JPL PUBLICATION 77-4

Parallel Compilation: A Design and Its Application to SIMULA 67

National Aeronautics and Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91103

JPL PUBLICATION 77-4

Parallel Compilation: A Design and Its Application to SIMULA 67

Richard L. Schwartz

February 1, 1977

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91103

PREFACE

The work described in this report was performed by the Telecommunications Science and Engineering Division of the Jet Propulsion Laboratory.

ACKNOWLEDGMENT

I would like to thank Dr. Daniel M. Berry of the University of California for his help in improving the original version of this paper.

Contents

PARALLEL COMPILATION: A DESIGN AND ITS APPLICATION TO SIMULA 67

Abstract

This paper presents a design for a parallel compilation facility for the SIMULA 67 programming language. The proposed facility allows top-down, bottom-up, or parallel development and integration of program modules. An evaluation of the proposal and a discussion of its applicability to other languages are then given.

1. INTRODUCTION

Since the early days of FORTRAN, the need to segment large computer programs has been recognized. Whenever large programs are developed, it is necessary to have some means for considering only small segments of a program at one time, whether by top-down programming, bottom-up programming or some other method of problem confinement.

It has been shown to be beneficial to segment the program into "modules" containing segments of the program with high intraconnectivity and low interconnectivity ([SMC, Mey]). These segments can then be compiled and tested separately. Conceptually, each module can be thought of as a separate program, which, given certain input, performs a certain task.

As understanding in the field of computer language design progressed, the need for providing a reliable interface between communicating modules was recognized ([Den,LuE] and others), and attempts have since been made to provide a computer verified module interface.

Various languages have implemented schemes for allowing separate compilation. Each scheme has attempted to provide some means of secure communication between modules.

The notions of parallel and serial separate compilation are introduced in this paper to further distinguish between methods of separate compilation. A discussion of the separate compilation facilities found in FORTRAN, PL/1, ALGOL 68C, and DEC-10 SIMULA 67 explores the problems with present approaches.

A design for a new parallel compilation facility for the SIMULA 67 programming language is then presented as an illustration of how the facility can be incorporated into existing languages. SIMULA was chosen for illustration because of its wide range of module definition and communication concepts. An evaluation of the proposal, a discussion of its

applicability to other languages and some general remarks about programming environments conclude the paper.

This paper assumes some knowledge of the design and implementation of the general class of procedure-oriented algorithmic languages. Some specific knowledge of SIMULA 67 is also helpful.

2. A DEFINITION OF PARALLEL AND SERIAL COMPILATION

For the purpose of this paper, the notion of separate compilation has been further classified into the notions of serial and parallel compilation, with the following definitions:

Parallel Compilation:

> The ability to compile program modules in any order, or in parallel, with the module interface not being resolved at compile-time. That is, no knowledge of the other program modules need be present at compile time.

Serial Compilation:

> The ability to compile program modules separately in a particular partial ordering which allows the resolution of the module interfaces at compile-time. That is, knowledge of other program modules may be required for compilation of

an individual module.

3. A SURVEY OF PRESENT COMPILATION MECHANISMS

This section presents a short chronological survey of the methods for separate compilation currently used in FORTRAN, PL/1,ALGOL 68C, and DECsystem-10 SIMULA 67.

3.1 Parallel Compilation in FORTRAN

Program modularization is achieved in ANS FORTRAN ([FOR]) through the use of external subroutines. The program is structured by dividing it into a number of separately compiled subroutines. These subroutine modules communicate by means of formal subroutine parameters, and through COMMON data. All parameters must be listed in the the subroutine heading, and may be explicitly declared. The declarations are used only to determine the size and displacement for the formal parameters in order to compile code to access the parameters.

In FORTRAN there need not be an explicit declaration of an external subroutine. Any call to a subroutine for which no subroutine body can be found is assumed to be a reference to an external subroutine. In the case where the name of an external subroutine is passed as a parameter, there must be an explicit EXTERNAL declaration.

Non-parameter data are transmitted to the external subroutines through the use of the COMMON declaration. This declares the usage of a block of data, labeled or unlabeled, which may be referenced by other program modules. Each module using the COMMON data must contain a COMMON declaration. FORTRAN rules state that there must be identity in type for all entities defined in the corresponding storage position from the beginning of the COMMON block.

The following program illustrates one method of communication between separately compiled program segments. The dotted lines delimit a separately compiled segment.

```
------------------------------------
        C     MAIN PROGRAM
        C     THIS PROGRAM OPERATES ON A FILE OF INTEGERS
              INTEGER IN,OUT,FILE(100),POINTR
              COMMON/FILE/FILE(100),POINTR/
              PTR=1
              DO 20 I=1,100
                READ(5,10)IN
        20    CALL ADDFIL(IN)
        10    FORMAT(I4)
              END
------------------------------------
              SUBROUTINE ADDFIL(ELEM)
        C     THIS MODULE ADDS AN INTEGER TO THE FILE
              COMMON/FILE/FILE(100),PTR/
              INTEGER FILE(100),PTR,ELEM
              FILE(PTR)=ELEM
              PTR=PTR+1
              RETURN
              END
------------------------------
```

The ANS FORTRAN definition requires that the type and order of the parameters in a subroutine call exactly match that in the subroutine declaration, and that the declarations for the the

corresponding COMMON storage positions in each module be consistent.

Since a parallel compilation mechanism is used, there can be no checking of the module interface at compile-time. In order to check the module interface it would therefore be necessary to employ a type-checking linkage-editor, or to use a pre-linkage-editor to do the type checking. Unfortunately, to the author's knowledge, the linkage-editors used to bring together FORTRAN modules do not have a type-checking capability, and there is no means for checking the module interfaces. Most linkage-editors deal only with making the addresses of defining occurrences known to each applied occurrence.

