Softwaro Dovolopmcnt Control Based on Module Interconnection

Walter F. Tichy
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

Constructing large software systems is not merely a matter of *programming*, but also a matter of *communication and coordination.* Problems arise because many people work on a joint project and use each other's programs. This paper presents an integrated development and maintenance system that provides a controlling environment to insure the consistency of a software system at the module interconnection level. It assists the programmers with two facilities: Interface control and version control. Interface control establishes consistent interfaces between software modules and maintains this consistency when changes are made. Version control coordinates the generation and integration of the various versions and configurations. Both facilities derive their information from an overall system description formulated in the module interconnection language INTERCOL. A demonstration system is sketched that runs under the UNIX time-sharing system.

1. Introduction

Constructing a large software system is not merely a matter of *programming*, but also a matter of *communication and coordination.* Communication and coordination problems arise because of the complexity and size of software systems. Size makes it necessary to work with many people rather than with a single programmer or a small team[1]. This introduces all the difficulties inherent to human interaction. With many people working on numerous, interrelated parts, it becomes extremely difficult to ensure the consistency of these parts. In addition, the asynchronous nature of the development activities and the mass of information involved quickly exhaust human capabilities to retain a clear picture of the actual development stage of the system. As another example for the need of communication and coordination consider the phenomenon of continuing change[2]: All large and widely used systems are subject to an endless stream of corrections, improvements, customizations, and diversifications, because it is usually cheaper to modify them than to rebuild them from scratch. However, continuing modification means continuing communication of the overall system structure, design decisions, and implementation details to the modifiers, and continuing control to avoid (or slow down) the deterioration of the system into unfixable unstructuredness.

Current research in programming methodology, programming languages, specification and verification techniques takes a different view: The main goal is to *reduce* the complexity and size of software systems to manageable dimensions. However, no matter how successful these approaches are, computer applications are subject to the Parkinsonian law that their scale will expand to meet the limits of technology. Thus, we shall always be faced with building large, complex systems and shall have to solve the communication and coordination problems inherent to their development.

This paper presents the prototype of an integrated software development and maintenance system, whose goal it is to control the development of large software products by addressing the communication and coordination problems outlined above. Its facilities can be viewed as guaranteeing the integrity of a software system at the module interconnection level[3]. At this level, a software system appears as an organized collection of modules which interact with each other through interfaces. Our system provides two facilities at this level: *Interface control* and *version control*. Interface control establishes consistent interfaces between modules and maintains this consistency when changes are made. Version control performs version selection and automatic (sub-)system generation (or regeneration) in a multi-version/multi-configuration system.

Both interface control and version control derive their information from an overall system description, formulated in the module interconnection language INTERCOL. Such a description specifies a software system at the module interconnection level. It indicates how the various modules and their versions are grouped into (sub-)systems and how they interface with each other. An INTERCOL specification is separate from actual program code.

Although INTERCOL[4] is not a subject of this paper, we need to introduce a few of its notions in the following section. In section 3, we present a careful analysis of the mechanisms for inter-compilation type-checking, because type-checking across compilation units is the most important facility of interface control. Then we discuss our methods of interface control and version control in sections 4 and 5, respectively. Finally, a pilot implementation running under the UNIX time-sharing system is described in section 6.

2. Terminology

A *resource* is any entity that can be declared and named in a programming language (e.g., variables, constants, procedures, generics, type-definitions, etc.). A *subresource* is a resource that is part of another resource (e.g., a field of a record, an operation of an abstract data type, etc.). The *type* of a resource determines the ways in which the resource may be used in a program. *Type-checking* verifies that the resource is used in only those ways. Type-checking is done by inspecting the program, rather than by executing it.*

A *module* consists of program text, documentation text, test data, and status information. An *exported* or *provided resource* of a module is a resource that is declared in the program text of the module and may be used in the program text of other modules. An *internal resource* of a module is declared in the program text of that module, but may not be used by another module. An *imported* or *required resource* of a module is one that is not declared in the module, but may be used in it. The set of a module's provided resources together with the set of its required resources constitutes the *interface* of that module.

A *system* consists of a collection of components, viz. modules or other systems, plus documentation text, test data, and status information. Thus, a difference between systems and modules is that there is no program text associated with a system. A system also has an interface of provided and required resources. However, since there is no program text, a system's provided and required resources are in turn provided and required by its components.

