SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 9, 921–924 (1979)

A Simple Technique for Controlled Communication Among Separately Compiled Modules

DAVID R. HANSON
Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, U.S.A.

SUMMARY

A simple technique for communication among separately compiled modules using the existing facilities of most operating systems is proposed. The access control capabilities of existing hierarchical file systems can be used to control access to shared objects (e.g. procedures, types, data structures). Information about shared objects, such as type and date/time of compilation, is stored in description files, and access to a description file implies access to those objects. Declarations cause the appropriate information to be maintained in the description files. The advantage of this approach is that it is based on existing mechanisms with which most programmers are well acquainted.

KEY WORDS Separate compilation Module interconnections Access control Program maintenance File systems UNIX

INTRODUCTION

One of the major problems in the development of a large program is controlling the communication among program components such as procedures and data modules.[1, 2] Such communication is usually controlled by programming language constructs, e.g. parameter transmission techniques, scope rules, block structure and access to shared (global) objects. The facilities of most languages, however, are designed to provide a means of sharing within a single compilation unit. Global references are usually the only means of communication among separately compiled modules.

Separate compilation is an invaluable aid to the development of large programs. Not only does separate compilation conserve computing resources, but it is much simpler to work with the components of a program (procedures, data modules, etc.) than with the system as a whole. It might be impossible to work with the entire system if several programmers are working on different components, each at their own speed. In addition, debugging a component in the environment of an incomplete and untested system is extremely difficult and error-prone. In short, separate compilation permits programmers to work efficiently on small, manageable components of the system without interfering with each other.

There are several disadvantages of separate compilation as embodied in most languages. First, most languages provide only an 'all or nothing' external reference capability. There is no way to control the scope of external definitions and references. As a result, naming conflicts and references to the wrong objects are common problems. Second, type information is usually available only during compilation. Many of the advantages of a strong type system are lost when referring to external objects. For example, most languages rely on programmer-supplied type information for such references. There is, however, a way to

0038–0644/79/1109–0921$01.00 *Received 21 February 1979*

check this information at link time.[3] Finally, there is usually no way to specify dependencies among separately compiled modules. When a module is changed, the programmer must understand the entire system well enough to determine which other modules require recompilation. While there exist programs that aid in this task,[4] dependency relationships can be determined automatically if a more suitable means of intermodule communication is provided.

Reconciling the conflicting goals of separate compilation and strong type checking at compile time is a difficult problem.[2] Much recent research in programming languages has been directed toward integrating strong typing with facilities for intermodule communication and access control.[1, 2, 5, 6, 7, 8] Comparatively little work has been done in proposing facilities for communication among separately compiled modules that can be applied to existing languages supporting separate compilation.

This paper proposes a simple technique for controlling intermodule communication. The scheme described below is not intended to be a comprehensive solution to this difficult problem. For example, it does not provide access control capabilities as comprehensive as some other proposals.[1, 6] It can, however, be easily incorporated into existing languages, and, perhaps more important, is based on facilities with which most programmers are well acquainted. It therefore provides a usable vehicle for experimentation upon which to base subsequent research.

THE IMPORT/EXPORT FACILITY

The basic idea is to use an existing mechanism for controlled sharing instead of adding a new mechanism to an existing language or design an entirely new language. The file system supplied by the operating system is an example of the kind of existing facility that provides some of the necessary features. The access and protection mechanisms imbedded in hierarchical file systems, such as provided by UNIX,[9] can be used to control communication among separately compiled modules. In addition, programmers are likely to be intimately familiar with the use of their file system. Thus programmers can control communication among separately compiled modules without having to acquire additional knowledge.

The file system can be used for this purpose by storing appropriate information about external objects, such as type and time of compilation, in files whose access is controlled by the programmer.

The declaration

export *x* **to** *filename*

directs the compiler to store type information and other characteristics of *x* in file *filename*. Object *x* is accessible from other modules only if filename is accessible to the programmers responsible for those modules. These files, called description files, are constructed and maintained by the compiler; they are never edited by programmers.

The declaration

import *x* **from** *filename*

causes the compiler to obtain the characteristics of *x* from the description file *filename*, provided it is accessible. Note that if *x* is a procedure or a variable, references to *x* are external references, which must be resolved at a later time. The type checking associated with operations on *x* can be performed at compile-time, however, since the necessary information is kept in the description file. Link-time type checking can also be performed as a check on the correctness of the compiler.

