An Efficient Separate Compilation Strategy for Very Large Programs

Andres Rudmik
Barbara G. Moore

GTE Laboratories
40 Sylvan Road
Waltham, Massachusetts 02254

Abstract

This paper describes the design of a compiling system that supports the efficient compilation of very large programs. The system consists of front ends for different languages, a common program database to store the intermediate code, and various back ends, optimizers, debuggers and other development tools. The compiling system achieves efficiency of use by minimizing the number of system components that must be invoked when a small change is made in a program.

A new separate compilation strategy is presented that is both easy and natural to use and does not require language extensions for its use. The database provides the necessary contextual information to support separate compilation and to facilitate complete compile-time checking. Also, the use of this database affords a unique opportunity to reduce substantially the cost of recompilation and to support an efficient source patching facility.

Introduction

The CHILL Compiling System (CCS) is an integrated set of tools designed to support the development and maintenance of very large programs such as those typical of telecommunications applications, i.e., of the order of millions of lines of source code. Some of the requirements of a large scale programming environment that the CCS identifies and addresses are:

- the separate compilation of progam units in an indeterminate sequence with complete compile time interface checking;
- an efficient method for processing small changes in very large programs;
- a system of support tools that makes use of the results of analysis performed during compilation;
- a configuration control system implying the existence of multiple versions of the program and of its components.

Implicit in these and all other requirements for the system is the overriding requirement that it should operate with an economy of time and space attuned to the magnitude of the programs under development. Furthermore, in order to extend the usefulness of the system over the broadest possible range of applications, it was required to have the flexibility to support an entire family of block structured languages, and to adapt to a variety of target and host machines.

The architecture of the CCS is illustrated in Figure 1. The hub of the system is a program database, which stores the intermediate code produced by a front end and makes it available to a series of machine-specific back ends as well as to other development tools. The front end and intermediate code technology were chosen to support the processing of CHILL (the standard international high level language for telecommunications operations) (1), several dialects of Pascal, and Ada. The CCS currently supports both CHILL and Pascal and there are plans to add Ada in the near future. The back ends implemented include code generators for IBM-370, TANDEM, VAX 11/780, I8086, and the new Intel microprocessor, I432.

The central database is the most innovative component of the CCS, making possible the system's unique approach to separate compilation and serving to minimize and manage the tasks of all the other interfacing components. Of the many functions of the database, support for separate compilation was selected as the most critical, and consequently has been most influential in shaping the system design.

Strategy for Separate Compilation

Separate compilation of program parts was identified as a key issue and selected as a starting point for structuring the database design for several reasons:

- In the environment of large scale program development, it is essential to formalize the communications between humanly managed modules so that accurate module interconnections can be enforced throughout the development process. This task of maintaining inter-module consistency would be the responsibility of the database.
- Not all the languages envisioned would, like Ada, have built into their syntax a means of designating the program units destined for separate compilation. It was desired to formulate a generalized method of permitting any block structured language to communicate to the database in its own syntax the structure of a program viewed as a collection of separately compilable modules.
- Some of the most critical factors that would determine the overall performance of the system are linked to separate compilation, in particular, the system's response to a request for recompilation of a randomly selected unit.

The literature attests to the fact that a satisfactory solution to the problems posed by separate compilation is difficult to find. Three basically different approaches emerge from a survey of published implementations:

1. The task is viewed primarily as a responsibility of the link editor; this view requires that the link editor undergo promotion to a high level tool and perhaps have a module interconnection language of its own (2,3). Among the disadvantages of this approach is the fact that mismatches at the interface level go unnoticed by the compiler and are detected only after the cost of generating object code has been incurred. For large programs, which require considerable time for the code generation phase of compilation, this approach is unrealistic.
2. A module interconnection scheme is specified that restricts separate compilation to modules defined at the global level, effectively masking the block structuring character of the language (4,5,6). There are some interesting arguments in support of this kind of solution which deserve to be weighed as we gain experience in the use of the CHILL Compiling System (7,8). First, it is argued that the packaging of data exchanged or shared among sibling modules at the global level

