EXPERIENCE WITH A MODULAR TYPED LANGUAGE: PROTEL

by P.M. Cashin, M.L. Joliat, R.F. Kamel, D.M. Lasker

Bell-Northern Research
Ottawa, Canada

Abstract

The support for modular software and the ability to perform type checking across module boundaries are becoming the mainstay of recent high level language design. This is well illustrated by languages such as MESA and the US Department of Defence's new standard language ADA. At Bell-Northern Research, PROTEL, one of the first modular typed languages, has been used since 1975 to implement a substantial software system. The experience accumulated in building this system has given us a unique perspective. It has shown that the confidence of language designers in modular typed languages is well founded. It has also revealed some pitfalls which others will undoubtedly encounter. The purpose of this paper is to share our experience by outlining the nature of the problems and our solutions to them.

Keywords : modularity, languages, separate compilation, type checking, ADA

Introduction

The technique of constructing systems from separately compiled modules has existed since the earliest days of programming. Fundamental to this technique is the ability to divide a system into modules and to share data and procedures among these modules. The evolution of approaches to system partition and information sharing has been a major trend in programming language design.

Early languages, as typified by FORTRAN [ANSI 66], JOVIAL [Pe 66] and PL/1 [ANSI 75], provided indiscriminate sharing through global data pools or through explicit access to public data. Such languages required no interface definition and performed neither validity checking of procedure calls and their parameters, nor consistency checking of access to shared data. Any errors in these would go undetected until linkage time or until program execution anomalies revealed their presence. The pain of debugging and supporting large systems written in such languages has marked a generation of programmers.

More recently the notions of interface specification, data hiding and strict compile time type checking of inter-module data and procedure references have evolved. Many new languages support these notions. Most notable among them are MODULA [Wi 77], EUCLID [LHLM 77], CLU [LSAC 77], MESA [GMS 77], and more recently ADA [ADA 79].

Early experience with these languages has been favourable but, to date, there has been no reported experience with a substantial commercial system which tests the limits of modular typed programming in field situations. At Bell-Northern Research, such a system has been implemented in a language called PROTEL [FJKM 79].

The PRocedure Oriented Type Enforcing Language (PROTEL) was designed, in 1975, for the production of software for digital switching systems. The need for PROTEL arose as part of the development of the DMS-100 family of telephone switches [ICC 78]. DMS-100 is a large software controlled real time system which implements a diverse set of telephony applications, as well as an extensive maintenance and administration capability. A DMS-100 system configuration may contain from 150,000 lines to 250,000 lines of source. In addition to telephony software, PROTEL has also been used to write systems programs such as compilers, linkers, operating systems and database systems. The development and support team for all of DMS consists of nearly 200 people and three levels of management. So far, over 750,000 lines of PROTEL source have been developed.

Our experience in modular software development, acquired during the construction and maintenance of the above systems, is considerable. It has given us a unique perspective on large system development using modular typed languages. This perspective leads us to believe that the benefits of using modular typed languages for system development far exceed those offered by any alternate approach. Our experience has also revealed some technical issues which others will inevitably have to face. The purpose of this paper is to share our experience by reviewing the benefits, describing the pitfalls and outlining their solution.

The paper is divided into five major sections. The first of these has placed our work in perspective. Section 2 describes the concept of modularity as supported by PROTEL. Section 3 describes the PROTEL development environment. An extensive discussion of our experience is contained in section 4, and section 5 outlines our conclusions.

Modularity in PROTEL

The module is the basic building block of PROTEL. It typically defines operations expressed as PROTEL procedures which may be invoked by other modules. A module consists of one or more separately compiled units called SECTIONS. Sections fall into two categories: INTERFACE and IMPLEMENTATION. Interface sections usually contain procedure headings for the operations provided by the module as well as type definitions for the procedure parameter types. Interfaces may also contain constant and/or variable declarations, however PROTEL programming standards discourage the use of variables in interfaces. Implementation sections contain the executable statements of the interface procedures along with additional type, data and procedure declarations. Symbols defined in interfaces are accessible from outside the module, while those in implementation sections are hidden.

A module may contain several interface sections and several implementation sections. Multiple interfaces are used to logically group module functions. Examples of the use of multiple interfaces may be found in [La 79]. Multiple implementation sections allow the programmer to structure the contents of a module. This feature also reduces recompilation since the whole module need not be recompiled when a change is made to its implementation.

