Changing Module Interfaces

by

SCOTT R. TILLEY
B.Comp.Sc., Concordia University, Montréal, 1986

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in the Department
of
Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. H. A. Müller

[redacted]

Dr. R. N. S. Horspool

[redacted]

Dr. A. Astbury

[redacted]

Dr. N. Cercone

University of Victoria
May 1989

Supervisor: Dr. H.A. Müller

Abstract

A common problem in large, evolving software systems is the difficulty of changing the interfaces of low level components. The traditional (re)compilation rule in strongly typed, separately compiled programming languages is the recompilation of all modules that use resources provided by the changed interface. In many cases this conservative rule causes unnecessary recompilations thus wasting time and resources bringing the system back up to a consistent state.

The *global interface analysis algorithms*, as proposed by Hood, Kennedy and Müller [HoKM 87], analyze, predict, and limit the effects of a change to a basic interface in a software system. The CMI (*Changing Module Interfaces*) implementation of these algorithms is targeted to the programming languages Ada, Modula-2, and CMI. The implementation is based on an attributed graph model of the software system, and solves the persistent problems of interactive change consequence analysis and minimal interface recompilation. CMI is a module interconnection language developed to capture essential interface relationships. It features a strict, explicit, and concise mechanism to describe the inter-object dependencies and the flow of resources through the network.

CMI is intended to be an integral part of the Rigi software development environment for programming-in-the-large, but can also be used as a stand-alone tool for both software development and maintenance. It is written in portable, object-oriented C and was developed on an IBM 3090 running VM/CMS, but may be easily re-hosted on a variety on operating systems using various compilers and support tools.

Examiners:

[redacted]

Dr. H.A. Müller

[redacted]

Dr. R.N.S. Horspool

[redacted]

Dr. A. Astbury

[redacted]

Dr. N. Cercone

Table of contents

List of tables

List of figures

Acknowledgements

I am grateful to Hausi Müller for his long distance guidance. His encouragement and support throughout my work on this thesis was invaluable. Using the post, tie-lines, couriers, fax machines, and electronic mail to communicate across a continent can sometimes be cumbersome, but he was always available and willing to help when it was needed (which was often).

I am fortunate to have always been surrounded by helpful friends and colleagues, from Concordia University to the University of Victoria, and the IBM laboratories in Toronto and Santa Theresa. They have all made an impression on me, both personally and professionally.

I would like to thank my parents for all their support over my many years of study, and I especially thank Faith for following me from coast to coast.

Finally I would like to express my gratitude to IBM Canada for their use of equipment and allowing me to spend time away from my other responsibilities to complete this.

For Faith, and my parents.

"I shall not waste my days trying to prolong them. I shall use my time."

James Bond, *You Only Live Twice*

CHAPTER 1

Introduction

1.1 The software lifecycle

Consider the following scenario: a programmer is given the task of implementing a performance enhancement to an existing project. The program consists of a large number of modules that were written over a long time frame by many different people (who may no longer be on the project). The change to the module involves altering one of the heavily used service routines and its external interface. Given this task, the programmer may have no idea what the effect of the change on the rest of the system will be. Even with reams of documentation that may be available on the original system design (which may have since evolved, probably without the documentation being kept up to date), this type of maintenance chore quickly becomes very complex. The unforeseen consequences the seemingly simple change will have may prove costly, both in person and machine time. It is important, particularly in the development of large systems, to know not only what is changed, but also what effect the change has on the system, since those effects can be far reaching and perhaps unanticipated.

Unfortunately, situations similar to that outlined above, changing the interfaces of low-level components (also termed *basic interfaces*), are a common problem in large, evolving software systems, and occur in both the development and maintenance phases of the software life cycle. During the maintenance process, the changing of a low-level interface involves the tracking down of all its clients and making sure that they are recompiled in the correct order to maintain a consistent system. Programmers become unwilling to alter a basic interface since they are unable to

estimate the effects of the change on the entire system. Indeed, because of the time needed to observe the effects of a change to an interface, programmers often try to breach the intermodule type-checking facilities to avoid the tedious wait of a (possibly lengthy) recompilation [Rain 84].[1] This evasion of compiler checks is contrary to the software engineering principle of modular encapsulation and the strong typing facilities provided by imperative programming languages. Programmers may also attempt to alter existing programs so that recompilation is minimized, often to the detriment of a proper design.

It is commonly taken for granted that languages such as Ada and Modula-2 support incremental development of large software systems through their facilities for separate compilation. This belief is not entirely accurate. These languages in fact only support a restricted form of incremental development. This form is governed by the compilation rules of the language (*i.e.*, the order in which compilation units must be submitted to the compiler or library management system). The traditional (re)compilation rule usually enforced requires that the interface, or specification, part of a module be available for analysis before its corresponding implementation part be submitted. While this rule implies that implementation modules may be developed in any order, as long as the appropriate interface modules have already been analyzed and accepted by the compiler, it also means that the interface parts of a software system must be developed in a strictly bottom-up order. Ada's subunits are an attempt to alleviate this restriction, but they are not fully successful in doing so. This restriction forces the construction of low-level modules to begin before any beneficial analysis can be done at the higher levels. In addition, since the interface part of a module often reflects the design decisions made concerning the modulari-

---

[1] However, this is impossible in some of the newer and better software development environments for Ada.

zation and interface relationships of a system, the rule forces those decisions to be made from the bottom-up (if they are to be subjected to incremental analysis) [WoCW 85b, WoCW 89]. In the rapidly changing development environment, this rule leads to interfaces that are frozen before they are sufficiently explored and tested [Teit 84a].

1.2 The problem

The benefits of software engineering principles such as modularization, separation of concerns, information hiding, and syntactic interface specifications applied to the construction of large software systems are well documented in the literature. However, the recompilation and coordination costs of changing a syntactic interface in large software systems is often prohibitive since so many software components depend on the interface.

This problem is exacerbated when the changing interface is a low-level component; a low-level component is a basic interface that is at or near the bottom of the system hierarchy [HoKM 87, Mull 86]. Altering a basic interface poses two main problems: since they form the foundation for the rest of the system, they are usually defined early in the design of a system and, as such, may not benefit from knowledge and experience gained at a later stage of development as more client implementations are written. The second major problem is that basic interfaces, by their very nature, have many clients. Changing an interface involves tracking down all the clients of the interface to make sure that they are still consistent.

Modularization is a structuring technique used by most strongly typed, separately compiled programming languages to decompose a large problem into several smaller

problems. Each of these "smaller problems" is termed a *module* in software development. Each module is usually split into (at least) two textually distinct parts: the *interface* part and the *implementation* part. A module communicates with other modules in the system through the interface part. Only the resources provided by the module are given; the mechanisms used to represent and/or implement these resources are hidden in the implementation part (and any of its variants).

Each of these modules communicate with the surrounding environment and other modules through *require* and *provide* lists. Though the syntax used by different programming languages to express this *require-provide* relationship between modules may be different, the semantics are much the same; a *producer-consumer* relationship is implicitly set up between various modules in the system. The modules providing resources are the *producers*, and the modules requesting resources from the producer are *consumers*. The *require* list specifies how the requirements of a module are to be resolved, and the *provide* list describes those objects that are provided by a module and are thus visible outside of it. Interconnection models are used to define these interactions between objects; visibility control mechanisms are used to govern requisition and provision of access to a resource. Different programming languages provide different visibility control mechanisms, which also differ in the degree of preciseness that they provide. For example, it may be possible to request a specific subset of items from a module, but that module may have no way of limiting access to the resources it provides. Contrasting the modification frequency and compilation time of the interface and implementation parts, interfaces are rarely modified (they contain less text) and compile much faster (no code generation is required). Nonetheless, consistency checking and recompilation due to the alteration of a basic interface are usually much more time consuming than changing an implementation part [Tich 82], since numerous other interfaces and implementations in the system

may directly or indirectly depend on a basic interface, while the implementation part has only local dependencies.

When the implementation part of a module is changed, the interface part is usually not affected. Thus, any other modules in the system that *depend* on that module are not affected. When the interface part of a module is changed in some manner, any other module in the system that *uses* resources that the changed module provides may be affected; the changes may ripple through the *require-provide* lists of the modules in the system. Thus, a seemingly innocent change in one interface may affect a large number of modules in the system. These affected modules must then be recompiled, even though they may not at all be affected by the changes made in the original module.

The traditional rule used in strongly typed, separately compiled programming languages to guarantee type and parameter consistency across compilation unit boundaries is the recompilation of all modules that use resources provided by the changed interface. Often this conservative rule causes unnecessary recompilations, thus wasting time and resources returning the system to a consistent state, since many modules do not use the resource that was actually changed in the altered interface. Moreover, this cascaded recompilation of modules may cause an unnecessary explosion of versions, since each compilation can be thought of as creating a new version of the object code, rather than replacing the old one.[2] This method may also severely handicap persons responsible for maintaining large software systems because it becomes costly, in both machine and person time, to modify a low level interface. This negates the benefit of experimenting with various design alternatives

---

[2] For example, on VAX/VMS a compilation produces a new version of the object code, and does not destroy the old copy.

in an evolving software system, and forces freezing of code before it has been explored sufficiently.

Implementations of this traditional recompilation rule for strongly typed, separately compiled programming languages typically use file modification time stamps, as exemplified by Feldman's *make* utility [Feld 79] (a standard tool of the Unix operating system), to determine whether a compilation unit needs to be recompiled. Since the interface's contents are not analyzed, a modification to it causes the recompilation of its implementation part and of its client modules, even if the latter are not at all affected by the change. Languages in this category include Ada [Ichb 79], C [ANSI 88], Modula-2 [Wirt 85], Modula-2+ [Rovn 86], Modula-3 [DJCL 89], Mesa [MiMS 79], Cedar [Teit 84, SwZH 86], Oberon [Wirt 87], and some the recent *modular* Pascal dialects [Josl 87, Bord 88].

In an improvement on this conservative recompilation rule, Tichy and Baker [TiBa 85, Tich 86] developed the *smart recompilation* algorithm which analyzes the contents of the interface and implementation parts of a module after a change to the interface by computing *change* and *reference* sets for the interface and implementations parts, respectively. This algorithm is especially useful for languages such as C or modular Pascal dialects, which use context, or `.h` (header), files as interface mechanisms, and source code proper (`.c` or `.p`) files as the implementation part. Programming environments such as $\int_C$ (Integral C) [Ross 87] use a modified version of these algorithms for incremental compilation of C programs. The smart recompilation algorithm determines if recompilation is required for the implementation part of a changed interface. It does not aid in limiting the effects of the change as they propagate through the network of client interfaces. The Tichy-Baker algorithm is concerned with interface/implementation pairs, as found in *independently* compiled

languages such as C or Pascal, while the global interface analysis algorithms concentrate on *separately* compiled languages such as Ada or Modula-2.

1.3 The approach

In contrast to the Tichy-Baker algorithm, the *global interface analysis algorithms*, as proposed by Hood, Kennedy, and Müller [HoKM 87], analyze, predict, and limit the effects of a change to a basic interface in a software system. They determine the effects of a change to an interface part on the clients (both direct and indirect) of the interface by propagating the set of affected resources from the altered interface through the *require-provide* lists of the interface network. When used in conjunction with the Tichy-Baker algorithm, they can minimize the recompilations of both interfaces and implementations that result from an editing change on an interface.

The CMI (*Changing Module Interfaces*) implementation of these algorithms is targeted to the programming languages Ada and Modula-2, and the CMI module interconnection language. The global interface analysis algorithms assume an environment that provides efficient access to the compilation dependencies among the modules in the system. Though CMI is intended to be an integral part of the Rigi software development environment for programming-in-the-large [Mull 86, MuKl 88], it was developed in a different environment. While Rigi is currently hosted on a network of Sun Workstations, the development environment for CMI was an IBM 3090 running VM/CMS. Because of this, the design of CMI differs from what it would have been had it been developed from the start as part of Rigi. In particular, CMI constructs the necessary environment for the global interface analysis algorithms by extracting the interface descriptions from the source text and storing

the information in its internal data base, which represents a semantic network data model of the software system under investigation.

Extracting the necessary inter-module relationship information can be accomplished in two ways: either an existing compiler for the supported target language is altered to provide the interface information in some way, or another tool must be built that parses the source text and gleans the information without the compiler's aid. In part because of time and resource restraints, and in part because of the philosophy of CMI, the existing compilers were not altered. Rather, a parser was built for each supported language, which allows incremental support for new languages to be added at a later date. It also avoids the legal implications of changing source code in (sometimes proprietary) compilers. The CMI module interconnection language was initially developed as an intermediate form that consisted only of the essential interface relationships among modules in the system. It soon became clear that the language by itself is a viable method for describing explicit *require-provide* relationships in a strict, clear, and concise fashion. It also allowed the internal implementation to abstract itself away from the different "features" of various programming languages, and to concentrate on the information extracted from the equivalent CMI description of a particular interface. Thus, the only language-dependent part of the CMI implementation is the parsing.

CMI can be used for both software development and maintenance. As a stand-alone tool, it can be used as an aid for carrying out *consequence analysis* on interface changes and as an efficient recompilation mechanism for managing interface changes. The implementation is written almost entirely in C, and is portable across a variety of operating systems using various compilers and support tools. It can be used as an interface translator (compiler) between any of the supported languages, or interactively in either a command-line mode or in a menu-driven environment. When integrated into Rigi, CMI will use its window-based user interface.

1.4 Benefits

CMI can be used to solve two persistent problems: *interactive change analysis* and *interface recompilation*. Interactive change analysis allows a user to experiment with changes to an interface and to preview the effects the change would have on the entire system before it is actually implemented. Minimal interface recompilation requires that, when a system is rebuilt, only those modules affected by editing changes be recompiled, and in the correct order. The algorithms are used during the (re)compilation process to limit the effect of the interface change, and thus limit the explosion of versions as well.

With CMI, experimental changes may be made to a basic interface to determine the effect on the rest of the system. Interface designs need not be frozen early in the life cycle, since changes can be efficiently managed at a later date. This is an improvement over simplistic, monolithic, and crude recompilation algorithms that are found in most systems nowadays, as exemplified by the well-known *make* program or the current Cedar environment [Lamp 84, Teit 84b]. The design of a system should not be hampered by poorly constructed "work-arounds" introduced by programmers trying to avoid massive recompilations due to a small change in a low-level or "busy" interface. The CMI implementation solves this problem by managing the change, rather than avoiding it. The programmer can minimize the time needed for recompilation, thus allowing various alternatives in the interface design to be explored before a final decision is reached.

In a software development environment such as Rigi, aided by CMI, the maintainer of a large, complex system can easily decide whether a change in a basic interface can be implemented or not by computing the set of affected modules. If the set is acceptable, the change can be implemented; otherwise, it can be undone.

CMI aids in the support of incremental development by distilling out program analysis from the compilation mechanism, which is where it has historically been confined. It is desirable to be able to perform both syntactic and semantic analysis at different stages of the software life cycle. CMI provides feedback through the development and maintenance processes.

1.5 A guide to this thesis

Chapter 2 presents some background on pertinent issues of modularization and syntactic interface specifications with respect to the targeted programming languages Ada and Modula-2. In Chapter 3 the global interface analysis algorithms are discussed in detail, and Chapter 4 describes the CMI implementation of these algorithms. Chapter 5 illustrates the use of CMI as a stand-alone tool in its various modes of operation, presents the results of using CMI on sample systems, and comments on how it would be used as part of an integrated software development environment. Finally, Chapter 6 presents conclusions and possible directions for future research.

CHAPTER 2

Background

2.1 Introduction

This chapter reviews work related to this thesis and presents background material on the topics of programming-in-the-large and software development environments. This chapter also discusses modularity and interface specification in modern strongly typed, separately compiled programming languages and interface description languages, as well as visibility control, interconnection models, and recompilation.

Programming-in-the-large refers to those aspects of software development concerned with the design, integration, and maintenance of software systems above the level of a single module. A description of the differences between programming-in-the-large and programming-in-the-small is given, outlining some of the special problems that programming-in-the-large poses for conventional programming environments.

Software development environments, and particularly those concerned with programming-in-the-large, have been very slow to evolve [Perr 87]. A brief overview of selected software development environments for programming-in-the-large is presented, with emphasis placed on the Rigi system [Mull 86, MuKl 88].

Modularity and interface specification in modern programming languages such as Ada and Modula-2 play an important role in complex software system design. Interface description languages such as IDL [StNe 87] and Intercol [Tich 79] are a

more formal method of stating *producer-consumer* relationships between modules in a software system. An overview of the use of these modularization aids is presented.

Classical programming languages rely on *nesting* as their visibility control mechanism [WoCW 86, WoCW 88]. A description of other access mechanisms is presented, especially those suited for modular programming systems.

Interconnection models are used to define the interactions and relations between objects in a software system. Various models and how they relate to modularity, visibility control, and recompilation strategies are presented.

Finally, the recompilation problem is discussed. Traditional methods of recompilation used for programming languages that are used in programming-in-the-large are not optimal and often cause many unnecessary recompilations to occur. An overview of current recompilation methods is given, and the problems associated with these are highlighted.

2.2 Programming-in-the-large

A programming environment provides support for a single user developing a single module. This support typically consists of tools such as syntax-directed editors, incremental compilation, and source-level debuggers [Till 87]. These tools are used in the creation, editing, compiling, testing and debugging of a *single* module. This aspect of software development is termed programming-in-the-small [DeKr 76].

Programming-in-the-large refers to those aspects of software development concerned with the design, integration, and maintenance of software systems above the level of a single module. The term software development environment is usually

used in place of programming environment to describe programming-in-the-large development support environments. A software development environment that provides support for programming-in-the-large must address issues such as the organization and representation of system structure, module decomposition, component dependence analysis, precise interface control, separate compilation, subsystem and composition identification, managerial support, method independence, as well as version and release control [Klas 88, Mull 86, WoCW 85b].

Besides programming-in-the-small and programming-in-the-large, a third term is often used to describe software development environment support: programming-in-the-many [HaNo 86]. Aspects of software development concerned with programming-in-the-many include several programmers working on the same project, as well as projects that stretch over long periods of time. Access control, mutual exclusion, documentation, change history, network access, and project management are also concerns in this area.

In the past, most software development environment work has focussed on the programming-in-the-small aspect of software development. Projects in this category include the work on intermediate forms for program representation and structured editor support by Caplinger [Capl 85a, Capl 85b], the Cornell PSG [TeRi 81, BaSn 85], Gandalf [HaNo 86, Notk 85], the $\int_C$ (Integral C) development environment for C [Ross 87], Interlisp [TeMa 81], the Magpie development environment for Pascal [DeMS 84], PECAN [Reis 84], SODOS [HoWi 85, HoWi 86a, HoWi 86b], SRE [BuHZ 85], and the object-oriented SW2 system [LaHa 85]. The emphasis on programming-in-the-small neglects important aspects of software development, and does not provide support in all phases of the software life cycle. Table 2.1 [Till 87] shows the three main software development environment areas. Only recently have systems for programming-in-the-large emerged. These are discussed below.

PIS Programming-in-the-small	PIL Programming-in-the-large	PIM Programming-in-the-many
Construction, analysis, compilation, execution, optimization, debugging, testing, monitoring of a single module.	Design, integration, and maintenance above the level of a single module. System descriptions, module decompositions, module dependence analysis, interface control, separate compilation, subsystems, libraries, compositions, versions, releases.	Several programmers working on a large project, large time frame, access control, mutual exclusion, documentation, change logs, network access, project management.

Table 2.1. Programming in the small, large, and many

2.3 Software development environments

As mentioned above, in the past most research in the area of software development environments has focussed on the programming-in-the-small aspects of software construction. The development of sophisticated systems for managing a large software project through all phases of the software life cycle is still relatively new. The sheer magnitude of the task has meant that software development environments for programming-in-the-large have been slow to evolve. Recently, however, several new and promising projects have emerged. These environments support the programming-in-the-small aspect of software development, but also address issues related to the design, integration, and maintenance phases of the software life cycle.

This thesis is mainly concerned with language, tool, and environment support for describing the relationships among objects in a software system, and the effects a

change in one of these objects has on the rest of the system. Some of the software development environments that provide support for this type of analysis include the APSE (Ada Program Support Environment) [NoHa 81, MOPT 88, Ober 88] and common interface set for Ada, the Arcadia research project [TBCO 88, TCOW 86] and its predecessor Arcturus [StTa 84, TaSt 84, TaSt 85], Xerox PARC's Cedar system [Dona 85, SwZH 85, SwZH 86, Teit 84a, Teit 84b] and the related Mesa system [LaSa 79, MiMS 79, Swee 85], the CENTAUR system [BCTI 88], Appollo's DSEE [LeCh 84, LeCh 87, LeCM 85, LeMc 85], the Infuse tool for automatic change management [PeKa 87], which is part of the Inscape environment [Perr 87b], the Ipsen system at the Aachen University of Technology [Scha 87, Lewe 88], the K2 project [MHHL 87, MHHL 89] at the University of Victoria, MicroScope [AmOd 87, AmOd 88], MUPE-2 [MaPT 86] at McGill University, the PCTE and PCTE+ systems [BGMT 88], the PIC system for Ada-based programming-in-the-large [WoCW 84, WoCW 85a, WoCW 85b], the PUNS program understanding system [Clev 88], the Rigi project [Mull 86, MuKl 88] at the University of Victoria, the R^n environment for FORTRAN at Rice University [CoKe 88, CoKT 85, CoKT 86a, CoKT 86b], and the TEAM environment [ClRZ 88] for experimentation with various testing and analysis tools.

Of the software development environments mentioned above, Rigi is the most important for this thesis, since CMI is an implementation of the global interface analysis algorithms, one of the subprojects associated with Rigi. Rigi is a framework for programming-in-the-large currently consisting of four main subprojects:

1. Rigi Model
2. Reusability-in-the-large
3. Global Interface Analysis Algorithms
4. Rigi Editor

Rigi is also part of the K2 software development environment for programming-in-the-large as proposed by the Software Systems Group at the University of Victoria.

The Rigi model is a *semantic network data model* for representing and organizing the components and dependencies of complex systems [Mull 86, MuKl 88]. The model is expressed in graph terminology, with special meaning given to the arcs and nodes in the network. Each node represents a component in the software system, while the arcs represent dependencies between system components. *Abstraction mechanisms* are used to organize the information presented by the nodes and arcs of the network. Rigi models the basic *bricks* and *mortar* of a large, integrated software system. These bricks can range from the module level to the composite objects of a system or subsystem.

The second subproject of Rigi is concerned with the reuse of design information, system structures, and domain knowledge for programming-in-the-large beyond the module level. The third subproject is the global interface analysis algorithms, which are the subject of this thesis. The fourth subproject, the Rigi editor, is a tool for maintaining and understanding the structure of a large system. The system description documents produced by the Rigi editor should run as a common thread through the life of a software project. Two implementations of the Rigi editor have been completed: the first was a prototype on an Apple Macintosh [Mull 86], and the most recent on a Sun workstation, which is the subject of [Klas 88].

