FILOMAT (Niš) 9:2 (1995), 303–313

Filomat '94, Niš, October 22-24, 1994
Geometry. Computer Sciences

DETERMINING MODULE DEPENDENCIES IN MODULAR PROGRAMS

Lehel Szarapka and Zoran Budimac

ABSTRACT. *A short and precise algorithm for determining a module initialization order in modular programming languages is described. This algorithm is compared with a classical technique of dependency analysis of module names. It is also shown how an algorithm for determining a module compilation order is drawn from a given algorithm.*

1. Introduction

Modular programming languages enable division of a program into a set of *modules* that limit the scope of their identifiers. In this way is the design and maintenance of large programs easier, especially in team projects.

An identifier from module A is visible in module B if it is *exported* from module A and *imported* in module B. Exporting and importing of identifiers is achieved by specialized programming language constructs. Among the most popular modular languages (Ada, Modula-2, Modula-3, ...) the definition of (every) module M consists of (at least) two parts:

(1) an *interface* (or *definition module*, *package specification*, ...) of the module M, which lists all identifiers which module M exports,
(2) an *implementation* (or simply *module*, *package*, ...) of the module M, which implements (i.e., defines exported identifiers) the interface of module M.

In the rest of the paper we shall assume that a main (i.e., program) module contains a "dummy" interface. In this way *all* modules have interface and implementation parts. An implementation of a module can have (possibly

1991 *Mathematics Subject Classification.* 68N20.

The second author is supported by Science Fund of Serbia, Grant #0403 through Mathematical Institute, Belgrade.

empty) initialization part: the sequence of statements to be executed before the main program starts its execution.

Both interface and its implementation can import identifiers from other modules, via the special language constructs (import lists). The scope of the imported identifiers is only the module that has imported them.

Implementations of truly modular programming languages (Ada, Modula-2, Modula-3, Oberon, Oberon-2) should keep a complete "bookkeeping" of their modules to correctly maintain a module *compilation order* and *initialization order*. Implementations of other programming languages that only enable independent compilation (C, C++) are usually supported by separate utilities (for example *make*) to do the same task.

In this paper the algorithms for both activities are presented. It is shown how the algorithm for determining compilation order can be successfully drawn from an algorithm for determining initialization order, thus merging two activities into only one. The main contribution of the paper however is a construction of a small and efficient algorithm that can be included into a compiler of a modular language, which *precisely* determine the module initialization order. This is especially important in the presence of circular dependencies among modules, where many programming languages allow uncertainty.

The rest of the paper is organized as follows. The second section emphasizes the importance of the module compilation and initialization order and different approaches to its determination. The third section describes dependency analysis - a "classical" technique for determination of dependency order. The fourth section introduces the new algorithm, while the fifth section compares two approaches. The sixth section extends the algorithm for initialization order to an algorithm for compilation order. The last section concludes the paper.

2. Definitions and previous work

2.1 Initialization order.

According to definitions of all modular programming languages, an initialization part of every implementation of a module M is to be executed:

(1) exactly once,
(2) after initialization parts of all modules (in arbitrary order) which module M imports, and
(3) before execution of the main program.

Definition 1. Initialization order *is the order in which all modules constituting the program are initialized, such that the above three conditions are fulfilled.*

Example 1. Let module A import modules B, C and D, and modules B, C and D import nothing (which means, that they are independent of other modules.) The initialization order is the following: (B, C, D), A, where the order of B, C and D is arbitrary.

In languages where mutual imports (i.e., circular dependency) of module implementations is allowed, the initialization order is undefined (see for example [7] and [3] for Modula-2 and Modula-3 respectively.)

Example 2. Let module A import modules B and D, module B import module C, module C import module E and module D is independent. The structure of the modules is the same as in Figure 1 except that modules C and D are not connected. The possible initialization orders are the following: E, C, B, D, A, or E, C, D, B, A, or E, D, C, B, A, or D, E, C, B, A. Note that module E is always initialized before module C, and module C is always initialized before module B. Module D must be initialized before module A.

In real programming projects, circular dependencies among module implementations are not rare. If in such cases the initialization order is not defined, the programmer alone must take care of the correct and explicit initialization, to produce reliable and portable code (which is not in the "spirit" of modular programming languages.)

Example 3. Let module A import module B, module B import module C and module C import module B (circular dependency among modules B and C.) Initialization order is the following: (B, C), A, where initialization order of modules B and C is *not defined*. Note the difference between this example and Example 1, where initialization order of modules B, C and D was *arbitrary* (and always correct.) In this example the order chosen by the target compiler might be "incorrect", i.e. different from the programmer's intentions.