The sections of a module form an inverted tree structure as shown in figure 1. The first statement of a module's root section names the module and lists the names of interfaces it imports. The first statement of each of the remaining sections of the module determines their position in the module tree. The section header also indicates whether a section is an interface or an implementation section.

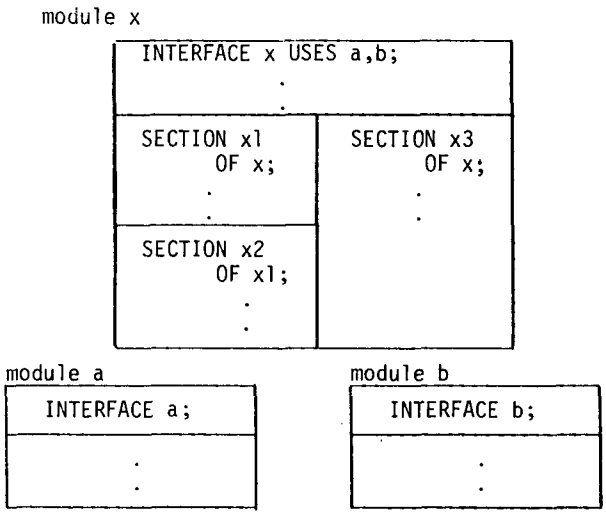

FIGURE 1 : PROTEL MODULE STRUCTURE

The items imported by a module are those declared in sections explicitly USED by the module, and those declared in interface sections above the used interfaces in their respective module tree. The items visible in a section of a module are those imported by the module and those declared in sections which appear on the path from the section to the root of the module. For example, in figure 1, items declared in the interfaces of a and b are visible throughout module x. All these items, plus those declared in the interface of x are seen throughout sections x1, x2 and x3. The items of x1, however, are not seen by x3 or visa-versa. The items of x1, but not those of x3, are visible in x2.

The PROTEL Environment

The PROTEL system supports the concept of modularity throughout the software development process. This system consists of a compiler, a linker, a loader, a library control system and a number of other utilities. All these tools are operational on both the IBM/370 and on the DMS-100 processor except for the loader which is only available on DMS-100.

The Compiler

The compiler accepts PROTEL source for a section and generates an object file which contains symbolic information as well as partial data and code segments for procedures and constants declared in the section.

To support type checking of inter-section and inter-module references the compiler performs a process we have christened 'imbedding'. This process divides into two parts: outbedding and inbedding. Outbedding is the act of writing symbolic information to an object file. Inbedding consists of reading that information for all sections visible to the one being compiled and using it to initialize the compiler symbol table. With the symbol table so initialized, full compile time checking of all references can take place.

It is worth noting that imbedding imposes a partial ordering on the compilation of a system: before a module can be compiled, symbol table information from all modules it USES must be available.

The Linker

The linker reads section object files and produces a single loadable file per module. To do so, it strips off the symbolic information from the object files and concatenates the partial code and data segments into complete ones. Note that sections disappear in the linker and are not visible beyond this point in the development process: modules are the basic units of system building.

The Loader

Unlike most systems, DMS-100 has no system wide linking process; our so-called linker

operates on one module at a time. Instead, an on-line loader is used to incrementally add new modules to an existing system. This was done to allow for flexible configuration using modules as building blocks for different systems and to allow additional modules to be added on-line to an in-service system. The loader commands necessary to install a module form the configuration description mechanism of PROTEL. These commands specify such information as the grouping of modules into processes and initialization control. They are similar in purpose to configuration languages such as C/MESA [LS 79].

The Library System

The PROTEL Library System (PLS) is used to control the storage, editing, compiling, linking and configuration of PROTEL sections, modules, subsystems and complete systems. It maintains a database describing all the components of one or more complete systems, the interconnection and ordering of these components, the section source code and the loader configuration commands.

The initial version of PLS was implemented in 1977. It was a single version system, roughly equivalent to the one prescribed for ADA in the ADA rationale [ADA 79]. While this system was better than none at all, we soon found ourselves unable to deal with on-going development in parallel with maintenance of previously released versions of the system. Multiple versions were 'supported' by creating complete copies of libraries and inserting common changes in multiple copies manually. This process has been referred to as 'forking' [HS 79] and is rightly looked upon with horror.

