Title	An interactive support system for constructing programs in SIMPL
Sub Title	
Author	永田, 守男(Nagata, Morio) 三嶋, 良武(Mishima, Yoshitake) 鹿野, 芳之(Shikano, Yoshiyuki) 原田, 賢一(Harada, Kenichi)
Publisher	慶應義塾大学理工学部
Publication year	1982
Jtitle	Keio Science and Technology Reports Vol.35, No.9 (1982. 8) ,p.153- 167
JaLC DOI	
Abstract	This paper proposes an approach to provide inter-module and inter-procedural information to programmers who are working together in a program development project. This approach is based on the following idea: such information can be easily obtained by modifying some components of the existing compiler to output key information of source programs and implementing the module information manager and analyzer. The SIP (SIMPL Interactive Programming) system on the basis of this approach has been designed and implemented as a tool for supporting SIMPL program construction. The effectiveness of the approach is described here by giving the actual output from this system. Our approach is also useful to create and modify programs written in the procedure-oriented language with the separate compilation facility.
Notes	
Genre	Departmental Bulletin Paper
URL	https://koara.lib.keio.ac.jp/xoonips/modules/xoonips/detail.php?koara_id=KO50001004-00350009-0153

KEIO SCIENCE AND TECHNOLOGY REPORTS
VOL. 35, NO. 9, pp. 153-167, 1982

AN INTERACTIVE SUPPORT SYSTEM FOR CONSTRUCTING PROGRAMS IN SIMPL

Morio NAGATA, Yoshitake MISHIMA* and Yoshiyuki SHIKANO

Department of Administration Engineering, Keio University
3-14-1 Hiyoshi, Yokohama 223, Japan

Ken'ichi HARADA

Institute of Information Science, Keio University
4-1-1 Hiyoshi, Yokohama 223, Japan

(Received July 1982)

ABSTRACT

This paper proposes an approach to provide inter-module and inter-procedural information to programmers who are working together in a program development project. This approach is based on the following idea: such information can be easily obtained by modifying some components of the existing compiler to output key information of source programs and implementing the module information manager and analyzer. The SIP (SIMPL Interactive Programming) system on the basis of this approach has been designed and implemented as a tool for supporting SIMPL program construction. The effectiveness of the approach is described here by giving the actual output from this system. Our approach is also useful to create and modify programs written in the procedure-oriented language with the separate compilation facility.

1. Introduction

As one of programming methodologies, it has been proposed that the program should be divided into modules as sets of logical and functional units. Moreover, incremental programming has come to be recognized as a promising methodology. Here, term of "incremental programming" means a program development process with taking advantage of separate compilation facility for each module in an interactive environment. Combining these notions, this paper presents an inter-module analysis tool for definition and use of identifiers (variables, procedures and

* Present address: Mitsubishi Research Institute, Inc., Time and Life Building, 2-3-6 Ohtemachi, Chiyoda-ku, Tokyo 100, Japan

functions) which works cooperating with a compiler.

Really to construct a procedure by using the structured programming language in the incremental programming environment[3], inter-procedural information at that state of programming is useful. For example, global variables used in other procedures are important to construct a new procedure. But it is difficult to obtain such information, and as the state of programming progresses, the configuration of a program always changes. Thus, our attention is concentrated upon inter-procedural and/or inter-module information at any stage of the program construction process. Notice that the word 'procedure' is used to mean both a procedure and a function in this paper. Every module consists of a sequence of procedures and functions that can access any set of global variables, parameters or local variables.

Our system, called the SIP (SIMPL[5] Interactive Programming) system, provides the following inter-procedural information in accordance with programmer's request: attributes and cross reference lists of all identifiers in each module or over modules, data binding list on global variables, a call graph representing invocation relationships between each module to the other modules, statistical information of programs, and check list of interface of a module and the other modules. Whenever a programmer wants to get such information, he will use the SIP system interactively.

The main design policy is to provide useful facilities by a little modification to the existing compiler. The SIP system has been implemented by adding only a function for producing Line Reference Table (LRT) to the SIMPL compiler.