2.2. Compilation order.

According to definitions of all modular programming languages, an *interface* of module M is to be compiled:

(1) before compilation of implementation(s) of module M and
(2) after compilation of all interfaces that the interface of M imports.

Similarly, an *implementation* of module M is to be compiled:

(1) after compilation of interface(s) of module M and
(2) after compilation of all interfaces that the implementation of M imports.

Definition 2. Compilation order *is the order in which all modules constituting the program are compiled, such that the above four (two + two) conditions are fulfilled.*

Example 4. Let module A import modules B and D, module B import module C, module C import module D and module D is independent. The compilation order is the following: D, C, B, A.

In order to make a compilation a deterministic process, circular dependencies among interfaces are *not* allowed. Note that circular dependencies among module implementations are allowed.

Example 5. Let module A import module B and module B import module A. In this case the compilation order can not be established, because it is not clear which module should be compiled first.

2.3. Previous work.

Most of the research and publicly available compilers of modular programming languages:

(1) rely on external tools to establish compilation order and
(2) separately deal with compilation order and initialization order.

For example, **MOCKA** Modula-2 compiler [4] provides a separate utility which maintains the dependency graph to establish correct compilation order of modules. Modula-2* compiler [6] uses also dependency graph to establish compilation order, but later relies on Unix *make* utility to maintain it. Modula-2 implementation described in [5] leaves the responsibility of the compilation order to the programmer.

Almost all publicly available compilers implement initialization parts of modules as separate procedures which are called in order of their appearance in import lists. To avoid multiple calls, an indication variable for every module is set to `TRUE`, when initialization procedure is called.

Algorithms and strategies for determining compilation and initialization order are not publicly available in commercial implementations of modular programming languages.

In the next section we describe in more detail dependency analysis, as a tool for establishing correct dependencies. Up to the sixth section, we concentrate only on algorithm for determining initialization order.

3. Dependency analysis

Dependency analysis is a classical technique applied when exact dependency between some entities is to be determined. As stated in [2], it consists of the following three steps:

(1) construct a directed graph (dependency graph), such that a node a

is connected to a node b if and only if the entity a in a real world domain depends on entity b,

(2) find all strongly connected components of the dependency graph,

(3) sort strongly connected components of the dependency graph into dependency order. This is usually achieved by coalescing all components into single nodes and by sorting them topologically.

The graph transformed in the described manner shows dependencies between entities represented as graph nodes, where all nodes coalesced into one node are mutually (i.e., circularly) dependent. More details about the construction of dependency order can be found for example in [1] p. 221.

The graph for determining module dependencies consists of module names (as nodes of the graph) and links between them. More formally, the initial graph (step 1 of the above algorithm) is constructed in the following way:

(1) Construct a graph which consists of isolated nodes $M_i, i = 1, ..., n$, where $M_i, i = 1, ..., n$ are module names,

(2) Connect M_i to M_j if and only if module M_i imports module M_j.

Circularly dependent module names can be initialized in any order (which is in accordance with definitions of most modular languages.)

Dependency analysis is time and space consuming, no matter how efficiently graphs are represented (see the fifth section for details.) Dependency analysis is therefore not very suitable in direct inclusion in compilers. In the next section we proceed with a description of a more efficient solution.

4. Another solution

Basic design decisions of the new solution are:

(1) In the absence of circular dependencies, dependency analysis is equivalent to (much more efficient) depth-first search of a graph.

(2) Therefore, circular dependencies have to be resolved in a deterministic way, if possible.

The most natural way to resolve circular dependencies in a deterministic way is to establish a precise initialization order. A natural solution is to initialize modules in the order in which they appear in import lists. However, the rule that initialization of a module must be executed *after* initialization of all imported modules, must be obeyed. For example, if main module M imports module A, module A imports module B and module B imports module A, then the initialization of module B is to be executed prior to initialization of module A.

Once accepting this principle, an algorithm is simple and straightforward. The Modula-2 (pseudo-)procedure `Analyze1` implements the proposed algo-

rithm. The following variables and procedures are used:

(1) `initialization` - a list of module names in the initialization order,
(2) `InsertBefore(M,Module,List)` - a procedure which inserts module name `M` before the module name `Module` in a list `List`,
(3) `Member(M,List)` - a function procedure which returns logical truth value `TRUE` if name `M` belongs to a list `List`,
(4) `IntfOf(M)` - a function procedure which returns the name(s) of the interface(s) of module M, and
(5) `ImplOf(M)` - a function procedure which returns the name(s) of the implementation(s) of module `M`.