In a production environment, however, the need for parallel support of both old and new software is a fact of life. Therefore, a successor, PLS-II, was designed to simplify the handling of multiple versions of both source code and system structure. A tool called source manager, similar in spirit to SCCS [DHM 78], is used to simplify maintenance and reduce storage for nearly identical source versions. Compatible lineups of system versions are maintained in the PLS database which refers to the named issues kept by source manager. PLS-II is the subject of a forthcoming paper.

Experience

There is no doubt in our minds that modularity and inter-module type checking as supported by PROTEL are important and useful concepts. Many of the benefits of interface specification have been clearly explained by Parnas [Pa 78]. Practical experience in MESA [GMS 77, LS 79] has also substantiated these benefits.

Although these benefits are very real, on occasion we have had to overcome some difficulty in order to achieve them. In this section, we present the benefits we have experienced and the pitfalls we have overcome.

Interfaces as Documentation

Interfaces serve as a basis for module specification. The environment imported into a module is defined by the first statement of the module. The types, objects and operations exported by the module to other modules are defined in the interface. A number of recent languages, for example MODULA [Wi 77], EUCLID [LHLM 77] and CHILL [CHIL 80] provide facilities for modular programming but do not include interface definition features. Our experience with PROTEL indicates that its tools for interface specification are among the most important, if not the most important features of the language. We feel strongly that any language that excludes these features would be at a definite disadvantage for large system construction.

Also, in PROTEL, the full specification of the imported environment is restricted to the first statement of the module. The remainder of the module is prevented from further elaborating what is imported. This is contrasted with EUCLID [LHLM 77] where import declarations may occur at the procedure level and with ADA where the USE clause may occur anywhere in the code.

In practice, we have found the localization of import and export definitions to be important to clear design and readability. It allows the easy collection of interfaces for documentation purposes. It also permits easy maintenance of the module interconnection information kept by PLS since that information can be extracted automatically from the first statement of the module.

A general difficulty is the writing of a lucid description of the operations defined in a module's interface without outlining some of the implementation detail. We have discovered that, rather than relying on interface specifications, many programmers find it helpful not only to look at examples of a module's use but also to occasionally peek inside to see how it works. The separation of function and implementation is a skilled job which we support with standards and design reviews.

Modularity as a Project Manager's Tool

Modularity allows a large software project to be divided into a number of modules with well defined interfaces. This dividing process dictates a methodology which improves system structure. It also allows better allocation of work among a number of programmers, it lessens interaction problems and it permits parallel development once the interfaces are well defined.

However, due to the partitioning and hiding facilities provided by modularity we have found that, often, programmers tend to concern themselves with the specific part of the system in which they are directly involved. There is little motivation to cross module boundaries and delve into the implementation details of modules they did not write. As a result, the tasks of integration, testing and debugging must be a cooperative effort. To help with this, we have formed a

specialist team to assist the less experienced programmers.

Increased Reliability

By performing strict type checking on inter-module references, we have found that many of the errors which, in other languages, would have been detected at run time are flagged by the compiler. This permits easy repair early in the development process when bugs can be fixed cheaply. We have also found that the localization of the effects of an implementation error inside a module's body has allowed easier debugging with the security of knowledge that any changes are not likely to impact the rest of the system. The above has improved our ability to integrate separately compiled modules with a minimum of trouble. Our experience also echoes that of MESA [LS 79] in that we have found that a running system can undergo interface changes and, upon being successfully recompiled, it will run with the same reliability as before.

Organization within a Module

We have found multiple interfaces to be an extremely useful concept. They allow the separation of module functions by user. Thus, functions intended for general users may be separated from those intended for more 'privileged' users. For example, a file system may have a general user interface which supports operations such as OPEN, CLOSE, GET and PUT. A second, more privileged, interface may support operations which are only used by maintenance and audit software.

Multiple implementation sections, however, have not been as useful. Although they have often fulfilled their purpose in improving the internal structure of a module and in reducing recompilation, they have also permitted the development of overly large modules. Furthermore, they have allowed the merging of what would otherwise have been separate modules into a single entity.

In PROTEL, as in MESA, it is impossible to either nest modules within one another or to define more than one module per file. These restrictions do not exist in ADA. However, we feel the additional flexibility provided by ADA packages is unnecessary and overly complicates the language. We feel that control of access to modules is a problem best handled in a module interconnection language [DK 76, Ti 79] or library system and should not be a programming language feature.