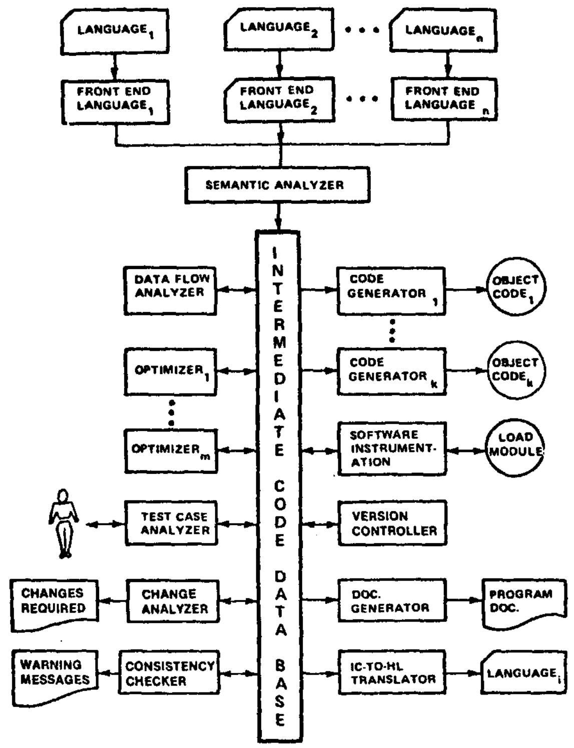

Figure 1. The CHILL Compiling System

introduces visibility constraints that provide benefits roughly equivalent to those of block structuring. Second, these authors believe that the monolithic, rather than the hierarchical view of a program is the more intuitive one, and in a language that offers both possibilities (such as Ada or CHILL), this view is likely to become the programmer's choice. The third argument defending this approach is one which the CHILL Compiling System addresses throughout the course of its design: in a block structured program, the recompilation of top level modules can force the recompilation of nested modules to an extent that in the worst case cancels out the advantages of conserving the output of previous compilations (8).

3. The third approach to separate compilation is one that both supports compile-time interface checking and upholds the block structured character of the Pascal-like languages. However, when based on modifications of traditional compilers, published implementations of this approach revealed undesirable limitations (9). Some implementations impose a partial ordering upon the compilation schedule, a side-effect of an on-the-fly visibility determination strategy. Some add so many language extensions that they draw criticism for the obscurity and extra

complexity they introduce into Pascal (8). In this respect, it might be observed that at least in some cases, the obscurity results from attempting to press mechanisms for separate compilation into support for a makeshift data abstraction construct. Yet, without this packaging capability, block-structured languages remain plagued with latent obstacles to proper interface checking (9).

In orienting our design, we believed that the introduction of a database, as well as the presence of data abstraction capabilities in the new languages (CHILL, Ada, and the GTE Laboratories' (GTEL) dialect of Pascal), would overcome the difficulties cited in the third approach. Thus we decided not to introduce any extraneous restrictions or non-standard syntax into the subscriber languages. Instead, we proposed to allow each language to support two "modes" of compilation: the first, called STRUCTURE mode, would communicate to the database the intended program skeleton, naming the separately compilable components and defining their structural relations; the second, called PROGRAM mode, would position the database key to a selected unit in the skeleton, and, after processing the source in the usual way, store the output of front end analysis in the database in the custody of the selected compilation unit key. Compilation in STRUCTURE mode could be repeated indefinitely during the course of program development; it would support program expansion in a manner homomorphic to design by stepwise refinement.

This strategy places no limitations on the order of compilation or on the semantics of the chosen partitioning. While we recognized that the most suitable unit of compilation is the data abstraction construct (the PACKAGE in Ada, the MODULE in CHILL and in the GTEL dialect of Pascal), we elected to allow the PROCEDURE also to become a unit of compilation, thus encouraging algorithmic portions of the program to be isolated and introduced into the public arena as well. The following is an introduction and illustration of the method.

