be used at many places in the programs.

Example 5

Let us refine the following procedural part of the first specification in Example 3(b):

search $(OBJ: room\,(\$, j1\,..\,j2), SO: room\,(i1\,..\,i2, j1\,..\,j2)$,

$$COND: \bigwedge_{j=j1}^{j2} EQ\,(room\,(\$, j), req\,(j)), GO: ans\,(i1\,..\,i2, j1\,..\,j2)) \qquad (13)$$

SAPRE searches the module library and finds that the procedural expression of the table-search module is unifiable with Expr. (13) by the following substitutions:

$x \leftarrow \lambda u1u2 \cdot room\,(u1, u2)\cdot$,

$m1 \leftarrow j1, m2 \leftarrow j2, n1 \leftarrow i1, n2 \leftarrow i2$,

$P \leftarrow \lambda u1\,(v1\,..\,v2)\,u2\,(v1\,..\,v2)\cdot \bigwedge_{j=v1}^{v2} EQ\,(u1\,(j), u2\,(j))\cdot$,

$r \leftarrow \lambda u \cdot req\,(u)\cdot$,

$z \leftarrow \lambda u1u2 \cdot ans\,(u1, u2)\cdot$,

The application of the above substitutions to the *OP* part of the module and the adoption of block structure yield the following refinement:

make block
$\forall i \in (i1 \,..\, i2)\,(\bigwedge_{j=j1}^{j2} EQ\,(room\,(i,j), req\,(j))$
$\rightarrow add\,(OBJ: room\,(i, j1\,..\,j2), GO: ans\,(i1\,..\,i2, j1\,..\,j2)))$.

Subsequently, the part of 'add' is rewritten by further refining the part and adopting caution as follows:

$k := k + 1$;
$k \geqslant i2 \rightarrow$ call monitor;
$z\,(k, j1\,..\,j2) := room\,(i, j1\,..\,j2)$;

4.2. Modification

If there are several modules each of which contains only a part of the specifications as an instance, refinement of the specifications is tried by modifying the specifications and partitioning them. Some of the main rules are shown as follows:

(1) partition of specifications

(a) $(IN: P_1\,(X), OUT: Q_1\,(x, z), OP: pr_1\,(OBJ: x, GO: z))$,
$(IN: P_2\,(x), OUT: Q_2\,(x, z), OP: pr_2\,(OBJ: x, GO: z))$
$P_1\,(x) \wedge Q_1\,(x, z) \rightarrow P_2\,(x)$

$(IN: P_1\,(x), OUT: Q_2\,(x, z), OP: pr_1\,(OBJ: x, GO: y_1); pr_2\,(OBJ: y_1, GO: z))$

(b) $(IN: P\,(x), OUT: Q_2\,(x, z_1), OP: pr_1\,(OBJ: y, GO: z_1))$,
$(IN: P\,(x), OUT: Q_2\,(x, z_2), OP: pr_2\,(OBJ: x, GO: z_2))$
$z_1 \neq z_2$

$(IN: P\,(x), OUT: Q_1\,(x, z_1) \wedge Q_2\,(x, z_2), OP: \{pr_1\,(OBJ: x, GO: z_1), pr_2\,(OBJ: x, GO: z_2)\})$

The above configurations mean that the precedent conditions over a horizontal line lead the consequent conditions under the line. When SAPRE finds that given specifications involve a library module as a part, it tries to apply the partitioning rules to the specifications from the consequent conditions to the precedent conditions.

Example 6
$(IN: GIV\,(OBJ: x\,(1\,..\,n))$,
$OUT: z = sum\,\{(x\,(i) - average\,(x\,(1\,..\,n)))**2 \mid x\,(i)\,\epsilon x\,(1\,..\,n)\}/n)$

If the average function is not ready for a procedure call, the specifications are divided into two parts by the above rule (a) as follows:

$$(IN\text{: } GIV\,(OBJ\text{: } x\,(1\,..\,n)),\ OUT\text{: } u_1 = average\,(x\,(1\,..\,n)),$$
$$OP\text{: } average\,(OBJ\text{: } x\,(1\,..\,n),\ GO\text{: } u_1))$$

$$(IN\text{: } GIV\,(OBJ\text{: } x\,(1\,..\,n))\ \Lambda u_1 = \text{average}\,(x\,(1\,..\,n)),$$
$$OUT\text{: } z = \text{sum}\{(x\,(i) - u_1)**2 \mid x\,(i)\ \epsilon x(1\,..\,n)\}/n,$$
$$OP\text{: } z := \text{sum}\ \{(x\,(i) - u_1)**2 \mid x\,(i)\ \epsilon x\,(1\,..\,n)\}/n)$$

(2) Modification of Specifications

In order that a library module is applicable to a part of given specifications, the specifications are also modified by using specific knowledge which are stored in a knowledge base according to their special fields. The limited version of unification in the second order logic is also used in the modification if it is required.

