A note on the checking of interfaces
between
separately compiled modules

C. Bron
E.J. Dijkstra
Department of Computing Science
State University of Groningen
Netherlands

T.J. Rossingh
Department of Mechanical Engineering
Twente University of Technology
Netherlands

1. INTRODUCTION

The subject matter of this note was originally contained in a paper presented at the conference quoted in reference [Dau]. As we were preparing the final manuscript for the conference proceedings, we realized that this subject was somewhat out of order within the context of that particular paper, reason why it is being left out of the final version. On the other hand, in view of the increasing interest in the maintenance of (large) libraries of software modules, as spurred by Ada developments, we deemed it worthwhile to report the material in a separate note.

This note is based on our experience with a language, Modular Pascal[Mod]] (or ModPas for short), providing separate compilation of modules with the benefit of full (Pascal) type checking across module boundaries.

Each module that is part of a program (or system) described in ModPas, may use (import) declarations from other modules. Complimentarily, modules may export some of their declarations to be available in importing modules. Imported modules act as outer blocks to the importing module.

2. COMPILE-TIME TYPE CHECKING

For the compilation of a module, the descriptions of the exported features of the imported modules are needed. These descriptions form the environment for that module. Some languages like Ada[Ada], Mesa[Mes], and Modula-2[Mo2] have a separate syntactic entity for such descriptions (e.g. Definition Modules in Modula-2).

For a ModPas module, the compilation results in the generated machine code (code file) and in the description of the exported features of that module (specification file). The specification file contains (in Pascal-like source format) the exported declarations. The specification files of imported modules are prefixed to the source code of a module during its compilation, and treated as outer blocks of that module.

In this way, type checking can be done by the compiler in a manner similar to that for monolithic (Pascal) programs. It is easily seen that this arrangement imposes an ordering on the compilation of a set

of interdependent modules. However, in most cases, if a module is recompiled, recompilation of dependent modules is not necessary.

3. LINK-TIME TYPE CHECKING

The above scheme for compile-time checking of module interdependencies is insufficient if the compilation phase is separated from the phase in which modules are bound together to form a program, or are embedded in the environment of an already operational system. In this phase it must be checked that an actual module being imported corresponds to its specification file as was used when the importing module was being compiled. E.g. if A imports B, B may have been changed and recompiled after A and thus its interface with A may have been disturbed.

A solution we considered for this checking problem was a time-stamping mechanism. If the specification file of B is time-stamped, it can always be checked that the code of A is "younger" than the interface of B. Furthermore, we were to suppress the updating of the time-stamp of B's interface if the specification file, upon recompilation of B, is identical to its previous copy.

In the Ada compiler, developed at the University of Karlsruhe the following approach is taken: If B's interface undergoes changes, all modules dependent on B are marked as candidates for recompilation upon a change in the interface of B [Goo]. A recent paper [Dau] shows how to save on this effort to some extent at the expense of a more detailed dependency analysis. The Karlsruhe approach may turn out quite costly if large libraries start to build up, or if there is a fair chance that B's interface will be subject to more changes before the majority of dependent modules will run in B's environment. And how does one detect -in the library- dead alleys of modules that are not supposed to become operational anymore? The solution is even unworkable if some libraries are maintained off-line.

We prefer to postpone the "time-stamp check" until the startup (or linkage) time of dependent modules.

As we were about to implement time-stamping in the ModPas environment, we realised that time-stamping can only be used in a universal module space, and that inconsistencies are bound to arise in a hierarchical file system where the same name may appear in several directories, and where files may be moved between directories or renamed in an arbitrary way. The above observations led to a much simpler approach. For every exported interface a checksum is calculated, which -to all intents and purposes- changes with every change in the interface. The chance for accidental equality of the checksums of two different interfaces can be made arbitrarily small by calculating a multi-length checksum. In the extreme case the 'value' of the specification file itself may act as the 'checksum'. This is, however, both impractical and unnecessary.

The code of each module is now labeled with its own (interface)checksum. At the time when A is linked to B, it is checked that the code of B bears the same checksum as was valid for the interface of B during compilation of A. This approach necessitates a

minimal amount of recompilation and no library searching at all. In exchange for these advantages, it must be admitted that the need for recompilation is detected at the latest possible stage.

In Mesa, unique 'checksums' are attached to interfaces, which are derived from the date the interface is compiled. This form of checksum calculation has the disadvantage that an interface can never be restored to a previous stage, e.g. after some files have been lost.

In ModPas, if the compilation of a revised version of a module does not result in a new interface, the checksum (calculated by a deterministic algorithm) remains the same. This allows for the changing of the body of an exported procedure or the addition of non-exported declarations.

4. USE OF NAMES APPEARING IN INTERFACES

In the ModPas environment, the interface (and its checksum) change with a change in the name of an exported item. This is in line with usage in monolithic Pascal programs: a change of a declared identifier invalidates the program (unless the identifier is not used at all). Formal parameter names in exported procedures do not affect the interface. This again is in line with practice in monolithic programs. Pascal procedures are called with positional parameters and therefore, at the site of call, the names of the formal parameters are irrelevant.

The relation between name change and interface change can be used as part of a software engineering discipline: if the semantics of an exported item (usually a procedure) changes, this by itself has no syntactic consequences. However, it is good practice to change the name of the exported item together with its semantics. Subsequently, inconsistencies in the inter-module semantic interactions will be detected by the interface checking mechanism described in this note.

More generally, the lesson to be learned is this one: exported names (i.e. names with a large scope) should be meaningful and should, as much as possible, reflect the semantics of the item they name.

5. REFERENCES

- [Ada 81]
U.S. Dept. of Defence, The Programming language Ada - Reference Manual
- [Dau 84]
M. Dausmann, Reducing recompilation Costs for Software Systems in Ada, Implementation Languages: Experience and Assesment, IFIP WG 2.4 Working Conference, Univ. of Kent, 17-19 Sept. 1984
- [Goo 81]
G. Goos, G. Winterstein, Problems in Compiling Ada, Third E.C.I. Conference, Munich (Germany), Oct. 1981 Lecture Notes in Computer Science #123, Springer Verlag
- [Mes 79]
J.G. Mitchell, W. Maybury, R. Sweet, Mesa Language Manual 5.0, Xerox

PARC Report CSL-79-3, April 1979

[Mod 80]

C. Bron, Report on the Programming Language Modular Pascal, TW
Memorandum #296, Dept. of Applied Mathematics, Twente University of
Technology, Febr. 1980

[Mo2 82]

N. Wirth, Programming in Modula-2, Springer Verlag 1982