Separate Compilation: An Example

The user first compiles a skeleton of the program containing only headings for the modules, regions, procedures and processes that are expected to become units of compilation. This is called a STRUCTURE mode compilation; it results in the initialization in the database of an undecorated program tree called the Structure Directory. To compile any program unit, the user must specify a unit name that matches the name of a preexisting node of the Structure Directory tree. Figure 2a illustrates the input to a CHILL compilation in STRUCTURE mode, and Figure 2b shows an abbreviated listing of the corresponding Structure Directory constructed from the input. The actual listing reports the status of each compilation unit, (initially, "uncompiled"), and assigns sequence numbers to the units, which can later be used to resolve any ambiguity of names.

```
/STRUCTURE/

program: MODULE
   global_procedures: MODULE
   END global_procedures;
   transformations: MODULE
      table_builder: MODULE
      END table_builder;
      interpreter: MODULE
      END interpreter;
   END transformations;
   input_analyzer: MODULE
   END input_analyzer;
   main_driver: PROCEDURE
   END main_driver;
END program.
```

Figure 2a.

```
/LIST STRUCTURE DIRECTORY/

1.0 MODULE program;
   2.0 MODULE global_procedure;
   2.1 MODULE transformations;
      3.0 MODULE table_builder;
      3.1 MODULE interpreter;
   2.2 MODULE input_analyzer;
   2.3 PROCEDURE main_driver();
```

Figure 2b.

Suppose that the first unit scheduled for development is "interpreter", which contains some key procedures requiring the use of global data. The programmer may wish to begin by creating the environment for the target module. Therefore, an initial version of "global_procedures" in which only data declarations appear is added to the program database. To accomplish this, an invocation of the compiler in PROGRAM mode (i.e., the normal mode of compilation, given the existence of a Structure Directory) specifies "global_procedures" as the unit of compilation.

Next, the Structure Directory tree is extended to incorporate the "interpreter" (Figure 3a). A partial listing that results from a new compilation in STRUCTURE mode is presented in Figure 3b.

It is now possible to select "stack_handler" and compile the code for it in PROGRAM mode. A second programmer can be working simultaneously on "translate", which perhaps requires the use of some procedure to be declared in "stack_handler". In agreement with CHILL semantics, the compiler allows code to contain references to as yet undeclared procedures, assuming the responsibility for resolving them as the declarations appear. Thus, separate compilation of "translate" can proceed, since, via a strategy described in a later section, the CCS instructs the database to store the information necessary to complete the compilation and will issue only a warning at this point.

```
/STRUCTURE/

interpreter: MODULE
   output_formatter: MODULE
   END output_formatter;
   file_handler: MODULE
      get_data: PROCEDURE()
      <> ALIAS get_data_1;
      END get_data;
   END file_handler;
   translate: MODULE
      stack_handler: MODULE;
      END stack_handler;
      get_data: PROCEDURE();
      <> ALIAS get_data-2;
      END get_data;
   END translate;
END interpreter;
```

Figure 3a.

```
/LIST STRUCTURE DIRECTORY/

3.1 MODULE interpreter;
   3.2 MODULE output_formatter;
   3.3 MODULE file_handler;
      4.0 PROCEDURE get_data();
          ALIAS get_data_1;
   3.4 MODULE translate;
      4.1 MODULE stack_handler;
      4.2 PROCEDURE get_data();
          ALIAS get_data_2;
```

Figure 3b.

At some stage in the program development, an individual programmer may wish to stabilize his private view of the ensemble while he proceeds within his selected environment undisturbed by the interactions of other team members. Furthermore, the programmer may need a temporary version of the program to save the output of his compilations until he makes the decision to promote it to a permanent version. To implement these concepts the compiler recognizes three classes of versions: a permanent and official version under the control of the chief programmer, called the "baseline" version; a temporary version, which is the default version for output, called the "test" version; and a set of programmer-initiated versions which represent filtered views of the program. A separate tool, the Version Controller, is invoked to create versions, and to add or delete compilation unit members of a version. A password protects the baseline version, and may also be applied optionally to any programmer-initiated version; in this way, only an authorized user can alter a password-protected version. In the examples we have seen, it would be possible for the developer of "global-procedures" to promote to the baseline version his compilation unit containing data declarations, and subsequently to recompile his unit to the "test" version or to a private version after having added procedures that are still in the test phase. Thus, changes in the status of "global_procedures" would not have an impact on compilations that need only to see the baseline version.