3. Inter-Compilation Type-Checking

Type-checking verifies the type-correctness of programs. We would like to stress that we consider type-checking across module boundaries to be more important than type-checking inside a module, for the following reasons. We assume that a module is the responsibility of a single programmer or of a small, closely interacting team. In order to implement a module's provided resources, the programmers will usually declare some other, more primitive internal resources. One can assume that the programmers rarely make type-errors in using these resources, since they are dealing with their own creations. However, programmers working on different modules cannot interact as closely. In

*We do not wish to give a more specific definition of the latter two terms, in order to keep the discussion independent of a particular programming language. Clearly, we have 'algolic' languages in mind, like PASCAL, ALPHARD, or ADA.

particular, the implementors of a resource exported from a module are not identical with the programmers using them. Experience shows that misunderstandings at module interfaces are the rule rather than the exception. We believe that type-checking can detect many of these errors. The value of type-checking across module boundaries has been confirmed with the use of the MESA-system[5, 6].

There is a small number of language systems which provide for inter-compilation type-checking, most notably SIMULA67[7], PLISS[8], and MESA[9]. The following paragraphs classify the various techniques that they use. The classification scheme is based on the observation that the strategy in which compilation (i.e., code generation) is performed can be decoupled from the strategy in which interfaces are checked. The compilation strategies we wish to distinguish are as follows:

1. Monolithic compilation, i.e., no separate compilation;

2. Incremental compilation, i.e., compilation units are processed according to some partial ordering, such as bottom-up;

3. Independent compilation, i.e., compilation units may be processed in any order.

Inter-compilation type-checking can be categorized using the same attributes:

1. Monolithic type-checking, i.e., no type-checking across compilations;

2. Incremental type-checking, i.e., the interfaces of the compilation units are checked according to some partial ordering, such as bottom-up;

3. Independent type-checking, i.e., the interfaces of compilation units may be checked in any order.

Combining the two categorizations results in an array of nine possibilities, as shown in Fig. 1. We have entered a few example systems into the grid.

We shall discuss each column of the array in the following paragraphs.

type-check / compilation	monolithic	incremental	independent
monolithic	1,1 PASCAL		
incremental	2,1 FORTRAN PL/I libraries	SIMULA67 ALGOL68C	
independent	3,1 FORTRAN PL/I	PLISS type-checking linkers	CLU-library MESA

Figure 1: Inter-compilation type-checking

3.1. Monolithic Type-Checking

Any language system that does not provide for separate compilation fits into square [1,1] (monolithic compilation, monolithic type-checking). The square [3,1] is the most densely populated one, since most language systems today offer separate compilation, but pass the responsibility for inter-compilation type-checking on to the programmers. These systems do not offer any more help even if one restricts the development to an incremental, bottom-up order (square [2,1]), in which the necessary information for type-checking is always available. For example, ALGOL-libraries are compiled before they are used, so that the types of the library entries are known. However, it is still the responsibility of the programmer to make sure he is using the library routines correctly.

3.2. Incremental Type-Checking

Let us first look at square [2,2], incremental compilation and type-checking, which is applied in

SIMULA67[7] and Algol68C[10]. Compilation of a module M that, for instance, defines a class C, results in object code and symbol table information. The symbol table information is stored in an auxiliary file and can be used by the compiler at a later time. When a module N is compiled which contains an external declaration of class C, the type specifications describing class C are read into the symbol table for module N so that full interface checking can be performed (see Fig. 2). Thus, a bottom-up compilation order must be observed. The Algol68C system has a similar arrangement, except that it is best used with a top-down processing-order: The separately compilable units are nested blocks, which can be inserted at a later time. This is implemented by transmitting symbol-table information from outer blocks to inner blocks through auxiliary files.

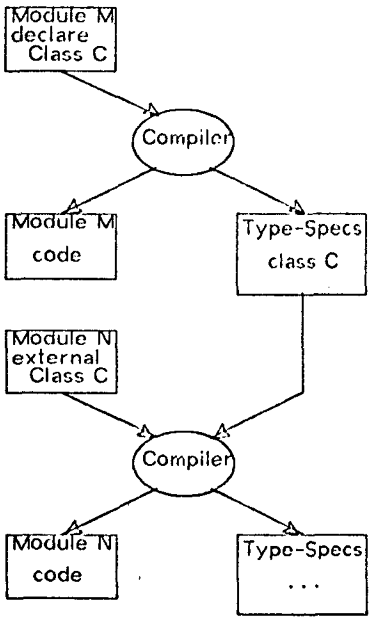

Figure 2: Incremental compilation / type-checking

Refining the idea of the extracted symbol table information leads to a technique that allows compilation in any order, but performs the type-checking in an incremental fashion (square[3,2]). It differs from the incremental compilation order in that a resource may be used before its type is in the auxiliary file. In that case the type of that resource is derived from the usage site (if possible) and tentatively entered into the auxiliary file. It will later be matched with the actual type (see Fig. 3). The PLISS-system[8], based on PL/I, uses that approach. A disadvantage is that the type-checking of a module's interface may be delayed until long after the time the module is compiled.