We shall assume that `IntfOf (Impl (module))` is undefined, i.e. returns a null value, and that `Impl ( Impl (module)) = Impl (module)`.

Prior to calling `Analyze1`, the list `initialization` contains only the name of a (main) module implementation. The procedure is as follows.

```
PROCEDURE Analyze1(module: ARRAY OF CHAR);
  FOR every import list of module DO
    FOR every module name M in import list DO
      IF NOT Member(ImplOf(M), initialization) THEN
         InsertBefore(ImplOf(M), module, initialization);
         Analyze1(ImplOf(M))
      END
    END
  END
```

After procedure `Analyze1` returns, the list `initialization` contains the list of module names in their initialization order. Note that initialization order depends only on module implementations, and not on module interfaces.

5. Comparison of two algorithms

Our algorithm gives the same initialization order as dependency analysis, if circular dependencies are not present among modules. However, when circular dependencies are present, our algorithm sometimes gives a different initialization order than dependency analysis.

In the example of module dependencies displayed in Figure 1, A imports B and D (in that order), B imports C, C imports D and E (in that order), and D imports C. Dependency analysis gives the following initialization order: E, (C, D), B, A, where the order of C and D is arbitrary. Our algorithm however, gives the slightly different, but deterministic initialization order as follows: D, E, C, B, A. If the order of imported modules of module C were

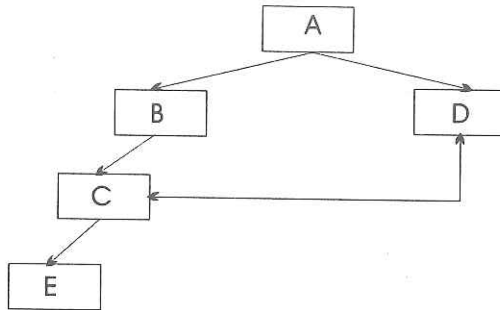

Figure 1. An example of module dependencies

changed into E and D (instead of D and E, as displayed), our algorithm would give the following order: E, D, C, B, A, which is same as the result of dependency analysis, but is deterministic. If A would import D before B, the initialization order would be E, C, D, B, A, no matter what C imports first, which is again the same as a result of dependency analysis.

Dependency analysis has a computational complexity of $O(n^2)$, where n is a total number of modules. Our algorithm has complexity proportional to hn, where h is the deepest possible nesting level of modules constituting the program. Since in most cases $h \ll n$, the complexity of our algorithm is $O(n)$. Only in degenarate cases (when $h = n$), the complexity of our algorithm is equal to the complexity of dependency analysis.

However, in reality (i.e., when n is finite) performances of our algorithm are much better than those of dependency analisys (and better than computational complexity shows.) In the following table some characteristics of both approaches are compared. Compile-time sizes of implementations of both approaches are given in the number of lines of source (Modula-2) code, while the run-time size is given in bytes. Graphs in dependency analysis are implemented as adjacency matrices of static size, but appropriate dynamic implementation would not be much smaller.

Feature	Dependency Analysis	Our Algorithm
Speed (29 modules)	4.40 sec	0.72 sec
Speed (21 modules)	3.41 sec	0.55 sec
Code size (Compile-time)	555 lines	85 lines
Code size (Run-time)	4450 bytes	800 bytes
Data size (Run-time)	$340n$ bytes	$n+16h$ bytes

6. Compilation order

The algorithm for determining compilation order can be easily obtained by the appropriate extension of the algorithm for determining initialization order. Since circular dependencies are not allowed in interfaces of modules, our algorithm will always give the same results as dependency analysis. Let us recall that for establishing initialization order only module implementations are taken into account. In order to produce an algorithm for determining compilation order, however,

(1) module interfaces have to be taken into account as well, and
(2) a list of visited module names has to be maintained to report any violation of circular dependency restriction.

Besides the already introduced variables and procedures, the following new variable and procedures are needed for the implementation of a new algorithm:

(1) `visited` - a list of visited interface names which are imported from interfaces,
(2) `InsertFront(M, List)` - inserts module name `M` at the front of a list `List`,
(3) `RemoveFront(List)` - removes a module name from the front of a list `List` (the above two operations are analogous to the `Push` and `Pop` operations on stacks), and
(4) `MakeEmptyList()` - returns an empty list.

If the procedure `Analyze` is to be called to determine *compilation* order, the list `initialization` has to contain the module name to be compiled. If the procedure `Analyze` is to be called to determine *initialization* order, the list `initialization` has to contain `Impl(module)`. The parameter `check` is set to `TRUE` if an interface is to be analyzed.