Example 7

Let us assume in Example 5 that the room table is separated into a room-reservation table *tr* and a room-characteristic table *tc*. The reservation table is to be updated every day while the characteristic table is almost invariable. Under the assumption that the number of the tables searched by the procedure 'search' is restricted to only one at a time, the output condition corresponding to the Expr. (13) takes the following form:

$$\begin{aligned} \text{ans}\cdot 2 = \{(tr\cdot 1, tc\cdot 1) \mid\ & tr\cdot 1 \epsilon tr\cdot 2, Pr\,(tr\cdot 1, rr\cdot 1), \\ & tc\cdot 1 \epsilon tc\cdot 2, Pc\,(tc\cdot 1, rc\cdot 1), \\ & tr\cdot 1\,(key) = tc\cdot 1\,(key)\}, \end{aligned} \tag{14}$$

where the following abbreviated forms are used:

$tr\cdot 1 \triangleq^{d} tr\,(\$, j1\,..\,j3),\ tc\cdot 1 \triangleq^{d} tc\,(\$, j3+1\,..\,j2),$

$tr\cdot 2 \triangleq^{d} tr\,(i1\,..\,i2, j1\,..\,j3),\ tc\cdot 2 \triangleq^{d} tc\,(i1\,..\,i2, j3+1\,..\,j2),$

$rr\cdot 1 \triangleq^{d} rr\,(j1\,..\,j3),\ rc\cdot 2 \triangleq^{d} rc\,(j3+1\,..\,j2).$

$(tr\cdot 1,\ tc\cdot 1)$ denotes a joined form of tuples $tr\cdot 1$ and $tc\cdot 1$. $rr\cdot 1$ and $rc\cdot 1$ denote the room-reservation part and the room characteristic part of a request respectively. $tr\cdot 1$ (key) denotes the key-attribute value of a tuple $tr\cdot 1$.

By inspecting Table 1, SAPRE finds the OUTput predicate of the procedure 'join-key' which includes a part of Expr. (14), then tries to modify Expr. (14) to a closer form to the OUTput predicate. Using the axiom

$$P\,(x) \rightleftarrows x \epsilon \{y \mid P\,(y)\} \tag{15}$$

in the knowledge base, SAPRE generates the following modified form of Expr. (14):

$$\begin{aligned} ans\cdot 2 = \{(tr\cdot 1, tc\cdot 1) \mid\ & tr\cdot 1 \epsilon \{y1\cdot 1 \mid y1\cdot 1 \epsilon tr\cdot 2, Pr\,(y1\cdot 1, rr\cdot 1)\} \\ & tc\cdot 1 \epsilon \{y2\cdot 1 \mid y2\cdot 1 \epsilon tc\cdot 2, Pc\,(y2\cdot 1, rc\cdot 1)\}, \end{aligned}$$

Table 1 A part of library modules

	Arithmetic operation

(1) PROC
IN
OUT
TYPE
OP
	f·∗ (*OBJ*: *g* (*x* (*i*1 . . *i*2)), *GO*: *z*)
	GIV (*OBJ*: *x* (*i*1 . . *i*2)), *f*∈binary arithmetic operator, *g*∈unary arithmetic operator
	z = *f*·∗ (*OBJ*: *g* (*x* (*i*1 . . *i*2)))
	x (*i*1 . . *i*2): array of ?select (real, integer), *z*: var of ?select (real, integer)
	TYPE *y*: var of ?select (real, integer);
	?select (block) → putstring ("make block");
	y: = *g* (*x* (*i*1));
	do (*FOR*: *i*: = *i*1 + 1, *TO*: *i*2, *OP*: *y*: = *f*(*y*, *g*(*x* (*i*))));
	z: = *y*