There are some systems that allow for independent compilation, but perform the type-checking across compilation units at link-time. This is an extreme case of incremental checking. It has the serious disadvantage of delaying the checking until integration time, long after the programmer is done with implementation.

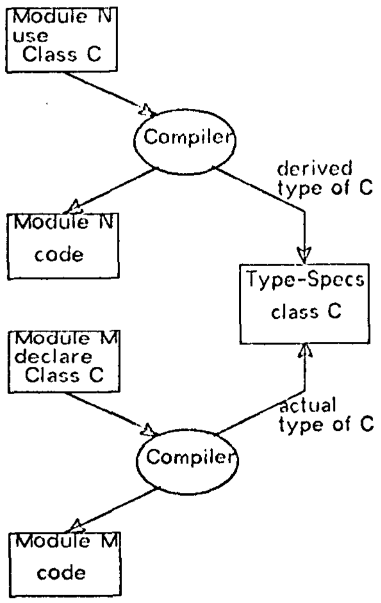

Figure 3: Independent compilation, increm. check

3.3. Independent Type-Checking

Let us consider square [3,3], fully independent compilation and type-checking. The characteristic of independent type-checking is that the interfaces between the modules are explicit. They are designed in the form of type-specifications and entered into a data base before any program code is written. That may seem like an undue restriction at first sight, but in fact it is the only way programmers can start writing separate modules: If there were no precise specifications of the interfaces, they could not program with each other's resources. With the interfaces in machine-readable form, it is a straight-forward matter for the compiler to pick up the necessary information before checking a module. In the MESA-system[9] for instance, the data-base of type-definitions consists of a set of *definitions files*. Each of these files contains the resources provided by a particular module in the form of type-specifications. The interface of a module is described in terms of definitions files: One for the provided resources, and zero or more for the required resources. The compiler simply reads these files before processing the module (see Fig. 4).

So far we have only discussed cases where interface-checking occurs at the same time as compilation, or later. Square [2,3] represents a case where type-checking is performed before actual code-generation. As an example where this is desirable, consider inline procedures. An inline procedure is a restricted form of a macro in that its body is expanded in-line, yet it has the same semantics as a regular procedure. Suppose that an inline procedure is called in several different modules. If the header of that procedure is available (for example in a definitions file), we can type-check all calls to it, but we cannot generate code since its body may not be programmed yet. This compilation problem can be solved with a two-phase translation. The first phase generates intermediate code which contains pseudo-calls to inline procedures. This phase also checks the interface and can be executed fully independently

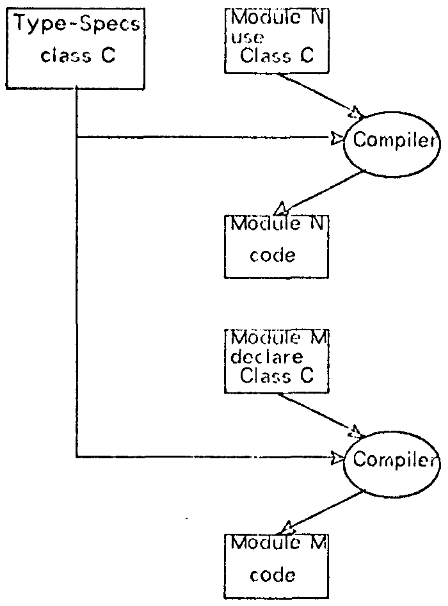

Figure 4: Independent compilation/type-checking

(like the MESA-scheme). The second phase generates the final code. However, this phase has to be delayed until the bodies of the needed inline procedures are available. This we can implement with an incremental scheme similar to the SIMULA67 system. We shall return to this problem in a more general framework when we discuss version control (section 5.4).

We have not considered the squares [1,2] and [1,3]. They are of little interest since they deal with monolithic compilation.

4. Interface Control

In practice, it turns out that it is difficult to establish and maintain consistent interfaces among the various components of a large, evolving system. However, such a consistency is absolutely crucial for the correct functioning of a system. Interface control is designed to guarantee this consistency. It helps to establish correct interconnections between components by checking their interfaces in an INTERCOL description of the overall system structure. It also extracts information from that description to ensure that the various programmed modules adhere to their interface specifications: First, each module must really provide the resources that are expected from it. Second, the types of the provided resources must be as specified. Third, no module may use more required resources than specified. Fourth, all required resources must be used correctly with respect to their types. The checking of these four conditions is implemented with inter-module type-checking, discussed in the following subsection. We also describe how the scope for interface errors can be limited with information hiding techniques, and how the problem of interfacing system parts written in different languages can be handled in a type-safe way. Finally, we discuss the actions necessary if the interconnections between the system components change.