2. An Overview of the SIP System

2.1 Structure of the SIP System

The SIP system consists of the SIMPL compiler and the inter-module analyzer. Although the SIP system does not allow any dynamic editing or compiling, a user can get a great many static information from the SIP system. At each time of a compilation, the SIP system collects the internal information of a program given by the compiler. As a result, the SIP system can respond to the user's query on his programs interactively. Figure 1 shows a structure of the SIP system.

2.2 Example

Using a small SIMPL program, we illustrate the effectiveness of information in the process of the program construction. The program (COMPEXP) computes A**B under the restriction that the addition (ADD) and multiplication (MULT) operations for given integers are only allowed with operations of successor (SUCC) and predecessor (PRED). The procedure EXPS(A, B, C) to return a value of A**B in C is called by the main procedure COMPEXP. Let us assume that all 6 procedures are separately compiled. The whole program COMPEXP is shown in Figure 2.

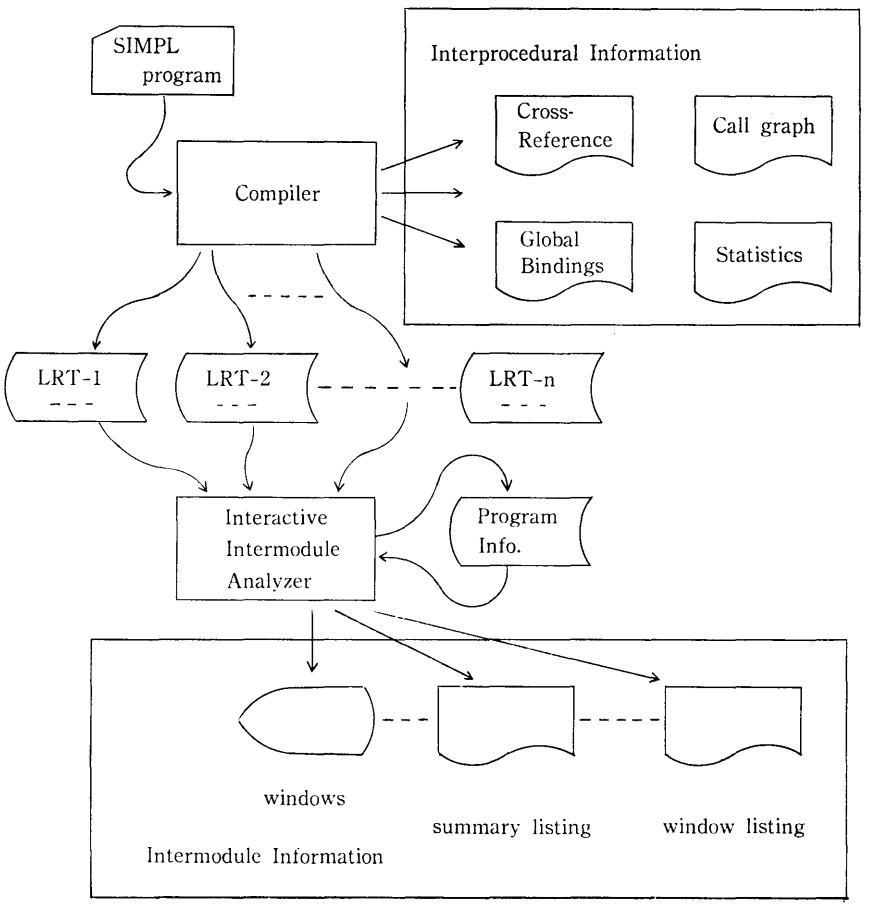

Fig. 1. Structure of the SIP System.

In the process of program construction, the previous procedures are treated as if all of them were the internal procedures in one module. That is, the SIP system gives a programming environment that above programs can be considered as one program illustrated in Figure 3.

Note that another example of COMPEXP (i.e. Figure 3) is compiled as a real program, and the SIP system can process above 6 internal ones, because of analyzing the complete flow of control using the merged call graph shown in Figure 4.