(2) PROC IN OUT TYPE OP	sum (*OBJ*: *g* (*x* (*i*1 . . *i*2)), *GO*: *z*) *GIV* (*OBJ*: *x* (*i*1 . . *i*2)), *g*∈unary arithmetic operator *z* = +·∗ (*OBJ*: *g* (*x* (*i*1 . . *i*2))) *x* (*i*1 . . *i*2): array of ?select (real, integer), *z*: var of ?select (real, integer) +·∗ (*OBJ*: *g* (*x* (*i*1 . . *i*2)), *GO*: *z*)
	Table operation
(1) PROC IN OUT TYPE OP	add (*OBJ*: *x* (*m*1 . . *m*2), *GO*: *z* (*n*1 . . *n*2, *m*1 . . *m*2)) *GIV* (*OBJ*: *x* (*m*1 . . *m*2)), *n*≜maxpointer (*z* (*n*1 . . *n*2, *m*1 . . *m*2)) *n*: = *n* + 1, *x* (*m*1 . . *m*2) ∈*z* (*n*1 . . *n*2, *m*1 . . *m*2) *x* (*m*1 . . *m*2): array of character, *z* (*n*1 . . *n*2, *m*1 . . *m*2): array of character, *n*: var of integer ?select (block) → putstring ("make block"); *n*: = *n* + 1; ? select (caution) → putstring ("*n* ⩾ *n*2 → call monitor"); *z* (*n*, *m*1 . . *m*2): = *x* (*m*1 . . *m*2)
(2) PROC IN OUT TYPE OP	search (*OBJ*: *x* (∗, *m*1 . . *m*2), *SO*: *x* (*n*1 . . *n*2, *m*1 . . *m*2), *COND*: *P* (*x* (∗, *m*1 . . *m*2), *r* (*m*1 . . *m*2)), *GO*: *z* (*n*1 . . *n*2, *m*1 . . *m*2)) *GIV* (*OBJ*: *x* (*n*1 . . *n*2, *m*1 . . *m*2), *P*, *r* (*m*1 . . *m*2)) *z* (*n*1 . . *n*2, *m*1 . . *m*2) = { *x* (*i*, *m*1 . . *m*2) | *P* (*x* (*i*, *m*1 . . *m*2), *r* (*m*1 . . *m*2))} *x* (*n*1 . . *n*2, *m*1 . . *m*2): array of character, *z* (*n*1 . . *n*2, *m*1 . . *m*2): array of character, *r* (*m*1 . . *m*2): array of character, *n*: var of integer, *P*: two place predicate ?select (block) → putstring ("make block"); $"optimize your test condition"; ∀*i*∈ (*n*1 . . *n*2) (*P* (*x* (*i*, *m*1 . . *m*2), *r* (*m*1 . . *m*2)) → add (*OBJ*: *x* (*i*, *m*1 . . *m*2), *GO*: *z* (*n*1 . . *n*2, *m*1 . . *m*2)))
(3) PROC IN OUT TYPE OP	join·key (*OBJ*: (*x*1 (*l*1 . . *l*2, *m*1 . . *m*2), *x*2 (*l*3 . . *l*4, *m*3 . . *m*4)), *KEY*: *x*1 (∗, *key*1) = *x*2 (∗, *key*2), *GO*: *z* (*n*1 . . *n*2, (*m*1 . . *m*2, *m*3 . . *m*4))) *GIV* (*OBJ*: (*x*1 (*l*1 . . *l*2, *m*1 . . *m*2), *x*2 (*l*3 . . *l*4, *m*3 . . *m*4))) *z* (*n*1 . . *n*2, (*m*1 . . *m*2, *m*3 . . *m*4)) = { (*x*1 (∗, *m*1 . . *m*2), *x*2 (∗, *m*3 . . *m*4) | *x*1 (∗, *m*1 . . *m*2) ∈ *x*1 (*l*1 . . *l*2, *m*1 . . *m*2), *x*2 (∗, *m*3 . . *m*4) ∈ *x*2 (*l*3 . . *l*4, *m*3 . . *m*4), *x*1 (∗, *key*1) = *x*2 (∗, *key*2)} *x*1 (*l*1 . . *l*2, *m*1 . . *m*2): array of character, *x*2 (*l*3 . . *l*4, *m*3 . . *m*4): array of character, *z* (*n*1 . . *n*2, (*m*1 . . *m*2, *m*3 . . *m*4)): array of character ?select (block) → putstring ("make block"); ∀*l*∈ (*l*1 . . *l*2) (*l*1 := *l*3 ; do (*WHILE*: *x*1 (*l*, *key*1) ≠ *x*2 (*ll*, *key*2) ∧ *ll* ⩽ *l*4, *OP*: *ll*: = *ll* + 1); *ll* ⩽ *l*4 → add (*OBJ*: (*x*1 (*l*, *m*1 . . *m*2), *x*2 (*ll*, *m*3 . . *m*4)), *GO*: *z* (*n*1 . . *n*2, (*m*1 . . *m*2, *m*3 . . *m*4))))