2.4 Modularity and interfaces

One definition of programming-in-the-large states that it involves two complementary activities: modularization and interface control [Tich 79]. Modularization

involves dealing with a large collection of modules to form a system, and it is a different activity from the construction of each module [DeKr 79]. Describing the major system objects and their interactions is the primary concern of architectural design at the high level, while maintaining correct and consistent interfaces is an overriding concern during the implementation and maintenance of a software system.

Since good software engineering design suggests that modules be kept relatively small, the number of modules in a large system is significant. For example, in a system of 500,000 lines, with roughly 200 lines per module, there would be 2,500 modules. This is an order of magnitude more than there are lines of code in each module. With such a large number of modules, development of the proper interface relationships remains a complex task [Parn 72, Tich 79, PaCW 84, WoCW 84]. Relatively little attention been given to how a system is best decomposed into modules, and how these modules should interact with each other. Interface control can be described as the specification and control of the interactions among entities in different modules. Perry has done empirical studies that indicate that misunderstandings at module interfaces are the rule rather than the exception [Perr 87c]. This thesis argues that better visibility control mechanisms could be used to reduce this inter-module communication problem.

Many programming languages supporting programming-in-the-large, such as Ada [Ichb 79], Modula-2 [Wirt 85], Modula-2+ [Rovn 86], Mesa [MiMS 79], Cedar [Teit 84a, SwZH 86], as well as some of the module interconnection languages such as C/Mesa [MiMS 79], IDL [StNe 87], Instress [Perr 86], Intercol [Tich 79], NuMIL [NaSc 85], and PIC/Ada [WoCW 84, WoCW 85a, WoCW 89] (an extension to Ada), provide facilities for the modular construction and description of large programs. Such a programming language separates the interface specification of a module from its implementation, and enforces type and parameter checking across compilation unit

boundaries. The interface part serves two purposes: it is the syntactic specification for the corresponding implementation, and it is the interface between the module and its environment. More advanced systems allow for multiple alternatives and/or versions of the implementation to co-exist; the user can select the version desired as late as link time [MiMS 79, LeCM 85].

Throughout this thesis, a *module* is loosely defined as synonymous with a *compilation unit*. Though this statement is not strictly true for all programming languages, and some may not agree to this equating of a logical unit with a pragmatic (*i.e.*, implementation defined) unit, it does serve as a guide to a practical example of the module concept. Most implementations of Modula-2, for example, either suggest or demand that each module be in its own compilation unit for easier interface consistency verification by the compiler.

Since much of programming-in-the-large is concerned with defining interfaces [NoHa 81] and modularization, an overview of the modularity and interface specification mechanisms in modern programming languages, focussing on Ada and Modula-2 in particular, is presented. Emphasis is placed on the different methods each programming language uses to implement modularity and interface specification.

2.4.1 Ada

An Ada program is a collection of modules or compilation units called *library units*. The implementation chooses how to specify which library unit is the main module. A library unit can consist of one or more of the following [Booc 83]:

- generic declaration

- generic instantiation
- subprogram declaration
- subprogram body
- package declaration
- package body
- subunit

Any library unit can be separately compiled. There are two philosophies for programming-in-the-large, bottom-up or top-down; Ada supports both of these through stubs and separate subunit compilation.

The main mechanism provided by Ada for decomposing systems into modules is the **package** construct. A module is a **package** and its corresponding **body**. Although Ada is orthogonal in that library units may be nested in any order to an arbitrary depth, usually the generic or package structure is the enclosing block. We concentrate here on packages and package bodies as the main modularization technique, with the package and its body usually being in separate compilation units.

Ada packages serve the same purpose as modules do in Modula-2 and clusters do in CLU. They are the mechanism used to structure programs for programming-in-the-large above the level of procedures and/or functions (which themselves may be arbitrarily nested). The package is broken into two (or more) textual parts, which need not be in the same file. These are the *interface* and *implementation* parts. Figure 2.1 shows the syntax for module decomposition in Ada.

Ada has only two levels of resource provision: all or none. A client gains access to the resources provided by another package through the **with** statement. The **with** statement causes all of the package's resources that are visible to be made

```
package name is            -- interface
    ...
end;

package body name is       -- implementation
    ...
end name;
```

Figure 2.1: Module decomposition in Ada

available.[3] The **use** clause can then be used to provide unqualified access to resources, as long as they do not conflict with already defined objects; otherwise, selected component notation can be used (*e.g.*, `name.resource`). Figure 2.2 shows an example of how a client would gain access to objects in package "name". The **with** statement declares that the current block is *importing* the resources provided by "name". The **use** clause makes the resources provided by "name" part of the local name-space, so that the dot selector notation is not needed to access these resources. The **use** clause is optional. This clause is analogous to the "with" statement for records in Pascal.

By default, all resources in the **package** are *exported* (*i.e.*, completely seen by any other structure that contains a "**with** package_name;" clause). The package writer may choose to hide certain parts of the package structure by declaring items to be **private**. A further alternative is the **limited private** type, similar to Modula-2's opaque types, except that even the equality and assignment operators are not provided by default, and must be provided by the package writer (if desired). By

[3] That is, all those objects that are not **private** or **limited private**.

default, the **package** STANDARD is automatically included, to give a variety of useful, system-dependent operations and types for the Ada programmer.

Ada also provides **generic**s as another powerful facility to further abstract the programmer and system designer from the actual use of a resource and from its definition. A **generic** is an entity that is only partially compiled. It takes parameters when it is instantiated to enable a package to be used for tasks that have a similar function but operate on different types of data. Examples are a generic sorting routine or a stack. The latter is well documented, and Barnes gives a good example in [Barn 84]; a stack's operations are independent of the type of data being stored in it.

2.4.2 Modula-2

The modularization mechanisms of Modula-2 include three types of modules: **DEFINITION**, **IMPLEMENTATION**, and the main program's **MODULE**.[4] Every module in Modula-2 (except for the main module) has both a definition part and an implementation part. These correspond to Ada's **package** and **package body**, respectively. The definition module is the interface to its clients.

```
with name; use name;
```

Figure 2.2: Client access in Ada

---

[4] The main module is, in fact, an implementation module [Wirt 87].

Modula-2's modules serve the same purpose as Ada's packages. They are the mechanism used to structure programs for programming-in-the-large above the level of procedures and/or functions. The module is broken into two (or more) textual parts, which may or may not be in the same file.[5] These are the *definition* and *implementation* parts. Figure 2.3 shows the syntax for module decomposition in Modula-2.

The greatest advantage that Modula-2 has over its predecessor[6] Pascal is the **MODULE** construct. This enables a programmer to split a task into distinct pieces, above the level of procedures. A Modula-2 program is made up of one main module, and zero or more definition/implementation module pairs. While modules may be nested,[7] they usually only enclose type or constant declarations and subprograms (procedures and functions).

The most recent version of Modula-2 [Wirt 85] implicitly exports all objects defined in a definition module. The initial definition of Modula-2 required the programmer to explicitly list the exported objects. Both the definition and implementation parts can import objects, but only the definition part can export objects. An example of the interface syntax is given in Figure 2.4. There is a one-to-one correspondence between definition and implementation modules. A client may import some of the objects provided by a module, or it may import everything in the

---

[5] The separation depends on the particular Modula-2 compiler's implementation.

[6] Modula was the direct predecessor of Modula-2. However, this language never received wide release, and was developed specifically for real-time control systems.

[7] Local modules are not considered, since they are not visible outside their enclosing scope, which is another local module, an implementation module, or the main program module.

```
DEFINITION MODULE name;          (* interface *)
    ...
END name.

IMPLEMENTATION MODULE name;      (* implementation *)
    ...
END name.

MODULE name;                     (* main or local module *)
    ...
END name.
```

Figure 2.3: Modularization in Modula-2

module. Only identifiers are used in the import and export lists; the Modula-2 syntax does not require one to specify the category of the identifier (e.g., `constant`, `type`, `variable`, `procedure`), since there is no overloading in Modula-2. There is an optional **QUALIFIED** clause that may be applied to exported identifiers from a local module. If given, the importing module must always fully qualify references to such objects. In Ada this would be a **with** clause without a corresponding **use** clause.

2.4.3 Other languages

Many other programming languages are equally powerful in supporting programming-in-the-large, including interface description languages and design languages. Programming languages such as Cedar [Teit 84a, SwZH 86], Modula-2+ [Rovn 86], Modula-3 [DJCL 89], Oberon [Wirt 87], and PIC/Ada [WoCW 84, WoCW 85a] all provide modularization facilities and interface mechanisms to allow for

program decomposition with strong inter-module consistency checking (to varying degrees). Design languages such as Anna [LuHe 85] and SEDL [IBM 88d] and module interconnection or interface description languages such as IDL [StNe 87], Instress [Perr 86], and Intercol [Tich 79] extend these facilities to allow a stricter and more precise description of the interactions between various modules in the system. The CMI module interconnection language provides a mechanism to describe inter-object relationships precisely; the language is explained in detail in Chapter 4.

Finally, it should be noted that languages not usually associated with programming-in-the-large such as C or Pascal are evolving to incorporate some type of inter-module verification facility. The earliest and crudest mechanism is probably from Fortran and its use of labelled (or unlabeled) COMMON blocks. The latest draft of ANSI C [ANSI 88] and implementations of Pascal such as Turbo Pascal 4.0 [Bord 88] both have a flavor of inter-module checking similar to Modula-2. PL/I has also had facilities similar to this since 1970 [IBM 87b]. The difference between these languages and Ada or Modula-2 is that the programmer is not forced to use these facilities;[8] the older "unsafe" style of programming can still be employed. In addition, the level of checking that the compiler does is not usually as complete as

```
FROM x IMPORT a,c;     (* specific resources *)
IMPORT y;              (* the whole module *)
```

Figure 2.4: Client access in Modula-2

[8] For example, in ANSI C, a program can still be coded in older *K&R* style [KeRi 88].

that in Ada and Modula-2. This may be attributed to the relative crudeness of the visibility control mechanisms used in these languages.

2.5 Visibility control

The term *visibility control* essentially describes the mechanism that controls the scoping of objects. In the past, the predominant visibility control mechanisms have been *ad hoc* and based on nesting [WoCW 85b, WoCW 86, WoCW 88]. An object's declaration, scope, and binding determines to whom it is available. This lexical scoping of blocks, procedures, modules, and subsystems forms the basis for an implicit *producer-consumer* relationship, where the enclosing scope is the producer, and all enclosed scopes that reference objects declared in the surrounding scope are consumers.

This nesting of scopes is not a precise mechanism for describing visibility control. In languages such as Pascal [Wirt 71] or PL/I, a subprogram's local entities are unavoidably made visible to other subprograms or blocks nested within that subprogram, but this fact is only implicitly stated in existing formal descriptions of nesting (usually given in the language's reference manual). There is no formal method of providing access to a particular object (or set of objects) to a selected set of clients, just as there is no way to formally request access to only certain objects that a parent or sibling module may be providing. Modules are essentially producers and consumers of resources; ideally, relationships should be specified from both the producer's and the consumer's point of view. A more formal and precise method is based on the more general concepts of *requisition of access* and *provision of access*. A visibility control mechanism can therefore be defined as a means for specifying

requisition and provision of access, where "access" is defined as making reference to, or using, an entity.[9]

Requisition of access occurs when an object (implicitly or explicitly) requests the right to potentially refer to some set of entities (or entities "owned" by another object). For example, in programming languages, a subprogram requests access to itself and any locally declared entities (plus some non-local objects). Provision of access occurs when an object (implicitly or explicitly) offers, to some set of objects, the right to potentially refer to that object (or entities that the object "owns"). For example, in procedural programming languages, access to a subprogram is typically provided to the subprogram itself (*i.e.*, recursive calls), and in languages that support nesting, to the subprogram's parent(s), siblings, and descendants.[10]

Given this definition of visibility, an actual reference by object o_i to object o_j is possible iff o_i requests access to o_j, and o_j provides access to o_i. In other words, one module may request access to an object in another module, but this request is only valid if the providing module in turn has made provision of this object to the requesting module. The distinction between requisition and provision alters with different approaches taken to visibility in different programming languages. For example, in Pascal [Wirt 71], requisition and provision are essentially mirror images: those objects requested by an object are also provided to that object and vice

---

[9] An entity is a nameable object in a programming language such as a `constant`, `type`, `variable`, or `procedure`. Throughout this thesis, the terms *resource* and *object* are used interchangeably with *entity*. The reason is that the definition of a resource changes with the level of the system hierarchy. For example, at the module level, a resource may be a `const`, but at the system and variant level, a resource may be a subsystem or module.

[10] For languages that do not support nesting, such as FORTRAN, only siblings apply.

versa (in a syntactically correct program). In the designs of languages intended for large and complex systems, the desire for greater control over entity visibility has resulted in mechanisms that address requisition and provision in separate and often unequal ways.

Both Modula-2 and Ada support nesting as a visibility control mechanism, but both also support a flavor of a *require-provide* mechanism as well. Each language supports provision and requisition of access differently. In Modula-2, provision of access may be qualified by forcing the requesting module to fully qualify objects imported from a local module (through the **qualified** clause). Provision of access of objects to a particular set of modules is not supported; if an object is provided by a module, all other modules in the system have access to this object implicitly. Requisition of access requires that the name of the providing module be given when requesting an object, but it is also allowed to import all objects that another module is exporting. Thus, the *require-provide* mechanism in Modula-2 is not as strict as is desired.

Ada has the same provision of access scheme as Modula-2 with the exception that the representation of a subset of a package's exported objects may be hidden (through the **private** and **limited private** clauses).[11] This method is still limited, since one may only hide an object from all other objects (packages) or none. There is no qualification mechanism for provision. Ada's requisition mechanism is even coarser than that of Modula-2; the **with** clause imports all visible objects to the requesting module.

---

[11] The name of the entity is still visible, just its representation is hidden.

The PIC/Ada language extends Ada by adding **provides to** and **requires** clauses to the Ada grammar. The programmer then has greater control over the provision of resources by explicitly providing objects to specific modules. The clients, in turn, can request the whole package (as in standard Ada) or only the required parts of a package. The PIC method is based on a directed graph model of module interfaces that is used to represent a set of interface relationships; the database (a development library)[12] maintains the results of previous module analysis. It also provides an incompleteness construct to alleviate some of the bottom-up restrictions of Ada development.

The *provide-require* relationships that exist between objects in a system are based on the visibility control mechanism used. One may easily model these relationships as a graph structure, with a node representing an object, and the arcs connecting the nodes indicating requisition and/or provision relationships. Chapters 3 and 4 study this graph structure as a compilation dependency representation; however, one may also study it in terms of a visibility graph that uniquely represents a particular set of visibility relationships among a set of objects.

Wolf *et al.* [WoCW 88] define a visibility graph $G = (N, A_r, A_p)$ as a directed graph where N is a finite set of nodes labeled by unique names of entities,[13] A_r is a finite set of ordered pairs of nodes (n_i, n_j) denoting the requisition relationship n_i "requests" access to n_j, and A_p is a finite set of ordered pairs (n_i, n_j) denoting the provision relationship n_i is "providing to" n_j. The ordered pairs in A_r and A_p determine the arcs in the graph. The visibility graph can be derived from the syntax tree representation

---

[12] The development library is a synthesis of an Ada program library with an operating system file structure.

[13] If overloading is allowed, as in Ada, type information is also needed, since names need not be unique.

of a program by applying the rules of a particular visibility control mechanism to the objects in the representation. Chapters 3 and 4 show how this information is extracted for CMI.

A visibility control mechanism is said to be *precise* if it is both *requisition precise* and *provision precise* [WoCW 86, WoCW 88]. This definition of preciseness can be used as a measure of a programming language's modularization and interface control mechanisms, particularly for programming-in-the-large. Languages such as Pascal are imprecise because they do not have any type of *require-provide* mechanism other than the implicit one of nesting. Ada is requisition-precise, but not provision-precise. The reason is that, in addition to Ada's nesting capabilities, Ada provides access to a package's objects through the **with** clause. The selection of objects from a (nest-free) package is on an all-or-none basis; an object[14] in a package is either visible to all other packages in a system, or it is hidden from all. Thus, it is not provision-precise. It is requisition-precise because a program can be constructed that has no nesting at all, and all services a package provides can be garnered through the **with** clause. This is somewhat odd because Ada is designed with the expectation that many systems will be built from libraries of general-purpose modules, many of whose elements are not needed at any given time.

Modula-2 is also requisition-precise, but not provision precise. While the request mechanism in Modula-2 is finer grained than Ada (*i.e.*, a client can request a subset of objects that a module provides), a definition module cannot provide services to a particular set of modules; it is an all-or-none construct. Note that this model of visibility does not insist on a particular interpretation of (in)consistency.

---

[14] That is, a visible object, according to Ada semantics. For example this excludes the representations of objects that are **private** or **limited private**.

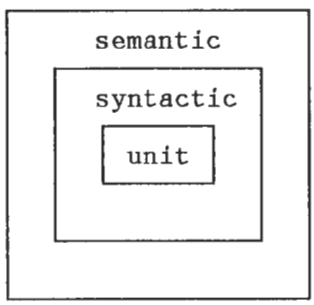
In Ada, for example, a minimum condition for the consistency of a set of object visibility relationships is that each object requested is in fact provided. Both PIC/Ada and CMI are provision and requisition precise. Chapter 4 shows that the module interconnection language that CMI uses is based on explicit and precise *require-provide* relationships. A complete formal treatment of visibility control mechanisms is given in the excellent papers by Wolf, Clark, and Wileden [WoCW 86, WoCW 88].

2.6 Interconnection models

Interconnection models describe the relationships among objects in a software system. The granularity of the inter-object relationships depends on the programming language (since the modularization and interface schemes available vary by programming language), the software development environment (because the tools provided by a software development environment for programming-in-the-large vary), and the visibility control mechanisms used. Perry introduced three possible interconnection models [Perr 87a], which are shown in Figure 2.5. Perry's formalism can be used to define an interconnection model as a set of tuples as follows:

$$IM = (\{objects\}, \{relationships\})$$

This set of tuples is easily mapped to a graph structure, with the objects being nodes and the relationships being arcs with semantics between the nodes. This is the basis for the CMI graph model.



The three layers of interconnection models are:

1. unit (coarse) (*e.g.*, *make*)
2. syntactic (fine) (*e.g.*, CMI with Rigi)
3. semantic (extra fine) (*e.g.*, Inscape)

Figure 2.5: Interconnection models

2.6.1 Unit model

The unit model defines relations between various units (modules or objects) in a software system. Examples of this model include *make*, unadorned Ada systems,[15] and most Modula-2 compilers. While the unit model is useful in encouraging modular construction, determining compilation contexts, determining recompilation strategies, providing change notification, and describing system models, it is useful only in the context of very large-grained objects, since the basic units in this model are either files or packages (compilation units), which themselves usually contain multiple entities. Often only a small amount of a context provided in this manner is

[15] Systems that do not provide a "smart make" facility.

actually used. This model forces many redundant recompilations because the change that occurred is with some resource within the basic unit; by its definition, changes can only be detected on a unit-based scale. It is sometimes advantageous to compose systems out of smaller pieces than files (compilation units) or modules. Thus, a finer grain of interconnection than is available with the unit interconnection model is needed for effective management of evolution in software systems.

An example of a system based on the unit interconnection model is the C programming language. A compilation unit gains access to type definitions and function prototypes (among other things) by textually including a context file by using the preprocessor `#include` directive; the context file is also called a header, or `.h`, file. A plain Ada system also uses the unit model for requisition of objects from other packages as well as to determine the recompilation order. Most Modula-2 compilers use a unit model similar to Ada's to determine recompilation order. Finally, *make* uses file time stamps to determine (re)compilation order. These unit interconnection models are shown in Figure 2.6.

2.6.2 Syntactic model

The syntactic model focuses on smaller units than the module; the relationships between modules are refined to more detailed relationships among the syntactic elements of programming languages. In generic terms, it has a set of entities such as objects, types, and variables and a set of relations such as ("is used at", "requires from", "provides to", "is called by", "is the parameter of", etc.) This model is useful because it localizes the interconnections to programming language entities, instead of operating system implementation restrictions such as files (compilation units). Where the unit interconnection model only indicates the general

C unit interconnection model:

```
IM = ({files}, {"includes"})
```

Ada unit interconnection models:

```
IM = ({library units}, {"with"})
Recompilation IM = ({library units},
                    {"with", "changed more recently than"})
```

Modula-2 unit interconnection models:

```
IM = ({modules}, {"import"})
Recompilation IM = ({modules},
                    {"import", "changed more recently than"})
```

make unit interconnection model:

```
IM = ({.o files, .c files, etc},
      {"depends on", "changed since generation", etc})
```

Figure 2.6: Unit interconnection models

location of changes, the syntactic model indicates the syntactic objects that are changed (*i.e.*, a finer degree of locality).

Examples of this model include CMI, Inscape's Infuse [PeKa 87], and Interlisp's Masterscope [TeMa 81]. Its uses include change management, static analysis, smart recompilation, and system modelling. A classic (but somewhat limited) example is the cross-reference utility provided by many compilers. By consulting the cross-reference listing, a programmer can locate the exact places affected by a change and manually propagate the change.[16]

[16] This manual check and propagation is of course extremely error prone for large systems.

Interlisp's Masterscope builds a database of relationships and provides change management by querying the database. Similar facilities are available in Cscope [Stef 85] and Smile [Notk 85]. Static program analysis tools such as *lint* are well known to Unix programmers; *lint* was extremely useful in pre-ANSI C days, but its function has now been incorporated into most of the newer C compilers that support ANSI C.

The most interesting use of the syntactic model occurs in smart recompilation algorithms that Tichy applied to the Berkeley Pascal prototype of "smart make" [TiBa 85, Tich 86][17] and the subsequent extension to this system by Morgenstern for C [Morg 87]. Tichy uses the syntactic model to determine the effects of a change within a module and if recompilation is required. As in the unit model, the relations of change, addition, and deletion are added to the relations of the syntactic model and used to determine whether a particular change necessitates recompilation. For example, a change within a procedure only requires the recompilation of that procedure; a change to the parameter list may necessitate the recompilation of all of its uses. The smart recompilation syntactic interconnection model is given in Figure 2.7.

```
IM = ({entities}, {"is used at", "is deleted from", ...})
```

Figure 2.7: The smart recompilation syntactic interconnection model

While the syntactic interconnection model improves on the unit model in terms of detecting the granularity of changes and recompilation algorithms, there is no

[17] Tichy's work is explained in more detail in Chapter 3.

information concerning *why* the relations between objects exist, only that they do. This problem is addressed by the semantic model.

2.6.3 Semantic model

The semantic model is similar to the syntactic model; in addition, semantic information is placed on the arcs joining the nodes in the system graph, not just the fact that "they do". This model may capture what the original designers and/or builders had in mind when they used the objects to initially build the system. No software development environment known fully supports the semantic model. However, Infuse [Peka 87], the change management component of Inscape [Perr 87b], does use semantic pre- and post-conditions to determine change extents. Given that the environment can determine the implications and extent of changes, Infuse provides the facilities to simulate these changes in addition to propagating them. In this way, the developer can determine whether a set of changes might have too adverse an affect before committing to those changes. Infuse also guarantees that the implications of the changes are carried out completely and consistently. Chapter 4 shows that while the current CMI implementation does not fully support the semantic model, the design allows for easy completion at a later date.