4.1. Inter-Module Type-Checking

In the previous section we have characterized the various separate compilation and type-checking mechanisms. Now we specify and motivate our choices.

Clearly, for individual modules we would like to have completely independent mechanisms, so that the various programmer teams need not wait for each other. The independent type-checking mechanism requires that we specify the module interfaces in some way. We noted already that this is not a severe restriction -- in fact, it is the only way in which different teams can use each other's programs. As for the compilation mechanism, we do not want to exclude inline procedures and generics, which means that we have to use some mixture of independent and incremental compilation mechanisms.

To speed up compilations, we may also want to translate pieces of modules separately, for example single routines. However, we cannot use an independent type-checking mechanism here, because the interfaces between module pieces are implicit -- in fact, it would be quite a nuisance to specify the evolving interfaces between module parts. Therefore, we chose an incremental type-checking mechanism which derives the interfaces automatically. As for the compilation mechanism, we chose the incremental one as well, because the independent one would require a facility to derive type-specifications from the usage of resources, which is in general impossible.

We have marked our choices of mechanisms in Fig. 1 by enclosing the corresponding squares with a dashed line. In the remainder of this subsection, we describe our implementation of the independent inter-module type-checking based on INTERCOL. (The incremental/incremental mechanism for module parts can be implemented in a way similar to the SIMULA67-scheme and will not be discussed here.)

Consider the following INTERCOL-program describing a fragment of an (unrealistic) parser.

```
system PARSER

  module IO
     provide
     function SrcChar : char;
     const    lineLn;
     type     Line = record { lineNum : int;
                              lineBuf : array[1..lineLn] of char; };
     procedure ListLine(outLine : ↑Line);
  end IO

  module LEXAN
     provide
     type     Lexeme = (Keyword, Operator, Identifier);
     function NextLex : Lexeme;
     require
     SrcChar, lineLn, Line.{lineNum,lineBuf}, ListLine
  end LEXAN

end PARSER
```

The system PARSER consists of the module IO (for input/output) and the module LEXAN (for lexical analysis). IO provides the following resources: function SrcChar for reading source characters, procedure ListLine for printing a line on the listing medium, a constant indicating the length of a listing line, and the internal representation of a line. Module LEXAN implements NextLex, a function that delivers the next lexeme. It makes repeated calls to SrcChar. LEXAN is also responsible for generating the listing; hence it needs the definitions of Line and the procedure ListLine. Note that we have omitted the type-specifications in the require-clause of LEXAN. (In general, types need to be specified only once, usually in the module where they originate.) We have also omitted the origin of the required resources; they are contained implicitly in the structure. This enables us to reuse LEXAN in other systems, where its required resources may come from different places.

Note that the above piece of text must be processed before we can start compiling either IO or LEXAN. Compiling an INTERCOL description results in a set of environments, one for each module. Such an environment consists of a pair of preludes, one containing the required, and one the provided resources. For the above example, we obtain the following preludes:

```
IO.require:
    <empty>

IO.provide:
    forward function  SrcChar : char;
    forward const     lineLn;
    type    Line = record { LineNum : int;
                          lineBuf : array[1..lineLn] of char; };
    forward procedure ListLine(outLine : ↑Line);

----------------------- ------------------------------

LEXAN.require:
    extern  function SrcChar : char;
    extern  const  lineLn;
    type    Line = record { lineNum : int;
                          lineBuf : array[1..lineLn] of char; };
    extern  procedure ListLine(outLine : ↑Line);

LEXAN.provide:
    type    Lexeme = (Keyword, Operator, Identifier);
    forward function  NextLex : Lexeme;
```

Now let us examine how LEXAN is compiled. Its program text contains a compiler directive that indicates its identity:

```
::module LEXAN::
```

As soon as the compiler encounters that directive, it will read in the corresponding preludes. LEXAN.require contains the types of the external resources, and their use can now be type-checked. LEXAN.provide specifies the types of the provided resources in the form of forward declarations. This has two effects: First, if the programmer forgets to implement one of these resources, the compiler will complain about an undefined forward declaration. Second, if the programmer makes a type-error in the implementation of one of the resources, this will be noted as an inconsistency with a forward declaration. This makes sure that a module really provides the resources that are expected from it and that their types are correct.* Note further that the type-definition for Line has been transported from IO to LEXAN.