Rather than increasing complexity by allowing module nesting as in ADA, our experience leads us to believe that even our current structure may be more complex than necessary. Instead, it may be sufficient to have a structure consisting of a linear chain of interface sections and a single implementation section. This would increase the efficiency of the support system since symbolic information would no longer need to be placed in implementation section object files thus eliminating the need for a linker.

Higher Level System Organization

We have come to appreciate that some interfaces are more important than others, and that there needs to be an explicit mechanism to make this clear. The collection of interfaces between an operating system and its base applications, for example, needs to be explicitly separated from interfaces that are intended to be internal to the operating system itself. As the number of interfaces grows larger, a monitoring and tracking mechanism, although essential, is just not adequate for design management. For this reason, it is convenient to group modules into subsystems where a subsystem corresponds to a configurable package of features. We found the identification of sets of modules which are to be treated as subsystems to be a necessary part of large system design, development and management.

In the early stages of the DMS project, we viewed the system as a homogeneous collection of subsystems. As our library of subsystems grew, its size increased to the point where a single library became unmanageable. We noted, however, that some parts of the system, while quite complex internally, interacted with the rest of the system via a narrow well defined interface or small set of interfaces. Eventually, the system was divided into a small number (currently six) of AREAS. As is the case with modules and subsystems, the USES relationship [La 79, Pa 78] between AREAS is hierarchical. AREAS are managed quite independently and may be developed on separate machines and possibly in different locations. New AREA releases are delivered to client programmer teams by distributing copies of the object files for the interfaces which are visible outside the area, copies of the module load files and copies of the subsystem configuration files.

Modularity as a Tool for Flexible Configuration

One of the requirements for DMS was the ability to tailor large numbers of slightly differing systems easily and inexpensively. This rules out any configuration scheme based on conditional compilation as the amount of time required to recompile a system of this size is large. Instead, we adopted a simple configuration process based on selection of subsystems from a common library. Invocation of optional software cannot be done explicitly as the name of an optional component would be unknown and treated as an error in systems from which the option is excluded. Instead, we invoke optional software via procedure variables. When an optional module is loaded, it "binds" itself (i.e. initializes procedure variables to point at its procedures) into the system and is subsequently invoked through these procedure variables. Operating system and language support is required to identify and execute module initialization code. It is interesting to note that ADA does not support procedure variables whereas software in both MESA and PROTEL makes extensive use of this feature.

A noteworthy special topic under the heading of flexible configurations is the use of multiple

module implementations: Since interface specifications allow a clear distinction between what a module provides and how it provides it, it is possible to design a single interface and several implementations for the same module. Any of these implementations can then be used transparently in different system configurations.

Module Qualification

In order to refer to an item imported from an external interface, the name of the item is used. Originally, PROTEL did not support a method for qualifying the name of the item with the name of the interface from which it was obtained. This sometimes reduced readability of code and also created a development difficulty: It was impossible for a module to use two interfaces which defined the same name since the ambiguity caused an error message. Such situations were not detected until a module which used the interfaces was created and compiled. We feel that it was a design error not to allow module qualification and have since corrected the situation. In our opinion, the lack of module qualification in other modular languages such as MODULA and CHILL is a serious shortcoming.

It should be noted that the use of constructs such as the Pascal WITH statement and the ADA USES clause raise similar problems in a more local context. Experience with an early version of PROTEL which supported a WITH statement was unsatisfactory and caused us to replace that construct by a BIND statement [LHLM 77] which allows local short-hand notations.

Type Transitivity

In PROTEL, the user of an interface has access to all type names which occur in that interface. Often, however, some of these types have been declared in another module. An example of this is the type of a procedure's formal parameter where the procedure is declared in an interface but the type of the parameter is defined in a different module.

This type transitivity results in two problems: first, since items can be imported transitively, the USES list of a module no longer represents an accurate account of the module's imports. Second, when a type definition in an interface is changed, such changes can ripple transitively through a system instead of being restricted to affecting only modules which directly use the interface. The result is added complexity in the support software to determine the true scope of an interface type change. How this can be done is discussed in the next section.

An alternative that avoids these problems is to ban type transitivity and prevent user modules from referencing names unless all interfaces defining these names are explicitly used. This, however, is not achieved without cost: the number of interfaces in a module's USES list will increase, sometimes significantly. The result is visibility of items which should be hidden and a rapid degeneration of the information distribution characteristics of a system. Further, the increased number of used items creates an unnecessary, and sometimes intolerable, imbedding overhead when the user module is compiled.