Thus, FORTRAN has a primitive but effective method of program segmentation and parallel compilation. Each segment of the program can be developed separately, and later brought together by the linkage-editor. With all implementations known to the author, there is no module interconnection verification or type checking performed, although programs with erroneous interfaces are not included in the language.

These separate compilation decisions appear to be consistent with the basic philosophy of FORTRAN and the usual implementation of the language.

3.2 Parallel Compilation in PL/1

The parallel compilation mechanism in PL/1 ([IBM]) is essentially the same as that used in FORTRAN. A module in PL/1 is a MAIN or an external procedure. A procedure must contain an EXTERNAL ENTRY declaration for each external procedure it uses. This EXTERNAL declaration contains the attributes of the procedure (i.e., information from the procedure heading). This declaration is used for checking the types of the parameters and the returned value of calls to the external procedure, and for generating code for these calls.

The following program illustrates a multi-module program.

```
------------------------------
        MAIN:PROCEDURE OPTIONS(MAIN);
             /* THIS PROGRAM READS PAIRS OF INTEGERS X,Y
                FROM THE INPUT STREAM AND OUTPUTS X MOD Y */
             DCL MODULO EXTERNAL ENTRY(BIN FIXED,BIN FIXED)
                                 RETURNS(BIN FIXED);
             DCL (X,Y) BIN FIXED;
             ON ENDFILE(SYSIN)STOP;
             DO WHILE('1'B);
               GET LIST(X,Y);
               PUT LIST(MODULO(X,Y));
               END;
             END MAIN;
------------------------------
        MODULO:PROCEDURE(X,Y)RETURNS(BIN FIXED);
               DCL (X,Y)BIN FIXED;
               DO WHILE(X>Y);
                 X=X-Y;
                 END;
               RETURN(X);
               END MODULO;
------------------------------
```

PL/1 requires, and to the author's knowledge, never gets, type checking of the module interface. While the language specification states that the ENTRY declaration in the main

module must agree with the procedure declaration in the external module, there is no verification of this condition. Thus, if the number and/or type of parameters of the module interface do not agree, the result will be undefined. Again as in FORTRAN, the use of a parallel compilation facility requires post-compilation interface checking. Thus, in PL/1 it is possible to have parallel compilation, but with an implicit warning of caveat programmus. Unfortunately, this lack of adequate interface error detection is consistent with the overall lack of error checking in the widely available implementations of the language (see [MoW] for a PL/1 subset with some nice error checking).

3.3 Serial Compilation in ALGOL 68

While ALGOL 68 ([vWi]) has not yet adopted an official modules facility at the time of writing this paper, various modules facilities have been proposed and implemented.

ALGOL 68C ([BBW,KTU]), the ALGOL 68 compiler, developed at Cambridge University in England, contains an ENVIRON mechanism for serial compilation, allowing a module to be compiled in a specified external environment. It is the ALGOL 68C facility which will be briefly described (see [Cle] for a more complete description).

In order to develop large programs in small segments, a system of partial compilation is used. This allows the development of program segments during separate compilations.

A module consists of module text and an environment, called an environ, in which the compilation takes place.

A module is invoked by the use of an ENVIRON statement. The ENVIRON statement is used to declare the block of code which is to be separately compiled. The block must be in what ALGOL 68 calls a "strong position" and be "voided". This statement causes all declarations visible at that point to be made available to the invoked module (to be compiled later) in the form of an environ table.

Each module contains a USING statement which specifies the environment in which the module should be compiled. For the main module, the standard environment containing all standard declarations is specified. For a submodule, the environment specified is that which surrounded the point of invocation (by the corresponding ENVIRON statement). In the implementation, this means that the file containing the environment information generated by the corresponding ENVIRON statement is read in at compile time, prior to parsing. All the declarations visible at the point of the ENVIRON statement are now visible to the module. Thus, it is as if the invoked module were compiled in the program at the point of the invoking ENVIRON statement.

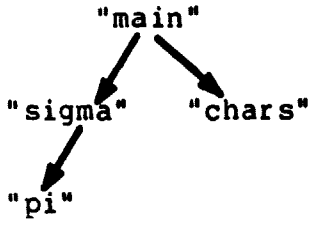
The following program should clarify what has been said.

```
-------------------------------
        main
        USING MACHINE FROM "STANDARD" # std env#
        BEGIN
          BOOL fill;
          CHAR y;
          INT x:=5;
          INT result;
          ENVIRON CHARS;
          ENVIRON SIGMA;
          print(result,x);
          print(y)
        END main
-------------------------------
        sigma
        USING SIGMA FROM "main" #atr file from "main" #
        BEGIN
          INT i;
          result:=0;
          FOR i:=1 TO x
            DO
              result:=result+i

            OD;
          ENVIRON PI
        END sigma
-------------------------------
        pi
        USING PI FROM "sigma" # atr file "sigma"#
        BEGIN
          INT t:=result;
          result:=0;
          FOR i:=2 TO t
            DO
              result:=result*i
            OD
          x:=0;
          y:="z"
        END pi
-------------------------------
        chars
        USING CHARS FROM "main" # atr file from "main" #
        BEGIN
          y:="a"
        END
-------------------------------
```

The above program is comprised of the four modules "main", "sigma", "pi", and "chars". The accessing relationship between the modules, given by the ENVIRON and USING statements, is

illustrated by the following graph, where a⟶b means module a accesses the external module b.



This interdependence has imposed a partial ordering on the compilation sequence of the four modules. The module "main" must be compiled before the modules "sigma" and "chars", while the module "pi" must be compiled after the module "sigma". The four modules must be serially compiled in any order such that:

"main" < "sigma","main" < "chars", and "chars" < "pi"

where a < b means a is compiled before b.