If the compilation of a module succeeds, the programmer has the guarantee that its interface will fit into the rest of the system. This is verified at the same time as the internal type-consistency of the module, and completely independently of the status of other modules.

4.2. Information Hiding

INTERCOL makes it possible to apply the principle of minimal privilege at module interfaces: No module must be given more resources or access rights than it actually needs. This is achieved with heterogeneous interfaces, name control and write-protection.

Heterogeneous Interfaces: Different modules may access different subsets of a common module's resources.

```
module A   provide a, b, c

module B1  require a, b

module B2  require   b, c
```

No matter how large the system is in which these three modules are embedded, they will not be able to use any other resources besides the ones specified in their require-clause. This compares favorably with the usual programming environments, where every module has unrestricted access to the complete name space of the system.

Name Control: Suppose module A provides resource x with subresources x1 and x2. Each module can be given access to exactly those

*The keywords **forward** and **extern** are not essential; their effects can be achieved with compiler directives as well.

subresources that it needs:

```
module A   provide type x = record { x1 : int; x2 : real; };

module B   require x.x2
```

The require-prelude of B will indicate that only x.x2 can be accessed. This can be accomplished with a record definition containing an *opaque* field, i.e., one that cannot be named and therefore not referenced:

```
     type x = record { int; x2 : real; }
```

Write Protection: Some programming languages specify write protection as part of the declaration of data-resources (variables, parameters, record fields, etc.). This is to protect global variables from accidental modifications, while still allowing read-access in other program parts. With INTERCOL, this can be controlled explicitly at the interfaces. (The default is that all imported variables and record fields are read-only.)

```
module A   provide variable  i1 : int;  -- read-write
                   readonly  i2 : int;  -- read-only

module B1  require i1, i2        -- read-only

module B2  require #i1, #i2      -- # means read-write
```

B1 has read-only access to resources i1 and i2. Module B2 has read/write access to i1, but the specified write-access to i2 is in error, since that variable is exported as read-only. Finally, A has read-write access to both i1 and i2, since they originate in that module and presumably need to be modified there.

All three of the above facilities have been proposed as programming language features before. Although some of the proposed mechanisms are adequate, we think that it is inappropriate to burden the languages for programming-in-the-small[3] with them. Information hiding local to a module does not make sense - why should a programmer hide something from himself? On the other hand, information hiding at the module interconnection level can be used to improve system flexibility[11]. It follows that global information hiding principles should not be implemented at the detailed level of program code. Instead they should be specified where they are needed: at the module interconnection level. (Nevertheless, features like opaque record fields and write-protection must be supported by the languages themselves so that a compiler can enforce the restrictions on the required resources.)

4.3. Multiple Languages

Suppose modules can be implemented in a number of different programming languages. How does interface control work in those cases?

INTERCOL is independent of the programming languages used. Even the sublanguage for type-specification in INTERCOL itself can be replaced. (We have chosen a modified PASCAL in the previous examples.) Let us assume that we use a customization of PASCAL for the type-specifications, and PASCAL, ALGOL, and assembler as the programming languages. For modules written in PASCAL, we generate preludes with extern- and forward-declarations as before. For modules written in ALGOL, we need to translate these preludes into equivalent ALGOL-declarations. Thus, we need a source-to-source translator or "decoder", that maps PASCAL-declarations to ALGOL. Of course, this mapping must be restricted to compatible language constructs (for example, primitive data types, procedures, functions, and value parameters). For assembler, we could develop a decoder that generates special macros defining the correct calling sequences, record offsets, etc. But now suppose our PASCAL compiler translates to assembly language. In that case, there is no need to write a special decoder at all -- we can use the existing compiler itself.

Of course, if a language as low-level as assembler is used for some modules, there is little interface consistency that INTERCOL can guarantee. In particular, if there are no types in a language, no automatic type-checking can be expected. On the other hand, closely related languages, or languages belonging to a family can be interfaced with a high degree of type safety.

4.4. Propagation of Interface Changes

An important problem in large projects is the management of interface changes. With our system, these can be processed automatically, because all the needed information is contained in the INTERCOL description. Changes of interfaces are recorded by modifying that description and recompiling it. This will usually destroy the global interface consistency established before. The minimum action to be taken in this case is to make sure that inconsistent modules cannot be used until they are updated. For minor modifications (like changes to constants or lay-out of records), recompilations can be started. Before going through with a modification, the designer could also be informed about how much will have to be recompiled or reprogrammed, so that he can estimate the impact of a proposed change. More important still, the changes can be documented in a precise way and automatically sent to the programmers of the affected modules (by putting messages into mailboxes, for instance).