2.7 Recompilation strategies

Modular programming languages that support separate recompilation have various rules for determining the recompilation order when an interface or implementation changes. Most programming languages use a conservative compilation rule to guarantee consistency across compilation unit boundaries. The general rule states that the interface of a module is to be compiled prior to its implementation

and prior to its client modules. This section presents various recompilation strategies based on various interconnection models, and comments on their effectiveness and weaknesses. Chapter 3 details the recompilation problem, and the global interface analysis algorithms as a possible solution.

2.7.1 Traditional recompilation strategies

The traditional rule in strongly typed, separately compiled programming languages is the recompilation of all modules that use resources provided by the changed interface. In many cases, this conservative rule causes unnecessary recompilation thus wasting time and resources returning the system to a consistent state. This conservative rule is typically implemented using time stamps [Teit 84a], as exemplified by the *make* utility [Feld 79]. *Make* uses a depth first search on the dependency graph of a program to compute the set of modules affected by a change in another module, where the change is based on the unit interconnection model. In many cases, this rule causes the unnecessary recompilation of many modules that do not use the resource that was actually changed in the altered interface. For example, the usual treatment of changes and dependencies in Ada can lead to massive recompilations of code when a low-level entity in a frequently used package is changed, even if that change had no effect on the package interface. For C programs, usually, some kind of `global.h` file contains system-wide type and macro definitions (among other things), and it is included by virtually every compilation unit in the program. Using *make*'s primitive rules, if the header file is altered in any way, say just to add comments, the whole system is totally recompiled. The reason

is that the granularity of the change is dwarfed by *make*'s basic unit for analysis, which is the compilation unit as a single object.[18]

Make requires the manual construction of the `makefile` that it uses to determine recompilation order. This is a text file that tells *make* the dependencies that each compilation unit in the system has. A sample `makefile` fragment[19] is shown in Figure 2.8. Even though there are tools, such as *mkmf* [Wald 84], to generate the `makefile` that controls *make*, these `makefiles` quickly become too large and unwieldy to manage. Most Modula-2 compilers are monolithic and base their interface checking on time stamps rather than on an encoding of the interface types. This puts tight constraints on the compilation order of definition and implementation modules: if in a handmade or generated `makefile` the dependencies are even slightly wrong, the Modula-2 linker will complain.[20]

Some systems do not provide any type of automatic recompilation mechanism at all; the programmer must manually remember the dependencies among the modules. Modula-2 compilers do not usually provide a library manager. Most

---

[18] Personal experience in a large project, the IBM C/370 compiler, indicates that this method is unacceptable.

[19] Using IBM VM/CMS syntax.

[20] By contrast, in C you do not need to compile interface definitions (*i.e.*, `.h` files). Thus, it is relatively easy to write a näive makefile that works. However, the C compiler and linker let you produce an executable program from components compiled with incompatible interfaces. Normally, function prototypes can be used to avoid these interface errors, but the programmer is not required to use them. This type of interface fault can be very difficult to track down without using a good compiler or static analysis tools such as *lint*.

produce a `.sym` file from the compilation of a `.def` (**DEFINITION**) file [Powe 84]. This mainly consists of time stamp information and an encoded symbol table. Others simply reparse the `.def` file every time it is imported by some module [Fost 86]. Neither scheme provides automatic compilation of the affected modules when a definition module changes. Quite a few systems, such as the Modula-2 make utility [IBM 88e], provide a limited aid for recompilation by automatically determining the recompilation order, but they still force the programmer to inform the system of "what has changed". Obviously, such a manual mechanism is extremely error prone and can cause very difficult-to-find "bugs".

For Ada, the program library manager contains successfully compiled compilation units and history information about all the compilation units in the program library. The Ada compiler uses the information in the library to verify that dependencies are met. The history file is used to share information about compilation units so that they can be recompiled in the correct order. The Ada rules for compilation are as follows [Davi 84, Dod 83]:

- A compilation unit must be compiled after all library units named by its context clause. The context clause identifies all the library units on which the compilation unit is dependent.
- A secondary unit that is a subprogram or package body (library unit body) must be compiled after the corresponding library unit.
- Any subunit of a parent compilation unit must be compiled after the parent compilation unit. The subunit contains the **separate** clause that identifies the parent compilation unit.

The Ada recompilation rules are:

- A compilation unit is potentially affected by a change in any library unit named by its context clause.
- A secondary unit is potentially affected by a change in the corresponding library unit.
- The subunits of a parent compilation unit are potentially affected by a change of the parent compilation unit.
- If a compilation unit is successfully recompiled, the compilation units potentially affected by the change are obsolete and must be recompiled.

In addition, a library unit may use the "ELABORATE" pragma to inform the compiler of a dependence on another library unit.

The above (re)compilation rules show that the Ada language reference manual (LRM) [DoD 83] does not dictate that a particular implementation must do better than the unit model when it comes to recompilation strategies. This is stated in § 10.3: "The implementation may be able to reduce the compilation costs if it can deduce that some of the potentially affected units are not actually affected by the change." Some Ada compilers require the use of special constructs embedded in the source informing the compiler of the compilation order of the files that affect a package.

Program dependence and call graphs have been used for interprocedural analysis work on systems for programming-in-the-small and programming-in-the-large, using conventional data flow analysis [Burk 87, BuRy 87, Call 88, DeBi 87, FeOW 87, HoRB 88, Ryde 87, RyPa 88, Zade 84] on the elimination of unnecessary recompilation when a parameter change in a subprogram occurs [CKTW 87, CoKe 88, CoKT 85, CoKT 86a, CoKT 86b]. However, the elimination of unnecessary recompilations in software development environments due to interface changes is a slightly different problem.

```
#
# makefile for CMI; VM version
#

cc      = cc                              #  C/370
link    = cmod
options = test define(C370) define(VM)
:
global  = thglobal.h thdebug.h
:

cmi.$(exe): compile bind

bind:
     @echo Linking...
     @vmpush txtlib
     @global txtlib $(txtlib)
     @$(link) $(object0) $(object1) $(object2) ( $(linkopts)
     @vmpop txtlib
     @execos                                 # clean up storage

compile: thmain.$(obj) \
         thmem.$(obj)  \
         :
         :

thmain.$(obj): thmain.c stdio.h string.h signal.h \
             $(global) thmem.h thinit.h  thcntrl.h thsignal.h \
             therror.h thexit.h thhelp.h \
             thmain.h thmain.hp
       $(cc) thmain.c.* ( $(options)

thmem.$(obj): thmem.c stdio.h stdlib.h string.h $(global) therror.h \
      :
      :
```

Figure 2.8: A makefile fragment

2.7.2 The CHILL system

The CHILL system [RuMo 82] supports separate compilation with compile-time interface checking, and it incorporates a method of processing small changes in very large programs. It supports a variety of block-structured programming languages and uses a central database. The database maintains the inter-module consistency. The two main parts of CHILL are the Change Analyzer and the Consistency Checker. CHILL has some of the same goals as CMI, in terms of minimizing recompilations. However, the method used is quite different: the user must insert directives in the source code to inform the system of what has changed.

2.7.3 Smart recompilation

Tichy [TiBa 85, Tich 86] devised an improvement to Feldman's *make* rules, which he termed "smart recompilation", that does not depend only on file time stamps. The *depends* relation that *make* uses is augmented so that a compilation unit is only recompiled if the compilation unit itself changes or if a context[21] upon which the compilation unit depends changes. If the context is modified, a change set, which represents the differences between the old and new version of the context, is made. This change set is then intersected with the corresponding reference set of each dependent compilation unit. This reference set is computed by the transitive closure of dependencies among declarations.

---

[21] Tichy defines context as "a specification of which external objects a compilation unit may reference (import) and which internal objects must provide (export)".

This method is applicable to languages such as Berkeley Pascal and the Unix FORTRAN compiler (*i.e.*, most languages that use context files to communicate). Tichy reports that the method is extremely efficient, resulting in a net savings even if one recompilation is saved.

2.7.4 Smarter recompilation

One of the less desirable features of Tichy's method is that it has a strict notion of inconsistency, and if it determines that a module needs to be recompiled because it is now in an inconsistent state, the programmer can do little about it. Morgenstern [Morg 87] implemented the "smarter recompilation" algorithms proposed by Schwanke and Kaiser [ScKa 88] as a prototype for C, which he called the "Inconsistency Management System" (IMS). Morgenstern defined a module as an independent compilation unit that imports and exports resources (entities). The IMS system lets the programmer decide what inconsistencies are allowed to exist between modules after a context changes. One can experiment with changes to `.h` files, and see the effects the changes would have.

Smarter recompilation [ScKa 86, ScKa 88] is like Tichy's system but relaxes the consistency requirement so that it is better tuned to experimentation; it allows the programmer to choose which (apparent) inconsistencies are allowed to remain (temporarily). However, Tichy raised some objections to this technique [Tich 88].

2.8 Summary

This chapter reviewed some of the work closely related to this thesis, and presented background material on the topics of programming-in-the-large, software

development environments, modularity, and interface specification in modern strongly typed, separately compiled programming languages and interface description languages, visibility control, and recompilation.

Research into programming-in-the-large and the software development environments that support it is rapidly gaining interest. Some of the pertinent software development environments, and how they relate to this thesis, were presented. The benefits of software engineering principles such as modularization, information hiding, and syntactic interface specifications applied to large software systems were reviewed. How modularization and interface specifications are accomplished in a few strongly typed, separately compiled programming languages was also discussed.

The use of an explicit *require-provide* mechanism to augment the usual nesting method for visibility control was presented and a definition of *preciseness* was given. It was shown how the three interconnection models, unit, syntactic, and semantic, greatly affect the nature and usefulness of recompilation algorithms. They depend on the modularity and interface control mechanisms used by both the programming language and the software development environment, as well as the visibility control schemes in use.

Finally, the recompilation problem for programming-in-the-large was presented. Current methods for recompilation were presented, and their weaknesses were highlighted.

CHAPTER 3

Global interface analysis

3.1 Introduction

This chapter discusses the global interface analysis algorithms, proposed by Hood, Kennedy, and Müller [Mull 86, HoKM 87], as a solution to the recompilation problem outlined in Chapter 2. These algorithms analyze and limit the effects of a change to an interface in a software system. By using a syntactic interconnection model, they improve on traditional recompilation strategies, which typically use only the unit interconnection model to determine recompilation ordering. The algorithms are designed assuming a software development environment that provides efficient access to the compilation dependencies and module interfaces. Chapter 4 shows how these dependencies are extracted in the CMI implementation. The algorithms operate on recursive inter-module dependencies, since, in programming-in-the-large, such inter-module dependencies occur frequently.

Attributed graphs are well suited to represent structured sets of data objects [Rohr 87]. A graph model is the basis for the CMI system, and is the most important concept of the Rigi model [Mull 86, MuKl 88]. This chapter also presents some of the necessary background material on graph theory and graph terminology relevant to the algorithms, and with these definitions, give the graph model of a large software system.

Finally, the interface recompilation algorithms that operate on this graph model are detailed, and the theoretical efficiency of the algorithms is given. Several sections

of this chapter are patterned after the paper "Efficient Recompilation of Module Interfaces in a Software Development Environment" [HoKM 87].

3.2 The recompilation problem

To ease the burden of remembering exactly "who used what" in a relatively large system, tools have been developed to automate the process of bringing a system back up to date in a consistent manner after a change. Of course, the easiest method is to blindly recompile the entire system, but this is not practical for systems of any relevant size. As shown with the syntactic interconnection model and exemplified by Tichy's smart recompilation model, many factors affect how a change in one module is reflected in the rest of the system. The modularization mechanism determines the overall system graph structure, while the interfaces and visibility control mechanisms determine the topology of the graph, the semantic arcs that link the objects together. The interconnection model directly affects the success of the recompilation strategies since it is the granularity of the change, and how the change is propagated to the affected modules, that determines the suitability of a particular recompilation strategy.

An important problem in large, evolving software systems is the management of interface changes [Tich 79], especially those changes that occur in the interfaces of low-level components. One of the criticisms of separate compilation implementations is the proliferation of disorder in a system when a single interface is changed and then recompiled [RuMo 82]. A change-propagation/configuration management tool for software development has the important task of ensuring that changes in source text are correctly propagated to the modules used to construct executable versions of a software system while simultaneously attempting to minimize the

amount of recompilation [LiLW 87]. The problem of eliminating unnecessary recompilations is important since it is difficult to establish and maintain consistent interfaces among the various components of a complex system [Tich 79]. The recompilation and coordination cost of changing a syntactic interface in large software systems is often prohibitive because too many software components depend on it. This leads to interfaces that are frozen before they are sufficiently explored and tested [Teit 84]. Considering that, for very large systems, massive recompilations may prove very costly, wasting many programming hours waiting for the system to fully recompile, there is a need to manage the problem rather than avoid it.

Consistency checking and recompilation due to the alteration of a basic interface is usually much more time consuming than changing an implementation part [Tich 82], even though the interface is usually altered less frequently than the implementation part and compiles much faster (no code generation). Numerous other interfaces and implementations in the system may directly or indirectly depend on a basic interface while an implementation part has only local dependencies.

When the implementation part of a module is changed, the interface part should remain unaffected. Thus, any other modules in the system that *depend* on the module are not affected. However, when the interface part of a module is changed in some manner, any other module in the system that *uses* resources that the changed module provides may be affected. This problem is exacerbated when the changed interface is at or near the bottom of the system hierarchy; the changes may ripple through the *require-provide* lists of the modules in the system [HoKM 87]. Thus, a seemingly innocent change in one interface may affect a large number of modules in the system. These affected modules must then be recompiled, even though they may not at all be affected by the changes made in the original module.

The recompilation problem has different flavors during the various stages of the software life cycle. In the rapidly changing development environment, the time needed to observe the effects of a change to an interface often causes the programmer to attempt to circumvent the compiler's cross-module checking to avoid the tedious wait of a (possibly) long recompilation [Rain 84], or to alter existing programs in such a way that recompilation is minimized, often to the detriment of a "clean" design. This evasion of compiler checks is contrary to the safe modular philosophy that the language provides. It would be more beneficial if the programmer could be informed of the effects of the intended change before the change were made permanent. The programmer could minimize the time needed for recompilation, thus allowing various alternatives in the interface design to be explored before a final decision is reached.

During the maintenance phase of a software project, changing a low-level interface involves tracking down all its clients and making sure that they are recompiled in the correct order to maintain a consistent system. A software development environment for programming-in-the-large should aid the maintainer of a large, complex system by helping decide whether a change in a basic interface should be implemented by computing the set of affected modules. If the set is acceptable, the change can be implemented; otherwise, it can be undone.

The key to avoiding needless module recompilation is to capture both the exact nature of the dependencies between the changing interface and the rest of the system as well as the exact nature of the change (if any) between the older and newer versions of the interface. The modularization and interface mechanisms as well as the visibility control mechanisms are used to construct a semantic graph model of the system to capture inter-module dependencies (as opposed to intra-module dependencies).

3.3 Graph terminology

Given a *simple, directed* graph $G = (V, A)$ with n vertices and m arcs, the *Boolean connection matrix* or *adjacency matrix* C of G is an $n \times n$ array of 0's and 1's such that $C[i, j] = 1$ if there is an arc from v_i to v_j and $C[i, j] = 0$ otherwise.[22] This matrix model of a graph is conceptually simple but not very efficient for implementation with respect to space, since even for sparse graphs[23] the matrix requires $O(n^2)$ storage. A more efficient representation of a graph uses adjacency lists, where each node in a graph has a linked list of other nodes that the node is connected to in the graph.

The reflexive, transitive closure A^* of the relation A satisfies the condition vA^*w iff $v = w$, or there exists a directed path from v to w. Given this definition, the *reflexive, transitive closure matrix* C^* is an $n \times n$ array with the property $C[i, j] = 1$ if there exists a directed path of length ≥ 0 from v_i to v_j and $C^*[i, j] = 0$ otherwise. The C^* matrix may be used to decide if there is a path between any two nodes in the graph in $O(1)$ time, but again a space penalty of $O(n^2)$ must be paid to store the matrix. Using the adjacency list representation, the test can be performed in $O(n)$ time. Notice that the path length may be 0, which means that a node is only connected to itself.

A directed graph $G = (V, A)$ is *strongly connected* if there is a path from v to w and a path from w to v for all $w, v \in V$. A *strongly connected component* of a directed

---

[22] A simple graph has no parallel edges. A directed graph is one in which the direction of the arc that joins two nodes is of importance. Note that, throughout the thesis, the terms *vertex* and *node* are used interchangeably, as are *edge* and *arc*.

[23] Graphs with "few" arcs connecting various nodes together.

graph G is a maximal strongly connected subgraph of G.[24] An arc that connects two strongly connected components is called a *cross-component* arc. The interconnections among the strongly connected components can be represented by constructing the *reduced graph* of G. The vertices of the reduced graph are the strongly connected components, and the arcs are the cross-component arcs. The strongly connected components cover the entire graph G. The reduced graph is always an *acyclic* graph.

A *topological sorting* of the acyclic graph imposes an ordering of the nodes. Consider a directed, acyclic graph $G = (V, A)$ with n vertices. The set of arcs A defines a reflexive, antisymmetric, transitive relation and, hence, a *partial order* on the set of nodes V. The pair (V, A) is a *partially ordered set (poset)*. Define $(v_1, v_2, \ldots, v_n)$ to be a sequence of all n elements in V. Then $(v_1, v_2, \ldots, v_n)$ is a *topological sorting* of V relative to A iff for all $v_i, v_j \in V$, $(v_i, v_j) \in A$ implies $i < j$.

3.4 Attributed graph model of a software system

A software system may be modelled as an attributed graph, with the vertices in the graph representing objects (modules) in the system, and the arcs representing semantic dependencies between the objects in the system. This is the basis for the Rigi model. Alpern *et al.* [ACRS 88] use the terms *boxes* and *cables* to describe vertices and arcs in an attributed graph model of a program. The boxes are connected to one another via cables, which connect to *ports* in the boxes. The cable connections make explicit the inter-module requisitions and provisions or resources.

---

[24] The strongly connected components of a directed graph $G = (V, A)$ can be found in $O(n + m)$ time using a variant of the depth first search algorithm [AhHU 83].

Which entities in a system are chosen as nodes depends on the use of the graph model. In code optimization, a node may be a basic block or a single variable. For programming-in-the-small, a node may be a single entity such as a type or `variable`. In programming-in-the-large, a node is a module that may contain other entities.[25]

The arcs that connect the nodes in the graph can also have different meanings, and be interpreted in different ways by different tools, much like *views* in a relational database. In Chapter 2, visibility control and interconnection models were discussed. To describe visibility control, Wolfe *et al.* [WoCW 86, WoCW 88] used *require* and *provide* relationships as the criteria for joining two (or more) nodes in the graph. CMI is mainly concerned with compilation dependencies above the unit model; therefore, the *require-provide* relationships (among others) are also used.

3.4.1 Compilation dependency graph

The compilation dependencies of a set of modules of a software system or subsystem form a directed graph $G = (V, A)$ called the *compilation dependency graph*. A vertex $v \in V$ denotes a module. An arc $(v, w) \in A$ from v to w indicates that w uses, or *requires*, one or more resources that is provided by v. Let $G_i = (V_i, A_i)$, $1 \leq i \leq k$, $1 \leq k \leq n$, be the k strongly connected components of a compilation dependency graph $G = (V, A)$. Then the reduced graph of G, called the *acyclic compilation dependency graph*, is denoted $G' = (V', A')$, where V' is the set of strongly connected components G_i, and $A' = \{(v, w) \mid (v, w) \in A,\ v \in V_i, w \in V_j, i \neq j\}$.

---

[25] Chapter 4 shows that a node in CMI is an *object*, with the semantics of the various arcs that connect the nodes depending on the class of the object.

An arc $(v, w) \in A'$ from v to w indicates that v is to be compiled prior to w. Each $v \in V$ has a (possibly null) set of successor nodes that must be compiled after v. This set is defined as $Succ(v) = \{G_i \mid$ there is some $w \in G_i$ such that $(v, w) \in A'\}$.

The interface parts that correspond to the set of vertices V_i of a strongly connected component G_i are merged and compiled as a single unit.[26] The implementation parts that correspond to the compiled interface part may be subjected to the Smart Recompilation algorithm by Tichy and Baker, which determines if the implementation needs to be recompiled too.

3.4.2 Rooted compilation dependency graph

Given the compilation dependency graph $G = (V, A)$, and a distinguished vertex $r \in V$ in G, then the *rooted compilation dependency graph* of r is the graph $G(r) = (V(r), A(r))$, where $V(r) = \{v \mid v \in V$ and there is a path from r to $v\}$, and $A(r) = \{(v, w) \mid (v, w) \in A$ and $v, w \in V(r)\}$. The reduced graph of $G(r)$, called the *acyclic rooted compilation dependency graph*, is $G'(r) = (V'(r), A'(r))$. The rooted compilation dependency graph is used when analysis is done on the change of a particular node in the graph: the root node r.

$V'(r)$ therefore denotes the partitioning of those interfaces that would be recompiled under a traditional compilation rule, using the unit interconnection model, if r were changed. In addition, a topological ordering of $V'(r)$ relative to $A'(r)$ defines a possible recompilation order.

---

[26] If a system is designed using the principles of information hiding, then a strongly connected component of a compilation dependency graph contains either a *single vertex* or a *small number* of vertices.

3.4.3 Recursive compilation dependencies

The compilation dependencies of the *provide* parts of a set of modules in a system typically form a directed acyclic graph, since most strongly typed, separately compiled programming languages do not support recursive dependencies across compilation unit boundaries. However, in the realm of programming-in-the-large, recursive inter-module dependencies are as natural as intra-module dependencies. They are induced by indirect recursive type declarations and indirect recursive procedure calls. The algorithms presented below are not restricted to acyclic compilation dependencies; recursive compilation dependencies do not pose a special problem. However, the language definitions of most strongly typed, separately compiled programming languages preclude recursive dependencies between two (or more) compilation units. This restriction is explained in detail in Chapter 4.

3.5 Interface recompilation

Given the above definitions and the basis of the graph model, the interface recompilation algorithms may now be presented. The global interface analysis algorithms are based primarily on the syntactic interconnection model, but they also use some features of the semantic model as well. In order to determine the effects of a change in a basic interface on the entire system (*i.e.*, to separate the affected compilation units from the unaffected ones), the nature of the change as well as the contents of the modules involved must be analyzed.

First the change in the basic interface is analyzed to determine the effect of the change on the interface and its environment. The type of the change is classified as *inconsequential* (Type I), *local* (Type II), or *global* (Type III). If the change is inconsequential, no further action needs to be taken. If the change is local, then the

Tichy-Baker algorithm may be invoked to determine if the implementation needs to be recompiled. The environment of the module is not affected. If the change is global, then the change is propagated through the *require* and *provide* (import and export) lists of the interface's clients.

