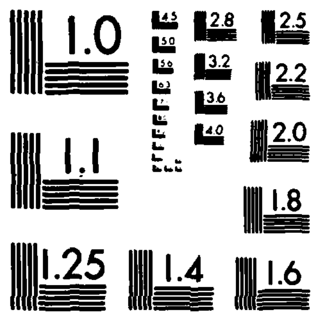
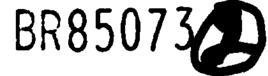
AD-A121 550 MODULAR COMPILATION SYSTEMS FOR HIGH LEVEL PROGRAMMING LANGUAGES(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) I F CURRIE ET AL. 82 RSRE-MEMO-3460 1/1

UNCLASSIFIED DRIC-BR-85073 F/G 9/2 NL

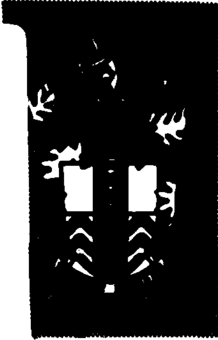


MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNLIMITED



AD A121550



RSRE MEMORANDUM No. 3460

ROYAL SIGNALS & RADAR ESTABLISHMENT

MODULAR COMPILATION SYSTEMS FOR HIGH LEVEL PROGRAMMING LANGUAGES

Authors: I F Currie and
N E Peeling

Supersedes AD-B068220

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

DTIC
ELECTE
NOV 12 1982
S D
E

82 11 12 151

DTIC FILE COPY

RSRE MEMORANDUM No. 3460

UNLIMITED

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3460

Title: MODULAR COMPILATION SYSTEMS FOR HIGH LEVEL PROGRAMMING LANGUAGES

Authors: I F Currie and N E Peeling

SUMMARY

This Memorandum will try to draw some conclusions from the experience gained by the different implementations of modular compilation in Algol 68. This does not mean that it is written only for the Algol 68 community. It is also produced for those working on new high level languages, notably Ada, in the hope that they may avoid some of the problems that have befallen the implementors of Algol 68.

Accession For		
NTIS GRA&I	X	
DTIC TAB	☐	
Unannounced	☐	
Justification		
By		
Distribution/		
Availability Codes		
Dist	Avail and/or Special	
A		

This memorandum is for advance information. It is not necessarily to be regarded as a final or official statement by Procurement Executive, Ministry of Defence

Copyright
C
Controller HMSO London

1982

S80/59

MODULAR COMPILATION SYSTEMS FOR HIGH LEVEL PROGRAMMING LANGUAGES

I F Currie and N E Peeling

CONTENTS

1 Introduction
2 What is modular compilation?
3 What is a natural subdivision of a programming task?
4 Simple procedure modules
5 Modules requiring elaboration
6 Comparison of the different systems
7 The Algol 68-R module system
8 The proposed standard for Algol 68
9 The modular compilation system on the RS Algol 68 compiler
10 The proposed standard for Ada
11 Conclusions
12 Acknowledgments
13 References

1 INTRODUCTION

This paper will try to draw some conclusions from the experience gained by the different implementations of modular compilation in Algol 68. This does not mean that it is written only for the Algol 68 community. It is also produced for those working on new high level languages, notably Ada, in the hope that they may avoid some of the problems that have befallen the implementors of Algol 68.

The proposed standard for Ada [1] seems to indicate that many of the lessons painfully learnt by Algol 68 implementors have not been passed on to the designers of Ada. Although one of the design goals of Ada was that the language should offer "support for separate compilation of program units in a way that provides the same degree of checking as within the unit", the proposed standard describes a system for modular compilation that is more dangerous than any such system that has been implemented in Algol 68.

2 WHAT IS MODULAR COMPILATION?

A modular compilation system allows a large program to be subdivided into a number of smaller modules which can be submitted for compilation separately.