After an interface change, we first have to determine the affected modules. Given the preludes described in a previous section, this can be implemented in a straight-forward manner: A simple comparison of the old and new preludes reveals exactly which module-interfaces were changed. Of course, the comparison can be refined to take into account whether a required resource was actually used in the implementation; changes make no difference for required resources that were not used. Once the inconsistent modules are isolated, the version control system can delete obsolete, precompiled versions, start some recompilations, and generate change notices for modules that need to be revised. In this fashion, simple change-requests as well as major design revisions can be managed efficiently and reliably.

5. Version Control

System maintenance in large programming projects is complicated by the proliferation of different versions and configurations. Even if a system is planned to be available in a single version, we nevertheless end up dealing with it in a number of versions internally: There is last week's version, a stable version, an experimental version, a testing and debugging version, programmer A's temporary version, etc. Other versions arise due to tailoring to individual user groups, functional enhancements, changes to the hardware and operating systems, reorganization, and last, but not least, error repair[12]. The management and maintenance of the numerous versions creates serious problems. Version control is designed to relieve the programmer from the tedious and error-prone task of organizing such a vast collection of components. More specifically, version control automates the following:

- *Version Selection.* With a set of rules and the INTERCOL description, version control determines which versions of which components should be combined to create a particular version of a particular configuration.

- *Construction.* Version control embodies detailed knowledge about the various construction processors that need to be invoked to generate a runable program.

- *Space/Time Tradeoff.* Version control maintains a data base of source text and the derived versions like object modules, partially linked subsystems, etc. It tries to optimize the space/time tradeoff of storing derived versions or regenerating them on demand.

- *Reconstruction.* If a new source version enters the data base or if interfaces change, version control determines which pre-constructed subsystems are obsolete. They will be reconstructed on demand.

In order to describe multiple versions and configurations, we have expanded INTERCOL to accommodate these notions. Each module or system in an INTERCOL description is now viewed as a *family*, whose members are the various versions. The description can then be exploited to automate the above functions. (For a different approach, the interested reader is referred to Cooprider[13].)

5.1. Module Families

We distinguish three orthogonal kinds of members of a module family.

1. *Implementations*. Implementations of a module family are the actual source programs. They share the same interface and abstract specifications, but may be implemented differently, in different languages, or tailored to different environments, operating systems, or user groups. For example, a program that runs under two different operating systems may have two different versions of the module that provides the idealized operating system environment.

2. *Revisions*. Each implementation exists as a sequence of revisions that succeed each other in the development history. Each revision is a modification of the previous one. For example, fixing a bug in the latest revision of an implementation generates a new one. All revisions of an implementation are linearly ordered by their creation time. Although the number of revisions can be quite large, the idea is that normally one deals only with the top few, for example the experimental, the stable, and the backup revisions.

3. *Derived Versions*. Derived versions are generated automatically from revisions of implementations. For example, each revision of an implementation written in portable Bliss can be compiled into a program that runs on a PDP-11, a PDP-10, or a VAX (given that no machine-dependent features of Bliss-16, Bliss-36, or Bliss-32 are used). Other derived versions can be produced by a compiler that generates optimized or non-optimized code, code with or without run-time checks of array-bounds, with or without debugging and instrumentation hooks. The versions that can be generated by different settings of conditional compilation switches fall in this class too.

5.2. System Families

A system family consists of a series of compositions. Each composition is a list of components which are other module or system families. Thus, each composition represents not only one, but a whole set of family members, depending on how rich the families of its components are. A specific member of a component can be selected by indicating implementation, revision, and derived version of a module family, or by indicating composition and components of a system family. This leads to a recursive description, which can be simplified considerably with defaults.

Example. Suppose M1 and M2 are module families, M1.1 and M1.2 are implementations of M1, and M2.1 and M2.2 are implementations of M2. If S.S1 = { M1, M2 } is a composition of a system family S, then the following examples represent three member of S:

```
S.S1(M1.1:79_06_14:Opt, M2.2:79_04_22:Opt)
S.S1(M1.2, M2.2)
S.S1
```

The first example indicates explicitly which implementations, revisions (by date), and derived versions (optimized) are desired. The second example suppresses revisions and derived versions; in this case, the newest revisions are selected. If precompiled derived versions of these exist already, they will be used regardless of whether they are optimized or not; otherwise, non-optimized derived versions will be generated. The third example does not mention implementations at all; in this case, the default implementations of M1 and M2 are selected.