$$tr\cdot 1\,(key) = tc\cdot 1\,(key)\Big\}.$$

The application of the partitioning rule (*a*) to the above yields the following procedures:

search (*OBJ*: $tr\cdot 1$, *SO*: $tr\cdot 2$, *COND*: *Pr* ($tr\cdot 1, rr\cdot 1$), *GO*: $z1\cdot 2$) ;

search (*OBJ*: $tc\cdot 1$, *SO*: $tc\cdot 2$, *COND*: *Pc* ($tc\cdot 1, rc\cdot 1$), *GO*: $z2\cdot 2$) ;

join-key (*OBJ*: ($z1\cdot 2, z2\cdot 2$), *KEY*: $z1\cdot 1\,(key) = z2\cdot 1\,(key)$, *GO*: $ans\cdot 2$) (16)

where

$ans\cdot 2 \stackrel{d}{=} ans\,(i1\,..\,i2, j1\,..\,j2)$,

$z1\cdot 2 \stackrel{d}{=} z1\,(i1\,..\,i2, j1\,..\,j2), z2\cdot 2 \stackrel{d}{=} z2\,(i1\,..\,i2, j3+1\,..\,j3)$.

5. Expansion to Program Expressions

After specifications have been refined to several modules consisting of well-defined formal expressions and optimized globally, these modules are expanded into a target language designated by users. Programs to be expanded consists of a declarative part and a execution part. The former part is constructed by using information about types and block selection specified in refinement. The latter part is generated from operation parts by a kind of rewriting rules of programming languages as shown in Expr. (6).

The next example shows the main execution part of Expr. (16) expanded to ISO Pascal.

Example 8

Suppose that the expression form of the module 'search' is designated to a block in Example 7 and test predicates *Pr* and *Pc* are defined in the declarative part of the main program, then succeeding to the declarative part the input statement of data and the procedure search, the execution parts are expanded as follows:

```
procedure search (x: array (...), z: ..., i: ..., r: ..., p: ...)
begin
  .
  .
  i: = 0;
  search (tr (i1 . . i2, j1 . . j3), z1 (i1 . . i2, j1 . . j3), i, rr (j1 . . j3), Pr);
  i: = 0;
  search (tc (il . . i2, j3 + 1 . . j2), z2 (i1 . . i2, j3 + 1 . . j2), i,
          rc (j3 + 1 . . j2), Pc);
  for l: = i1 to i2 do
    begin
      ll: = il;
```

```
        while (z1 (l, key 1) <> z2 (ll, key 2) ) and (ll <= i2)
          do ll: ll + 1;
        n: = n + 1;
        if n > n2 then monitor;
        for j: = j1 to j3 do
          ans [n, j]: = z1 [l, j];
        for j: = j3 + 1 to j2 do
          ans [n, j]: = z2 [ll, j]
        end
        .
        .
    end
```

6. Conclusion

SAPRE refined many specifications by the aids of library modules and expanded them to several compiler languages designated by the users. The unilateral unification was carried out with reasonable efficiency and useful for semiautomatic refinement by interaction with users.

Automatic linking facilities of modules and specifications by some restricted natural languages are to be introduced to SAPRE in the near future.

References

1) T. Pietrzykowski, A complete mechanization of secondorder type theory, JACM, 20 (2), 333 (1973).
2) J.L. Darlington, Automatic Synthesis of SNOBOL programs, Computer oriented learning processes, J.C. Simon (ed.), Nordhoff-Leyden, 443 (1976).
3) D. Teichrow and E.A. Hershey, PSL/PSA: a computer-aided technique for structured documentation and analysis of information processing systems, IEEE trans, SE-3 (1), 41 (1977).
4) D.R. Barstow, "Knowledge-based program construction", Elsevier North Holland, Inc. (1979).
5) J. Foissequ et al., Program development with or without coding. IFIP, 327 (1980).
6) Y. Fujita and F. Nishida, A Program Synthesis by hierarchical Definition System, IECE of Japan, Trans, J61-D (2), 103 (1978).
7) Fujio Nishida and Yoneharu Fujita, Semi-Automatic Program Refinement from Specification Using Library Modules, IPS of JAPAN, Trans, 25 (5), 785 (1984).