2.3 Output from the Compiler

On the contrary to a normal compilation, the compiler gives some useful information independently on the other separately compiled modules. The output information with some typical examples is as follows:

1. Cross reference list: The attributes and cross reference list of all identifiers in one program module are generated.

```
 1        ENTRY PROC SUCC(REF INT P)
 2          P:=P+1

 4        ENTRY PROC PRED(REF INT Q)
 5          Q:=Q-1

 7        ENTRY PROC ADD(INT X,INT Y,REF INT Z)
 8          EXT PROC PRED(REF INT), SUCC(REF INT)
 9          Z:=X
10          WHILE Y DO
11            CALL SUCC(Z)
12            CALL PRED(Y)
13          END

15        ENTRY PROC MULT(INT N,INT M,REF INT S)
16          EXT PROC ADD(INT, INT, REF INT), PRED(REF INT)
17          INT T
18          S:=0
19          WHILE M DO
20              CALL ADD(S,N,T)
21              S:=T
22              CALL PRED(M)
23            END

25        ENTRY PROC EXPS(INT E,INT F,REF INT G)
26          EXT PROC MULT(INT,INT,REF INT), PRED(REF INT)
27          INT H
28          G:=1
29          WHILE F DO
30            CALL MULT(G,E,H)
31            G:=H
32            CALL PRED(F)
33          END


35        PROC COMPEXP
36          EXT PROC EXPS(INT,INT,REF INT)
37          INT A,B,C
38          READ(A,B)
39          CALL EXPS(A,B,C)
40          WRITE(A,B,C,SKIP)


42        START
```

Fig. 2. COMPEXP (external).

2 Global data binding[1] list: The data binding information on global variables is produced by inter-procedural analysis for one program module. As the invocation relationships between procedures have been made clear in a form of internal representations, a binding analysis can determine a point, where a global variable is defined, and where the global is referenced to. Through these analyses, some logical errors such as side-effects with no care can be easily detected. Figure 5 gives an example of global data bindings. For instance, a global variable SYMTAB is referenced in procedures PROCFUNCATTR, GENCALLEDCHAIN and PRTCALLGRAPH, while it is not modified in PROCFUNCATTR. This means that PROCFUNCATTR may use a value of SYMTAB after modified in GENCALLEDCHAIN, and we can find a data binding for variable SYMTAB in these three procedures.

3. Call graph: By analyzing the invocation relationships between procedures,

```
 1                PROC SUCC(REF INT P)
 2     1    1         P:=P+1

 4                PROC PRED(REF INT Q)
 5     2    1         Q:=Q-1

 7                PROC ADD(INT X,INT Y,REF INT Z)
 8     3    1         Z:=X
 9     4    1         WHILE Y DO
10     5    2             CALL SUCC(Z)
11     6    2             CALL PRED(Y)
12                     END

14                PROC MULT(INT N,INT M,REF INT S)
15                     INT T
16     7    1          S:=0
17     8    1          WHILE M DO
18     9    2              CALL ADD(S,N,T)
19    10    2              S:=T
20    11    2              CALL PRED(M)
21                     END

23                PROC EXPS(INT E,INT F,REF INT G)
24                     INT H
25    12    1          G:=1
26    13    1          WHILE F DO
27    14    2              CALL MULT(G,E,H)
28    15    2              G:=H
29    16    2              CALL PRED(F)
30                     END

32                PROC COMPEXP
33                     INT A,B,C
34    17    1          READ(A,B)
35    18    1          CALL EXPS(A,B,C)
36    19    1          WRITE(A,B,C,SKIP)


38                START COMPEXP
```

Fig. 3. COMPEXP (internal).

```
Merged Call Graph
-----------------

0       1       2       3       4

COMPEXP  **Main-Proc**

        EXPS(int,int,ref int)  **Ext-Proc**

                MULT(int,int,ref int)  **Ext-Proc**

                        ADD(int,int,ref int)  **Ext-Proc**

                                SUCC(ref int)  **Ext-Proc**

                                PRED(ref int)  **Ext-Proc**

                        PRED(ref int)  **Ext-Proc**

                PRED(ref int)  **Ext-Proc**
```