By making the separate modules as self-contained as possible they can be used in the production of more than just the one program. To this end each module has a specification which defines which parts are visible outside itself. This allows the possible interactions between modules to be checked. The specification defines a module to the outside world so that a module can be altered and recompiled without affecting any other modules provided that its specification remains the same. It can be seen that if a

separate compilation system admits a natural subdivision of a programming task it will provide a useful means of dividing a large problem into manageable sized portions as well as minimising the amount of recompilation necessary during development.

3 WHAT IS A NATURAL SUBDIVISION OF A PROGRAMMING TASK?

Programming is often described in terms of the "top down" and the "bottom up" approaches. The top down approach starts with a high level description that breaks the problem down into a number of steps which are only specified in general terms. Each of these steps is then tackled in a similar manner by breaking it down into even smaller steps. At each level more detail is introduced until a complete solution has been generated.

The bottom up approach starts by defining the lowest level of primitives first (for example defining the data structures and basic procedures for manipulating them). These primitives are then used to produce a more powerful set of facilties which can then in turn produce yet more powerful ones until the problem can be easily solved using the facilities that have been built up. The building of a subroutine library is a naturally bottom up activity with each new level producing routines of greater sophistication but less wide ranging applicability.

The top down approach tends to be used to solve a specific problem while the bottom up approach is particularly suited to the provision of a set of utilities that can be used selectively to help solve a wide range of problems.

Modular compilation systems need to provide two different types of module to cater for the top down and the bottom up approaches. All separate compilation systems produced for Algol 68 have catered for the bottom up approach (after all this is the sort of facility offered by most FORTRAN compilers), but only a few have provided a type of module suitable for top down usage. In real life, a problem will tend to be solved by a mixture of the two approaches, so where two different types of module are provided it is usually possible to combine them in a natural manner.

Some systems draw a distinction between modules and compilation units because the visibility rules for modules do not have to be linked to a particular compilation mechanism. For the sake of simplicity we will treat them as one and the same.

We will only be considering bottom up modularisation because it is the more important of the two types.

In its simplest form bottom up programming provides the facility to separately compile some (possibly restricted) piece of program, to which has been added a means of publishing identifiers declared in it for use by other modules or programs ("keep" and "pub" constructions have been used for this purpose). Kept identifiers are made available to other modules or programs if a simple directive is included in their text ("with", "use" and "access" have all been used for this sort of construction).

4 SIMPLE PROCEDURE MODULES

The simplest and safest piece of program allowed as the unit of separate compilation is a single procedure declaration (we will refer to these as simple procedure modules). Simple procedure modules correspond to the units of separate compilation in most FORTRAN systems. Large libraries have been written in FORTRAN so it is reasonable to ask why high level languages such as Algol 68 and Ada need a different type of module. The answer is that if Ada and Algol 68 are satisfied just to copy the facilities that can be provided by FORTRAN libraries then there is no reason why simple procedure modules should not suffice. Writers of Algol 68 and Ada can however make great use of the data structuring provided by such languages and may well wish to declare and initialise data structures and make them available in the library.

A separate compilation system based on simple procedure modules has been implemented in Algol 68 (the CDC system). It provides a little more than this by having a single module (called a prelude) in which objects other than procedures can be declared and initialised. The prelude is automatically obeyed before all programs that use the library. The CDC system does present certain practical problems. If data space is to be created by the library it can only be generated locally in a procedure, or locally in the prelude which is then global to all users of the library, or by using a global generator which will use the heap with its associated overheads. The CDC systems can also give rise to very large preludes for very large libraries. For these reasons the simple procedure modules are unpopular with the producers of large libraries (eg the NAG library).

Procedures in Algol 68 (and their equivalents in Ada) are restricted in that they cannot be produced dynamically (in particular a procedure cannot produce a new meaningful procedure as its result). This restriction is imposed to allow efficient "stack based" implementations of the languages. If this restriction is removed, procedures become increasingly attractive as the basic unit for separate compilation. A module could most naturally be treated as a procedure delivering keeps as its result (hopefully using a nice structuring facility that allows easy access to the different fields). The module's parameters would either be procedures (unevaluated modules) or keeps (evaluated modules). This approach is still not a complete solution because you cannot say "only evaluate this module if it has not been evaluated by some previous module". It is because modular compilation systems are trying to get the effect of dynamically produced procedures, without abandoning "stack based" implementations, that leads to all the complexities that are assumed to be inherent in separate compilation systems.