A theoretical basis for treating transitive types does not yet exist. However, our current view is that transitive types are the logically correct approach provided adequate library system support is available to deal with the ripple effect.

Change in a Modular Environment

By dividing a module into interfaces and implementations and hiding the latter, one obtains a measure of encapsulation. This allows modification of data structures and algorithms often to be transparent to other modules.

Further, since interface changes are of potentially greater consequence than implementation changes, a clear distinction between the two classes of change enhances the visibility of the former. The Protel Library System can then mechanically determine the potential impact of an interface change so that automatic recompilation can take place.

Impact of Implementation Changes

Although implementation sections are hidden from the outside world, there is a fundamental problem in ensuring that an implementation change has not altered the semantics of the interface, even when there is no direct change to the interface itself. The detailed implications of changes to all users requires careful consideration which is not encouraged by the knowledge that most changes really are well hidden and do not compromise system integrity.

Implementation changes sometimes require an interface change. Consider a data type that is defined in an interface,but whose details are refined in the implementation. PROTEL allows these details to be hidden from users of the interface, but if the implementation changes the size of such a type then user modules must be recompiled. This problem will also exist in languages such as ADA and MESA but is overcome in languages such as CLU which use an extra level of indirection for referencing data objects.

Impact of Interface Change

The MESA designers described the ease with which fundamental data structures could be reorganized with full confidence that any missed references to them will be flagged by the compiler [LS 79]. We also have found this to be the case. However, we have discovered a complimentary problem - since interfaces are so easy to change, everyone wants to do it.

Whenever a change is made to an interface, it is necessary to insure that the modules in the system are still compatible. To do this, currently

it is necessary to recompile all modules which can potentially be affected by the change to the interface. These not only include the immediate users of the interface but, also, any transitive users of these since they may be indirectly affected. For example, consider the interface segments below. If, for some reason, the type T1 is changed to, say, a half_int then the affect will be felt as far as interface d.

```
INTERFACE a;
TYPE t1 int;

INTERFACE b USES a;
TYPE t2 STRUCT....
          f1  t1,
        ... ENDSTRUCT;

INTERFACE c USES b;
TYPE t3 TABLE [1 to 100] OF t2;

INTERFACE d USES c;
DCL var t3;
```

In a large, highly connected system the amount of recompilation required for the "potentially affected" set is considerable and the expense is often intolerable. Although such a set is complete in the sense that its recompilation guarantees compatibility; in practice, it will contain many modules which are not "actually affected". Experience has shown that the ratio between potentially affected and actually affected modules is often high and that significant work can be saved by compiling only the actually affected set. The above problem has also been experienced by the MESA designers. We now believe we have a reasonable solution to the problem.

The scheme we have designed to determine the actually affected set is to compare the values of attributes which affect compatibility between this version of the interface and those obtained from the imbed file of the previous version of the interface. These attributes include items such as the type, offset and size of defined symbols. If these attributes match then the interface is upwards compatible and none of its users need be recompiled. If they do not match then all users of the interface need to be recompiled. The same procedure is applied recursively. Thus, little more than the affected modules are processed. We intend to implement this process later this year.

Building Systems

The process of releasing a system places a heavy load on the library support software and system construction tools. There is a major cycle of integrating changes, debugging and testing. For example, there is only a narrow window at the start of such a cycle when any interface changes are allowed. Changes to low level interfaces can result in several elapsed hours of recompilation. Compilation errors in other interfaces, resulting from the changes also require further time for system recompilation. This leads to a syndrome where "every bug costs a day". The problem is compounded by the fact that scheduling a recompilation is tantamount to an invitation to programmers to make additional interface changes that have been waiting in the wings. When a number of interface changes have been made, the odds are not in favour of a speedy system release. Typically, a new release requires several days spent thrashing final changes needed to produce a load, during which time no 'almost working' system appears to be present. To keep this troublesome problem under control the change management, design controls and support software must all be very strictly organized. We feel that the difficulty inherent in this process is preferable to the mammoth debugging sessions which are required when analogous changes are made to programs written in more loosely typed languages.