Fig. 4. Merged Call Graph for COMPEXP.

```
        Global Data Bindings List

PROC/FUNC       Globals/Bindings -- Line Numbers

PROCFUNCATTR
                SEGNO
                        [Accessed]  69, 118
                PLINE
                        [Modified]  71, 72, 75, 77, 78, 83, 91, 96, 101, 106
                        [Accessed]  115
                BLANK
                        [Accessed]  71
                CALLTBL
                        [Accessed]  73
                SYMTAB
                        [Accessed]  74, 78, 80, 99, 104
                PROCTBL
                        [Accessed]  77
                PROCFLAG
                        [Accessed]  81
                ENTRYFLAG
                        [Accessed]  94
                FWDFLAG
                        [Accessed]  109

GENCALLEDCHAIN
                SEGNO
                        [Accessed]  130
                CALLTBL
                        [Accessed]  132, 158
                REFS
                        [Accessed]  136
                SYMTAB
                        [Modified]  139, 159
                        [Accessed]  137, 139
                PROCFUNC
                        [Accessed]  137
                PROCTBL
                        [Modified]  149, 157
                        [Accessed]  142
                CALLED
                        [Modified]  147, 151
                        [Accessed]  143, 145
                GAVAL
                        [Modified]  152
                        [Accessed]  147, 149, 151, 152

PRTCALLGRAPH
                SYMTAB
                        [Modified]  202, 207
                        [Accessed]  171, 173, 188
                PLINE
                        [Modified]  172, 174, 180, 182, 190
                        [Accessed]  174, 180, 182, 185, 190, 192
                COLN
                        [Accessed]  184, 187
                EXTFLAG
                        [Accessed]  188
                BLANK
                        [Accessed]  192
                PROCTBL
                        [Accessed]  197
                CALLED
```

Fig. 5. An Example of Global Binding List.

Listing of Procedures and Functions

No.	Name	Calls	Called	Attribute
1	PROCFUNCATTR	1	1	PROC
2	GENCALLEDCHAIN	0	1	PROC
3	PRTCALLGRAPH	2	3	PROC, REC
4	CALLINGLEVEL	1	2	PROC, REC
5	CALLGRAPH	3	1	PROC
6	DUMP	0	0	PROC
7	CALLGRAPHGEN	3	0	PROC, ENTRY
8	PUTGLINE	0	2	PROC
9	MANYGLOBALS	0	2	PROC
10	PRTGLOBALBIND	4	1	PROC
11	GLOBALSOFSEG	2	1	PROC
12	GLOBALDATA	2	1	PROC, FWD
13	GLOBALBIND	1	0	PROC, ENTRY

C A L L G R A P H

```
0       1       2       3

CALLGRAPHGEN(316)  **Entry**
        GENCALLEDCHAIN(123)
        PROCFUNCATTR(56)
                NAME(20)  **Ext**
        CALLGRAPH(241)
                CALLINGLEVEL(213)
                        CALLINGLEVEL(213)
                PRTCALLGRAPH(166)
                        NAME(20)  **Ext**
                        PRTCALLGRAPH(166)
                PRTCALLGRAPH(166)

GLOBALBIND(534)  **Entry**
        GLOBALSOFSEG(474)
                GLOBALDATA(51)
                        MANYGLOBALS(431)
                        MANYGLOBALS(431)
                PRTGLOBALBIND(442)
                        NAME(20)  **Ext**
                        NAME(20)  **Ext**
                        PUTGLINE(382)
                        PUTGLINE(382)
```