The execution of the above four modules is defined to be as though the following program were run.

```
-----------------------------
        main
        USING MACHINE FROM "STANDARD"
        BEGIN
          BOOL fill;
          CHAR y;
          INT x:=5;
          INT result;
          BEGIN
            y:="a"
          END
          BEGIN
            INT i;
            result:=0;
            FOR i:=1 TO x
              DO
                result:=result+i
              OD
          END
          BEGIN
            INT t:=result;
```

```
        result:=0;
        FOR i:=2 TO t
          DO
            result:=result*i
          OD
        x:=0;
        y:="z"
      END
      print(result,x);
      print(y)
    END main
-------------------------------
```

As D. M. Berry pointed out in his assessment of the ALGOL 68C separate compilation facility ([Ber]), it

1. appears to be a distinct improvement over that of PL/1.

2. supports the top-down programming and testing methodology described by Mills ([Mil]) and by McGowan and Kelly ([McK]), in that:

    1. The top level calling code is written first.

    2. This level can be tested with the use of stubs (null procedures) in place of the not yet present separate procedures.

    3. Each body can then be written (expanded) and tested in the same manner.

The serial compilation in ALGOL 68C <u>is</u> an improvement over the parallel compilation in PL/1, in that the module interface is made both more flexible and more secure. Full type-checking of the module interface is done at compile-time, with the

linkage-editor required to resolve only the beginning address of each module.

The module interface is more secure as a result of the partial ordering of the compilation of program modules, which requires that a submodule is compiled only after all declaring modules have been compiled. This means that full type checking of the interface may occur when the submodule is compiled.

The module interface is made more flexible in that the interface no longer has to take place at the program's global level. The call to a separately compiled procedure does not have to parameterize all the variables necessary for the called procedure. A module, consisting of any block or procedure, is compiled at the same nesting level as where the ENVIRON invocation occurred, and it may access all objects visible at the point of invocation as non-locals. This facilitates dividing the program up into modules with a minimum of interface problems.

Unfortunately, this type of serial compilation has several important disadvantages:

1. Since the submodule cannot be compiled until the declaring environment is known, bottom-up programming is not practical. The partial ordering of the modules dictates that the "bottom" modules must be the last to be compiled. Bottom-up programming can still be done through the use of dummy drivers, but the module being tested must be recompiled when the test

driver is changed, and when the driver is replaced by another module.

2. In order to compile code within the submodule to reference data within the declaring environment, it is necessary for the environment file generated by the declaring module to contain information on how to access the object. The module then uses this information to compile code to refer to the external object. If at a later time, a change in the declaring module's environment causes a change in the location of any data referenced by any submodule, then each affected module must be recompiled. For an implementation using a run-time stack with a display, such as ALGOL 68C, this means that if the i,j pair representing the base and stack offset of a referenced datum is changed by the addition, deletion or modification of any variable with storage earlier in the activation record (see [Weg] for a definition), then all dependent modules must be recompiled. Thus, it is not only changes in the actual module interface that force recompilation of the submodules. Any changes to identifiers stored in an activation record at an offset preceding an identifier which is referenced as a non-local by the submodule, force recompilation of the submodule.

In the ALGOL 68C program previously given, this means that if the declaration for the variable "fill" in module "main" is taken out, then the dependent modules "sigma", "chars", and "pi" must all be recompiled.

3. While the ability to link modules at a non-global level simplifies the segmentation of the program, it can introduce high module interconnectivity. The ability for a module to access any of the variables statically visible at the point of the ENVIRON statement can lead to confusion in both the declaring module and the submodule as to which variables constitute the interface. There is no explicit statement of the module interface as there is in PL/1, where the only reference to non-local variables is through the parameter interface. This implies that a module is not necessarily understandable by itself, but instead it can be understood only after determining which non-local identifiers are used, and the types of the identifiers. There is no type information in the submodule for the non-local identifiers referenced. This information must be obtained by searching through the environment surrounding tne declaring ENVIRON statement, looking for the declarations.

This criticism stems from the author's personal experience working with the ALGOL 68C compiler which itself is written in ALGOL 68C. It is virtually impossible to understand any submodule without considering the invoking module. By the same token, one cannot determine from the module containing the ENVIRON invocation which of its variables will be referenced and possibly modified by the submodules.

The sample ALGOL 68C program given earlier illustrates these problems. It is not possible to tell from considering only the module "main" whether the variable y is used in a

submodule. It may be that the programmer of the module "main" was not aware that the variable y was modified in the module "sigma" as well as in the module "chars". Also, looking at the module "sigma", it requires careful examination to see that the variables x and result are non-local references.

All of these interface problems are caused by the use of a non-explicit module interface scheme. Enough information is present for the compiler to generate correct code, but not for the user to clearly see the module interdependencies.

4. Since the module interface is not made explicit, it is not known during compilation of a module which of its variables and procedures will be referenced by its submodules. Because of this, the environment file produced must include the attributes of all variables visible, even though only a small percentage of these variables will actually be referenced by submodules.

Another, similar but more complex, scheme ([Lin]) has been proposed by Charles Lindsey as an ALGOL 68 standard.

3.4 Serial Compilation in SIMULA 67

The SIMULA 67 Common Base Language Definition ([DMN]) does not include semantics for a separate compilation facility. The definition states that if an implementation permits user-defined procedure and class declarations to be separately compiled, then a program should have means of making reference to such declarations as external to the program. Suggested syntax for

an EXTERNAL statement is given, but without a semantic definition.

In 1971, Jacob Palme, of the Norwegian Computing Center, proposed a system of Part Compilation ([Pal]) similar to that used in FORTRAN, but with full module interface verification. This system has been introduced in the DECsystem-1Ø SIMULA 67 implementation ([BEOP]), and is the one that is described here.

Serial compilation in DEC-1Ø SIMULA is a bottom-up partial ordering of modules, rather than the top-down ordering found in ALGOL 68C.

The declaration of a class or procedure in SIMULA as EXTERNAL indicates the use of a separately compiled module. In any module, e.g., the main program, these declarations can be put anywhere a procedure or class declaration is allowed. The separately compiled modules will then be available anywhere within the scope of the EXTERNAL declaration. According to SIMULA rules, separately compiled prefix classes must be copied into a program in the same block as their subclasses. This is required to prevent dangling reference problems.