Configuration Control

In spite of all the above controls, it is still possible, in a highly dynamic environment, to configure an inconsistent set of modules. Such configurations may occur when, for example, a rapid fix is required and an interface is changed but not all modules affected by it are recompiled. Configuration of such an inconcistent set must be identified as an error. To check for such errors, we insert in each load file the module's version code and the version code of each module in its uses list. This information is readily obtainable since the compiler and linker run under control of the library system. A version code consists of two parts: A major number that is incremented when non-upwards compatible changes are made and a minor number incremented when upwards compatible changes are made. Many, but by no means all, interface changes are upwards compatible. The check used for consistency is as follows: when a module is loaded, the major number of each module in its USES list is compared with that of the version of the module already existing in the system. If they are not equal, the configuration is inconsistent. If they are, then compare the minor numbers. If the minor number in the USES list is greater than that of an existing module then it is possible that the module to be loaded relies on features not yet implemented in the existing module and the configuration is inconsistent.

Efficiency Problems

Probably the greatest problem we have encountered in type checking across module boundaries is the inefficiency of the process. The amount of information imported is often very large and it is not unusual for the compilation time of a 500 line section to be swamped by that required for inbedding. This inefficiency is seen as being due to three factors: First, the number of object files that need to be opened. Second, the amount of information copied from these files to the symbol table. Third, the large size of the symbol table which, in turn, leads to degeneration in paging characteristics. A number of ways of reducing this inefficiency have been found. Three of these are now presented.

AREA Interfaces

It was observed that, often, the same group of interfaces occurs in several USES lists. This will happen if, for example, several modules wish to use interfaces supporting the file system (FILESYS), the storage allocator (STORSYS) and the utility package (UTIL) of an operating system. Without AREA interfaces, compilation of such user modules would require that imbed files for all used interfaces be opened. An area interface allows the creation of a single "super" interface whose imbed file contains the same information as all the interfaces it USES. Thus, using an area interface OS which, in turn, uses interfaces FILESYS, STORSYS and UTIL is equivalent to directly using these interfaces. However, the number of files which need to be opened is reduced from three to one. Also, since compaction of multiple names is possible in areas, the amount of information in the area can be reduced.

We have found that area interfaces not only improve efficiency but, also, provide a powerful system structuring facility. However, sometimes, their usage has led to problems: some programmers, in order to avoid the tedium of writing lengthy USES lists, tend to use areas which contain many more interfaces than the ones they require. This leads to precisely the inefficiency area interfaces were introduced to combat.

Overall we believe that areas are a worthwhile feature but stress the need for rigorous control over their construction and usage.

Reimbed Feature

Although the unit of compilation is a section, it is not uncommon to compile several sections in a single invocation of the compiler. When this is done, it is normal that the compiler sections are closely related: for example, in the same module. Thus, they will tend to have highly similar import requirements. In such cases, it is possible to reduce the cost of inbedding by retaining part of the symbol table between sections. An example best illustrates the approach. Consider the three section headers below.

```
INTERFACE x USES a,b,c,d;
SECTION x1 OF x;
SECTION x2 OF x;
```

After x is compiled, its symbol table would appear as a stack containing, from bottom to top, the symbols of a,b,c,d and x. If the symbol table were reinitialized prior to compiling x1, then that compilation would build an identical table prior to adding its own symbols. It is straightforward to avoid this inbedding by simply retaining the symbol table between the compilations. When x2 is compiled, the situation is slightly more complex. The part of the symbol table pertaining to x1 is removed before x2 is compiled. The approach is easily generalized to cases where random parts, rather than the top of the table, are removed. However, it was felt that this would lead to severe fragmentation of the table and cause an increase in the already large amount of store required by the compiler.

Selective Inbedding

In the largest recorded case of inbedding 60 files were opened and 10,000 symbol definitions were entered into the symbol table. The total space occupied by that table was over 600 Kbytes. However, of the 10,000 entries in the table less than 1,000 were used in the compiled section. If the compiler had read only these symbols, the table would have required less than 60 Kbytes.

The above observation led us to develop a method which only enters into the table the symbols required by the section being compiled. This method exploits the fact that there is a clear division between the two passes of the PROTEL compiler. Pass1 parses the source and creates an abstract syntax tree representation of the input text. Pass2 performs semantic processing and outputs object code. In previous compilers, inbedding was performed prior to pass1 so that identifiers encountered in the source were immediately correlated with symbol table entries. The new strategy is summarized below.

1. Perform pass1. Any identifiers which are encountered but whose definition is not available are entered into the symbol table and marked as "not_yet_defined".
2. Perform inbedding. Open all required object files and scan them. If a symbol in a file matches one "to_be_defined", it is entered in the table in place of the undefined entry.
3. Perform pass2.