Fig. 6. An Example of Call Graph.

```
    --- Entry variables defined ---

                                                    attribute of variable
        NOFPARAM            108   INT, ENTRY  <- - - -
        PARAMSTR            117   STRING[20](20), ENTRY
        PARAMVAR            114   INT(20), ENTRY
        ..... ↖                   ↑
            variable name         |---line number where the variable is declared

Information for module - PSTAT4.SPL.12(02/08/82-13:49:03)

    --- Entry variables defined ---

        None.

Information for module - PSTAT5.SPL.12(02/08/82-14:09:07)

    --- Entry variables defined ---

        None.

Information for module - PSTATM.SPL.18(01/27/82-21:33:32)

    --- Entry variables defined ---

        BLANKS              36   STRING[132], ENTRY
        CELL                69   INT(1024), ENTRY
        CURRMODNO           31   INT, ENTRY
        DUMMY1              51   INT(97), ENTRY
        ENTRYDEFC           72   INT, ENTRY
        EXTREFC             73   INT, ENTRY
        FUNCNO              65   INT, ENTRY
        IDSYM               67   STRING[8](1024), ENTRY
        INFPSIZE            30   INT, ENTRY
        LIBFUPDT            32   INT, ENTRY
        LINENO              62   INT, ENTRY
        MODDV1              58   INT, ENTRY
        MODDV2              59   INT, ENTRY
        MODNAME             61   STRING[80], ENTRY
        MODPTR              50   INT(50), ENTRY
        MODULES             49   STRING[20](50), ENTRY
        NOFENTPF            74   INT, ENTRY
        NOFENTV             75   INT, ENTRY
        NOFEXTPF            76   INT, ENTRY
        NOFEXTV             77   INT, ENTRY
        NOFMODULE           47   INT, ENTRY
        NXTAVAIL            48   INT, ENTRY
        PFCHAIN             71   INT, ENTRY
        PROGNAME            46   STRING[20], ENTRY
        REFS                70   INT(4000), ENTRY
        SEGNO               64   INT, ENTRY
        STMTCNT             63   INT, ENTRY
        SYMTAB              68   INT(3038), ENTRY
        TEXT                35   STRING[80], ENTRY
        TOPPTR              66   INT, ENTRY
        .....
```

Fig. 7. An Example of "LIST" Command.

```
Rererenced in -- PARSEQUADS.SPL.9(02/03/82-18:09:53)

   IDNAME            STRING <------ type of identifier
                     459* <-------- value is assigned at line 459.
                                    "*" denotes an assignment.
     identifier to be searched


Rererenced in -- SCANDECL.SPL.5(02/03/82-18:07:19)

   IDNAME            STRING
                     139, 142, 144, 267, 269, 422*, 428, 432*, 432, 434,
                     638*, 667, 671*, 671, 675, 677*, 677, 682, 687, 849,
                     1050


Rererenced in -- SCANMACRO.SPL.7(02/03/82-18:06:05)

   IDNAME            STRING
                     115*, 124*, 126*, 131*, 136, 138, 143, 145, 147, 181,
                     183, 184, 187, 198, 206, 208, 218, 220, 230, 232, 236*,
                     236, 236, 266, 399*, 401*, 405, 408*, 408, 413, 414


Rererenced in -- SCANMAIN.SPL.8(02/03/82-18:08:27)

   IDNAME            STRING
                     110, 119, 214, 215*, 232, 233*, 251, 261*, 268, 285*,
                     355, 382, 391, 400, 409, 430, 438, 446, 463, 472, 513,
                     678, 734, 746, 751, 928, 946


Rererenced in -- SCANTOKS.SPL.20(02/03/82-18:05:14)

   IDNAME            STRING
                     423*, 425, 445*, 1144, 1146, 1151, 1166, 1188, 1274,
                     1286, 1289, 1327*, 1338*, 1338, 1348*, 1348, 1376*,
                     1376, 1406*, 1406, 1430, 1434, 1436, 1446*, 1446, 1499,
                     1568, 1771, 1773


Defined in -- SYMTAB2.SPL.3(02/03/82-17:59:36)

   IDNAME               13  STRING[256], ENTRY
                     48, 49, 52, 55, 70, 74
```

Fig. 8. An Example of "SEARCH" Command.

procedure invocations are represented in a call graph. In general, the structured programs consist of many logical components expressed by procedures. It is very powerful to debug and maintain a relatively large size program that the procedure invocations are exactly examined in their relationships. In addition, after all call graphs of program modules compiled separately are collected, the call graph for the executable (absolute) program would be acceptable by merging all graphs into one. Nothing is more accurate in the invocation order than this complete merged call graph. A call graph shown in Figure 6 tells us following facts;

1) This module has two entry procedures CALLGRAPHGEN and GLOBALBIND.