3.5.1 The object sets *Inside* and *Outside*

When a change occurs in module d, the affected objects are stored in the sets *Inside(d)* and *Outside(d)*. The elements of these sets are nameable entities in a programming language, such as `procedure`, `constant`, `package`, `type`, or `variable`. The object set *Inside* of a module d describes the affected objects of d.[27] The object set *Outside* of d describes those objects that may affect the environment of d (*i.e.*, the changes that can be seen by the clients of d).

For each module visited, the change propagation algorithm computes the object sets, tests whether the sets are empty, and proceeds according to the following case analysis:[28]

- $Inside = \emptyset$:

  The module is not affected (i.e., neither the interface nor its implementation is affected).

- $Inside \neq \emptyset$:

---

[27] It corresponds to the *change* set in the Tichy – Baker recompilation algorithm, which is used to determine the effects on the implementation variants of d.

[28] The four possibilities are not mutually exclusive (*i.e.*, more than one case can occur simultaneously).

The interface is affected and has to be recompiled. The Tichy – Baker algorithm is then used to determine whether its implementation is affected.

- *Outside* $= \emptyset$:

  None of the clients of the interface are affected.

- *Outside* $\neq \emptyset$:

  The clients of the interface may be affected. The affected objects are propagated through the pertinent require and provide lists.

The algorithm to extract the change type is depicted in Figure 3.1.

3.5.2 Sorting the dependence graph

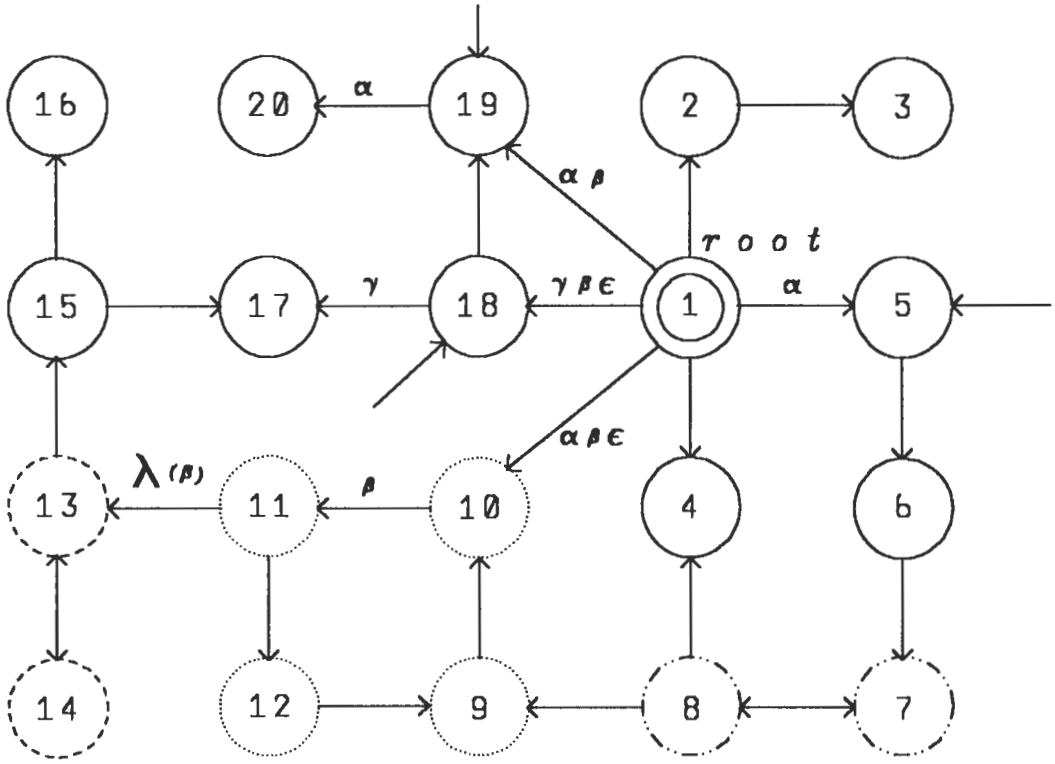
The object sets of the changed interface r must be propagated throughout its rooted compilation dependency graph $G(r)$ to determine the set of affected modules after an interface change. It is assumed that the compilation dependency graph *CDG* has already been transformed into an acyclic compilation dependency graph *ACDG* by reducing the graph into subgraphs of strongly connected components. This reduced graph is now sorted topologically to produce a sorted, acyclic, compilation dependency graph *SACDG*. This transformation process is outlined in Figure 3.2. A sample compilation dependency graph is depicted in Figure 3.3.

$$\text{system} \Rightarrow CDG \Rightarrow ACDG \Rightarrow SACDG$$

Figure 3.2: Compilation dependency graph transformations

```
type
  ChangeSet = record
    modified, added, deleted : set of resources;
    modreq, addreq, delreq : set of requirements;
    modprv, addprv, delprv : set of provisions;
    oldprv : set of provisions;
  end;

  ChangeType = (inconsequential, local, global);

  ChangeInfo = record
    outside : set of resources;
    require : set of requirements;
    ct : ChangeType;
  end;

function GetChangeInfo (old, new : vertex) return ChangeInfo;
var
  C : ChangeSet;
  ci : ChangeInfo;
  inside : set of resources;
  GetChangeSet : external function (vertex, vertex) return ChangeSet;

begin
  C = GetChangeSet (old, new);
  inside := C.modified ⋃ C.added ⋃ C.deleted;
  ci.outside := (inside ⋂ C.oldprv) ⋃ C.modprv ⋃ C.addprv ⋃ C.delprv;
  ci.require := modreq ⋃ addreq ⋃ delreq;
  if info.outside ≠ ∅
    ci.ct := global;
  elsif info.Imp ≠ ∅
    ci.ct := local;
  else
    ci.ct := inconsequential;
  return ci;
end GetChangeInfo;
```

Figure 3.1: Change type algorithm



This compilation dependency graph is made up of twenty nodes. The number in each node represents the topological sort number made from traversing the graph from the root node. The incoming arcs at nodes 5, 18, and 19 suggest that this is a subgraph of a larger system. The root node 1 is where a change has occurred. Nodes that are made up of the same type of dashed line are members of the same strongly connected component. For example, nodes 6 and 7, nodes 9, 10, 11, and 12, and nodes 13 and 14. The letters on the arcs indicate the flow of resources from one node to another. The special lambda character from node 11 to node 13 indicates a transitive dependence on resource beta.

Figure 3.3: Sample rooted compilation dependency graph

The SACDG algorithm presented below takes a compilation dependency graph G as input and produces a (topologically) sorted, reduced graph $s(G)$. It avoids backtracking by visiting the vertices of $G(r)$ in topological order (*i.e.*, it is guaranteed

that a node is only visited if all of the non-empty object sets of its predecessors are available).

3.5.2.1 SACDG algorithm

Input: A compilation dependency graph $G = (V, A)$.

Output: A sorted, reduced compilation dependency graph $s(G)$.

Method: The algorithm consists of two steps:

1. Determine the reduced acyclic compilation dependency graph $G' = (V', A')$ by computing the strongly connected components of G.

2. Determine the sorted, acyclic compilation dependency graph $s(G) = (V'', A'')$ by computing a topological ordering of V' relative to A' , and label each node with its topological number in the range 1 to $|V'|$.

The algorithm may be performed once for the entire compilation dependency graph G, or on demand for a rooted compilation dependency graph $G(r)$. If the system structure is fairly stable, as it usually is during the maintenance phase of a software project, then these preprocessing steps are computed for the entire graph G. If the system structure changes frequently, as it typically does during the design and implementation phase, then the strongly connected components and the topological sort are only computed for the rooted graph $G(r)$.

The SACDG algorithm above executes in $O(n + m)$ time. The proof is given in [Mull 86], but stated informally, the dominant factors in the algorithm are the computation of the strongly connected components in Step 1 and the topological sorting of the reduced graph in Step 2. Both steps may be done with a variant on

the depth-first search algorithm, in $O(n+m)$ time [AhHU 83, Mehl 84]. Chapter 4 presents the implementation details of this algorithm.

3.5.3 Minimal interface recompilation algorithm

After the SACDG algorithm has reduced the compilation dependence graph G to a sorted, acyclic, compilation dependence graph $s(G)$, the minimal interface recompilation algorithm (MIRA) may be applied. This algorithm computes the minimal (*i.e.*, optimal) set of modules that must be recompiled after a change to an interface, and then recompiles the affected modules in the correct order.

3.5.3.1 MIRA algorithm

Input: A compilation dependency graph $G=(V, A)$, its sorted reduced graph $s(G)$, a module $r \in V$ whose interface has changed, and a set of identifiers *Change* that denotes changed entities in r.

Output: The set of modules whose interfaces are affected by *Change* is recompiled in the correct order. If a module's interface does not compile successfully, then the algorithm requests that the interface be edited before it resumes the sequence of module compilations. If this editing operation alters the interface of the module, then the algorithms must be restarted, since the topology of the compilation dependence graph may have changed.

Method: The interface recompilation algorithm first determines the object sets of r and the magnitude of the change. The magnitude of change is characterized as either *inconsequential* (Type I), *local* (Type II), or *global* (Type III), where an inconse-

quential change has no effect, a local change affects the module but not its clients, and a global change potentially affects the clients of r.

If the change is of Type III, then the algorithm computes the object sets for the affected nodes in topological order. An arc is followed only if the set *Outside* of its source node is not empty and if its destination node has not been visited previously. The algorithm selects which node to visit next by maintaining the possible strongly connected successors in a *heap* data structure (*i.e.*, a working set on which a *min* operation may be performed efficiently). The minimal interface recompilation algorithm is shown in Figure 3.4.

The minimal interface recompilation algorithm outlined computes the set of affected modules and the correct recompilation order of these modules in $O(n \log n)$ time, where $G'^*(r) = (V'^*(r), A'^*(r))$ is the *affected* rooted compilation dependency graph, and $n = |V'^*(r)|$. The full proof for this bound is given in [Mull 86], but it follows easily from the above discussion and may be stated informally as follows. The amount of work done by the algorithm is dominated by three statements:

1. `R := DeleteMin (S);`
2. `PropagationSets (R);`
3. **if** (•) **and** (•) **then** S := S $\bigcup$ Succ (d);

It is clear that $|S|$ can at most be n; therefore, statement (1) is executed at most n times. If the set S is implemented as a heap, any one execution of statement (1) can be performed in $O(\log |S|)$ time. The algorithm only visits those nodes that are affected by a change in r, and a node is only visited if all of its predecessor nodes have been visited. Since the maximum size of $|S|$ is n, the total time required for all executions of statement (1) is bounded by $O(n \log n)$.

```
procedure InterfaceRecompilation (old, new : vertex; sG : graph);
var
  ci : ChangeInfo;
  S : heap;     (* heap with key field = topological number *)
  R : scc;      (* strongly connected component *)
  d : vertex;   (* module *)
  GetChangeInfo : external function (vertex, vertex) return ChangeInfo;
  CompileEdit : external procedure (vertex);
  DeleteMin : external function (heap) return scc;
  PropagationSets : external procedure (var scc);
begin
  ci = GetChangeInfo (old, new);
  select ci.ct
    when inconsequential => nop;
    when local => CompileEdit (new);
    when global =>
      "set all outside sets (except new's) to ∅"
      S := Succ (new);
      while S ≠ ∅ do
        R := DeleteMin (S);       (1)
        PropagationSets (R);      (2)
        for each d ∈ R do
          if d.inside ≠ ∅ then
            CompileEdit (new);
          if ("d is source of a cross component arc") and
             (d.outside ≠ ∅) then
            S := S ⋃ Succ (d);    (3)
        end
      end
  end
end InterfaceRecompilation;
```

Figure 3.4: Minimal interface recompilation algorithm

For the same reason as above, statement (2) is executed n times. Since, by assumption, the *require* list of each interface is of constant size, each node has a constant indegree. Thus, each execution of PropagationSets can be done in time proportional to the size of the strongly connected region R. Since each vertex in V

is in only one strongly connected component (guaranteed from the SACDG algorithm), statement (2) does a total of $O(n)$ work for all of its executions.

Finally, for statement (3), note that any one strongly connected component can occur in the Succ (d) set of module d for at most a constant number of different d's (*i.e.*, any strongly connected component can be the sink of a finite number of sources, bounded by n). Thus, any one strongly connected component of the affected region is inserted at most a constant number of times into S. Since each insertion into set S can be done in time $O(\log |S|)$ (S is implemented as a heap), and $|S| \leq n$, the total amount of time for all executions of statement (3) is $O(n \log n)$.

3.5.4 Change sets

The global interface analysis algorithms must determine the differences between a newer and an older version of a changed interface. These differences are captured in the object sets *Inside* and *Outside*, while the magnitude of the change is reflected in the *change type*, as outlined above. The *change sets* are defined as the object sets of a changed interface, and the *change propagation sets* as the object sets of the interface's clients. The mechanism used to compute these change sets is given below.

Given a compilation dependency graph $G = (V, A)$, and a specific changing interface $r \in V$, a subgraph of G, rooted at r, is formed. Two versions of the evolving node are created: r_{old} and r_{new} (a revision of r_{old}). Let $G_{old} = (V_{old}, A_{old})$ be the rooted compilation dependency graph of r_{old}. It is assumed that G_{old} was in a *stable* state (*i.e.*, all the nodes in V_{old} compiled correctly). The differences between r_{old} and r_{new} are recorded in the sets defined below.

An entity in the change set *Inside* denotes an added, deleted, or modified declaration. An entity is *modified* if its declaration appears in both r_{old} and r_{new} and was changed in some way. This includes entities that were modified in the *require* list.[29] For example, in r_{old}, the entire services of a particular module may have been requested, but in r_{new} only particular entities provided by the source module may be requested. An entity is *added* if its declaration is not present in r_{old} but is present in r_{new}. This includes entities that are added to the *require* list. An entity is *deleted* if its declaration is present in r_{old} but not in r_{new}. This includes entities that are deleted from the *require* list.

In addition to these entities, the set *Inside* includes those entities that are indirectly modified. For example, a structured type is indirectly modified if a subtype, which is used in creating the composite type, is modified. The set of indirectly modified entities can be computed by performing a transitive closure under declaration dependence on the source program; the semantics of this analysis varies with the programming language being used. Indirectly modified declarations may also include those affected by changes in storage allocation. For example, if the compiler assigns storage locations to successive variable locations, and a composite type has two (or more) of its subfield's ordering physically changed, but semantically remaining the same, then the modified composite type must also be included in the set *Inside*. This closure can be computed in constant time when the number of declarations in the interface of a module is kept small, as proposed by good software design principles. Computation of the set *Inside* is shown in Figure 3.5, and an example of an interface with a changed *Inside* set is given in Figure 3.6.

---

[29] It is important to note that modifications to the require or provide lists of a module may change the topology of the compilation dependency graph.

$$Inside = AddReq \cup DelReq \cup ModReq \cup AddDcl \cup DelDcl \cup ModDcl$$

$$Outside = (Inside \cap Provide) \cup AddPrv \cup DelPrv \cup ModPrv$$

Figure 3.5: Computation of the object sets *Inside* and *Outside*

The set *Outside* of a module *d* is obtained by combining the set of changes to the provide list with the set of those locally affected identifiers that are provided. Changes to the provide list include added or deleted entities, as well as changes to the visibility of a provided entity. For example, in r_{old}, an entity may be provided to all objects in the system, but in r_{new} the provision constraints may be tightened to provide the entity to only a subset of all objects in the system. Computation of the set *Outside* is shown in Figure 3.5, and an example of an interface with a changed *Outside* set is given in Figure 3.7. The set *Provide* are those entities in the *provide* list, and the sets *AddPrv*, *DelPrv*, and *ModPrv* are those entities added to, deleted from, or changed in the *provide* list, respectively.

3.5.5 Change types

The effects of an interface change are partitioned into three categories:

Type I A Type I change is said to be *inconsequential* if it has neither local nor global effects. This category includes layout changes, indentation changes, or the addition/deletion/modification of comments. A change is inconsequential if the set *Inside* is empty.

The old interface $m0_{old}$

```
DEFINITION MODULE m0;
  FROM m1 IMPORT v1, v2, t1;
  IMPORT m2;
  TYPE t2 = RECORD
      ac : ARRAY OF char;
      i : cardinal;
      t : t1
    END;
END m0.
```

The new interface $m0_{new}$

```
DEFINITION MODULE m0;
  FROM m1 IMPORT v1;      (* v2, t1 no longer imported *)
  FROM m2 IMPORT v3;      (* now specific request of v3, not all m2 *)
  TYPE t2;                (* type t2 now made opaque *)
END m0.
```

The set *Inside* for this changed definition module is {v2:variable; t1:type; v3:variable; t2:type;}.

Figure 3.6: An altered Modula-2 interface with a non-null *Inside* set

Type II A Type II change is said to be *local* if it affects only the changed interface and its implementation module and none of the direct or indirect clients of the interface are affected. This category includes changing the implementation of opaque types, or the addition or removal of a resource that no client was or is requesting. A change is local if its object set *Inside* is not empty, but its *Outside* set is empty.

The old interface $p0_{old}$

```
with p1;
package p0 is
  type t2 is record
    c : character;
    p : access p1.t1;
    t : p1.t1;
  end record;
  function f (m1, m2 : p1.t1) return p1.t1;
end p0.
```

The new interface $p0_{new}$

```
package p0 is
  type t2 is private;       -- representation of t2 now hidden
                            -- package p1 no longer required
                            -- function f no longer provided
  private
    type t2 is record
      c : character;
      t : access t2;
    end record;
end p0.
```

The set *Outside* for this changed package is {t2:type; f:function;}.

Figure 3.7: An altered Ada interface with a non-null *Outside* set

Type III A type III change is said to be *global* if it affects the clients of the changed interface. In this case the object set *Outside* is not empty. This category includes the addition, deletion, or alteration of any resource that was on

the provide list of the object and was or is being used by other client modules, as well as any change in preciseness in resource provision.

3.5.6 Change propagation sets

The *change propagation sets* are the object sets of a changed interface's clients. The global interface analysis algorithms determine the set of affected modules of a changed interface by first starting with the direct clients of the changed interface. Then the change sets (as computed above) are propagated through the *require-provide* lists of the clients to determine if any other modules in the system are affected. This process continues until no more affected modules are found. An excellent metaphor for this process is waves rippling in water: the first "splash" is made by the changed interface, and the first ripple affects the direct clients. As the wave spreads out, the intensity of the wave decreases until it finally vanishes. For recompilation, this "decreasing of wave intensity" is the filtering of the change though the *require-provide* lists, which may be thought of as filters or dampers, of the modules in the system. Usually, the set of changes that make it through layer $i+1$ of the interfaces are smaller than layer i, and so on until the change set becomes null. A graphic illustration of this filtering effect is given in Figure 3.8.

Given a compilation dependency graph $G = (V, A)$, and a changing interface denoted by $r \in V$, let $G(r) = (V(r), A(r))$ be the rooted compilation dependency graph of r, and let $G'(r) = (V'(r), A'(r))$ be its acyclic, rooted compilation dependency graph. The effects of a global change in r on the modules of $G(r)$ are determined by filtering the change through the contents of the strongly connected components of $G(r)$. If a strongly connected component contains exactly one module, then the filters are the *require* and *provide* lists of that module. If a strongly connected component contains

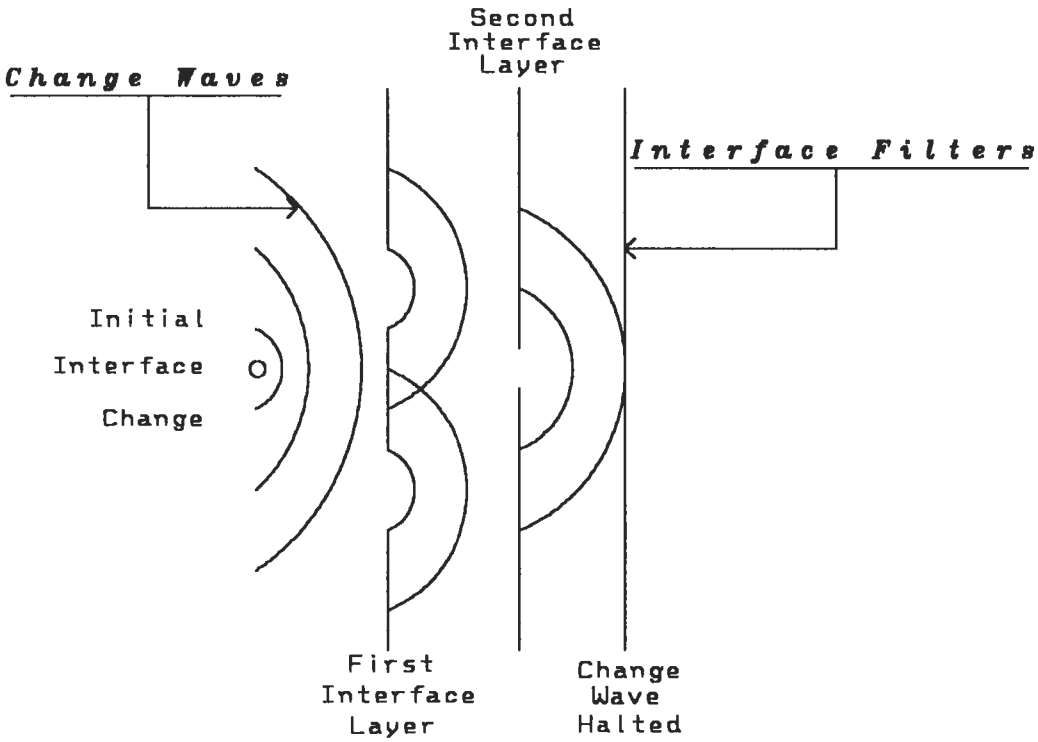


Figure 3.8: Propagating changes through interface filters

more than one module, then the filters are the *require* and *provide* lists of the modules that are incident upon the cross-component arcs.

Given that *Inside(r)* and *Outside(r)* are the object sets of r, let $R \in V'(r)$ be a strongly connected component of $G(r)$. $D(R) \subseteq V$ is defined to be the set of modules that belong to R. If $d \in D(R)$, then $Req(d) \subseteq V$ is defined to be the set of modules from which d requires objects. Also define $Require(d, x)$ to be the set of entities required by d from x. Given these definitions, the change propagation sets of the

strongly connected components of a compilation dependency graph G may be computed, as depicted by the algorithm given in Figure 3.10.

There are two possibilities to consider: a strongly connected component either contains a single node or more than one. These two cases are considered separately below.