At the beginning the list `visited` is always set to an empty list and is made local to the module. This is important because of the detection of the circural dependencies. When we analyze an implementation module a

new list will be created, and a new compilation order check will start. The algorithm for determining initialization order is not affected.

Because of features of procedures `Intf` and `Impl`, the `Member (Intf (module))` (seventh line of the following procedure) returns `TRUE` if a `module` is an implementation module (because a null value is a member of every list.) Similarly, `FOR` loop (18th line of the following procedure) will execute only once (for implementation part only).

```
PROCEDURE Analyze(module: ARRAY OF CHAR; check: BOOLEAN;
                  visited: List);
   IF check THEN
      InsertFront(module, Visited)
   END;
   IF module is an implementation module THEN
      IF NOT Member(Intf(module), initialization) THEN
         InsertBefore(Intf(module), module, initialization);
         IF M1 = Intf(M) THEN
            Analyze (M1, TRUE, visited)
         ELSE
            Analyze (M1, FALSE, MakeEmptyList());
         END
      END
   END;
   FOR every import list of module DO
      FOR every module name M in import list DO
         FOR M1 := Intf(M) TO ImplOf(M) DO
            IF NOT Member(M1, initialized) THEN
               InsertBefore(M1, module, initialization);
               Analyze(M1, M1 = Intf(M));
            ELSE
               IF check AND Member(M1, visited) THEN
                  report circular dependency;
               END
            END
         END
      END
   END;
   IF check THEN
      RemoveFront(visited)
   END;
```

7. Conclusion

An algorithm for establishing initialization and compilation order of modular programming languages is proposed. Its main contributions are:

(1) it establishes both compilation and initialization order;
(2) it is smaller and more efficient than "classical" dependency analysis, and thus can be included directly into a compiler;
(3) it improves the definition of modular programming languages by introducing deterministic initialization order in case of circular dependencies.

The third improvement of our algorithm over dependency analysis comes with a cost of producing (in some cases) different initialization order than dependency analysis would. However, according to [8], the emerging ISO Modula-2 standard (for example) gives also a clear advantage to the precise definition of module initialization order than to a classical (and sometimes vague) dependency analysis.

The presented algorithm can be applied without changes to any modular programming language regardless of how many interfaces and implementations of a module M are allowed. That includes languages like: Ada, Modula-2, and Modula-3. In languages like Oberon and Oberon-2, where circular dependencies are forbidden in implementation modules as well, a slight modification is required.

The proposed algorithm is included in a Modula-2 compiler, which is currently under development at the Institute of Mathematics in Novi Sad.

References

[1] A. Aho, J. Hopcroft, and J. Ullman, *Data Structures and Algorithms*, Addison Wesley, London, 1985.

[2] Z. Budimac, L. Szarapka, Z. Putnik, and M. Ivanović, *Dependency Analysis in a Compiler of a Functional Language*, Bull. Appl. Math. **1047/94 (LXXIV)** (1994), 43–50.

[3] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson, *Modula-3 Report (revised)*, SRC of Digital Equipment Corp. and Olivetti Research Center, Palo Alto and Menlo Park, 1989.

[4] H. Emmelmann and J. Vollmer, *GMD Modula System MOCKA - User Manual*, University of Karlsruhe, Technical Report, Karlsruhe, Germany, 1994.

[5] L. B. Geissmann, *Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith*, Ph.D. thesis no. 7286 ETH Zürich, Switzerland, 1983.

[6] S. U. Hänssgen, E. A. Heinz, P. Lukowitz, M. Philippsen, and W. F. Tichy, *The Modula-2* Environment for Parallel Programming*, Proc. of the Working Conf.

on Programming Models for Massively Parallel Computers, 1993, Berlin, Germany, (to appear).

[7] N. Wirth, *Programming in Modula-2*, fourth edition, Springer Verlag, Berlin, 1988.

[8] M. Woodman, *A Taste of Modula-2 Standard*, SIGPLAN Notices **28 (9)** (1993), 15–24.

University of Novi Sad, Faculty of Science, Institute of Mathematics, Trg D. Obradovića 4, 21000 Novi Sad, Yugoslavia
E-mail address: `ilehel@unsim.ns.ac.yu`

University of Novi Sad, Faculty of Science, Institute of Mathematics, Trg D. Obradovića 4, 21000 Novi Sad, Yugoslavia
E-mail address: `zjb@uns.ns.ac.yu, zjb@unsim.ns.ac.yu`