2) Through these two procedures there are two program control paths in the invocation order given by the procedures defined in this module.

3) Every procedure which is directly called from a procedure is listed on higher invocation level than the caller's. For example, GENCALLEDCHAIN, PROCFUNCATTR and CALLGRAPH with level number 1 are directly called by CALLGRAPHGEN with level number 0.

4. Program statistics analysis: The general grogram statistics information such as the number of each statement type, the number of significant tokens, or maximum and an average of nesting levels are printed in a chart.

2.4 Inter-module Information

If the following commands are given by the user, the SIP system provides inter-module information by using LRT from the compiler. Some commands have several subcommands.

1. APPEND: adds current program information created by the SIMPL compiler to the SIP system. If the SIP system has information for the same module, it is replaced by a new one.
2. GET: specifies the module name to which the following commands are to be applied.
3. LIST: outputs statistical information of the module specified by the GET command. Any combination of the following information is available.
   - global variables and/or procedures defined
   - entry variables and/or procedures defined
   - external variables and/or procedures referenced
   - number of source lines, statements, proc/func and etc.

   Above statistics can also be obtained for every module collected in the SIP system instead of the specified one.

   Figure 7 shows a result of executing the command which specifies "list all entry variables declared in each module in the program library." This listing contains for each module in the SIP library; names of modules, names of variables, types of variables and line numbers where the variables are declared.
4. SEARCH: searches for definition and/or reference points of specified identifiers of the module. This function can be extended over all modules.

   Figure 8 is the part of a result of executing "SEARCH" command. In this case, variable "IDNAME" is searched over all modules and the following

```
Interface consistency check for --- PSTAT5.SPL.12(02/08/82-14:09:07)

   ! ! ! Inconsistent with module -- PSTATM.SPL.18(01/27/82-21:33:32)

        GETPAGE2          PROC EXT       (INT, INT)<- - - - - - - - - - - - - - -
                                                                                  |
      procedure name    external procedure    parameter list                      |
                                                                                  |
Interface consistency check for --- PSTATM.SPL.18(01/27/82-21:33:32)              |
                                                                                  |
   ! ! ! Inconsistent with module -- PSTAT5.SPL.12(02/08/82-14:09:07)             |
                                                                                  |
        GETPAGE2            225  PROC ENTRY       (INT, INT ARRAY)<- - - - - - - -
                                                                    |
                  line number   entry procedure    formal parameter list

   ## Undefined Identifiers ##

        AFFECT              PROC EXT <---------- used as external procedure

        S$SIGN              STRING FUNC EXT<--used as external string function

        SHOW                PROC EXT


   ## Unused Identifiers ##

        .DUMMY1               51   INT(97), ENTRY<--- entry integer array with
                                                      97 elements
        .MODDV1               58   INT, ENTRY

        .MODDV2               59   INT, ENTRY
                                    |
                                    |---line number where the variable declared

Interface consistency check for --- PSTATU.SPL.7(02/01/82-21:27:22)

   ## Undefined Identifiers ##

        U$CLFL              INT FUNC EXT

        U$WTFL              INT FUNC EXT
```

Fig. 9. An Example of "CHECK" Command.

results are printed out: a name of the module where the variable is declared and/or referenced, and line numbers referring to the variable. The asterisk following a line number shows an assignment for that variable.