Case 1 $|D(R)| = 1$. The strongly connected component R has only one node. Then

$$Inside(d) = \left[\bigcup_{x \in Req(d)} (Require(d, x) \cap Outside(x)) \right]^*$$

where the * represents transitive closure, and

$$Outside(d) = Inside(d) \cap Provide(d)$$

A graphic depiction of the flow of resources through a node that is a member of a strongly connected component with only one member is given in Figure 3.9.

Case 2 The strongly connected component R has more than one module. The set $D_{c\text{-}c} = \{d \mid d \in D(R)$ and there is an arc $a \in V'(r)$ such that $(a, d) \in A'(r)\}$ is defined as the set that contains those nodes of $D(R)$ that are the sinks of cross-component arcs. The sets *Inside(d)* and *Outside(d)* must first be calculated for every $d \in D$. This calculation

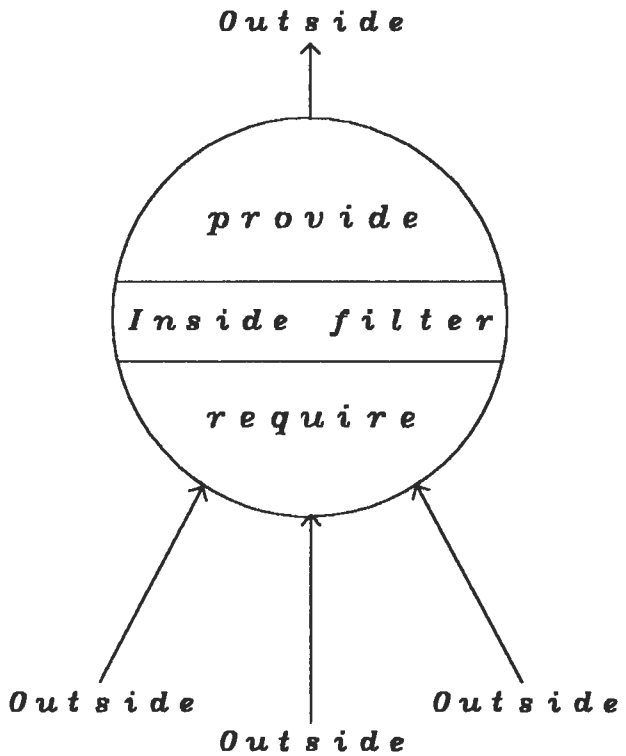


Figure 3.9: Propagation set filter for Case 1

may be done using data flow analysis by solving the following two simultaneous equations [Kenn 81]:

$$Inside(d) = \bigcup_{p \in Req(d)} \Upsilon_p\,(Outside(p))$$

$$Outside(d) = \Upsilon_d\ (Inside(d))$$

where $\Upsilon_d(\Delta)$ is the set of all declarations in d that are affected if the set of changes Δ is propagated into d. Note that, since vertices have been visited in topological order, then for all $p \in Req(d)$ where p is not in R, $Inside(p)$ and $Outside(p)$ have already been calculated. (These are precisely the sets of modified resources that are being propagated into d from outside of its strongly connected component.) If the set $Outside(d)$ of a module d is empty, then all its emanating arcs are deleted from R. If the set $Inside(d)$ of a module d is empty, then the vertex d is deleted from R. If the remaining graph is acyclic, then the vertices of the graph are topologically sorted and recompiled in the topological order. If the remaining graph still contains cycles, then the interfaces corresponding to the vertices in a cycle are recompiled as a monolithic block.

The algorithm given in Figure 3.10 computes the change propagation sets *Inside (d)* and *Outside (d)* for a particular strongly connected component d. The CMI implementation of this algorithm currently does not allow $|D(R)| > 1$. By allowing only $|D(R)| = 1$ a unique solution to the equations is guaranteed. Moreover, most programming languages such as Ada and Modula-2 do not allow recursive inter-module dependencies.

3.6 Summary

This chapter has presented the global interface analysis algorithms, which analyze and limit the effects of a change to a basic interface in a software system.

```
procedure PropagationSets (var R : scc);
var
  D : set of vertex;    (* the set of modules in R *)
  d : vertex;           (* a module in D *)
  DefMods : external function (scc) return set of vertex;
  TransitiveClosure : external procedure (var set of vertex);
begin
  D := DefMods (R);
  if |D| = 1 then
    d.inside := TransitiveClosure [ ⋃ (Require(d, x) ⋂ x.outside)];
                                    x ∈ Req(d)
    d.outside := d.inside ⋂ Provide (d)
  else
    (* calculate d.inside, d.outside, for all d in R  *)
    (* -- this will be done using standard dataflow   *)
    (* -- analysis techniques                         *)
    (* -- Solve assumes that d.inside, d.outside have *)
    (* -- already been calculated for all modules     *)
    (* -- outside R that propagate into R.            *)
    Solve (R);
    for each d ∈ D(R) do
      if d.outside = ∅ then
        "delete all outgoing arcs of d";
      if d.inside = ∅ then
        "delete vertex d";
    end
    "sort the remaining vertices topologically"
  end
end PropagationSets;
```

Figure 3.10: Propagation sets

These algorithms operate on recursive compilation dependency graphs and compile the modules that are affected by an interface change in the correct order. The efficiency of these algorithms is measured by three parameters: the preprocessing time, the space required to store the sets *Inside* and *Outside*, and the time of the recompi-

lation algorithm. The preprocessing time is linear in the size of the compilation dependency graph, the space requirement is linear in the size of the affected graph, and the recompilation algorithm performs in $O(n \log n)$ time, where n is the size of the affected compilation dependency graph.

In addition, an attributed graph model of a software system was presented. This graph model is the basis for the Rigi model and the backbone of the CMI implementation presented in Chapter 4.

CHAPTER 4

The CMI implementation

4.1 Introduction

This chapter describes the Changing Module Interfaces (CMI) implementation of the global interface analysis algorithms discussed in Chapter 3. The current implementation is targeted to strongly typed, separately compiled programming languages such as Ada or Modula-2, and the CMI module interconnection language in particular. The environment in which CMI was developed was not the same as the target environment for integration of CMI with Rigi. This placed some special requirements on the design of CMI that might otherwise have been unnecessary, but also produced some very desirable results, such as allowing CMI to run as a stand-alone tool. This environment, the restrictions it imposed, and the benefits it produced are discussed.

The internal design of CMI is presented. As in the Rigi editor, the internal database is an attributed graph that represents the software system under investigation. Each node in the graph is an *object*. The arcs in the graph represent various types of *dependencies* between objects in the graph, as well as the flow of *resources* between producer and consumer objects. The object classes form a logical hierarchy that CMI uses as a graph structuring mechanism. This data structure (graph) may be thought of as a semantic network with many different "views", like the views that a relational database provides.

The CMI module interconnection language is presented. This language was developed initially as a common intermediate form for targeted programming

languages, but quickly grew into a powerful language in its own right. It is shown how targeted programming languages can be transformed into CMI, and what role the attributed graph model, which is the basis for CMI, played in the design of the language.

The implementation details of the algorithms for global interface analysis and minimal recompilation are discussed. These algorithms manipulate the objects in the internal graphical database, using the dependencies and inter-node resource flow to analyze and limit the effects of a change to an object (a basic interface) in the system graph. Details of how the inter-object (inter-module) dependencies are extracted from the source text are also given.

The limitations of the current implementation are discussed. No implementation is perfect, and there are some places where time and/or resource restraints have imposed restrictions on what could be done at the present time. Adding support for other programming languages is also discussed, with some ideas on the work such additions would entail given.

Finally, a discussion on the porting of CMI is given. The current implementation of CMI has already been tried with various compilers and on various operating systems. The changes needed to port CMI to a new system are outlined, and the task of integrating CMI into Rigi is discussed.

4.2 Implementation philosophy

CMI is intended to be part of the Rigi software development environment. As such, it will be part of a larger and more complex system for programming-in-the-large. However, the implementation was carried out while the

author was working off-campus; this meant that direct contact with the Rigi project was impossible, and provisions had to be made for CMI to be usable as a stand-alone tool, but such that a future integration could be accomplished without too much difficulty. One of the most important design factors of the current implementation was to take into account the fact that the global interface analysis algorithms assume a software development environment that provides efficient access to the compilation dependencies and the module interfaces of the various components in the software system [Mull 86, HoKM 87]. Without working directly in the Rigi system, this meant that a separate environment had to be created first in which all testing of CMI could be carried out. More importantly, it meant that considerable work had to be done to extract the compilation dependencies from the source text of the compilation units in the system. In part because of time and resource constraints, and in part because of the philosophy of the tool, the existing Ada and Modula-2 compilers were not altered.[30] Instead, CMI is implemented as a preprocessor. It parses the source text and extracts the module interconnection information necessary for the global interface analysis algorithms and stores the information in the CMI database. This is similar to the *integrated catalog database* technique proposed by Marti in [Mart 85]. At the same time, a new module interconnection language was developed (which is discussed in detail below). This technique allowed development to continue without requiring complex compiler front-ends to be written, or for them to be available.

As a stand-alone tool, CMI can be used in three different ways: by command-line invocation to compile an input source file, in interactive (line-oriented) command mode, or in menu mode. The use of CMI in all of these scenarios is presented in Chapter 5.

---

[30] The source for available existing compilers is normally protected by legal copyrights, and thus cannot be changed.

4.3 Attributed graph model

As stated above, the global interface analysis algorithms were written under the assumption that they would operate in an environment that provides efficient access to the (compilation) dependencies among modules in the system. The CMI implementation has chosen a general schema of having a software system modelled as a collection of objects that interact with each other in various ways; the compilation dependencies arise from *provide-require* relationships among certain classes of objects in the system. The resultant system and its inter-object relationships are modelled as an attributed graph.

This attributed graph is the internal data base that CMI uses. Each node in the graph represents an object of some particular class (*e.g.*, syntactic interface specification as represented by the `oc_def` class, or an implementation module, an object with class `oc_imp`). Each object in the directed graph may have arcs both incoming and outgoing. These arcs represent relationships and dependencies between objects. Currently, CMI keeps track of the dependence classes shown in Figure 4.1.

The *implicit* dependence is meant to reflect a dependence that arises between two objects but is not explicitly stated. An example of an implicit dependence is the relation between an implementation module and its corresponding definition module; the implementation module implicitly imports the definition module, but does not do so explicitly. An *implementation* dependence is one imposed on the system by the support environment for implementation reasons. Examples are the restriction of one module definition per compilation unit that many Modula-2 compilers impose. A *hierarchical* dependence is created through the logical hierarchy imposed on objects of different classes. For example, a definition module has a hierarchical dependence on a compilation unit, which in turn has a hierarchical dependence on a system. A

```
typedef enum {                          /* Dependence Class          */
  dc_unknown = 0x0,
  dc_implicit,                          /* implicit                  */
  dc_implementation,                    /* implementation            */
  dc_hierarchical,                      /* hierarchical              */
  dc_structure,                         /* provides                  */
  dc_compilation,                       /* requires                  */
  dc_semantic,                          /* semantic dependence       */
  dc_transitive,                        /* transitive closure        */
  dc_revision,                          /* revision                  */
  dc_MAX
} DependenceClass;
```

Figure 4.1: Dependence classes

structure dependence is one of provision of resources. If an object provides resources to another object, then a structure dependence is created between producer and consumer. A *compilation* dependence is the mirror image of a provision dependence. If an object requires resources from another object, then a compilation dependence is set up between the consumer and the producer. Figure 4.3 gives an example of a structure and compilation dependence. A *semantic* dependence is one imposed for logical reasons. It is currently unused, but has been included to allow CMI to be extended to the semantic interconnection model at a later date. The source text could be augmented with special directives to allow CMI to extract these semantic dependencies. An example of how to do so is given in Figure 4.2. A *transitive* dependence is a synthetic relation between two resources that is created when the **based** keyword is applied to a resource (see discussion of the CMI module interconnection language below). The *revision* dependency is currently unused but has been added to allow possible future integration with Rigi's version control mechanisms.

The objects in the system are arranged in a hierarchy. The valid object classes are shown in Figure 4.4, and the object hierarchy is shown in Figure 4.5. The master objects in the system are those of class `oc_sys` (system objects). The data structures have been implemented so that systems may be nested (*i.e.*, subsystems within systems are possible), but the current implementation only works with a single master system at a time. The only objects that the system currently contains are compilation units (class `oc_cu`). This is more for pragmatic purposes than anything else; there is no semantic reason that a system should contain only compilation units, but a compilation unit is the easiest object to implement with most file systems. The alternative is using an external database for object support (but this is not desired at the moment).

A compilation unit is made up of zero or more objects. Again, though any object class is possible, semantically only objects that are releases (class `oc_rel`), main programs (class `oc_pro`), modules (class `oc_mod`), definition modules (class `oc_def`), implementation modules (class `oc_imp`), generics (class `oc_gen`), alternatives (class `oc_alt`), revisions (class `oc_rev`), or accessories (class `oc_acc`) are allowed (*i.e.*, a compilation unit cannot logically contain another compilation unit).

```
(*!SEMANTIC  ....*)      ⇒   Modula-2

--!SEMANTIC  ....        ⇒   Ada and CMI
```

Adding a "semantic" or "info" field to declarations in the grammar would enable CMI to extract this information. It could be hidden in comments (to allow normal processing by traditional compilers).

Figure 4.2: Augmenting programming languages with semantic rules

---

The following Modula-2 code fragment

```
DEFINITION MODULE v;
  FROM w IMPORT x,y;
END v.
```

imposes an explicit compilation dependency on *w* by *v*, and an implicit structure dependence on *v* by *w*, as shown in the diagram below.

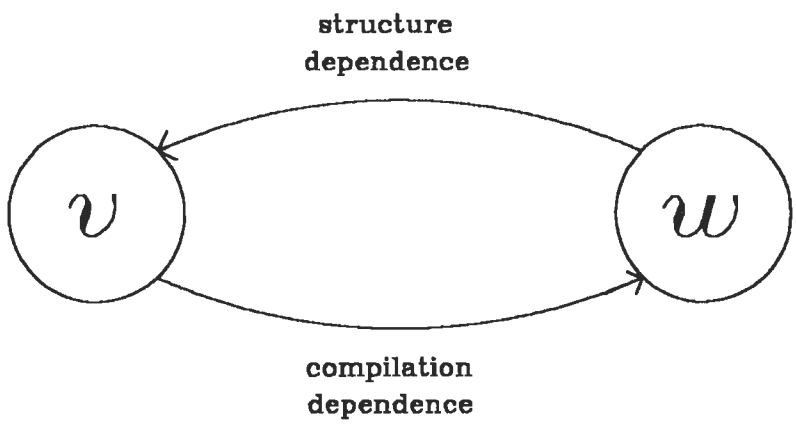


Figure 4.3: Structure and compilation dependencies

4.4 The CMI module interconnection language

4.4.1 Introduction

The attributed graph model discussed is used to internally store the relationships among objects in a system. These relationships must first be created before they can be modelled and stored. Since CMI must provide the environment that the global interface analysis algorithms assume (one in which the inter-module compilation dependencies are readily available), the dependence relationships among entities in a target programming language must first be extracted. There are two ways to do this: either force the programmer to explicitly list the dependencies among the objects in the system (such as *make* requires the programmer to provide a

```
typedef enum {                          /* Object Class              */
  oc_unknown = 0x0,
  oc_sys,                               /* system or subsystem       */
  oc_cu,                                /* compilation unit          */
  oc_rel,                               /* subsystem release/variant*/
  oc_pro,                               /* main program              */
  oc_mod,                               /* module                    */
  oc_def,                               /* definition part of mod    */
  oc_imp,                               /*implementation part of mod*/
  oc_gen,                               /* generic definition mod    */
  oc_alt,                               /*alternative implementation*/
  oc_rev,                               /* revision                  */
  oc_acc,                               /* accessory (misc.)         */
  oc_MAX
} ObjectClass;
```

Figure 4.4: Object classes

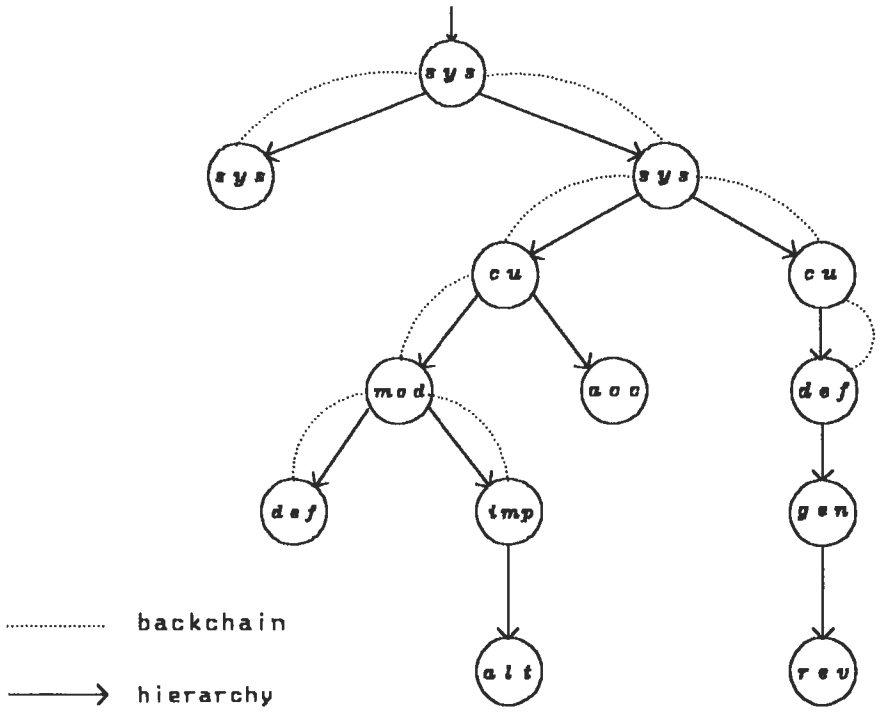


Figure 4.5: Object class hierarchy

makefile), or generate these relationships automatically. The relationships must go beyond the unit interconnection model to be truly useful. For example, in Ada it is not enough to know that there is a "with B;" clause in package A; which resources provided by B are used in A, and how they are used, must also be extracted. For languages such as Ada, this task is not trivial. It requires running the front end of the compiler against the source text. Time and resource restraints precluded writing a parser for each desired programming language that CMI was to run on. At the same time it was not desired to force the programmer to write down the object

relationships (inter-module dependencies) explicitly. Being forced to recode the module dependencies in another language, one that is not the programmer's implementation language, would be unacceptable. Thus, a compromise solution was chosen. A new language was invented, one that is small and reflects the essence of the various relationships among objects in the system being investigated; this language is called CMI.

The CMI module interconnection language can be thought of as a program design language (PDL) at the module, rather than the procedural, level. Only the interactions among various interfaces in the system are discussed; the implementation modules can also be sketched, but since they cannot provide resources, only require them, they play a less important part in the global interface analysis algorithms than the interface parts. It was initially thought that CMI would simply be an intermediate form, one that contained the essence of the interface description of a Modula-2 module or an Ada package. However, the language soon evolved into a serviceable mechanism to describe all module interface relationships that the global interface analysis algorithms needed. Moreover, since it provides a superset of the modular definitions available in Modula-2 or Ada (with one restriction, as shown below), it is possible to map the different interface mechanisms provided by various programming languages into a single language; this enabled the CMI implementation to focus on the relationships between objects, and less on how they were extracted. If other strongly typed, separately compiled programming languages are to be added to CMI at a later date, they may be incrementally enabled by updating the mapping, or cross-compilation mechanism, between the new programming language's interface facilities and the CMI module interconnection language.

For any language to support a superset of several programming language's interface mechanisms, and yet remain simple and concise, is somewhat of a challenge. It must retain the information necessary to fully describe relationships

between modules, yet be simple enough to manually write and automatically generate. The current language definition is at least a step in that direction. Functionally similar to Intercol, it provides a superset of the modular definitions available in standard Modula-2 or Ada, with one main exception: it does not allow the nesting of interfaces.[31] The reason for disallowing nesting of interface descriptions is both philosophical and pragmatic. From a philosophical (language design) point of view, the author believes that allowing the nesting of interfaces needlessly clutters both the language design and the subsequent code written with that language. A perfect example is the extremely complex nesting and visibility mechanism provided by Ada. Interface descriptions can be nested to an arbitrary degree, and in any order. This nesting allows a procedure to have internal packages that can have internal packages, and so on. When the separate compilation facility that Ada provides is added, the complications only multiply. Modula-2 does not allow the nesting of interface modules, but it does allow the nesting of local modules inside of an implementation module. Again, somewhat strange visibility rules arise from a local module exporting resources from a lexically nested level to an outer scope. Since it is possible to describe any system and its requirements using only single-level interfaces that communicate totally through an explicit *provide-require* mechanism, there is no need to burden the language further with nesting. From a pragmatic standpoint, allowing nested interfaces is more complex than single-level interfaces. Scoping and name visibility rules must be extended to allow for lexical levels of interfaces, and this is further complicated if enclosed modules are allowed to export resources to an enclosing level (sort of a reverse inheritance). If one also allows a nested definition to be separately compiled, as Ada does with its **separate** mechanism, the name state space must be saved externally so that when the separate interface is recompiled, all

---

[31] The object structure used internally by CMI does not preclude this nesting structure, and may be expanded (if desired) at a later time.

names that would have been available, had the interface been in-line, are available. For CMI's purposes, this goal was a little too ambitious. Thus, it was decided to disallow nesting in the current implementation. Note, however, that there is no semantic restriction in the internal design that would disallow adding this extension at a later date. It would simply entail allowing a definition module (an object of class `oc_def`) to contain a set of `oc_def` objects (recursively).

4.4.2 The CMI language definition

A CMI compilation unit is made up of zero or more object definitions. While any of the object classes given above may be entered, the current implementation is only concerned with those of the `oc_cu`, `oc_pro`, `oc_def`, `oc_imp`, and `oc_mod` classes (and the `oc_def` is the most important of all). The syntax of the CMI language is given in Figure 4.6 in a format similar to Extended Backus-Naur Formalism (EBNF); the syntax diagram of the CMI language is given in Appendix A.

Several features of the language deserve attention. First, the keyword **changed** was added to allow incremental testing of the implementation of the change propagation algorithms, but it has been left in because of its usefulness. By tagging a resource as **changed**, the recompilation algorithms are triggered; there is no need to actually alter the definition of the resource. One can easily predict the effect of a change on a provided resource, without changing its definition. By simply adding the **changed** keyword, CMI computes the set of affected objects in the system. As an aid in a maintenance task, it may be used to easily track down all dependencies on a particular resource.

```
cmi_cu          = {object}.
object          = object_id [require] [declare] [provide] end [object_name] ".".
require         = require {object_id ["←" resources] ";"}.
declare         = declare {resources}.
provide         = provide {[object_ids "→"] resources [qualified] ";"}.
resources       = resource {resource}.
resource        = [changed] resource_id [based "(" resources ")"].
object_ids      = object_id {object_id}.
object_id       = object_name ":" object_class.
resource_id     = resource_name ":" resource_class.
object_class    = SYSTEM | RELEASE | PROGRAM | MODULE | DEFINITION |
                  IMPLEMENTATION | GENERIC | ALTERNATIVE | REVISION |
                  ACCESSORY.
resource_class  = TASK | PROCEDURE | CONST | TYPE | VAR | IDENTIFIER.
```

Conventions for syntax: Square brackets [ ] mean that the enclosed form is optional. Curly braces { } mean that the enclosed form is repeated zero or more times. A vertical bar | separates a list of alternative choices. Uppercase names represent terminals; non-terminals are written in lower case. Language keywords are in bold and must be entered in lower case.