To use a separately compiled module "a" inside another separately compiled module "b", the EXTERNAL statement for "a" is placed before the beginning of the separately compiled module "b". The EXTERNAL declaration for module "a" must then precede that of module "b" in each module that uses b. This will be illustrated shortly.

SIMULA 67 has a HIDDEN PROTECTED feature to increase the reliability and security of large programs by controlling the interface between submodules. Attributes of classes which are declared PROTECTED are visible only at a prefix level equal to or inner to the class containing the PROTECTED specification. Attributes declared HIDDEN are invisible at a prefix level outer to the class containing the HIDDEN specification. Thus, attributes declared HIDDEN PROTECTED are visible only inside the body of the class with the HIDDEN PROTECTED specification.

To implement serial compilation, the compilation of each separately compiled class or procedure produces an attribute file containing an entry for each externally accessible attribute of a class module, or each parameter of a procedure module. This entry lists the identifier and its type. When an EXTERNAL statement or declaration is encountered, the attribute file for that module is read in by the compiler. Full type checking is then performed.

As a consequence, a serial bottom-up compilation sequence must be performed. Each separately compiled class and procedure must be compiled before being referenced. Although the DEC-10 SIMULA handbook does not specify this, one ramification of this seems to be that a separately compiled procedure can communicate only through its formal parmeters since the environment of the EXTERNAL statement, within each, is established later. This also means that no GOTOs to external labels are allowed.

There is an additional requirement that when a class is separately compiled, the block level of the place where it is copied into the main program must be given as a parameter to the compiler.

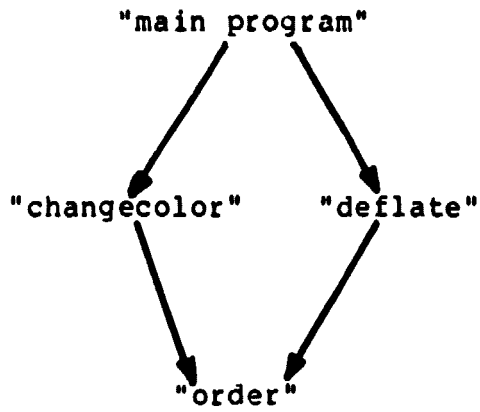
The following program illustrates the module communication:

```
------------------------------
        CLASS order(account,color,quantity);
          TEXT color;
          INTEGER quantity,account;;
------------------------------
        EXTERNAL CLASS order;
        PROCEDURE changecolor(object);
          REF(order)object;
          BEGIN
            IF object.color = "green"
                  THEN object.color:="blue";
          END;
------------------------------
        EXTERNAL CLASS order;
        PROCEDURE deflate(object);
          REF(order)object;
          BEGIN
            object.quantity :=object.quantity//2;
          END;
------------------------------
        BEGIN COMMENT main program;
          EXTERNAL CLASS order;
          EXTERNAL PROCEDURE changecolor;
          EXTERNAL PROCEDURE deflate;
          REF(order)get;
          get :- NEW order(411,"green",2);

              .
              .
              .
          changecolor(get);
          deflate(get);
              .
              .
              .
        END;
------------------------------
```

The above program comprises the four modules consisting of: the CLASS order, the PROCEDUREs changecolor and deflate, and the main program. The accessing relationship

between the modules is illustrated by the following graph.



This interdependence causes a partial ordering of the serial compilation sequence, namely

"order" < "changecolor", "order" < "deflate",

"changecolor" < "main", and "deflate" < "main".

Thus, the serial compilation mechanism in SIMULA is an adaptation of the parallel compilation mechanism in FORTRAN. By serially compiling the modules, full interface verification can take place at compile-time, so that the linkage-editor need only resolve the entry address of each module.

The scheme allows the security of declaring a procedure or class at a non-global level, without the advantage of non-local external reference found in ALGOL 68C. This gives a well-defined interface, not present in ALGOL 68C, but forces more parameters to be passed to external procedures.

Just as separate compilation in ALGOL 68C was designed for mainly top-down integration, separate compilation in SIMULA is designed mainly for bottom-up integration. Top-down integration is possible, but only with numerous recompilations of the program modules during the testing stages as the stubs are replaced by actual modules.

The attribute file is not as susceptible to module changes as the ALGOL 68 environment file. In SIMULA, it would seem that the attribute file has to contain only the externally accessible attributes of a class, or the parameter information for a procedure. This would imply that no internal change to an external class or procedure should change the attribute file. Only actual interface changes should force recompilation. The DEC-10 SIMULA compiler evidently has made some decision to alter this, since the handbook states only that "in most cases no other module need be recompiled". It may be that the temporary locations for expression evaluation have been mixed into the activation record. If so, it seems to be a design error.

4. PHILOSOPHY OF DESIGN FOR PARALLEL COMPILATION

As a result of considering the separate compilation facilities present in existing languages, a basic design philosophy has been formulated regarding what the characteristics of a separate compilation facility should be. This philosophy, briefly stated, is that:

1. Modules compiled and/or developed separately should have only explicitly stated interfaces.

2. Each separately compiled module should be understandable by itself, without reference to other modules.

3. The recompilation of one module should not force the recompilation of any other module unless a change in the actual module interface is made.

4. Complete type checking of the interface should be done.

5. Bottom-up and top-down programming should both be accommodated without undue overhead.

6. The module interface should not be unnecessarily restrictive.

7. The overhead associated with providing enough information for type checking and non-local reference should be low.

5. THE PROPOSAL

A proposal which has been designed using the philosophy outlined in Section 4 is now given for the design of a parallel compilation mechanism for the SIMULA 67 language. This section presents the proposal, and illustrates its usage. Sections 6 and 7 will then discuss the merits of the proposal, and how it

can be applied to other languages.