Database Designed for Efficiency

The database is a special purpose implementation designed expressly to meet the needs of the compiler in a natural and direct manner. Thus, the Structure Directory forms its hub, organizing it as a hierarchy of compilation units, each serving as a primary key. Nodes of the Structure Directory fan out into version nodes, storing the mapping information needed to fetch version-dependent entities; queries reach the target node by specifying a compilation unit key qualified by a version. For efficiency of access, the system generates a unique surrogate key for each compilation unit, reached by a hashing scheme that makes use of the source compilation unit name and/or the unique user-supplied alias.

The principal entities stored in the database under compilation unit key are a symbol table segment for that unit and a set of intermediate code trees that represent the procedural portions of the unit and denote operands by referencing the symbol table. Also stored in the database independent of compilation unit key is an identifier table, which stores all names referenced in the symbol table.

While any system that strives to maintain consistent separate compilation must store complete symbol table information relevant to the context for a given unit, our decision to store the intermediate code in addition to symbolic information was an innovation. This choice was made for the sake of the interfacing tools, in order to eliminate restrictions on their capabilities and to economize the lengthiest phase of the processing they must do, i.e., analysis of the source. Thus, since the intermediate code for the entire program is available from the database, the data flow analyzer can perform meaningful interprocedural flow analysis. Furthermore, a substantial advantage is won if the data flow analyzer, optimizers, and so on, can operate directly on the predigested intermediate code, which is generated only once and lends itself easily to manipulation and modification. The time savings is particularly important to the code generation phase of compilation, which can be deferred under this plan until the moment of actual need, and is then accomplished in a fraction of the time that the source analysis has consumed.

These goals in the time dimension of the system's efficiency demanded that a proportionate amount of attention be paid to the constraints of the compiler's space resources. A guideline applied to the design of system-based data structures was that their total size should not exceed the size of the source code. It is not the intent of this paper to delineate the principal data structures; a previous paper describes in detail the design of the intermediate code (10). However, some general comments might serve to communicate the flavor of this design approach.

The intermediate code is modeled on a high level data flow graph implemented as a forest of spanning trees that capture the program's abstract syntax. The database views both the intermediate code and the symbol table macroscopically as trees whose leaves sometimes explode into linked lists. Within the compiler is a common facility for managing linked lists, and by extension of this same capability, for traversing trees; this facility is at the service of the symbol table, the intermediate code, and of the Structure Directory tree of database keys. A distinctive feature of the compiler's list processor is that it is capable of viewing the structures it manages not only as collections of "atoms," e.g., representations of individual symbols or code phrases, but also as "molecular" structures, i.e., groups of atoms which may be organized in a fixed variety of shapes and may be manipulated globally at a level adapted to the I/O requirements of the database. For example, an intermediate code tree corresponding to a procedure block can be retrieved from the database by a single operation. This strategy not only streamlines the compiler space requirements by substantially reducing the number of pointers that must be maintained at the atomic level; it also improves the time performance of the database by enabling more efficient I/O management.

To help prevent database I/O from becoming a performance bottleneck, the compiler implementation language (GTEL Pascal) has been equipped with an explicit heap facility that builds linked data structures in a position-independent manner and allows the entire contents of a heap to be written to or read from the database files as a stream of buffers. The manipulation of pointers and heap objects is as in standard Pascal with the additional requirement that pointers be bound to a user defined heap. Currently under development is an additional facility for "virtual heaps" that will support linked structures whose size exceeds the addressable region available to the compiler. Each compiler module that presides over a complex data type has the ability to invoke the heap facilities to handle its I/O in an efficient, type-safe manner.

A critical task of the database is to compose from individual segments the entire symbol table context for a selected unit, and to construct for the segment undergoing compilation the links required to insert the segment into the context. Thus, symbols are not stored redundantly in the database as components of the various contexts of individual compilation units; instead, the database saves only enough information to enable the context of a unit to be reconstituted as needed. Support for this strategy is provided by a hashing scheme that enables the internal symbol table references to be independent of the physical position of a symbol in its home segment. Such a scheme is also helpful in reducing the amount of repair work the compiler must do when a unit is recompiled, as the next section will point out.