Figure 4.6: Syntax of CMI language

The **based** keyword is used to indicate a transitive dependency of one resource on another. The dependency can be nested in the declaration, so that type t_i can be based on t_j which can in turn be based on t_k. The predominant use of this keyword is the modelling of composite types in languages such as Ada and Modula-2. For example, a pointer type in Modula-2 may point to a type imported from another module. This implies the pointer type is based on the imported type. There is a restriction on this (somewhat simplistic) method of transitive dependency: there is no indication of implementation or compiler specific details such as successive storage locations. This type of information is only available from analysis of the symbol table of a compiler front-end for the programming language in question. CMI uses the **based** keyword as a sort-of "catch all" to indicate transitive dependencies.

The requisition and provision mechanism used in CMI is precise (as stated in Chapter 2). An interface may explicitly provide a set of resources to a particular set of clients, and it may also require a subset of the resources being provided to it by another producer. This is similar to the PIC/Ada language [WoCW 85b] in that, although a resource may be made *globally* available, it may also be explicitly provided to only a subset of modules. The same facility is available in Anna through the **provide to** clause [Lulle 85]. This provides an additional level of control over resource provision and requests, allowing more precise descriptions of the intended use of a resource [Habe 79]. For example, one may request all available resources from another object:

require d1:definition;

or one may request a particular set of resources:

require d1:definition ← v1:var t1:type;

In addition, an object can provide a resource to all other objects in the system:

provide c1:const t2:type;

or it may provide resources to a selected subset of objects:

provide d2:definition d3:implementation → c2:const p1:procedure;

An example of resource requisition and provision in CMI is depicted in Figure 4.7.

The **declare** keyword is currently unused. It was added with the idea that perhaps in the future there would be a need to locally declare resources in a module,

```
d0:defintion
require
  d1:definition ← t1:type t2:type v1:var;
  d2:definition;
provide
  t3:type based (t2:type);
  d4:definition d5:implementation → d2.v2:var;
end d0.
```

In this CMI code fragment, the definition module d0 is requesting the resources t1, t2, and v1 from definition module d1, and all available resources from definition module d2. It is then providing the type t3, which is based on the imported t2 type, to all objects in the system, and the variable v2 from definition module d2 to definition module d4 and implementation module d5 only.

Figure 4.7: Sample resource requisition and provision in CMI

particularly a definition module, which would contribute in some way to resource(s) provided by that module. At the moment, local resources can be declared, but they play no part in the implementation.

This small grammar allows the CMI database to store the minimal information required for interface analysis. There are no provisions made for version control in the CMI system. However, the semantics of the Rigi model include version control facilities [Mull 86, MuKl 87].

4.5 Compiling programming languages to CMI

Throughout most of the development and testing of CMI, the CMI language was used as the primary mechanism to specify objects and their relationships. A

Modula-2 parser is also provided to allow input to come from existing Modula-2 programs; CMI then extracts the module interconnection information. This extraction of information is basically the compilation of one language to another; in this case, it is the compilation of various strongly typed, separately compiled programming languages to the CMI module interconnection language.

The essential information that must be extracted from the module interface is its set of requirements and its set of provisions of resources. The dependencies among resources must also be extracted. For example, the fact that a new type is an array of another type implies that the array type is dependent on its base type; since the base type may be imported from another object, this forms a transitive dependency between the two resources. In this fashion, the Modula-2 parser detects transitive dependencies for new types that are sets of some other type, pointers to another type, enumerated and/or subrange types, record types, and procedure types. A record type (or variable) is transitively dependent on each of the types of its subfields. Similarly, a procedure type is dependent on the type of each of its parameters (if any), and the return type (if any). Even though elementary data types[32] are provided by the language definition, and not by some external module, these dependencies are also tracked. An example of the transitive dependencies between resources extracted from a Modula-2 program is given in Figure 4.8.

4.5.1 Language extensions

Since the CMI language provides a superset of the interface mechanisms provided by the programming languages Ada and Modula-2, both of these languages

---

[32] In Modula-2, these are INTEGER, CARDINAL, REAL, BOOLEAN, CHAR, and BITSET.

Compilation of the following Modula-2 code fragment:

```
TYPE
  type1 = ARRAY [1 .. 10] OF module1.type1;
  type2 = POINTER TO CARDINAL;
PROCEDURE proc1 (p1, p2:type1; p4: ARRAY OF BOOLEAN) : type2;
```

Produces the following CMI code:

```
type1:type based ( module1.type1:type )
type2:type based ( CARDINAL:type );
proc1:procedure based ( type1:type BOOLEAN:type type2:type );
```

Figure 4.8: Extracting transitive relationships in Modula-2

could be extended to take advantage of the features that the CMI language provides, such as preciseness. To incorporate preciseness in Modula-2, the Modula-2 grammar can be extended to allow an **EXPORT** clause of the form

EXPORT [**QUALIFIED**] **TO** modules objects;

where *modules* and *objects* are lists of modules and objects, respectively. This refinement allows stricter control of object provision. If it is not present, then the usual rules of Modula-2 apply. It allows Modula-2 programmers to take advantage of the preciseness inherent in the CMI language. Since the most recent (third) edition of Modula-2 does not require the use of the **EXPORT** clause to export objects from a definition module (all resources in the module are implicitly exported), this extension would require the programmer to relist the identifiers that have been selected to be

provided to a specific set of modules (*i.e.*, are precise provisions).[33] This extension is currently provided by the existing Modula-2 interface.

An Ada interface could extend the grammar to allow a more refined importing of resources (instead of the all-or-none method currently used by Ada) with a construct such as

with package.(objects);

The default Ada rules could be used if the standard **with** clause were used. The PIC/Ada language features an extension that allows resource provision to be made precise by augmenting the Ada grammar with a `provide to` construct, similar to that outlined above for Modula-2.

4.6 Minimal recompilation

The steps taken to ensure minimal recompilation when an interface change occurs are outlined below. The implementation follows the global interface analysis algorithms that were detailed in Chapter 3.

During system verification, if a resource of class `rc_identifier` is found, it will be matched with the correct resource from the providing object (if possible).

---

[33] Of course, each identifier in this set must appear in the proper body of the definition module.

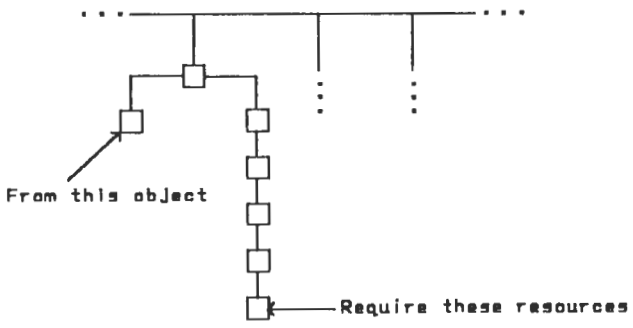
4.6.1 The *require* and *provide* data structures

Two of the most important relations between objects in the global interface analysis algorithms are imposed by the requirements that one object places on another, and the corresponding provision of resources that an object may support. Both are represented as sets; this is displayed graphically in Figure 4.9, with the corresponding C declarations given in Figure 4.11.

A definition module may both require and provide resources, while an implementation module may only require resources. The resources are identified by a name and a class indicator. Since there is no overloading of identifiers in Modula-2, resources are exported by name only. The consumer module imports the resource by just stating the name of the resource, without a class. CMI verifies that there is only one resource with that name being provided to the consumer module, and by doing so also makes possible the imported resource to be assigned a class. Ada requires type information in addition to the name to distinguish one resource from another. The resource classes supported by CMI are given in Figure 4.10. Resources are the single most important entity in the CMI system. Objects have dependencies on one another as the resources flow through the system graph, leaving a producer object and entering a consumer object.

The require data structure is used to represent the compilation dependencies among objects. Each object may have a set of requirements; each element of this set has the form of the object shown in Figure 4.11. Basically, each require set element represents a specified set of resources that are required from a certain producer object. If the set of resources required is null, the entire producer object is required; this corresponds to the Ada **with** clause, where all of the visible resources are requested. If the set is not null, a specific set of resources is requested.

Set of *require* structures:



Set of *provide* structures:

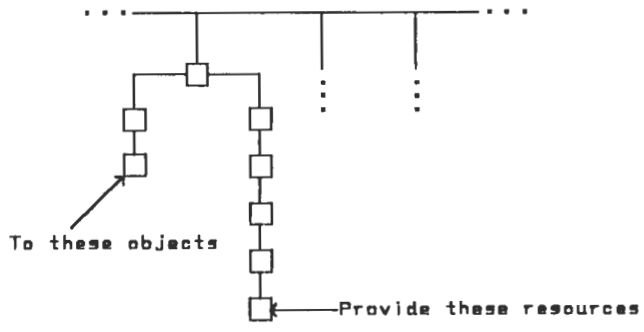


Figure 4.9: require and provide data structures

The provide data structure is used to represent the structure dependencies among objects. Since most programming languages are not precise (*i.e.*, they do not allow a producer object to specify its clients explicitly), there are two kinds of structure

```
typedef enum {                          /* Resource Class          */
  rc_unknown = 0x0,
  rc_task,
  rc_procedure,
  rc_const,
  rc_type,
  rc_var,
  rc_identifier,
  rc_MAX
} ResourceClass;
```

Figure 4.10: Resource classes

dependencies that CMI builds: explicit and implicit. The explicit dependencies are extracted from languages such as CMI that allow one to specify the limited group of possible consumer objects. The implicit dependencies are created by mirroring the require dependencies created by a requesting object. Some classes of objects may provide resources; those that can have a non-null set of provide data structures. Each element of this set has the form shown in Figure 4.11. This structure allows for each element of the set to specify a set of objects that is allowed to request the set of provided resources. If the set of possible consumer objects is null, unrestricted access is available; this interface mechanism appears in both Ada and Modula-2.

4.6.2 Graph transformation using the SACDG algorithm

The graph that is created to represent the system must undergo the transformations shown in Figure 3.2 before the minimal recompilation algorithms can be applied. The compilation dependencies for each object are represented by an attribute attached to the object, implemented as a set of pointers to these objects in the

```
/*****************************************************************/
/* For requirements and provisions, from each producer a 'set' of */
/* resources may be required.  Thus, in the def and imp data      */
/* structures, there is a list of modules.  For each entry in this */
/* list, a list of resources is either provided or required,      */
/* depending on the type of module.                               */
/*****************************************************************/
typedef struct require {                                  /* require */
  struct object *from;
  set resources;
} Require;

typedef struct provide {                                  /* provide */
  set consumers;
  set resources;
} Provide;
```

Figure 4.11: C declarations for *require* and *provide*

graph. This compilation graph is first made acyclic by computing the strongly connected components. From then on, each strongly connected component is treated as a single logical component. The algorithm used to compute the strongly connected components is a variant of Mehlhorn's [Mehl 84].

This acyclic graph is now topologically sorted; the topological numbers assigned to each object in the graph induce a partial order on the compilation order of the objects in the graph. Thus, each object has a sorted list of pointers to other objects in the system that would require recompilation if this object is changed. If a dependent object is part of a strongly connected component that contains more than one member, only the direct client in the strongly connected component is held on the list; the remaining members of the strongly connected component are implicitly

dependent on the original object by the fact that they are in a client's strongly connected component. This set of dependent objects is called the *make chain*, since it represents the complete set of objects that would require recompilation using *make*'s unit interconnection model.

This set of transformations from a compilation dependency graph to the sorted, acyclic compilation dependency graph can be applied once for each object in the system, or on demand for a particular object. When CMI first constructs a new system model, these transformations are applied to each object in the system. One can quickly determine the set of dependent objects (strongly connected components) for any object in the system since they have already been computed. When a new version of an object is submitted to CMI for analysis, the transformations are applied only to this new object; this is the root of the new compilation dependency graph. At the same time, system consistency is verified. If the new version of the object makes the system inconsistent (*e.g.*, it requires resources from a non-existent object), then the recompilation algorithms cannot be performed; (as currently implemented) they require a completely consistent system to function correctly.

4.6.3 Computing the change sets

The global interface analysis algorithms are invoked when an interface changes in some way. This may be through an editing change on the interface, or by having tagged one of the resources in the interface as **changed**. If an editing change has occurred, then CMI first proceeds to classify the change as either *inconsequential*, *local*, or *global*. The internal representation of the old and new versions of the interface are compared, producing the object sets *Inside* and *Outside*; this action is similar to comparing the abstract syntax trees of two objects in some programming language.

The comparison between old and new versions of an interface is non-trivial due to the nature of the set implementation. A set difference or intersection routine is not sufficient to reflect the nature of the change. Consider for example a new version of an interface that now provides the same set of resources, but to only a particular set of possible client modules. The older version had an unlimited export of these resources. The same can be true for requisition of resources. Perhaps the old version had requested the whole interface, but now has explicitly stated the set of resources required. Obviously, if this latter method is used, the number of recompilations will almost always be decreased.

4.6.4 Propagating the changes

If the change was inconsequential, no further work needs to be performed. This is a surprisingly common situation during development; an interface may be edited just to add comments or "beautify" the code layout.

A local change currently produces a textual message saying that the Tichy-Baker algorithms should now be applied to the implementation corresponding to the changed interface, to determine if the change affects it and if its recompilation is necessary.

A global change affects the clients of the changed interface (and possibly its implementation). The *Inside* set has been filtered through the set of provided resources in the changed interface to produce the *Outside* set. To propagate this set of changes, a heap is initialized to contain the set of direct clients of the changed object (*i.e.*, those objects that are on the adjacency list of the object). Note that this set is a subset of the *make chain*; it may be thought of as the "first layer of the onion" of change layers. to demonstrate the effectiveness of the global interface

analysis algorithms, CMI can optionally produce the set of objects that would require recompilation if *make*'s rules were used; this is simply a linear traversal of the make chain. The set of affected visible resources is then intersected with the requirement set of each of these objects in the heap. Note that if any of the objects found on the heap is a member of a strongly connected component that contains more than a single object, CMI stops minimal recompilation. For each client, the set of incoming affected resources is filtered through its require set. If no member of this changed set of resources is on the require list of the client object, then this object is removed from the heap; it is unaffected by the change and thus does not require recompilation. If the resultant set from the intersection is not null, then again the Tichy-Baker algorithm would be invoked on its implementation (and any of its variants); in addition, this (hopefully smaller) set of affected resources is filtered through the objects provision list. This procedure continues until the heap is empty (*i.e.*, all the objects that are affected by the change have been processed).

4.7 Selected implementation details

Many of the often-used functions in CMI are synthetically "in-lined" using macros in place of function calls. An example is the memory allocation macro `getmem()` which only calls the function of the same name if the pre-allocated chunk of storage is exhausted. A sample profile of execution time of CMI has shown that this simple technique drastically cuts down on the number of accessory function calls, and thus execution speed increases significantly.

The CMI code is written almost entirely in C (see also the section on portability below). To reflect the attributed graph described above, an object-oriented approach was used in the implementation. Every object in the system is an *object*, as defined by the C code fragment given in Figure 4.12. Various operations on each object are

defined; these operations are pseudo-inherited from the object class's definition. It would have been much easier to use C++ [Stro 87] than C for this, but C++ was not readily available. Instead, each object class has a single, program-wide, representative variable in which the operations on that object are defined. It was decided to use a single variable per class rather than have each object carry around information relating to the operations on its class. One reason for this design decision is that the memory allocation routines are quite fast; it was not desired to "clutter up" the code space with *n* memory allocation routines, where *n* is the number of distinct object classes. Besides allocating memory, it would be necessary to repeatedly copy the same set of function pointers to every variable instance of each class. For example, an object of class `require` invokes operations on its type by `(*op_req.insert)(...)`, rather than `(*req_var→insert)(...)`. All the representative global variables start with the letters *op_*, to represent operations, and the next three represent the object class (*e.g.*, *req* for the require data structure).

As with most applications that involve the manipulation of graphs and graphical objects, set manipulation plays an important role. Two separate abstract set representation techniques were written, one based on binary trees, the other on doubly-linked lists. Currently the latter representation is used. However, macros were also written to traverse this set type, so the implementation of the set could be altered at any time. Typical set manipulation routines were written, such as insert, locate, merge, and all are part of the set class; objects access these functions through the function pointer mechanism shown above.

Since the set routines were written to contain objects of any class, two macros were written to self-check the parameters passed to some of the functions, as well as to enforce semantic meanings on objects that must be cast to a desired type (they are pointed to by `void *` pointers). The first macro is called **`demand()`**, and acts like

```
/*****************************************************************/
/* An 'object' is the basic unit in CMI.  It may be any of the   */
/* supported object types (as distinguished by ObjectClass).  Each */
/* object also has a set of dependencies on other objects.       */
/*****************************************************************/

typedef struct object {                  /* Object                */
  char *name;
  ObjectClass oc;
  union {
    Sys *system;
    Cu  *compilation_unit;
    Pro *program;
    Rel *release;
    Mod *module;
    Def *definition;
    Imp *implementation;
    Gen *generic;
    Alt *alternative;
    Rev *revision;
    Acc *accessory;
  } class;
  set dependencies;                      /* of type 'Dependency'  */
  struct cdg *cdg;                       /* compilation dep. graph */
  unsigned changed : 1;                  /* flags                 */
  #if C2
    unsigned       : 15;
  #else
    unsigned       : 31;
  #endif
} Object;
```

Figure 4.12: The basic unit in CMI

the `assert` macro in the standard C library. If the condition that is required to be true is not, then CMI aborts with a fatal run-time error. The `semantic()` macro is used to ensure that objects passed to certain routines satisfy some logical condition, such as an object being of a certain class. For example, some accessory functions expect to work on objects of class `oc_def` only, but are passed a pointer to some unknown object. If the semantic condition fails, a runtime warning is produced but processing continues.

4.7.1 Restrictions of the current implementation

The current implementation has several shortcomings. Perhaps if CMI had been developed from within Rigi, less time would have been spent on the creation of the proper environment for the global interface analysis algorithms, and more on some of the finer details, such as change analysis. However, this was not so.

4.7.1.1 Recursive inter-module dependencies

The current CMI implementation only partially supports recursive inter-module dependencies. Even though Clarke contends that recursive compilation dependencies occur quite frequently in large software systems [Clar 85], few programming languages fully permit these to occur. The support is usually limited to opaque types (pointers). Neither Ada nor Modula-2 allow recursive inter-module dependencies in their language definition. For Ada, this type of cross-compilation unit dependency violates the elaboration rules as specified in the LRM [Dod 83], § 10.5: "The program is illegal if no consistent order can be found (that is, if a circularity exists). The elaboration of the compilation units of the program is performed in some order that is otherwise not defined by the language." This type of recursive cross-module

dependency is also not allowed in Modula-2, as specified in [Wirt 85], p. 169: "If circular references occur among modules, their order of initialization is not defined." Thus, this implementation restriction is not severe considering that neither Ada nor Modula-2 allow this situation to occur in a valid program, but it does go against the spirit of one of the main contributions of the global interface analysis algorithms.

4.7.1.2 Transitive dependencies and the change type algorithm

The computation of the transitive closure under declaration dependence of resources is currently based on the relationships formed by the **based** keyword. The Modula-2 interface provided has a rather severe restriction on what relationships are extracted from the source text as "based" relationships. The reason for this is that to extract the exact changes from one version of a module to another, detailed information on each resource must be maintained. This information is only available by performing a complete analysis of the symbol table usually created by a compiler's front end. The interface provided is by no means a complete front end system for Modula-2; it is a parser that extracts "obvious" and specific dependencies among resources. However, it does not perform full symbol table analysis.

The resource data structure currently used is tuned more to the CMI module interconnection language than to any programming language. In particular, the information stored about a resource is somewhat limited; only the name and the class of the resource is stored. No "subclass" information is kept. For example, the value of a const is not stored anywhere. Thus, if the same const variable is provided in two versions of a module, but its value changes, this change will go undetected. Similarly for types: there is no information kept on the subclass of a type (*e.g.*, if it was a set type, or a pointer type). This means that if in one version of a module, a type is

declared as a set of X, but in the next it is declared as a pointer to X, this also goes undetected. Furthermore, specific implementation details such as storage locations are not captured. Neither are changes in the size of a resource, say a record type.

The philosophy of providing a Modula-2 interface at all is to show that the algorithms are a viable mechanism to reduce compilations, but supplying a full Modula-2 front end is overly ambitious. Thus, detailed change analysis is somewhat less than desired. In particular, sets `C.modified` and `C.modprv` (as shown in Figure 3.1) are not as complete as they could be. The focus is on detecting resources that have been added or deleted from an object, or that are based on a type that has changed or been removed. The resource change will certainly be detected if it or any of the resources on which it is based is tagged as **changed**.

This simplistic view of resource representation is clearly the most severe restriction in the current implementation. This shortcoming would certainly have to be rectified if the Modula-2 interface were to be used in a production environment. As it stands, the current implementation will sometimes miss recompiling a module, since the Modula-2 interface has not provided enough detailed change information. This missed recompilation may cause serious problems by not maintaining a consistent system. Note however that it does not affect the CMI module interconnection language, just the Modula-2 interface. The solution is to augment the resource class data structure with finer details on the representation of resources. The const resource would have to carry around its value, and a type would have to have much more detailed information concerning its external representation and subclass (pointer, set, etc). This problem quickly grows into complete symbol table analysis, since it must now be possible to handle variant records, the size of a record type and any on its subfields, and so on. This is too ambitious for an initial prototype implementation such as CMI. Moreover, this information changes with each

programming language. For example, this problem would be compounded in an Ada interface because of the overloading of identifiers that is permitted, and the richer set of type subclasses available.

The Modula-2 interface included provides a way of alleviating some of the problems caused by this shortcoming. The Modula-2 grammar that is accepted by the parser has been augmented with a construct that explicitly informs CMI of a change in a resource. This is similar to the **changed** keyword of the CMI module interconnection language. The information is hidden inside a special comment to allow the same Modula-2 source to be processed by true Modula-2 compilers. This construct may be used to tag a resource as changed, even if it has not been (to perform consequence analysis as outlined above). The use of the construct is shown in Figure 4.13.

There is a second way of "helping" CMI out if it is needed and/or desired. When the Modula-2 module has been entered into the system, the equivalent CMI code may be produced by unparsing the internal data base of CMI (as shown in Chapter 5). This resultant code may then be edited by hand to add any missing details, such as whether the resource is based on another resource or not. It may be useful to use the **changed** keyword here as well. After editing, the CMI code is then recompiled into the system.