A note about word semantics: the word "object" is used to mean a variable, procedure, or class, rather than the SIMULA meaning ascribed to it. The word "pragmate" is defined to mean those attributes (in the PL/1 sense, not the SIMULA sense) which define the implementation of the object, thus leaving "attribute" for its SIMULA meaning.

5.1 A Module Definition

A module is a separately compiled entity of the form

MODULE <module identifier> MAIN <module body>
MODULE <module identifier> [<accessions>] <module body>

where <module body> is a main program, an external procedure declaration, an external class declaration, or an external statement (including a block).

The <module identifier> must be unique for the entire program (throughout all the modules), and need not be distinct from the normal program identifiers (because it is always possible to distinguish them syntactically).

A program consists of one MAIN module and a series of submodules containing external class and procedure declarations and external statements. The modules may be compiled in any order or in parallel. The meaning of <accessions> will be

described in Section 5.2 .

5.2 Module Communication

In a given module, any procedure declaration, class declaration or statement which is to be compiled as a separate module is replaced by a stub statement. Each stub statement is of the form

STUB <stub identifier> <stub interface>

Each <stub identifier> must correspond to a <module identifier> which identifies the segment of code to replace the STUB statement.

The STUB statement declares the presence of an external segment of code which is to be logically considered as being compiled at that point in the program (subject to interface restrictions). This functions in the same manner as the ENVIRON statement in ALGOL 68C, or the EXTERNAL declaration in DEC-10 SIMULA 67. There may be more than one STUB statement naming the same external module, as long as each appears in an environment providing the required interface (described shortly).

There are three kinds of STUB statements:

1. a procedure STUB

2. a class STUB

3. a statement STUB

The form of the statement depends on the nature of the construct the STUB statement replaces.

The <stub interface> in each STUB statement describes the interface that the declaring module assumes is present with the STUBbed module.

For a procedure or class STUB, the <stub interface> specifies the objects, if any, which are released for use by the STUBbed module, and the assumed pragmates of the STUBbed module, which may be used within the declaring module.

The RELEASE clause of the <stub interface> specifies the variables, procedures, and classes visible at the point of the STUB statement which may be used by the STUBbed module. In the case of a class, it is possible to RELEASE either the entire class, or only individual attributes of the class. In this way, it is possible to protect certain classes, procedures, or variables from being used in the submodule.

The ASSUME clause of the <stub interface> specifies the assumed externally accessible attributes of the STUBbed module. For a procedure submodule, this is the heading of the procedure. From this, the parameter and return value types may be deduced.

For a class submodule, the ASSUME clause gives the class heading together with a BEGIN-END enclosed sequence of variable declarations and procedure headings which may be referenced from outside the class body (according to SIMULA rules). Together, these constitute the external attributes of the class.

In the case when a statement (or BEGIN block) STUB is used, only a RELEASE clause is included in the <stub interface>, since SIMULA rules imply that there can be no new objects declared in the STUBbed statement which will be visible in the declaring environment.

In the <accessions> clause of the module specification, the STUBbed module must declare all non-local objects referenced. These objects, consisting of the non-local variables, procedures, and classes referenced (including the use of prefix classes) must be a subset of the objects RELEASEd by the corresponding STUB statement in the declaring module (except for system procedures and classes). For ACCESSed classes, it is necessary to include in the declaration only those attributes which will actually be accessed by the submodule (i.e., class attributes which will not be used by the submodule need not appear within the class heading in the ACCESS declaration).

The use of GOTOs to labels outside a module has not been included in this proposal since it violates the design philosophy by forcing a high degree of module inter-connectivity and decreases understandability of individual modules.

To illustrate what has been said, an example of each kind of STUB and corresponding module replacement is now given:

1) Procedure STUB and module replacement

```
------------------------------
        MODULE x MAIN
        BEGIN
          INTEGER flag1,flag2;
          TEXT options;
          CLASS tree(val,lson,rson);
            INTEGER val;
            REF(tree)lson,rson;;
          CLASS prog(input);
            TEXT input;;
            .
            .
            .
          STUB parse[RELEASE flag1,flag2,options,tree,prog;
             ASSUME
               BOOLEAN PROCEDURE parse(source,output);
                         REF(tree)output,REF(prog)source ]
            .
            .
            .
        END;
------------------------------
        MODULE parse
        [ACCESS INTEGER flag1,flag2;
                TEXT options;
                CLASS tree(val,lson,rson);
                  INTEGER val;
                  REF(tree)lson,rson;;
                CLASS prog(input)
                  TEXT input;; ]
        BOOLEAN PROCEDURE parse(source,output);
            REF(tree)output,REF(prog)source;
            BEGIN
              IF flag1=0 ∧ flag2=0 THEN scan(options);
              .
              .
              .
            END;
------------------------------
```

2) Class STUB and Module Replacement

```
MODULE y MAIN
BEGIN
  INTEGER x,y,z,w;
  CLASS prefix(row);
    INTEGER row;
    BEGIN
      REAL PROCEDURE width;
        BEGIN
          .
          .
          .
        END;
      BOOLEAN PROCEDURE sturdy;
        BEGIN
          .
          .
          .
        END
    END
  STUB   classa [ RELEASE x,y,z,prefix;
                  ASSUME prefix CLASS board(col);
                           INTEGER col;
                           BEGIN
                           REAL PROCEDURE leng
                           END ]
  REF(board)tray;
  tray:-NEW board(5,6);
  IF ¬tray.sturdy THEN tray.row:=tray.row-1
END
```

```
MODULE classa
[ ACCESS INTEGER x,y;
         CLASS prefix(row);
           INTEGER row;
           BEGIN
             REAL PROCEDURE width;
           END ]
  prefix CLASS board(col);
    INTEGER col;
    BEGIN
      REAL PROCEDURE leng;
        BEGIN
          .
          .
          .
        END
    END
```