5. CHECK: checks the interface consistency of the module against all other program modules. For each external and entry objects, following items are checked by this command.
   - types of variables
   - procedure or function type (with type of return value for the function)
   - number and types of parameters for procedure/function

   This command provides the error detection capability for module interface which can not be gained by the conventional linkage editor and prevents disastrous results at run time caused from erroneous declarations.

   Figure 9 represents a result of the interface consistency checking over all modules. This example shows that inconsistent parameter lists for the procedure "GETPAGE2" have been specified in the module "PSTATM" which has 2 integer parameters and "PSTAT5" which has the integer and integer array parameters. It also shows undefined (unused) identifiers which are used (defined) in one module but not declared (used) anywhere in other modules.
6. AFFECT: points out all module names coming under the influence of a modification to specific variables and/or procedures in other module. This function takes advantage of the invocation analysis and is especially useful for checking the affect of modification to be made in the maintenance process. In addition, an analysis on the global data bindings for all modules can be acomplished by applying a data flow analysis based on the merged call graph.

3. Design and Implementation

3.1 A View of Our Design

In the bootstrapping process[2] for the compiler enhancement, several useful facilities to support programming activities were thought to be added to compiler itself, if they can be developed with a little modifications to the compiler. The straightforward approach to this problem is to obtain information on the definition and reference of variables reflected with their line numbers in the source program.

Characterizing the overall design of this system, there are a minimal number of the modifications in order to create a skeletal information. The design policies in the modifications are as follows:

1. It is programmed so that the necessary information should be collected only when the compile option is specified.
2. Every information on data items should be packed in a relatively small table, and should reflect all accesses to variables with the line numbers where they appeared.
3. Such a table including the symbol table should be easily separated independently from the compiler.
4. All SIP functions work with the program information library file which

contains the set of tables created by the compiler mentioned above. The SIMPL compiler and the analyzer should be loosely connected only by a temporary file containing such tables.

3.2 Line Reference Table

We define a line reference table (LRT) that contains the source line numbers with all definitions and references of variables. This LRT is designed to give skeletal data for the debugging and analyzing facilities. The data items registered in LRT are identifiers (e.g. variable or procedure and function names) and the intrinsic procedures used in a program. Every item includes the line number in a source program where it appears and the indicator whether its value is defined or referenced to. In addition, every procedure binds its scope in LRT.

3.3 Implementation

(1) Inter-procedural analysis

The compiler at the first stage have already had a variety of testing, debugging and program analysis facilities. These are 1) attributes and cross reference listing, 2) traces available for line numbers, calls and returns, and variable values, 3) subscript and case range checking, 4) statistics at compile time. In generating a cross reference listing, especially, the line references have been managed in an LRT-like table. Therefore, we reconstructed this table as the LRT in order to facilitate a generation of cross reference listing and a call graph, and a computation of global bindings.

Call graph: The scanning for procedure names are done inter-procedurally only in the LRT, and the invocation relationships are represented in a directed-graph. A node involves both procedure name and referenced line number. While the graph listing can be provided for every compiled module, this graph is also a subgraph of whole call graph for all separately compiled modules.

Global bindings: The items in the LRT are also scanned inter-procedurally with respect to global variables. Definitions and references of variables are listed with line numbers. The globals include both internal and external variables. As the local ones, a reference preceding to an assignment of value can be checked if the flow of control is accurately analyzed through the call graph.

Module interface: As for a program consisting of several modules, the internal LRT and call graph for one module are written on the library file together with the symbol table. This interface facility automatically generates a call graph, and rearranges the LRT into another LRT which disables an access to local variables and represents the global or external ones in a linear list with lexicographical order.

(2) Inter-module analysis

The SIP system consists of many separate functions[4]. Some of them correspond to SIP commands on the top level. After a certain function is invoked, control goes into the subcommand mode of the function and the execution continues.

For the purpose of the inter-communication of these functions, the system work area (SWA) is prepared in SIP. The SWA contains LRT, inter-module call

graph and related information only for one module at a time. Inter-module analysis is done using SWA. Some analysis which requires inter-module information should get information of each module from the library file into SWA and create internal table over modules successively.