5 MODULES REQUIRING ELABORATION

Simple procedure modules consist of compiled code that is obeyed whenever the procedure is called. If separately compiled modules do more than just declare procedures, for example declare variables, the module may also contain code that is obeyed before the using program (the obeying of this code is referred to as the elaboration of the module). If more than one such module is being used, it becomes necessary to know if the order of elaboration of the modules is important. It may also be important to know how many times each module is elaborated.

An obvious extension to the simple procedure module is to make the unit of compilation as unrestricted as possible. To this end many systems allow any legal sequence of statements (with some expression of the module's external specification) to be compiled separately. This is the only unit of separate compilation in Algol 68-R [2], it is one of the units in the proposed standard for Algol 68 [3], and it is also one of the units in the proposed standard for Ada.

This is obviously a much more flexible unit than the procedure declaration. Unlike the procedure declaration these modules may require elaboration and a simple example will suffice to show that the order of this elaboration may have to be known if the result is not to be ambiguous.

In our examples modules will be headed by the word module followed by the name of the module. An (optional) use list of module names may be included after the module name; this use list will provide access to all the identifiers published by the modules named in it and will also cause their elaboration if required. The modules may publish identifiers in a keep list at the foot of the text.

```
MODULE aa =
        (INT i := 1)
KEEP i

MODULE bb USE aa =
        (....; i := i+1; ....)
KEEP ....

MODULE cc USE aa =
        ( ....; i := 2; ....)
KEEP ....
```

This example shows that there is an obvious partial order within a library of modules because module aa must be compiled before either bb or cc. It is thus easy to say that aa must be elaborated first, but any program that uses both bb and cc must know the order of their elaboration. For example, if the order of elaboration is aa, bb, cc the variable i refers to 2, but if the order of elaboration had been aa, cc, bb the variable i would have refered to 3.

We have so far assumed that there was only one elaboration of each module so that there was only the one copy of the variable i, which allowed the modules to communicate via the common reference. If any module that used i had its own copy (ie a module is elaborated as many times as it is used) such communication would have been impossible. It should be obvious that the number of copies of each module will affect any decisions made to define the nature of any communication between modules using common references (communication during the elaboration is often referred to as a side-effect). Is such communication a defect or a facility? It is easy to construct examples where you want communication and equally easy to construct examples where you do not (consider a module that defines a procedure that produces elements of a pseudo-random sequence - do you want modules to use the same random sequence, or do they each require their own?).

No one has managed to devise a system that allows the user to choose as many copies as required, which would be the best solution to the problem. If a copy of a module is taken every time it is used, the implementation is liable to become slow and use a lot of space. For this reason most systems take as few copies as is possible given the information known at compile time, usually just a single copy of each module. Because the proposed standard for Algol 68 allows the use of its access clauses within the body of the module text rather than the more usual approach of having a use list at the top of the module, it is not always possible to determine at compile time if a copy of a module already exists; in such cases a separate copy must be taken which can cause the most appalling confusions (a good reason for keeping the use list at the top of the module).

6 COMPARISON OF THE DIFFERENT SYSTEMS

Given that a decision has been made to permit only one copy of each module, what can be done about the possibility of communication between modules using common references.

We will examine the relative advantages and disadvantages of the Algol 68-R system, the proposed standard for Algol 68, the modules system for the RS Algol 68 compiler [4] and the proposed standard for Ada.

The Algol 68-R system is somewhat of an anachronism because it was the production of this system that showed up many of these problems in the first place. It is still worth examining because it has probably been more heavily used than any other system and the problems users have had with it were an important factor in the production of the RS module system.