3) Statement STUB and Module Replacement:

```
-----------------------------
        MODULE z MAIN
          BEGIN
          INTEGER x,z,y;
          CLASS classb;;
            .
            .
            .
          STUB blocka [ RELEASE x,z,classb ];
            .
            .
            .
          END;
-------------------------------
        MODULE blocka
        [ ACCESS INTEGER x,z;
                 CLASS classb;;]
        classb BEGIN
            .
            .
            .
        END;
-------------------------------
```

More formally, the syntax for each MODULE and STUB statement is given below. The productions are proposed as an extension to the syntactic description given in the SIMULA Common Base Language Definition. Syntactic classes referred to but not defined in this paper refer to syntactic definitions given in [DMN] and [Nau].

```
<module>::= MODULE <module identifier> MAIN
            <module body> | MODULE <module identifier>
            [ <accessions> ] <module body>

<module identifier>::= <identifier>

<module body>::= <procedure declaration>|
                 <class declaration>|
                 <statement>

<accessions>::= ACCESS <external accessions declaration list>

<external accessions declaration list>::=
     <accession declaration>|<accession declaration> ;
     <external accessions declaration list>
```

```
<accession declaration>::=
     PROCEDURE <procedure heading> |
     <class attribute heading> |
     <type declaration> | <array declaration>

<class attribute heading>::=
     <prefix option> CLASS <class identifier>
     <formal paramter part> ; <value part> <specification part>
     <virtual part> <local attributes option>

<prefix option>::= <prefix> | <empty>

<local attributes option>::= <empty> | BEGIN <local
     attributes list> END

<local attributes list>::= <local attribute> | <local attribute>
     ; <local attributes list>

<local attribute>::= <type declaration> |
     <array declaration> | PROCEDURE <procedure heading>

<stub statement>::= <procedure stub statement> |
     <class stub statement> | <statement stub statement>

<procedure stub statement>::= STUB <module identifier>
     [ <release declaration> ; <assumed procedure
     declaration heading> ]

<class stub statement>::= STUB <module identifier>
     [ <release declaration> ; <assumed class attribute
     heading> ]

<statement stub statement>::= STUB <module identifier>
     [ <release declaration> ]

<release declaration>::= RELEASE <visible list>

<assumed procedure attribute heading>::=
     ASSUME PROCEDURE <procedure heading>

<assumed class attribute heading>::=
     ASSUME <class attribute heading>

<visible list>::= <visible id> |
                  <visible id> <visible list>

<visible id>::= <variable identifier 1> |
     <class identifier> | <procedure identifier 1>
```

5.3 The Module Compilation Phase

Each module is compiled independently, without knowledge of the other program modules. The compilation takes place in the standard system environment, with all system classes, procedures, and identifiers visible within the module.

During the compilation, all objects which do not have corresponding local declarations are checked for appearance in an ACCESS or an ASSUME declaration. If so, compilation proceeds using the ASSUMEd or ACCESSed declared attributes for the missing external declarations. If the object does not appear in an ACCESS or ASSUME declaration, then it is assumed to be an unresolved external reference, and is reported as an error. A dummy reference (a null (i,j) pair) to the external object is generated in the object code, which will later be filled in by the pre-linkage-edit step (described in Section 5.4).

All variables, procedures, and classes which appear in the RELEASE clause of a STUB statement are checked for visibility at that point in the module. Any object appearing in the RELEASE clause which is not visible at that point constitutes an error.

A pragmate file and an object code file are generated by each compilation. The pragmate file contains:

1. The module name

2. The pragmates for each object which is RELEASEd by one or more STUB statements within the module

3. For each STUB statement, the STUB name, the nesting height within the module of the STUB statement, the name of each object RELEASEd by the STUB, and the ASSUMEd pragmates of the STUBBed module

4. The pragmates of each ACCESSed object

5. The pragmates of the procedure or class declaration if the module being compiled is a class declaration or a procedure declaration.

6. For each ACCESSed and ASSUMEd object, a list of its applied occurrences.

While the exact amount of information that must be present in the pragmate file will depend on the actual implementation, the pragmates for each object should include:

variable pragmates

1. variable name

2. type indicator

3. for RELEASEd variables , the (i,j) pair representing the nesting height within the module of the block containing the declaration, and the offset within the block (for a RELEASEd

variable not local to the module, the offset is not known, and an external tag should be used)

4. a HIDDEN/PROTECTED flag

procedure pragmates

1. procedure name

2. return type indicator

3. the type indicator and transmission type for each formal parameter in the proper order

4. a VIRTUAL/non-VIRTUAL flag for class procedures

5. a HIDDEN/PROTECTED flag

class pragmates

1. class name

2. prefix class name (if any)

3. the type indicator and transmission type for each formal parameter in the proper order

4. the pragmate for each attribute in the proper order:
    a) for a variable attribute, a variable pragmate
    b) for a procedure attribute, a procedure pragmate

5. a HIDDEN/PROTECTED flag

6. the location of any INNER statements

5.4 The Pre-Linkage-Edit Phase

In order to do complete type checking of the interface between modules, and to handle non-local object reference, it is necessary to have a linkage-editor preprocessor. This preprocessor accepts all of the program object modules and their pragmate files as input, verifies that the module interfaces are consistent, and determines the (i,j) pair for the reference to external objects so that it may be handled by a standard system linkage-editor program, as described below.

The ASSUMEd class and procedure attributes in each declaring module are checked against the class and procedure headings in the corresponding external class and procedure submodules. The ASSUMEd attributes must exactly match the actual attributes declared in the submodule, right down to identifier names. The order of procedure and variable declarations need not be the same in the ASSUME clause as in the module declaration.

The ACCESS declarations in each submodule are checked against the corresponding RELEASE clause of the STUB statement in the declaring module. All objects ACCESSed must have been RELEASEd by the declaring STUB. The order of appearance of the objects in the RELEASE declaration need not match the order of appearance in the ACCESS declaration, but the type of each object RELEASEd must match the type of the object ACCESSed. In the case of an ACCESSed class, the attributes declared in the ACCESS declaration need not be the complete set of attributes, but only the subset actually used, assuming it is consistent with the full set of attributes of the class. Thus, only those procedures and variables inside the class which are actually accessed need be declared.