Table 1 shows the current program size of the SIP system created by the SIP system itself. Currently, as the program size of each SIP function is relatively small and DEC-20 has 256 kw (36 bits/w) virtual address space, SIP function programs are linked together in the same address space. In the case when the number of functions is increased or each function becomes more complex and larger, segmentation or creation of another address space for each function is required.

Table 1. Program Module Statistics for —PSTAT—

Module Name	Ext	Cycl.	Lines	Stmt	PROC/ FUNC	Ent Var	Ent P/F	Ext Var	Ext P/F
1. PSTAT1	SPL	22	325	113	4/0	0	2	21	5
2. PSTAT2	SPL	49	993	403	12/4	0	7	24	7
3. PSTAT3	SPL	4	203	32	1/0	3	1	2	4
4. PSTAT4	SPL	11	251	90	2/0	0	1	12	10
5. PSTAT5	SPL	7	396	138	5/2	0	1	7	8
6. PSTATM	SPL	17	226	54	5/0	30	3	3	12
7. PSTATU	SPL	7	133	36	6/0	0	6	3	3
Total:			2527	866	35/6	33	21	72	49

4. Discussion

The SIP system has been in use by our collegues and ourselves only a few weeks. However, we conclude that this kind of system is useful for SIMPL or other procedure-oriented program construction.

During the development of the SIP system, we have been able to detect some errors by using facilities providing inter-procedural information. The traditional compiler or linkage editor can not support incremental programming. On the other hand, the programmer can use the SIP system interactively during the incremental programming process.

This system is written in the SIMPL language; therefore the user can modify the SIP system with a little efforts. Consequently, this system is flexible in its user's modification.

In contrast to other independent programming support tools, the SIP system effectively uses information obtained by the SIMPL compiler.

Besides the internal form and size of the library file, trade-off between response time for query and memory space plays an important role. Under the time sharing environment, response time have an influence on the system load. Process time for each command should be minimized. For instance, definition and reference list for identifier of each module can not be affected when other modules are modified. On the other, a call graph over modules should be reorganized each time

when invocation of external procedure in a certain module is changed.

In the latter case, when speed is important, all analyses should be done at each time of modification of modules prior to the next query. In contrast, as it takes time and space, creating call graph at each query is reasonable when such kind of query is in rare.

Only mutable information used by current commands is an inter-module call graph. Because of the low frequency of query using a call graph, we decided that it is recreated on each time of receiving the query. In future, it is preferable that a user can specify the trade-off between time and space for each command. In such a case, system generates internal information required for the commands prior to the query which is classified as time critical by the user.

5. Conclusion

We have constructed the support system for writing SIMPL programs which consist of a number of modules. However, our approach can be applied to other procedure-oriented languages and their processors with separate compilation facilities.

Finally, the SIP system supports an aspect of human programming activities, and there are tools which also support other aspects. Thus, for the construction of comfortable programming environment, the compiler, the SIP system, a structure editor, an automatic verifier and a dynamic program analyzer should be effectively combined.

REFERENCES

[1] BASILI, V.R. and TURNER, A.J., "Interactive Enhancement: A Practical Technique for Software Development," IEEE Trans. on Software Engineering, Vol. SE-1, No. 1 (Dec. 1975), 390-396.

[2] BASILI, V.R. and PERRY, Jr. J.G., "Transporting Up: A Case Study," The Journal of Systems and Software 1, Elsevier North Holland, Inc., 1980, 123-129.

[3] MEDINA-MORA, R. and FEILER, P.H., "An Incremental Programming Environment," IEEE Trans. on Software Engineering, Vol. SE-7, No. 5 (Sept. 1981), 472-481.

[4] OSTERWEIL, L.J., "Draft TOOLPACK Architectural Design," Dept. of Computer Science, Univ. of Colorade at Boulder, 1981.

[5] SHIKANO Y., et al., "System Implementation Language: SIMPL on DEC System-20," Proc. of DECUS Japan, Vol. 1, No. 1 (Oct. 1980), 1-27.