7 THE ALGOL 68-R MODULE SYSTEM

Algol 68-R modules can only be used if they are incorporated in a library (called an album), and they are date stamped when they are put into the library. The date stamping gives a total ordering within the library. When a program is run that uses modules from the library, the total set of modules required is obtained and they are elaborated once only in order of their date stamps (oldest first). It is obvious that if this system is used to build up a library from scratch it will, of necessity, obey the partial order. If however a module is changed we must decide if any modules that use it must be recompiled. We have already said that a module can be changed without affecting any other modules provided that its external specification remains unaltered. We will now consider what constitutes the specification of an Algol 68-R module. To allow the complete interface checking that will be necessary to implement a safe system it follows that if the contents of the keep list are changed (or the modes of the elements in it) then any module that uses any of the identifiers that have changed must be recompiled. For efficiency reasons it is usual to specify that any change in the keep list of a module will require the recompilation of all modules that use it. This means that an exact description of the keep list is part of the external specification of a module. Regrettably this is not sufficient for the external specification in Algol 68-R; the use list must also be included. The necessity of including the use list will be demonstrated by an example.

Consider compiling the following modules in the given order:

```
MODULE aaa = ....

MODULE bbb USE aaa = ....

MODULE ccc USE bbb = ....
```

If we recompile aaa and change its keep list we change its specification. The old version of module aaa has to be removed from the library before being replaced by the new. Module aaa now has a new date stamp. We have to recompile bbb because it uses aaa but can we amend it (amending means that the specification of the new module is the same as the old version so that the code referred to by the module bbb can be replaced by the new code and no module that uses bbb will have to be recompiled)? The answer is no because the partial order tells us that we must elaborate aaa then bbb, while unfortunately the date stamping tells us to elaborate bbb before aaa, which is nonsense, and so we have to remove bbb from the library and then put it in again with a new date stamp. This means that the use list of bbb complete with date stamps has formed part of the external specification. The module ccc must also be recompiled because bbb has a more recent date stamp. The result of adding the use list to the specification means that all modules that use aaa must be recompiled plus all the modules that use the recompiled modules no matter how indirectly. It will come as no surprise to know that the users of the Algol 68-R system have to rebuild their libraries quite frequently.

The Algol 68-R system imposes an external total ordering on the modules and all the recompilation problems arise because the total ordering can at times seriously contradict the necessary *partial order*. If a total ordering could be found that did not allow these gross anomalies with the partial order the system could probably allow a more liberal regime for amending a module.

The external total ordering does have some advantages over the proposed standard for Algol 68 which uses the syntax of the use list to give the total ordering. We will see that in the proposed standard for Algol 68 any changes in the order of the use list can alter the total ordering while in Algol 68-R this does not happen. This does not mean that the external total ordering totally freezes the side-effects as we will now demonstrate.

Consider a module

```
MODULE aaaa =
          (INT i := 1)
KEEP i
```

Imagine one person compiles a module bbbb that uses the fact that i is initialised to 1.

```
MODULE bbbb USE aaaa =
          (....; INT j := i; ....)
KEEP ....
```

Then imagine another person compiles a modules cccc that alters i

```
MODULE cccc USE aaaa =
          (....; i := 2; ....)
KEEP ....
```

If the external total order is aaaa, cccc, bbbb then any module (or program) that uses just bbbb will get the effect that the author of bbbb intended but if both bbbb and cccc are used it is likely that bbbb will not have the effect the author intended because j will refer to 2 after its assignment instead of referring to 1 as the author expected. It seems likely that the author of bbbb will feel he is being punished for the sins of the author of cccc.

The NAG Algol 68 library has been implemented in Algol 68-R and the only serious complaint is that the library is not tolerant of changes and so needs rebuilding too often. Experience with more naive users has shown that unexpected side-effects are also a serious problem.

8 THE PROPOSED STANDARD FOR ALGOL 68

The modules required are elaborated by a recursive procedure working from left to right in the use lists. If a module in a use list itself has a use list then the recursive procedure calls itself on this new use list. If a module has no use list or its use list has been exhausted it will be elaborated (provided that a copy is not known to exist).

This has the effect that all the use lists must be known if the total order is to be determined and that altering the order of a use list can sometimes alter the total order. Unexpected side-effects can only be avoided by knowledge, self-discipline or luck.