Inbedding in this strategy requires slightly more work than the previous, straightforward copying, version. However, this increase is offset by the improvement in paging characteristics in a virtual memory system due to the much smaller symbol table. Of more importance is the fact that PROTEL compilers may run efficiently on machines with a smaller amount of memory.

Conclusion

Our experience with PROTEL has confirmed that the benefits of using modular typed languages for system development are substantial. The major benefits we have experienced were in better control of project development and maintenance, improved capacity for change and system evolution, ease of configuring slightly different systems from common modules obtained from a system library and, most importantly, the detection of errors by the compiler at an early stage of system construction. We have also uncovered several difficult issues in the use of such languages for large system development. Three of these are the handling of transitive types, reducing the amount of recompilation required when interface changes are made and decreasing the cost of implementing type checking across module boundaries. We have outlined our solutions to these problems.

Acknowledgements

This paper represents the evolution of ideas and experience accumulated over many years by the members of the DMS design team. To all these, too numerous to mention here, the authors wish to express their gratitude.

Bibliography

[ADA 79] ADA Reference Manual and Rationale. ACM Sigplan Notices, Vol. 14 no 6 (June 1979).

[ANSI 66] ANSI X3.9 - 1966 (USA Standard Fortran).

[ANSI 75] ANSI X3.53 - 1975 (USA Draft Proposed Standard PL/1).

[CHIL 80] Proposal for a Recommendation for a CCITT High Level Programming Language (CHILL), Study group XI/3, February 1980.

[DHM 78] Dolotta, T.A., Haight, R.C., Mashey, J.R. "The Programmer's Workbench", Bell System Technical Journal, Vol. 57, No. 6, Part 2, July - August 1978, pp. 2177 - 2200.

[DK 76] DeRemer, F., Kron, H.H., "Programming-in-the-large Versus Programming-in-the-small", IEEE Transactions on Software Engineering SE-2 2 (June 1976), pp. 80-86.

[FJKM 79] Foxall, D.G., Joliat, M.J., Kamel, R.F., Miceli, J.J., "PROTEL: A High Level Language for Telephony", Proc. 3rd Intl. Computer Software & Applications Conf. (November 1979), pp. 193-197.

[GMS 77] Geschke, C.M., Morris, J.H., Satterthwaite, E.H., "Early Experience with Mesa", Comm. ACM, Vol. 20 no. 8 (August 1977), pp. 540-553.

[HS 79] Horsley, R.R., Lynch, W.C., "Pilot: A Software Engineering Case Study", Proc. Fourth Intl. Conf. on Software Engineering (September 1979), pp. 94-99.

[ICC 78] Proc. IEEE Intl. Conf. on Communications 1978, Vol II, Session 32, (5 papers on DMS-100).

[La 79] Lasker, D.M., "Module Structure in an Evolving Family of Real Time Systems", Proc. Fourth Intl. Conf. on Software Engineering (September 1979), pp. 22-28.

[LHLM 77] Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., Popek, G.J., "Report on the Programming Language EUCLID", ACM Sigplan Notices, February 1977.

[LS 79] Lauer, H.C., Satterthwaite, E.H., "The Impact of Mesa on System Design", Proc. Fourth Intl. Conf. on Software Engineering (September 1979), pp. 174-182.

[LSAC 77] Liskov, B., Snyder, A., Atkinson, R., Schaffert, C., "Abstraction Mechanisms in CLU", Comm. ACM, Vol. 20 no.8 (August 1977); pp. 564-576.

[Pa 78] Parnas, D.L., "Designing Software for Ease of Extension and Contraction", Proc. Third Intl. Conf. on Software Engineering (May 1978).

[Pe 66] Perstein, M.H., The Jovial (J3) Grammar and Lexicon, SDC Technical Report TM-555 (1966).

[Ti 79] Tichy, W.F., "Software Development Control Based on Module Interconnection", Proc. Fourth Intl. Conf. on Software Engineering (September 1979), pp. 29-41.

[Wi 71] Wirth, N., "The Programming Language PASCAL", Acta Informatica, 1, 1971, pp. 35-63.

[Wi 77] Wirth, N., "Modula: A Programming Language for Modular Multiprocessing", Software Practice and Experience, Vol. 7 (June 1977).