The kinds of objects shared using this mechanism are not restricted to variables and procedures. The import/export declarations can also be used to share type definitions, compile-time parameters and macro definitions, for example. Most programming languages distinguish between sharing compile-time definitions and external objects by supplying a different mechanism for each kind of sharing (e.g. %INCLUDE and EXTERNAL in PL/I). The use of description files permits both kinds of sharing to be handled by a single mechanism.

The general forms of the import and export declarations are

export *x*1, *x*2, . . . , *xn* **to** *f*1, *f*2, . . . , *fm*
import *x*1, *x*2, . . . , *xn* **from** *f*1

If the protection mechanisms of the available file system are inadequate, these declarations can be augmented with access control information that is placed in the description files. For example, an access list containing module names can be associated with each object. The names in the access lists specify to which modules the object may be imported. The information contained in description files would also be useful for the automatic generation of system documentation.

Another common problem in large systems is that a change in one module may require recompilation of other modules. This is particularly important when data structures have been changed. This kind of dependency information can also be kept in description files. The export declaration causes information concerning the definition of the objects, such as the name and location of the source file and the date/time of compilation, to be stored in the indicated description file. The import declaration causes dependency information to be added to the description file. For example, if the name of the dependent source file and the date/time of importation is added to the description file, subsequent exportation can trigger recompilation of the dependent module. This scheme is similar to that used by Make.[4]

Note that this scheme cannot handle 'mutual' dependencies. For example, suppose module *a* contains

```
export f to f1;
import g from f2;
procedure f(. . .)
  . . .
  g(. . .);
  . . .
end
```

and module *b* contains

```
export g to f2;
import f from f1;
procedure g(. . .)
  . . .
  f(. . .);
  . . .
end
```

Module *a* must be compiled before module *b* and vice versa. The obvious solution is to construct a module containing both *f* and *g*. Alternatively, a means for explicitly editing description files can be provided.

For instance, if the language is modified to permit the definition of procedure *f* without the accompanying procedure body, the 'first cut' at module *a* can contain only the export declaration and the definition for *f*. Module *b* can then be compiled. Finally, *a* can be

'fleshed out' and compiled using the information in description file *f2* to check the consistency of the definition of *f*. This approach is similar to the way in which abstractions are entered into the CLU library.[7] The specification of the exported object is all that is really required in order to make an entry in the description file.

Like most high-level language features, the import/export feature imposes a particular structure on programs. While cases that do not 'fit' into this structure can always be found, the imposition of structure is an aid to program construction just as are the various structured control statements and type systems found in recent languages.

CONCLUSIONS

The proposed import/export facility provides a means of partially controlling the access to shared objects among separately compiled modules at compile-time. The important aspect of this approach is that it is based on existing facilities. While there are drawbacks to using existing facilities, there seems to be a tendency to include new features in programming languages without sufficient experimentation. Inclusion of features that purport to solve some of the difficult problems associated with intermodule communication should undergo experimental tests. In many programming environments, existing file systems provide many of semantic capabilities necessary to control communication adequately among separately compiled modules. This paper has proposed a simple syntactic construct to make use of those existing capabilities thereby providing a usable experimental facility.

REFERENCES

1. F. DeRemer and H. H. Kron, 'Programming-in-the-large versus programming-in-the-small', *IEEE Trans. Software Eng.*, **SE-2**, 80–86 (1976).
2. J. W. Thomas, *Module Interconnection in Programming Systems Supporting Abstraction*, Ph.D. Diss., Tech. Rep. CS–16, Comp. Sci. Dept., Brown Univ., Providence (1976).
3. R. G. Hamlet, 'High-level binding with low-level linkers', *Comm. ACM*, **19**. 642–644 (1976).
4. S. I. Feldman, 'Make—a program for maintaining computer programs', *Software Practice and Experience*, **9**, 4, 255–266 (1979).
5. C. M. Geschke, J. H. Morris and E. H. Satterwaite, 'Early experience with Mesa', *Comm. ACM*, **20**, 540–553 (1977).
6. A. K. Jones and B. H. Liskov, 'A language extension for controlling access to shared data', *IEEE Trans. Software Eng.*, **SE–2**, 277–285 (1976).
7. B. H. Liskov *et al.*, 'Abstraction mechanisms in CLU', *Comm. ACM*, **20**, 564–576 (1977).
8. N. Wirth, 'Modula: a language for modular multiprogramming', *Software Practice and Experience*, **7**, 3–35 (1977).
9. D. M. Ritchie and K. Thompson, 'The UNIX time-sharing system', *Comm. ACM*, **17**, 365–375 (1974).