5.3. A Complete Example

We now present our earlier example of the primitive parser as a system family with two members that can execute on two different machines and under two different operating systems.

```
system PARSER

   module IO
      provide
      function SrcChar : char;
      const    lineLn;
      type     Line = record { lineNum : int;
                               lineBuf : array[1..lineLn] of char; };
      procedure ListLine(outLine : ^Line);

      implementation HYDRA.bliss
      implementation TOPS.bliss
   end IO

   module LEXAN
      provide
      type     Lexeme = (Keyword, Operator, Identifier);
      function NextLex : Lexeme;
      require
      SrcChar, lineLn, Line.{lineNum,lineBuf}, ListLine
   end LEXAN

   composition CMMP = { LEXAN, IO.HYDRA }:Target.PDP11
   composition PDP10 = { LEXAN, IO.TOPS } :Target.PDP10
end PARSER
```

IO has two implementations, one for the HYDRA-operating system running on C.mmp, and one for TOPS10, an operating system for a DEC-PDP10. The implementation of LEXAN need not be mentioned since there is only one; let us assume, the default language is compatible BLISS. By pairing LEXAN with the TOPS10-implementation of IO, and compiling for the PDP10 as the target machine, we obtain a version of PARSER for a PDP10. By pairing LEXAN with IO.HYDRA and compiling for a PDP11 target, we can generate a version that runs on C.mmp (a multiprocessor constructed out of 16 DEC-PDP11s).

5.4. System Generation

The INTERCOL description is used to set up a data base that stores the various components. The revisions of an implementation are maintained in a stack similar to SDC[14] and SCCS[15]. Derived versions are constructed and integrated automatically. This poses a classical time/space optimization problem. One can either consume machine time by regenerating derived versions on demand, or one can save that time by storing them on disk. Unfortunately, the optimum cannot be determined statically, because system generation activities fluctuate. For example, the modifications to a particular subsystem may go through cycles of high and low activity. As the subsystem approaches a stable state, it is appropriate to partially integrate it and store it for later use. However, if the subsystem is not needed for a long time, it wastes space.

As a simple, semi-automatic solution we propose the following: The version controller maintains the latest derived version of each module and (sub-)system that was requested. It is the programmer's responsibility to delete some of these in order to conserve space. The only time that the version controller deletes derived versions is when they become obsolete because of interface changes or new revisions. Re-generation occurs on demand only.

When a programmer is testing and debugging a module, the version controller can optimize the re-integration time for the test system in which the modified module is embedded. The typical work-pattern is that the programmer repeatedly goes through a cycle of modification, compilation, integration, and execution. Assume that the programmer is working in an experimental area (a sub-directory for instance) which contains a copy of the module M he is updating. Suppose that system X is the testbed for M. Whenever he issues the command to re-integrate X, he indicates that he wants to substitute the newly modified M. The first time he issues that command, the version controller also generates a partially integrated system X with only M unbound. In all subsequent integrations of X, only M has to be linked in. This saves the re-integration of X from scratch for every modification of M.

The above is a simple cache scheme which shortens the integration time for recently used subsystems. Combined with time-outs for deletions from the cache, it should take care of most of the space-management for program development and maintenance.

In the previous sections we discussed how the various family members are selected. System generation is performed with the compile/integration scheme. There may be several interleaved phases of compilation and integration. This is due to the fact that INTERCOL describes only logical interfaces. A logical interface contains enough information to perform the first compilation phase, namely syntactic and semantic checking and the generation of intermediate code. However, there is not enough information to generate final code because physical properties of the interface elements are missing. Examples are values of constants, array bounds, record sizes, field offsets, bodies of inline procedures, and generic definitions.

One could decide to make the logical and physical interface identical and include them fully in the interface specifications (this approach was taken in the MESA-system). However, this is undesirable because the designer of the overall system structure may be unable to provide enough detail to do the physical binding. Furthermore, this would severely restrict the ways different versions can be constructed. For example, if the bodies of inline procedures were prescribed in the interfaces, then all revisions of all implementations of a module family would have to function with the same inline body.

A more complicated compilation/integration scheme, which avoids these problems, works as follows: The first compilation phase performs complete type-checking of a module and translates it into intermediate code in which the physical interfaces of required resources are still unbound. This phase also extracts the physical interfaces of provided resources and stores them in the data base. Next, the integration phase collects the physical interfaces of the module's required resources, once for each configuration in which the module is used. Note that the origin of the required resources is needed to determine which physical interfaces to use. This is derived from the interconnections specified in the INTERCOL program. Finally, the last phase uses the physical interfaces to generate machine code, again once for each configuration.

Clearly, there is substantial bookkeeping involved in this task. In any realistic software project, it is totally hopeless to perform that task by hand; only machines can handle it efficiently and reliably.