The reference to non-local variables can be resolved without difficulty at this stage, since the pragmate file for each program module is available. The hierarchical structure of the whole program can be determined by the module name and stub identfiers within each pragmate file. With the knowledge of the overall structure of the modules, the declaration for each external reference can be found, and the overall block nesting height and storage offset within the block can then be deduced. This information can be inserted into the instructions within the object code which reference the variable. For an implementation using a run-time nesting height display, this involves substituting the actual display level and offset within the activation record for the dummy level and offset inserted during compilation.

with the knowledge of the class hierarchy, the reference to VIRTUAL procedures and split class bodies can be resolved, and the dummy reference may be replaced by actual code. The actual address determination will be done by the linkage-editor. Note that this implies that the identification of the actual body for a VIRTUAL procedure can vary with the use of different STUBs.

The only addresses left unresolved are the beginning address of each module, the procedure and class entry points, and references to VIRTUAL procedures and split bodies. These can be handled by a standard linkage-editor program.

6. AN ASSESSMENT OF THE PROPOSAL

The proposal outlined in the previous section appears to be an improvement over the current schemes for separate compilation found in the languages surveyed. The scheme given here allows the full power of a verified non-global interface with global object reference found in ALGOL 68C, but with an explicit interface specification. The development and compilation of program modules may proceed in parallel, with no imposed partial ordering of the compilation sequence. This allows bottom-up, top-down, or any other sequence of program development and testing. In addition, the recompilation of a module only forces recompilation of other modules if a change in the actual module interface is made.

By postponing the resolution of the (i,j) pair for each non-local object from the compilation phase to a pre-linkage-edit phase, there is no direct dependence on information obtained during the compilation of the other modules. This allows complete type checking of the variables, procedures and classes being used for the communication between modules, but without the forced top-down or bottom-up testing order imposed in ALGOL 68C and DEC-10 SIMULA. This achieves the less restrictive module interface obtained with the top-down serial compilation in ALGOL 68C, but without the partial ordering of the modules and the resulting high sensitivity of the environment file used to resolve external references. At the same time, it achieves the insulation of the pragmate file from internal changes to a module, found in the DEC-10 SIMULA scheme, but without sacrificing the ability to reference non-local variables, necessitated by the strict bottom-up ordering of SIMULA program modules.

Using this scheme, it is possible to have multiple invocations to a given module. The only restrictions are that a procedure or class submodule may not be invoked twice in the same range, and the module interface must be consistent with all STUB statements. Briefly stated, the use of multiple STUBs to a single submodule must be such that a consistent program is obtained by replacing each STUB statement by the submodule body. The ability to have multiple invocations allows a compile-time-like macro substitution, but with the resolution being done at pre-linkage-edit time. Effectively, this

functions as a macro substitution where the macro need not be known at compile-time.

All assumptions about the outside environment must be explicitly stated within a module. The STUB statement explicitly states which objects are RELEASEd, and may be changed by the submodule. This, coupled with the HIDDEN/PROTECTED attributes allows a well protected and more easily verified interface. The ASSUME clause of the STUB statement and the ACCESS clause of the MODULE heading together give the specification of each external object which is used in the module. This information assures that there are no objects referenced in the module which do not have local type information.

The overhead required to implement the proposed scheme is less than that in ALGOL 68C, and is comparable to that in DEC-10 SIMULA. In ALGOL 68C, the pragmates of all objects visible at the point of the ENVIRON statement must be placed in the environment file, since it is not known during compilation which of the objects will be accessed by submodules. In the proposed scheme, it is known at compile time which objects are RELEASEd for use by the submodule. Only the pragmates of these objects, together with the procedure or class pragmates for a procedure or class module need be included.

The total overhead for program development using the proposed parallel compilation scheme should always be approximately less than or equal to the total overhead using the

ALGOL 68C serial compilation scheme. If a change in an ALGOL 68C module does not cause a change in the environment file, then only that module must be recompiled, and the set of object modules must be linkage-edited again. If the change does cause a change in the environment file, then all submodules must be recompiled, and the set of object modules must be linkage-edited again. Using the parallel compilation scheme, any change not affecting a submodule interface causes only the recompilation of that module, another pre-linkage-edit run and another linkage-edit run. For a change that affects a submodule interface, the affected module and submodule have to be recompiled, and another pre-linkage-edit and linkage-edit step must be run.

As Berry noted ([Ber]), the STUB-replacing module facility supports top-down development and testing of programs, since each STUB statement may be compiled as:

1. an empty construct of its kind, returning the default value of its type

2. a call to the interpreter, which executes intermediate level code for the replacement module

3. calls to the user via the interactive console, for she/he to plug in values of the required type.

7. APPLICATION TO OTHER LANGUAGES

While the proposal for a parallel compilation scheme has been designed for the SIMULA 67 programming language, it can be applied equally well to any other procedure-oriented language. The overhead will vary from language to language, and from implementation to implementation depending on scoping rules, run-time organization, etc.

8. REMARKS

A few philosophical comments appear to be in order. Many people ([Pal, Ber, BBW] and others) have advocated complete compile-time resolution of the module interface through serial compilation. Jacob Palme stated ([Pal]) that using post-compile-time type checking of the module interfaces requires that "the loading (linkage-editing) programs must be modified, which is something you want to avoid since these programs are commonly used system programs".

One solution to this dilemma is to have a special purpose language-specific pre-linkage-editor do the type checking. This avoids making changes to a commonly used system program, and provides useful language-dependent features that a general purpose system program can not provide.