The compiler exploits in many ways the advantages of preserving in the intermediate code all the semantic information contained in the source, including references to as yet undefined procedures, variables, types and constants. Thus, the front end is able to recognize any correspondence between previously detected unresolved procedure references and new tree decorations appearing as a result of the current compilation. It can then make appropriate changes to previously generated intermediate code, repairing dangling pointers. While this scheme was devised primarily to implement the semantic rule of CHILL which allows forward invocation of procedures, it is also suitable for supporting the special case arising in a separate compilation setting where unresolved references to variables or procedures are presumed resolvable in some as yet uncompiled unit. A warning message apprises the user of the existence of such a reference; the same service is offered with respect to procedure and data definitions that are never referenced.

The security of protected entities is a natural responsibility of the database, and is easily implemented using standard database technology. Thus, database operations require authorization for modification of the Structure Directory, as well as for changes to the baseline version or to any version created with a password. In addition, the database mediates between contending concurrent requests that attempt to access the same version. While assembling the program context of a given version, the database applies a mutual exclusion protocol, deterring other users from making updates to that version. Conversely, it excludes other users from either reading from or writing to a version which is in the process of being updated.

Strategies for Efficient Recompilation

As our literature survey attests, the criticism of separate compilation implementations that pleads for the abandonment of block structure cites the proliferation of disorder in a program when a single unit is recompiled. Since in CHILL, visibility of encapsulated objects can be extended not only to nested units, but by the GRANT clause to enclosing modules as well, the problem becomes even more significant to the CCS. We have introduced a variety of approaches to this problem, while still allowing the order of compilations to be independent of compiler implementation considerations. The system considers the various degrees of impact as separate cases, each requiring a more expensive level of intervention than its predecessor, and each implying that the lower level interventions must automatically ensue. These cost levels are as follows:

1. regeneration of object code for an affected unit, with a resulting patch to its load module;

2. reanalysis of the intermediate code for an affected unit;

3. recompilation of source.

There are two system tools that detect recompilation side effects and trigger the approprite repair mechanisms: the Change Analyzer, which is automatically activated by the front end when a unit is recompiled within the same version, and the Consistency Checker, which the user invokes separately. What follows is a description of how these tools operate.

Change Analyzer. As a new symbol table is being constructed for the unit requesting recompilation, the Change Analyzer automatically compares each generated symbol to any existing symbol of the same name and visibility. Changes in the meaning of symbols are detected, as well as deletions of symbols from the predecessor table; the appropriate action for restoring consistency is then selected and set in motion. The following case is offered as an illustrative example. Suppose a dominant module A is recompiled with a change in the definition of its type, T1, imported by dependent module B. The Change Analyzer discovers the change, and, using a pointer to the cross reference list for T1, retrieves the list of T1's dependencies. The affected module B is thus identified, and a determination is made as to whether B must be recompiled from source, (if, for instance, B makes an incompatible assignment to an object of the new type T1) or perhaps requires only a regeneration of its code (if, for instance, the modification of T1 concerns only an extension of its range). In both cases, a message is issued to the user; the first action results in a change to the database entry for the status of module B: from "compiled" to "requires modification"; the second action is completed automatically by the CCS. It can be observed that the recompilation of module A has no impact at all on dependent units that import no symbols whose meaning is changed. This is because, as mentioned in the previous section, each unit references its external symbols in a data-independent manner, i.e., independently of the symbol's physical location, which is subject to change by recompilation.

Consistency Checker. This is a tool which the user must invoke to review the status of the whole program. Since the CCS will refuse to generate object code for an inconsistent program, this tool would normally be invoked in the sequence of user commands that seeks to activate the compiler back end. It should also be invoked before merging versions; e.g., before merging a private version with the baseline version. This tool flushes out inconsistencies of all kinds: those that have resulted from recompilation, as well as those that arise from references to unknown variables or procedures. In addition to notifying the user of the inconsistencies that require his intervention, this tool also effects whatever automatic adjustments and patches are appropriate.

Benefits From The Design

The provision of a totally supportive database that allows separate compilation to take place without forcing a monolithic structure or requiring semantic restrictions brings to the CCS many extra benefits that come as free or low cost bonuses. Some of the benefits are:

1. Design support. Although the CCS does not actually impose a design methodology, the embodiment of the program skeleton within the Structure Directory forces the design to become public and encourages it to grow by stepwise refinement. At the same time, by permitting compilations to germinate at any node of the Structure Directory tree, the CCS allows bottom-up experimentation to take place, either within an official "baseline" version, or in a less controlled way, in a programmer-initiated version.