4.8 Porting CMI

CMI was written with portability in mind. The development environment was an IBM S/370 Model 3090 running VM/CMS (SP5), with some early work done on an IBM RT PC running AIX. The target architecture for integration with Rigi is a network of Sun Workstations. Other parts of the Rigi system that have already been

```
CONST
  (*!changed*) name = value;
TYPE
  (*!changed*) name = type;
VAR
  (*!changed*) name:type;
PROCEDURE (*!changed*) name (...);
```

The added construct is hidden inside a comment. It must be entered exactly as shown: "(*!changed*)". Note that there is no need to apply this construct to opaque types, since they only provide a name, with no representation information. CMI automatically detects whether the opaque type has been added or deleted from the require or provide set.

Figure 4.13: Augmented Modula-2 grammar

completed, such as the Rigi editor [Klas 88], are written in C. With this in mind, the code was written in such as way that it should be straight forward to integrate CMI into Rigi, and as a stand-alone tool it should be easy to adapt CMI to various operating systems and compilers.

Almost all of the code for CMI is written in C. There is one module written in PL/I for the VM version. In addition to the C code, there are the input files to the lexical analyzer and the parser generator; though powerful parser generators were available on VM, they are IBM internal tools and thus not readily available on other hosts. It was for this reason that the standard Unix compiler development tools for language scanning and parsing, *lex* [LeSc 86] and *yacc* [John 86], respectively, were used. There are also a number of support files that are used when rebuilding CMI. These depend to a great extent on the environment in which CMI is being rebuilt. For both VM/CMS and MVS/TSO there is a set of Rexx [IBM 86a] EXECs and Xedit macros; for DOS there are a few .BAT files and sed scripts; for Unix there are

corresponding shell and sed scripts. On VM one of the uses of the Rexx EXECs is to extract file time-stamps from the operating system; this can easily be done in C for the DOS and Unix versions.

Independent of the operating system is the need to "fix" the output of lex and yacc; both tools produce C source file(s) from their respective input files. The problem is that neither program was written with the idea of having more than one scanner and/or parser active in the same program at the same time. Both use a two-letter prefix on all the variables introduced by the generated C source, which start with *yy*. The various Xedit macros or sed scripts are used to change the exposed variables in each file to be distinct and related to a particular scanner/parser pair.[34] The second 'y' in the reserved name is changed to reflect the use: 'c' for CMI, 'd' for the interactive command parser, 'm' for the Modula-2 interface, and 'a' for the Ada interface. In addition, a few multiple definitions of variables, such as `yylineno`, must be replaced by a single occurrence, since some linkers (such as those on DOS) cannot link a program with a multiply-defined external variable. The variables `yytext`, `yyerror`, `yyin`, `yyout`, and `yylineno` are shared throughout all scanners and parsers. The `makefile` used correctly repairs the output of lex and yacc for each scanner and parser when CMI is rebuilt by calling the appropriate EXECs, shell scripts, or batch files, depending on the settings of the os variable (as discussed below). If CMI is ported to a system not currently supported, then the equivalent accessory programs must be written. For example, various `.COM` command files would be needed for VAX/VMS.

---

[34] It is assumed that the standard lex and yacc driver files have been used. These are usually called NCFORM and YACCPAR, respectively. If these driver templates have been altered from the default versions in some way, then the accessory files may require some "tweaking" to reflect these changes.

As stated above, VM/CMS was the primary development environment. The IBM C/370 compiler [IBM 1988a] was used, as well as the INSPECT testing and analysis tool [IBM 1988b].[35] To have CMI run on MVS/TSO, it may be compiled on VM or MVS. In either case the resultant output object code is then link-edited on MVS and the module is placed in a pre-allocated PDS (Partitioned Data Set).

On DOS, both Turbo C [Bord 87] and IBM C/2 [IBM 87a] have been used. When compiled with Turbo C, only one parser may be used at a time; this has been chosen as the CMI language parser, making the Modula-2 and interactive command language parsing systems unavailable. The reason for this is that Turbo C does not provide a mechanism for splitting up the global data segment into separate chunks, even when compiled under the huge memory model; with all three parsers active, there is more than 64K of global data. Any single segment is limited to 64K, and thus the link step fails after compilation. The C/2 compiler has special compile-time switches that alleviate this problem by splitting up any segment that grows beyond a selectable size threshold.

When porting CMI to a new operating system or compiler, a few things should be kept in mind. First of all, yacc and lex must be available; if they are not the scanner and parser (.l and .y) modules must be rewritten. If a new C compiler is used, it should conform as close as possible to the proposed ANSI standard [ANSI 88]; at the very least it must support function prototyping. If it does not support bitfields of 32 bits, such as the C/2 compiler which limits bitfields to 16 bits, then in a few places this compiler must be added to the C/2 class for compilation to

---

[35] CMI has also been successfully compiled with the IBM C Program Offering (PO) [IBM 86b]. Using the new Program Product (PP) was an interesting experience; the C compiler was under development at the time, and so was INSPECT.

succeed.[36] A likely candidate to be a problem when moving to a new operating system is the file naming convention. For example, VM uses `name type mode`; MVS uses an assortment of methods, whether the dataset (file) is sequential, or a member of a PDS, as well as other factors; DOS uses `name.type` and Unix allows almost anything. Hand in hand with file naming conventions goes the extraction of file time-stamps from the operating system; this may be trivial or difficult, depending on the architecture used. Finally, as with most porting tasks, I/O is usually the most implementation dependent. CMI uses only standard C library routines for all of its I/O. The portion of the code that would benefit most from a change is the menu code; it is currently written for a generic "dumb" terminal. If CMI is moved to a system that has facilities for windowing, then this code should be changed. An abstract set of routines currently exists to do rudimentary tasks such as move the cursor down or to the right, clear the screen, and so forth. Only one module would have to change to use any new I/O facilities.

To rebuild CMI either for a different operating system, or a different compiler, there are two variables that must be changed in the `makefile`: `os` and `compiler`. By default they are set up as `os = VM` and `compiler = C370`. These values may be over-ridden when rebuilding CMI by supplying new values to the `make` program as exemplified in Figure 4.14.

Compilers that are currently supported are IBM C for System/370 (PO), IBM C/370 (PP), IBM C/2, Turbo C, Waterloo C (on S/370), and Unix C. Target operating systems currently supported are DOS, VM/CMS, MVS/TSO, and Unix. CMI

---

[36] These places currently are `thglobal.h`, `thlist.h`, and `thtree.h`. The restriction of bitfields to 16 bits in length is not currently a problem, since the individual bits in the bitfields are only used for flags, and very few of them are used yet.

- make (doall os=MVS compiler=C370 f thmake.makefile.a
- make -fthmake.mak -Dos=DOS -Dcompiler=TURBOC

The first example uses the syntax of the *make* program that runs on VM/CMS. It rebuilds CMI from scratch using the C/370 compiler for hosting on MVS. The second example uses the syntax of *make* provided by Borland's Turbo C. This second example rebuilds CMI for DOS using Turbo C. Note that in this case `thglobal.h` must be "touched" to trigger a full system rebuild.

Figure 4.14: Rebuilding CMI

has already been tested on an IBM PC/XT running DOS 3.2, an IBM PS/2 Model 80-386 running DOS 4.00, an IBM S/370 Model 3090 running VM/CMS (SP4 and SP5), an IBM S/370 Model 3084 running MVS/XA, briefly on a VAX-11/750 running 4.3BSD, and a Sun 3-280 running SunOS. Except for the limitations (such as program size) that environments such as DOS impose, CMI looks and runs exactly the same on all systems; the only difference is speed. Porting CMI to systems such as VAX/VMS should not be difficult; the only changes that may be needed are the interface to file naming conventions, extraction of file time-stamps, menu mode I/O (if required or desired), and a changing of some of the tools needed for rebuilding CMI (as outlined above).

A few small problems occurred during porting to the above mentioned compilers and operating systems that are of interest. "Trigraph" characters are not supported on Turbo C or Unix C, so they had to be changed to their normal character equivalents.[37] Not all compilers support the same set of signals, or even the same mech-

[37] Trigraphs are part of ANSI C to allow special C characters to be entered at terminals that do not have the keyboard required. For example, `'??/n'` can be used in place of `'\n'` (the new line character). See [ANSI 88] for a complete list of trigraphs and their character equivalents.

anism of raising and trapping them. CMI has complete control over all signal handling when running; if an exception occurs a special signal handler is called, and if compiled with C/370, INSPECT is invoked. Finally, the table size for the Modula-2 scanner must be altered from its default on DOS and Unix. This meant increasing %a to 4000 and %o to 6000 in the .l file for the scanner being built.

4.9 Adding support for other languages

CMI was written to be both portable and extendable. One of the design goals was to ease the integration of a new language into CMI. The most obvious choice for the next supported language is Ada (and some of the work has already been done), but any other strongly typed, separately compiled programming language would be a valid candidate. Some of them that would benefit the most include Oberon, Modula-2+, and Modula-3. To add support for a new language there are basically two choices: if a compiler for the language already exists and the code for it is available (*i.e.*, it may be altered), then the parser or front-end of the existing code may be augmented to either produce CMI as a target language, or it may be interfaced directly into the generic parsing routines that CMI provides. If the source code of the compiler is not available, then a new, separate parser must be created; this was the method used for the Modula-2 support that CMI currently provides.

If the source to CMI is not available either, then clearly the only available method would be to produce CMI code as output from the new parser. CMI could then read in this resultant file and process it. While slower than entering CMI directly, it is feasible. To interface directly with the generic parsing routines that CMI provides, the source to CMI must be available. The Modula-2 parser currently provided uses this latter method; it calls the same routines that the CMI parser does. It may happen that the language support to be added is too rich for the current level

of CMI's generic parsing routines to handle. In this case the code to CMI must be enhanced. This would be the case if full Ada support were to be added without the source of the Ada compiler.

In addition, if *lex* and *yacc* were used, then the tools used to post-process the output files would need to be altered to handle the new language.

4.10 Integrating CMI into Rigi

The integration of CMI into Rigi is the next logical step in CMI's evolution. It is in conjunction with Rigi that the benefits of the global interface analysis algorithms would be the most apparent. In addition, the use of a high-resolution bit-mapped display, such as that on a Sun Workstation (which is what Rigi currently uses), would be a definite improvement over the simplistic menu environment provided with the current implementation.

Since this integration has not yet taken place, but is the most likely candidate for a porting exercise, a few details on the job it will entail are given here. Since Rigi currently runs under SunOS, a variant of 4.3BSD and Unix System V.2, the lex and yacc tools are available, and the shell and sed scripts used should not require much change. The area most likely to require work is the interface to the invocation of the algorithms.

4.11 Summary

This chapter presented the details of the CMI implementation of the global interface analysis algorithms. The internal data base, which is an attributed graph

model of the software system, was presented. The hierarchical relationships among object classes and the flow of resources though the objects along the arcs of the graph was discussed.

The CMI module interconnection language was introduced. This language is used to efficiently describe the essential interface relationships between objects. It features a clear, precise, and explicit mechanism for describing an object's needs and provisions. The design of the attributed graph model both influenced and was influenced by the language.

The implementation of the minimal recompilation algorithms was described. They efficiently perform consequence analysis and compute the optimal recompilation sequence after a change to a basic interface. The implementation is written in portable, object-oriented C and may be re-hosted on a variety of operating systems using a variety of compilers and support tools.

CHAPTER 5

Experience with CMI

5.1 Introduction

This chapter illustrates some of the uses of CMI, and highlights the benefits that may be gained by using it as a stand-alone tool for performing consequence analysis and computing the minimal recompilation sequence after an interface change occurs. As a stand-alone tool, CMI may be used to compile (translate) an input source file to the equivalent language representation in any of the supported languages. This translation is for the interface part only. It may also be used in an interactive (line-oriented) command mode, or in menu-driven environment. Each of these uses is described below.

Since the primary development environment for CMI was VM/CMS, the examples shown are taken from that operating system. Appendix C provides sample screen images of CMI while running in menu-driven mode. Through various examples it is shown how the impact of a change in an interface can be minimized by using CMI. The set of recompilations that would occur had a *make*-like tool been used is shown to usually be much larger than that produced by CMI for the same change.

5.2 Using CMI as a stand-alone tool

As a stand-alone tool, CMI can be used in three different modes:

1. Command-line mode (compiler only)
2. Interactive (line-oriented) command mode
3. Menu-driven mode

The options passed to CMI at invocation-time determine the mode of operation. The syntax of the CMI command and a description of the command-line options is given in Appendix B. On-line help can be requested by typing '?' at a CMI prompt. There is also a set of help panels that may be used on VM/CMS by typing `help cmi` at the CMS prompt; the main help task panel for CMI is shown in Appendix C.

As a stand-alone tool, CMI operates as a preprocessor, parsing the source text of the interface and producing an intermediate representation which is recorded in the CMI data base. The input source can be in any of the supported languages. The setting of the current input language (which may be altered at any time by the user), determines the parsing interface used. By default, CMI is chosen as the input and output language. As each compilation unit is entered into CMI, the internal graph structure being created will usually be incomplete. This situation arises when an object requests resources from another object which is not yet in the system. For example, if the `-f` option is used to create a new system out of a set of compilation units, as each compilation unit is parsed it may have dependencies on other objects that have not yet been parsed, thus CMI does not (yet) know about them. Thus, several error and warning messages may be produced during the initial system build.

If a syntax error occurs while parsing the source text, very minimal error recovery is performed. Usually an error message indicating the type of error and the approximate line the error occurred on is produced. If no syntax error occurred, the successful compilation creates a file that is the current output language's representation of the input source. The name of the file is the same as the input file, but the

file type is either CMC, CMM, or CMA, depending whether the output language is CMI, Modula-2, or Ada, respectively. The default file type for the input file is .CMI.

When all relevant compilation units have been entered into the system, the initial state of the internal data base is said to be *unknown*. When in this state, no recompilation and little analysis can be performed. The user invokes the verification procedure that validates all requests and provisions of resources in the system. If any errors are found (*e.g.*, an object requests a resource that is not being provided to it), then the system remains in an *inconsistent* state. The system is inconsistent if one or more of its compilation units are inconsistent. Only when all errors have been resolved can the system be made *consistent* using the verification procedure. At this point consequence analysis and interface recompilation may take place.

Every CMI compilation or session produces a file called CMI.LOG (unless the -log option was specified) which contains a copy of all output produced by CMI. This may be used as a record of the session. Moreover, a file called CMI.ERROR (CMI.ERR on DOS) is also produced (unless the -e option was specified). It contains all the errors and warnings produced by CMI.

5.2.1 Compiler mode

When used as a compiler, CMI may be used to check the syntax of an input source file in any of the supported languages, or to produce a compiled (translated) version of its input. Both the input language and the output language are changeable by the user. The syntax of compiler-mode invocation and a complete description of the command-line options is given in Appendix C.

As an example of compiling from one language to another, Figure 5.1 shows a small Modula-2 input file which was submitted for compilation to CM1. The resultant output file is shown in Figure 5.2.

5.2.2 Interactive command-line mode

Interactive mode provides (nearly) the same services as the menu-oriented environment. However, the commands must be entered in full and follow the syntax of the interactive command language exactly. This syntax is given in Figure 5.3. Only a subset of the language is currently supported. The implementation is such that command-line mode support may be incrementally added at a later date. The user may switch from menu-mode to command-line mode and back from either environment.

5.2.3 Menu mode

Menu mode is the most useful environment in which to work. It provides easy access to all of the capabilities of CMI, and is the closest to a "real" environment that the stand-alone model of CMI gets.

When CMI is invoked in menu mode, an initial informational message and introduction appears for a few seconds. Then the primary option screen appears, which lists the main options available to the user. Selected sample screen images taken from various `CMI.LOG` files are shown in Appendix C.

```
DEFINITION MODULE d5;
IMPORT d2;
FROM d1 IMPORT v11, v2, t1;
  CONST
    c1 = 12;
  TYPE
    opaque1, opaque2;
  TYPE
    t5a = ARRAY [1 .. d2.max] OF t1;
    t5b = RECORD
            f1:REAL;
            f2:t5a;
            f2b:RECORD
              sf1:POINTER TO CHAR;
              sf2:PROCEDURE (REAL, d2.t1)
            END
          END;
    t5c = POINTER TO t1;
    t5d = PROCEDURE (t5c, VAR t5c) : t5c;
    t5e = RECORD
           f1:t5i;
           CASE caseS2:CARDINAL OF
                0: |
                17,18,30..50: r:REAL; ptr:opaque1 |
             1..5: field1, field2:t5a; field3:SET OF [1..5]
             ELSE catchAll:BOOLEAN
           END
         END;
  VAR
    bs:BITSET;
    tvar:t5b;
    PROCEDURE proc5a (p1, p2:t5b; VAR p3:BITSET; p4: ARRAY OF t5c);
    PROCEDURE proc5c() : opaque2;
END d5.
```

This file was compiled using the following command on VM/CMS:

```
cmi / -li modula-2 test.m2
```

The resultant CMI output file is shown im Figure 5.2.

Figure 5.1: Sample Modula-2 compilation

```
!Compilation unit test.m2 is part of the *linemode* system

d5 : definition
require
  d2:definition ;
  d1:definition ← v11:identifier v2:identifier t1:identifier ;
provide
  c1:const ;
  opaque1:type
  opaque2:type ;
  t5a:type based ( t1:type )
  t5b:type based ( REAL:type t5a:type CHAR:type d2.t1:type )
  t5c:type based ( t1:type )
  t5d:type based ( t5c:type )
  t5e:type based ( t5i:type REAL:type opaque1:type t5a:type
                   CARDINAL:type BOOLEAN:type ) ;
  bs:var based ( BITSET:type )
  tvar:var based ( t5b:type ) ;
  proc5a:procedure based ( t5b:type BITSET:type t5c:type ) ;
  proc5c:procedure based ( opaque2:type ) ;
end d5.
```

Figure 5.2: Translated input of Modula-2 source

5.2.3.1 Building a new system

If no compilation units are specified on the command line invocation of CMI, typically the first activity to take place is the creation of a new system (option B). If the user attempts to add a new object to a non-existent system, a warning is produced and the build routine is automatically invoked. The screen that appears for the build system option is shown in Appendix C. Either a single compilation unit or a file containing a list of compilation units may be chosen.

```
command     = build | add | delete | change | recompile |
              show | language | unparse | help | set | query | quit.
build       = BUILD systemname (CU | FOCU ) filename.
add         = ADD object_class name.
delete      = DELETE object_class name.
delete      = CHANGE object_class name.
recompile   = RECOMPILE (GIAA | MAKE).
show        = SHOW object_class name.
language    = LANGUAGE (INPUT | OUTPUT) langname.
unparse     = UNPARSE ("*" | filename) object_class name.
help        = HELP [BRIEF | ALL].
query       = QUERY.
set         = SET VERBOSE (ON | OFF) |
              SET WARNING (ON | OFF) |
              SET MODE (MENU | INTERACTIVE).
quit        = QUIT.
object_class = SYS | CU | REL | PRO | MOD | DEF | IMP |.
              GEN | ALT | REV.
```

Conventions for syntax: Square brackets [ ] mean that the enclosed form is optional. Curly braces { } mean that the enclosed form is repeated zero or more times. A vertical bar | separates a list of alternate choices. Round brackets () imply that only one of the choices may be selected. Uppercase names represent terminals and must be entered in lower case. Non-terminals are written in lower case.

Figure 5.3: Interactive command language syntax

The new system created completely replaces the old one. During compilation, the default system name is `*linemode*`. This may be changed by specifying a system name using the `-s` command-line option.

5.2.3.2 Adding, changing, or deleting an object

The add, change, and delete options first prompt the user to specify the class of the object chosen. The classes available are those listed in Figure 4.4; the screen that

appears is shown in Appendix C. CMI verifies that there is an active system to which an object may be added to, deleted from, or changed in. If no system is currently active, an error message is produced. Though any class may be selected, only the classes `oc_def`, `oc_imp`, `oc_pro`, and `oc_cu` are currently acted on. The support for additional object classes may be incrementally added at a later date. An object of class `oc_imp` or `oc_pro` cannot be directly changed; only a compilation unit and a new definition module may be given. If the change is a new definition module, the new version is typed in interactively at the terminal. However, it is easier to change an object in a compilation unit, since the system editor is invoked on the desired compilation unit. The full power of the editor may then be used to alter the object in the manner required. Interactively typing in a new definition of an object tends to be error prone, and the parser is very unforgiving with syntax errors.

When the object selected is a compilation unit, then all objects inside the compilation unit are analyzed. Thus, changing a compilation unit may in fact involve altering more than one definition module. The algorithms are used to analyze each object inside the compilation unit to detect changes between the old version and the new one. If the change is determined to be inconsequential, then there are no affected objects and no recompilation needs to take place. If the change is local, then only the object itself needs to be recompiled (and possibly its corresponding implementation module). When a global change is detected, CMI displays the list of objects that are affected by the change. This may also include an informational message that says the Tichy-Baker algorithm should be invoked on a particular implementation module, if its corresponding definition module is affected by the change. This only occurs if there is in fact a corresponding implementation module in the system. CMI's concept of consistency is not the same as that of a program linker/loader: it allows a system to be considered consistent even if a definition

module does not have a corresponding implementation module in the system at the present time (and vice versa).

After the set of affected objects has been displayed, an ordered list of compilation units is presented. This is the set of compilation units that must be recompiled because of a change in one (or more) of the objects that is inside the changed compilation unit. The system is smart enough to not recompile the same compilation unit twice in a row; this occurs if the topological sorting of object dependencies determines that two objects that are affected by a change exist in the same compilation unit. Note however that since CMI allows more than one definition module to exist in a single compilation unit, there may be wasted effort by the compiler if it must recompile a single definition more than once. This situation can occur if two (or more) definition modules in the same compilation unit are affected by a change, but they have a non-zero number of other compilation units that must be recompiled between one another. The second (and subsequent) recompilations do not present a consistency problem, since if a compilation unit is recompiled for the second (or more) time, no change in the surrounding environment can have any affect on the previously affected objects; if it had, they would have been recompiled earlier in the first place. This problem of redundant compilations can be alleviated by limiting a compilation unit to contain a single definition module. However, this is not a requirement, and if this advice is not followed there may be a slight recompilation penalty. Furthermore, from a software engineering viewpoint, since a compilation unit is typically the implementation model of a single module, putting more than a single module into a compilation unit goes against the modularity provided by the language and mapped to the operating system's file structure.

After the set of compilation units has been presented, the user is asked if the change set is acceptable. If it is, the new object is entered into the system and the old one deleted. If it is not, then the change is not incorporated. If the change(s)

made in the object cause the system to become inconsistent, no recompilation analysis takes place. However, the change may still be incorporated (if desired). This situation may be common if a set of changes is made one at a time. Each compilation unit is edited and returned to the system, even if it temporarily makes the system inconsistent. After the set of changes is completed, the user then selects the verify option to re-evaluate the system. If everything is correct the system, with its new objects, may be consistent again. System inconsistencies tend to occur when a resource that was previously available is deleted. In this case the requests of client objects cannot be satisfied.