6. Status (June 1979)

We have implemented a demonstration system to test and refine our ideas. The system runs under UNIX on a PDP11/40 and currently implements interface control as described in section 4. Its main components are compilers for an early version of INTERCOL and for the programming languages C[16] and TC[17]. TC is a strongly typed variant of C, supporting opaque and partially opaque data structures as well as write-protection for data. Write-protection is safe in the sense that it is impossible to gain write-access to write-protected data except through address arithmetic. (For example, passing parameters, creating or returning pointers, dereferencing, accessing records and arrays, and even type-breaching are all checked for potential protection violations.) C and TC differ enough to show the feasibility of interfacing closely related languages. We chose TC as the type-specification sublanguage for INTERCOL. Of course, strong type-checking and write-protection can only be guaranteed for modules written in TC. For simplification, we do not make a distinction between logical and physical interfaces at the moment.

Implementation of, and experimentation with, the earlier version greatly improved INTERCOL, especially with respect to the representation of system families. Reimplementation including version control is planned.

7. Conclusions

We have described an integrated software development and maintenance environment that guarantees the structural consistency of a programmed system with respect to interconnections and version integration. Summarizing, its facilities are as follows:

Interface Control guarantees the consistency of the interfaces between the system components. It ensures global type-correctness by performing inter-module type-checking. It implements information hiding with detailed name control and write-protection. It detects the inconsistencies caused by interface changes and takes appropriate action. Multiple languages can be accommodated.

Version Control performs automatic system generation in a multi-version/multi-configuration system. It ensures structural consistency by selecting versions correctly. It performs flexible version integration by handling logical and physical interfaces. It manages a data base of numerous components and optimizes the space/time tradeoff of storing/regenerating subsystems. It responds to changes by detecting inconsistent and deleting obsolete versions.

Acknowledgments: I wish to thank A. Nico Habermann for numerous discussions and valuable suggestions, and Frank Deremer for his help in the design of an early version of INTERCOL.

References

1. Belady, L.A. *Large Software Systems.* Rep. RC-6966, IBM Thomas J. Watson Research Center, Jan. 1978.

2. Belady, L.A. and Lehman, M.M. The Characteristics of Large Systems In Peter Wegner, *Research Directions in Software Engineering*, M.I.T. Press, 1979, pp. 106-138.

3. DeRemer, Frank and Kron, Hans H. Programming-in-the-Large vs. Programming-in-the-Small. *IEEE Transactions on Software Engineering SE-2*, 2 (June 1976), 80-86.

4. Tichy, Walter F. *INTERCOL User Manual.* Carnegie-Mellon University, Department of Computer Science, 1979. To be published.

5. Morris, James. The Sniggering Type-Checker Experiment. An experiment conducted with the Mesa type-checker at Xerox Parc, Palo Alto; private communication.

6. Lauer, Hugh C., Satterthwaite, Edwin H. *The Impact of Mesa on System Design*, pages ?. ACM, IEEE, ERO, GI, Sept. 1979.

7. Birtwistle, G., Enderin, L., Ohlin, M. and Palme, J. *DECsystem-10 Simula Language Handbook Part 1.* Rep. C8398, Swedish National Defense Research Institute, March 1976.

8. White, John R. and Anderson, Richard K. . Supporting the Structured Development of Complex PL/I Software Systems. *Software - Practice and Experience 7* (1977), 279-293.

9. Mitchell, James G., Maybury, William, and Sweet, Richard. *Mesa Language Manual.* Xerox Palo Alto Research Center, Feb. 1978.

10. Cleveland, J. C. *The Environ and Using Facilities of Algol 68C.* Computer Science Department, UCLA, April 1975.

11. Parnas, David L. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM 15*, 2 (Dec. 1972), 1053-1058.

12. Belady, L.A. and Lehman, M.M. A Model for Large Program Development. *IBM Systems Journal 15*, 3 (1976), 225-252.

13. Cooprider, Lee W. *The Representation of Families of Programmed Systems.* Ph.D. Th., Carnegie-Mellon University, Department of Computer Science, 1978.

14. Habermann, A. Nico. *A Software Development Control System.* Carnegie-Mellon University, Department of Computer Science, 1979.

15. Rochkind, Marc J. The Source Code Control System. *IEEE Transactions on Software Engineering SE-1*, 4 (Dec. 1975), 364-370.

16. Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language.* Prentice-Hall, 1978.

17. Tichy, Walter F. The TC-Manual. TC is a strongly typed version of the C-language, developed at the Computer Science Department of Carnegie-Mellon University.