2. Good communication. Since the CCS tools for program documentation are also oriented by the Structure Directory, it is a simple matter to project on to its listing additional kinds of information that synthesize a type of Module Interconnection Language. These may include, for example,

   a) Program static structure.

   b) Descriptions of module data abstractions together with the operations they define.

   c) The uses relation between modules.

   d) The list of modules undergoing side effects after the recompilation of a dominant module.

   e) The table of symbols visible at any selected node of the program tree.

   f) The names and parameter types of the undefined procedures invoked in each module.

3. Version control. The database supports version control of the intermediate code, allowing diverse views of the program under development to be defined and integrated into any given configuration control environment.

4. Program consistency. Working from the database, the Consistency Checker automatically ensures that as private versions are reintegrated into the baseline version, any lingering inconsistencies are flushed out.

5. Program. database. Storing the intermediate code in the database offers countless possibilities for improvement of

the services offered by auxiliary CCS tools. For example, data flow information for the use of optimizers is most conveniently extracted from intermediate code trees and stored in the database as parts of the same relation (10).

6. Improved efficiency. The database-anchored architecture frees the back end from awkward dependencies on the structure of the source, thus enabling the code-generators to achieve maximal flexibility (11). Furthermore, by invoking separate code space and execution time optimizer tools, the user can select an optimal level of upgraded code, thereby deferring some of the passes over the intermediate code until the point of usefulness.

Conclusion

The CCS is self-compiling in GTEL Pascal. It has been bootstrapped to run on the IBM 370, TANDEM, and Vax 11/780; other implemented back ends include code generators for the I8086 and I432.

The database containing complete symbolic information and intermediate code occupies file space equivalent to 80% of the source on the IBM 370. A further 25% reduction in space is expected to be achieved through the use of simple data compaction techniques.

Currently, the front end processes 40,000 lines per CPU minute on the IBM 3033. After the optimization tools have been completed and put to use, this speed is expected to double. The CCS back ends typically generate code in one-half the time consumed by the front end.

Bibliography

1. "Proposal for a recommendation for a CCITT high level programming language," CCITT Study Group XI, Brown Document, 1979.

2. Hamlet, R.G., "High-level binding with low-level linkers," Comm. ACM, vol.19, 1976, pp. 642-644.

3. Keiburg, R.B., W. Barabash and C.R. Hill, "A Type-checking linkage system for Pascal" Proceedings 3rd International Conference on Software Engineering, 1978, pp. 23-28.

4. Celentano, A., P. Della Vigna, C. Ghezzi and D. Mandrioli, "Separate Compilation and Partial Specification in Pascal," IEEE Trans on Software Eng., vol.SE-6, 1980, pp. 320-328.

5. Hanson, D.R., "A simple technique for controlled communications among separately compiled modules," Software-Practice and Experience, vol.9, 1979, pp. 921-924.

6. LeBlanc, R.J., and C.N. Fisher, "On implementing separate compilation in block-structured language," Sigplan Notices, Vol. 14, 1979, pp 139-143.

7. Clarke, L.A., Wiledon, J.C. and Wolf, A.L. "Nesting in Ada Programs is for the Birds," Sigplan Notices, vol. 15, 1980, pp. 139-145.

8. Hanson, D.R., "Is Block Structure Necessary?", Software Practice and Experience, vol.11, 1981, pp. 853-866.

9. Oldehoeft, R.R., W.D. Ralph and M.H. Tindall, "An Interactive Manager for Pascal Software," Software-Practice and Experience, vol.11, 1981, pp. 867-873.

10. Rudmik, A. and E.S. Lee, "Compiler Design for Efficient Code Generation and Program Optimization," Sigplan Notices, vol. 14, 1979, pp.127-138.

11. Waite, W.M., and L.R. Carter, "An Analysis/Synthesis Interface for Pascal Compilers," Software Practice and Experience, vol.11, 1981, pp. 769-787.