This system has not yet been implemented, so feed-back from users is unavailable. In the authors' opinion it is overly complex, possesses an unlovable syntax and is much too difficult for anyone who is not an Algol 68 lawyer to understand. It is however better than the proposed Ada system because given the complete text of all the modules involved all side-effects can be defined.

9 THE MODULAR COMPILATION SYSTEM ON THE RS ALGOL 68 COMPILER

The RS module system wished to avoid the shortcomings of simple procedure modules but without then encountering the problems with side-effects that have beset other systems. It was decided to extend simple procedure modules so that any declarations could be made in a module (for this reason they are called DECS modules). It was also found that other statements could be included without danger of side-effects. A DECS module can be any sequence of legal steps provided that two restrictions are enforced at compile-time. Firstly, the outer level of a module must not use any reference kept in another module. Since procedures are just code that is obeyed whenever they are called it possible to make a routine text free of restrictions if a second restriction, that procedures kept in other modules are not called at the outer level, is imposed. It is essential that routine texts be free of any restrictions so that procedures called by

programs that use the modules can communicate via non-locals that are also kept in the modules. These restrictions are sufficient to prevent any side-effects during elaboration, but unfortunately they prohibit the writing of some perfectly safe constructions. For example a procedure that does not alter any kept reference cannot be called in the outer level of another module even though it cannot cause a side-effect.

The designers of the module system of the RS Algol 68 compiler felt that communication during elaboration is so confusing that it is unreasonable to expect users to have to learn enough about the problem to be sure of always avoiding it, or to know enough about the module system to use the communication safely. The aim was to get the same effect (on communication) as multiple copying of modules but without the attendant overheads.

Two RS Algol 68 systems are already in use and so far the compile-time restrictions have not been found unduly difficult to work within.

10 THE PROPOSED STANDARD FOR ADA

Ada allows the separate compilation of a number of different compilation units. These include simple procedure modules (subprograms in Ada parlance) and unrestricted pieces of program (packages).

Ada only allows for one copy of each module and definitely regards communication during elaboration as a defect because the proposed standard says ([1], section 10.5) that if the order of elaboration is important then the program is erroneous, which means ([1], section 1.6) that if the program is executed then its results are ambiguous. This is a totally unsatisfactory solution to the problem because it will be very difficult to check if a program is erroneous. The authors feel that such ambiguities can best be avoided by compile-time restrictions, or failing that, the algorithm for the elaboration should be given.

The authors did not find this mechanism particularly easy to understand.

11 CONCLUSIONS

The authors' opinions of the relative merits of the various modular compilation systems have already been given. We would however like to add one more general comment. If, for efficency reasons, only one copy of each module is allowed then we feel that a minimal set of restrictions should be applied to get the same effect (on communication) as multiple copies. The restrictions in RS modules are one such set. The inability to call procedures in modules because there is no way of telling if they alter references non-locally is however quite a serious restriction. If a language contained a separate construct to describe a procedure that cannot alter non-locals then this sort of function could be called at the outermost level of a module. It is also possible that such a function could be made completely free of side-effects by prohibiting the use of reference parameters. A "side-effect free" function has a lot to be said for it in its own right because it would accurately reflect the mathematical idea of a function transforming one set of values into another set.

12 ACKNOWLEDGMENTS

The authors would like to thank Miss Susan Bond, Dr J.M.Foster and Dr D.P.Jenkins for all their advice and help.

13 REFERENCES

[1] Reference Manual for the Ada Programming Language - Proposed Standard Document. United States Department of Defense, (1980).

[2] Woodward,P.M. and Bond,S.G., "Algol 68-R Users Guide". Her Majesty's Stationery Office, (1975).

[3] Lindsey,C.H. and Boom,H.J., A Modules and Separate Compilation Facility for Algol 68, Algol Bulletin, 43, (1978).

[4] Bond,S.G. and Woodward,P.M., Introduction to the 'RS' portable compiler, RRE Technical Note, 802, (1977).

DATE
ILME
— 83