This alludes to a more general concept. We should be developing <u>total programming environments</u>. Instead of developing general purpose text-editors, language-specitic

compilers, general purpose linkage-editors, and general purpose debuggers, we should be developing a programming environment designed for the programming language. The text-editor should be intelligent, with features designed to aid in the coding of programs written in the language. The linkage-editor (collection program) should include facilities for resolving more of the interfaces than just the addresses. There should be a run-time system that includes an intelligent interactive debugger and tester.

In short, we should not develop compilers for programming languages, and then rely on general purpose system processors, which normally handle only the common subset of all language needs, to provide the user support. There's a great deal more we can do to aid in the production of reliable software.

Bibliography

[Ber] Berry, D.M., "Separate Compilation", Internal Memorandum, No. 147, UCLA, Computer Science Department, February 1976.

[BEOP] Birtwistle, G., L. Enderin, M. Ohlin, J. Palme, DECsystem-10 SIMULA Language Handbook Part 1, Report No. C8398, Swedish National Defense Research Institute, March 1976.

[BBW] Bourne, S.R., A. Birrell, I. walker, "ALGOL 68C Reference Manual", University of Cambridge, 1975.

[Cle] Cleveland, J.C., "The ENVIRON and USING Facilities of ALGOL 68C", Modeling and Measurement Note, No. 33, Computer Science Department, UCLA, April 1975.

[DMN] Dahl, O., B. Myhrhaug, K. Nygaard, Common Base Language, Publication No. S-22, Norwegian Computing Center, October

1970.

[Den] Dennis, J., "Modularity", Computation Structures Group Memo, No. 70, Project MAC, MIT, June 1972.

[FOR] Engel, F. (comm), "draft proposed ANS FORTRAN", X3J3/76, SIGPLAN Notices, Vol 11, No. 3, March 1976.

[IBM] IBM System/360 Operating System PL/1(F) Language Reference Manual, Form GC28-8201-3, IBM Corp., White Plains, New York, June 1970.

[KTU] Kearns, M., A. Tanenbaum, R. Uzgalis, UCLA ALGOL 68C Programmer's Guide, Computer Science Department, UCLA, May 1976.

[Lin] Lindsey, C.H., "Proposal for a Modules Facility in ALGOL 68", Algol Bulletin, AB39.4.2, February 1976.

[McK] McGowan, C.L., J.R. Kelly, Top-down Structured Programming Techniques, Petrocelli/Charter, New York, 1975.

[Mey] Meyers, G.L., "Composite Design: The Design of Modular Programs", Technical Report, TR00.2406, IBM, Poughkeepsie, New York, January 1973.

[Mil] Mills, H.D., "Top-down Programming in Large Systems", in Rustin, R. (ed.), Debugging Techniques in Large Systems, Prentice-Hall, Englewood Cliffs, New Jersey, 1971, pp. 41-55.

[MoW] Morgan, H.L., R.A. Wagner, "PL/C: The Design of a High Performance Compiler for PL/I", Research Report 70-83, Computer Science Department, Cornell Univ., October 1970.

[Nau] Naur, P. (ed.), "Revised Report on the Algorithmic Language ALGOL 60", Communications of the ACM, ACM, January 1963.

[Pal] Palme, J., "Part Compilation in High Level Languages", Report No. FOA-P-C-8306-M3(E5), Swedish National Defense Research Institute, Nov 1971.

[SMC] Stevens, W., G. Myers, L. Constantine, "Structured Design", IBM Systems Journal, No. 2, 1974.

[vWi] van Wijngaarden, et al., "Revised Report on the Algorithmic Language ALGOL 68", Technical Report TR 74-3, University of Alberta, March 1974.

[Weg] Wegner, P., Programming Languages, Information Structures, and Machine Organization, McGraw-Hill, New York, 1968.

1970.

[Den] Dennis, J., "Modularity", Computation Structures Group Memo, No. 70, Project MAC, MIT, June 1972.

[FOR] Engel, F. (comm), "draft proposed ANS FORTRAN", X3J3/76, SIGPLAN Notices, Vol 11, No. 3, March 1976.

[IBM] IBM System/360 Operating System PL/1(F) Language Reference Manual, Form GC28-8201-3, IBM Corp., White Plains, New York, June 1970.

[KTU] Kearns, M., A. Tanenbaum, R. Uzgalis, UCLA ALGOL 68C Programmer's Guide, Computer Science Department, UCLA, May 1976.

[Lin] Lindsey, C.H., "Proposal for a Modules Facility in ALGOL 68", Algol Bulletin, AB39.4.2, February 1976.

[McK] McGowan, C.L., J.R. Kelly, Top-down Structured Programming Techniques, Petrocelli/Charter, New York, 1975.

[Mey] Meyers, G.L., "Composite Design: The Design of Modular Programs", Technical Report, TR00.2406, IBM, Poughkeepsie, New York, January 1973.

[Mil] Mills, H.D., "Top-down Programming in Large Systems", in Rustin, R. (ed.), Debugging Techniques in Large Systems, Prentice-Hall, Englewood Cliffs, New Jersey, 1971, pp. 41-55.

[MoW] Morgan, H.L., R.A. Wagner, "PL/C: The Design of a High Performance Compiler for PL/I", Research Report 70-83, Computer Science Department, Cornell Univ., October 1970.

[Nau] Naur, P. (ed.), "Revised Report on the Algorithmic Language ALGOL 60", Communications of the ACM, ACM, January 1963.

[Pal] Palme, J., "Part Compilation in High Level Languages", Report No. FOA-P-C-8306-M3(E5), Swedish National Defense Research Institute, Nov 1971.

[SMC] Stevens, W., G. Myers, L. Constantine, "Structured Design", IBM Systems Journal, No. 2, 1974.

[vWi] van Wijngaarden, et al., "Revised Report on the Algorithmic Language ALGOL 68", Technical Report TR 74-3, University of Alberta, March 1974.

[Weg] Wegner, P., Programming Languages, Information Structures, and Machine Organization, McGraw-Hill, New York, 1968.