To determine what the affects of a change *would be*, it is easiest to simply tag the resource in question as **changed**. That way, full consequence analysis may take place in the "what if" scenario, without actually placing the system into an inconsistent state.

5.2.3.3 Recompiling

The recompile option may be selected to perform recompilation and consequence analysis on demand. The same services are invoked when the change option is selected. The user can choose to perform the recompilation using the global interface analysis algorithms, or using *make*-like rules.

5.2.3.4 System description

From the main screen, option S selects the system description operations. These may be used to describe the system or any of its parts, such as compilation units or definition modules. The information provided varies with the class of the object being displayed. For example, for a compilation unit the time of last modification

of the file and other operating system specific information is displayed, along with a list of the objects contained in it. For a definition module, the requests and provisions of the modules are shown. Information such as the transitive dependencies among resources, the adjacency list of the module, the heap of (possibly) affected clients (the *make chain*), and the strongly connected component members (if any) are shown. A sample description of a definition module is shown in Figure 5.4.

The services provided by this option may be used as an aid in program understanding. By starting at the system object and working downwards along the hierarchical links (for example, system to compilation unit to definition module), the overall structure of the program may be displayed. In addition, the dependencies of each object are shown. For example, the set of requirements a module has is shown, as well as the structure dependencies set up by other objects requesting resources from this object.

5.2.3.5 Language setting

Both the input and output languages are changeable to any of the supported languages. By default they are both set to CMI when the system starts up.

5.2.3.6 Unparsing

The unparsing routines simply traverse the internal graph model of a particular object and produce a textual representation of it. These routines are invoked automatically during compilation to produce the equivalent output language representation of the input language. The output may go to a file or to the terminal.

```
        Show definition module

        Enter name of the definition module:
Definition module d2

Requires:
  From d1:definition
    alpha:var
    beta:var
  From d3:definition
    d3_dummy1:var
Provides:
  To:
    d3:definition
  Provide:
    beta:var
    d2_dummy1:var
  To:
    d6:definition
  Provide:
    beta:var
    d2_dummy2:var
Dependencies:
  hierarchical dependence on compilation unit samp_d2.cmi (02/20/89 21:31:06)
  compilation dependence on definition d1
  compilation dependence on definition d3
  structure dependence on definition d3
  structure dependence on definition d6
  hierarchical dependence on module d2
CDG info:
  Last values: dfsnum = 2, tsnum = 0
  Adjacency list:
    d3:definition
    d6:definition
  Strongly connected components:
    d2:definition
    d3:definition
  Heap (topsort of affected scc's):
Key = 1     d6:definition d7:definition d5:definition
Key = 2     d10:definition
Key = 3     d9:definition


Press any key to continue...
```

Figure 5.4: System description of definition module

5.2.3.7 Verifying system consistency

A complete system verification can be made on demand if desired. This step is necessary if changes that were made have caused a system to become inconsistent; recompilation will not take place until the system is made consistent again.

5.2.3.8 Switching to interactive mode

Changing modes from menu-mode to interactive command-line mode and vice versa can be done from both environments. In addition, command-line mode may be chosen directly from the CMI invocation.

5.2.3.9 Message verbosity

Some of the diagnostics produced during parsing or system descriptions are informational. They may be turned off by toggling the "wordiness" of the messages.

5.2.3.10 Testing services

The test services are only available if CMI is compiled with its own internal debugging capabilities activated. In this case this option selects a screen of various testing tools, such as manipulation of the set routines, and the invocation of the INSPECT (if C/370 is the compiler used).

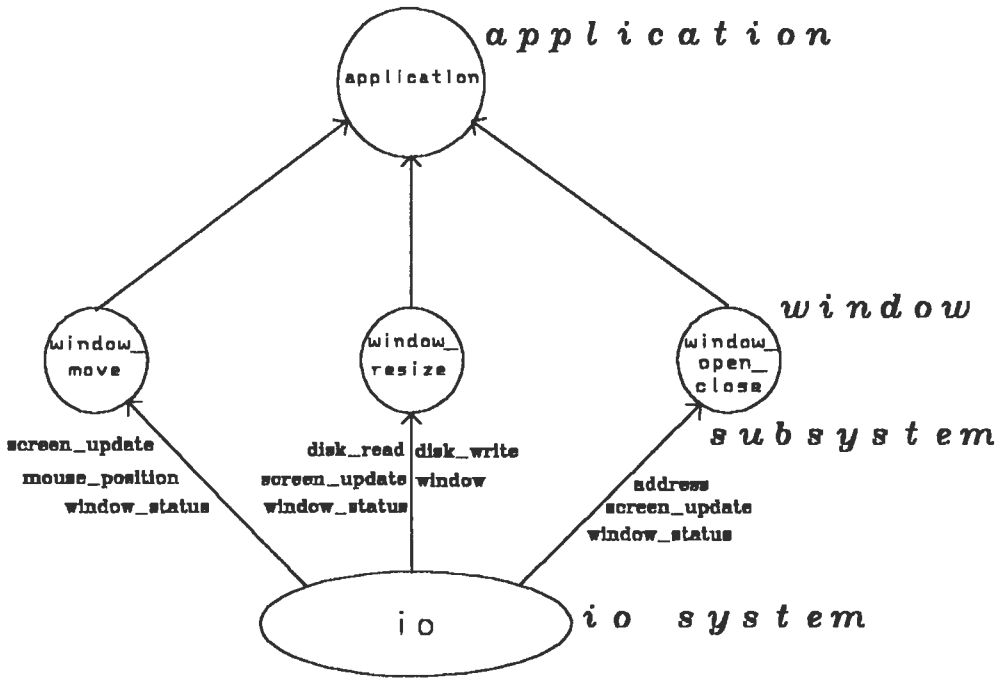
5.2.3.11 Exiting the system

Option X causes CMI to terminate.

5.3 Performance and results

Consider a simple system made up of five objects, as depicted graphically in Figure 5.5. The Modula-2 source which corresponds to this compilation dependency graph is given in Appendix E. The object `io` is a low-level definition module that provides input-output (I/O) facilities for some hypothetical graphics project. Sitting on top of this I/O system is a subsystem of interfaces for window manipulation. This subsystem is composed of three definition modules that use some of the services from the I/O module for performing machine-dependent operations such as interfacing to the screen and disk access, while providing services of their own, such as window resizing, hiding, movement, etc. A client module uses various resources provided by this window subsystem for some particular application program.

If a small change is made to the io definition module, say just to add or change some comments, *make* would still report that the three window subsystem modules and the application module must be recompiled, since they all use services from the io system, either directly or indirectly. Notice that the type `window_pointer` in the io module is currently unused by any other object in the system. If this type was removed from the io module, there is no real affect on the rest of the system, yet *make* would still require a complete system recompilation, since all it knows is that the io module has changed "in some way". In both of these cases, CMI reports that no clients of the io module require recompilation. The deletion of the `window_pointer` type would require the io definition module itself and possibly its corresponding implementation part (not shown here) to be recompiled, but that is all. Thus, in even a simple a system as this, using CMI has saved a significant number of recompilations.



The arcs connecting the window subsystem to the application are not labelled. This is because the application has requested all resources from each of the three window subsystem objects.

Figure 5.5: Sample graphics system graph model

In "real" systems, low-level library facilities like the io module exist in large numbers. However, it becomes increasingly difficult to provide enhanced support for the service libraries because of the recompilation that would take place because of a change. The above examples illustrated the recompilations that would be required if a change is either inconsequential or local. A very frequent maintenance change involves the addition of a new resource, say a new procedure, to some existing library

system. Since it is new, it is very unlikely that many existing objects have dependencies on this newly added function. Yet traditional recompilation strategies would require the recompilation of the entire library and of each application that used the library in any way. CMI detects that the newly added resource does not yet have any dependencies on it, and thus the recompilation wave is halted before it even starts; nothing but the definition module itself (and possibly the implementation) must be recompiled.

If the type of the variable `mouse_position` in the io module was changed, CMI would detect that the only object that actually has a dependence on it is the `window_move` definition module, and the application module indirectly. The procedure `winmov` in this module expects a parameter of the `mouse_position` type. *Make* would again recompile the whole system because of the change.

These recompilation savings can become much more significant in larger systems. The example shown above contains only five objects. A production program will likely contain a great many more modules, with several interface layers similar to that shown for the window subsystem. It can be seen that the halting of the recompilation wave as soon as possible is extremely beneficial, especially if the change occurred near the bottom of the system hierarchy.

Appendix E contains the CMI code which models a slightly larger system. This example system is given in [Mull 86]. The resource `delta` has been tagged as **`changed`** in the `SAMP_D1.CMI` compilation unit. This system was entered into CMI and consequence analysis was performed on the proposed change. Figure 5.6 illustrates that while *make* would force the recompilation of nine compilation units, CMI determines that only three compilation units are actually affected and require recompilation.

```
warning (1): can't find object d2 in system *linemode*
warning (2): can't find object d4 in system *linemode*
warning (3): can't find object d8 in system *linemode*
warning (4): semantic analysis failed on compilation unit samp_d1.cmi
             with 3 errors

Object d1:definition has had a Type III (global) change
The following objects are affected:
  d8:definition
  d13:definition


Press any key to continue...
Recompile the following compilation units in this order:
      1. samp_d1.cmi
      2. samp_d89.cmi
      3. samp_d13.cmi


Press any key to continue...
```

The error and warning messages are produced from the compilation of the SAMP_D1.CMI file. CMI reports the errors since objects d2, d4, and d8 are in other compilation units that have not yet been entered. The system was built with the following command:

cmi / -f sample.cmi -m m

The set of compilation units that would need to be recompiled using *make*'s unit interconnection (time-stamp) based model for the same change is shown in Figure 5.7.

Figure 5.6: Sample consequence analysis

To verify that the results obtained from the smaller examples were correct, change analysis was performed on a "real life" example: a Modula-2 compiler that runs on VM/CMS and is itself written in Modula-2. It is a port of Wirth's original compiler for the Lilith computer, and consists of 82 compilation units, split into 41

```
          Recompile the system using 'make'-like rules


info: Cu samp_d1.cmi is newer than system *linemode*.
       cu  samp_d1.cmi: ( 2/24/89  9:29:34)
       sys *linemode*: ( 2/24/89  9:29:26)
The following objects are affected by the change in cu samp_d1.cmi:
  d2:definition d3:definition
  d6:definition d7:definition d5:definition
  d10:definition
  d9:definition
  d4:definition
  d11:definition d12:definition
  d8:definition
  d13:definition

Recompile the following compilation units in this order:
      1. samp_d1.cmi
      2. samp_d2.cmi
      3. samp_d6.cmi
      4. samp_d10.cmi
      5. samp_d89.cmi
      6. samp_d34.cmi
      7. samp_d11.cmi
      8. samp_d89.cmi
      9. samp_d13.cmi


Press any key to continue...
```

Definition modules d2 and d3, d5, d6, and d7, and d11, d12 form three strongly connected components. The system represents each of these as a single line in the set of affected objects, since each is treated as a logical unit.

Figure 5.7: Set of affected objects using *make*-style rules

definition modules, 34 implementation modules, 1 program module, and 6 assembler modules. The compiler is broken into roughly six interface layers. The bottom three layers are system-dependent services and low-level library facilities, while the top three make up the compiler proper. The compiler can be thought of as a user application program sitting on top of a support layer, since the compiler itself is written in Modula-2. The complete compiler was put under control of the CMI system, and random changes were made to its source at each level. The results of this analysis are shown in Figure 5.8. For each layer, layout changes and comment modifications were made. Since this type of change is inconsequential, there is a column in Figure 5.8 for each layer that shows zero affected compilation units, and a 100% savings in compile-time. This is because for the same change, a *make*-like tool would recompile all of the objects on the *make-chain*. This set of objects is indicated in the figure by the column an arrow under it.

At the top of each column in the figure is the percentage of time saved by the elimination of redundant recompilations. The measurements were made on an IBM 3090 running at an average load factor of 14%. This time represents CPU time only, and not elapsed real time. On a heavily loaded system the elapsed time may be much longer. Included in the calculations of the time saved is the time that CMI required to compute the optimal set of affected objects. This is usually much less than the recompilation of a single implementation or even of a definition module. For example, to compute the set of objects from layer one for a change in the bottom-most definition module required 21% less time than to recompile the definition module itself. It can be seen from the figure that the benefits of change analysis are very pronounced for low-level routines. For example, the bottom-most layer of the compiler has 58 other modules dependent on it, and all would be recompiled if even a comment was changed in the source if a *make*-like tool was used. Even when 35

objects are affected by a change (instead of the 58 that *make* would report), a 61% savings is recompilation time is still produced.

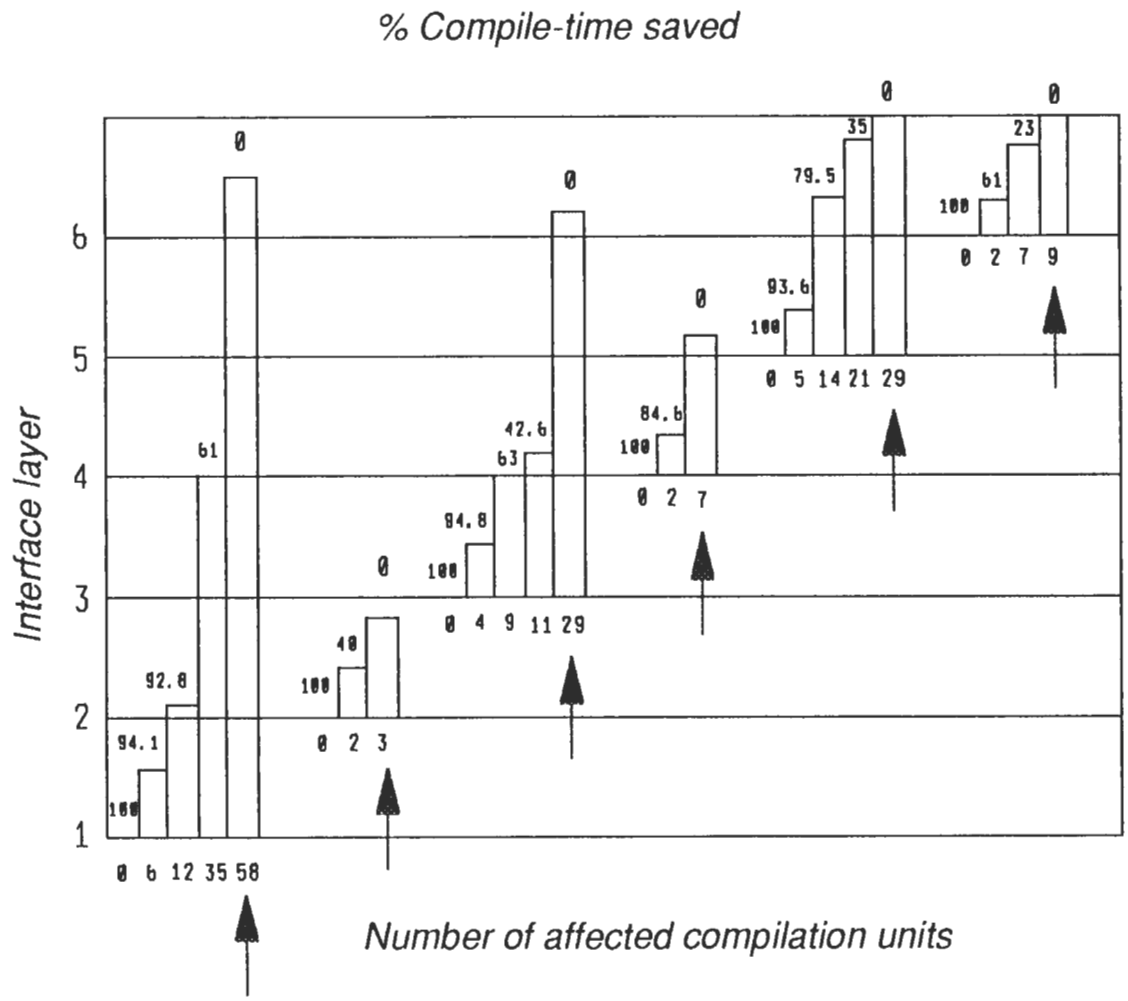
Other (non-trivial) modifications of resources were carried out. These included changing the representation of types, changing the parameter list of procedures, and tagging various resources as "changed". For example, one of the most far reaching changes occurred when the exported opaque type `LongInt` was tagged as changed in one of the definition modules in layer one. This definition modules provides 32-bit integer support for the compiler. This type is used in 35 compilation units, though the definition module itself was requested by 58 compilation units, as shown in Figure 5.8.

Non-trivial changes such as these usually produce savings not quite as large as with the first case (layout and comment alterations), but they are still quite significant. When one considers that many of the compilations being saved involve considerable machine time and resources, since they are implementation modules, this makes the elimination of as many recompilations as possible even more desirable.

A number of other sample programs were studied. When randomly tagging a resource in a definition module as changed, the number of objects that *make* reported were affected averaged between two and three times the number reported by CMI. Assuming the sample used[38] is indicative of the majority of programs, this number implies that the recompilation savings produced by CMI are significant.

---

[38] Most of the testing samples for Modula-2 were taken off an internal IBM network forum for Modula-2. A few others were taken from usenet and existing Modula-2 programs at the University of Toronto. All of the CMI test files were written by the author.



The numbers at the bottom of each column indicate the number of affected compilation units for some particular change. The numbers at the top of each column show the percentage of compile-time that is saved when fewer objects must be recompiled. This time also includes the time needed for CMI to compute the set of affected objects. The columns with the arrow underneath are the number of objects that *make* reports are affected by the same change, which remain constant no matter what type of change occurred.

Figure 5.8: Change analysis on the Modula-2 compiler

As discussed in Chapter 4, the current Modula-2 interface has a few restrictions. These cause CMI to incorrectly miss recompiling a module when it is in fact affected by a change. This problem is not one inherent in CMI, but rather the limits imposed by a prototype interface to Modula-2.

5.4 Summary

This chapter presented the use of CMI as a stand-alone tool. It was shown that CMI can operate in a variety of modes, and that the menu-oriented mode is the most convenient for program exploration and consequence analysis.

To illustrate the benefits of CMI, sample systems were introduced and the recompilations required by a tool such as *make* were compared to those recompilations required by CMI for the same change in an interface. In all cases CMI produces a significant savings by limiting the number of recompilations required to a number much less than that produced by *make*.

CHAPTER 6

Conclusions

6.1 Summary of results

This thesis presented CMI, a practical implementation of the global interface analysis algorithms. The current implementation is tailored toward the programming languages Ada and Modula-2, and the CMI module interconnection language in particular; however, the CMI implementation could be used as a basis for an implementation of these algorithms for any strongly typed, separately compiled programming language.

The primary motivation for this research was the need to alleviate the persistent problems caused by unnecessary recompilations in a large, evolving software system when a low-level interface is changed. Traditional recompilation strategies are based on the unit interconnection model and usually rely on operating system file time-stamps to determine the set of affected modules after an interface change. This conservative method often forces the recompilation of modules that are not at all affected by the change in the basic interface. Using CMI, the time and resources needed to bring a system back up to date after a change to a basic interface is greatly reduced. Consequence analysis may be performed to predict and limit the effects of the change on the entire system.

During the design phase of the software project, the designers could use CMI to determine the effects of an editing change on a basic component in a software system. When integrated into Rigi, a window could appear after an editing change on a module interface, listing the affected modules in the system and prompting for

answer to see if the change should in fact be carried out. A user can then explore alternate changes and estimate their effects on the entire system. Interfaces need not be frozen before they are sufficiently explored and tested. After an interface change, CMI determines the minimal set of affected client modules that has to be recompiled and thus avoids many redundant recompilations.

The maintainers of a software project could use CMI, in conjunction with Rigi, as a program understanding aid. Presentation of software development information, such as the topology of a large system, can be extremely complex. Since the maintainer of a system is quite often not the designer, they may be unfamiliar with the overall structure of the system. Changing a low-level interface can have far-reaching consequences, and may involve tracking down all of the interface's clients to ensure that they are recompiled, and in the correct manner. In a software development environment such as Rigi, aided by CMI, the maintainer could easily decide if a change made was acceptable; if it was, it may be implemented, otherwise it could be undone.

As a stand-alone tool, CMI can be used in several ways. It may be invoked as a one-line command to just check the syntax of source files, or to translate (compile) a source file from any supported language to any another. When used interactively, it may be used to build a system and experiment with changes to objects in the system. It may be used in a command-language mode, or in a menu-oriented environment. When it becomes integrated into the Rigi system, a sophisticated windowing system will be used.

In addition to the benefits that CMI provides as outlined above, the implementation was an exercise in portable, object-oriented programming in C. The basic elements in the CMI system are generic *objects* which communicate with each other through an explicit and well-defined *require-provide* mechanism. Operations on these

objects are pseudo-inherited through function pointers that reflect the base type of the object. Abstraction mechanisms were used to allow experimentation with different implementations of type representations, and modularization was employed (in as much as C allows it) whenever possible; most compilation units have a small number of exposed entry points, and the resources provided by the compilation unit are given in the `.h` file. By simply changing a few definitions in the `makefile` CMI can be re-targeted to various operating systems and compilers.

6.2 Future research

The current implementation has a number of shortcomings, and several interesting research areas were uncovered while working on the project.

The design of CMI is based on the Rigi model; integration of CMI into the Rigi software development environment is thus the next logical step. Chapter 5 presented the use of CMI as a stand-alone tool; used in conjunction with Rigi, the benefits would be compounded.

Though provisions for Ada have been made in this implementation, a working Ada interface has not been implemented yet. This was for several reasons. The nesting and visibility mechanisms of Ada are rather complex, since objects may be nested in a (nearly) arbitrary order and to any complexity. The language is so rich that the semantic checks the Ada compiler must perform in the front end are numerous and cumbersome; it is not uncommon for over 100,000 lines of code to be written to translate Ada source text to some intermediate representation, or high form, such as Diana. Other factors that cause extra work include overloading of operator names, ambiguous references that the **use** clause provides, name remapping provided through the **renames** clause, and the separate compilation facility provided

by the **separate** clause (to name but a few). The global interface analysis algorithms were written under the assumption that they would operate in a software development environment that would provide efficient access to the compilation dependencies in the system. The CMI implementation has to extract these dependencies from the source text, be it written in CMI, Modula-2, or Ada. Rather than implement a very restricted Ada interface, it was decided to leave the actual implementation for further work, and make provisions in the design for extensions to allow Ada to work under CMI at a later date. If an Ada compilation system was made available to be suitably altered to allow the internals of CMI to work in conjunction with the library manager, or if the internals of an Ada library system, with all of its stored high-form information on package contents and compilation dependencies, was available for CMI to query, then Ada support could be added with very little effort.

Although neither Ada nor Modula-2 support recursive inter-module dependencies, the CMI language does not explicitly preclude them. However, the current implementation only "tolerates" recursive inter-module dependencies; it does not fully support them. During the calculation of the change propagation sets, if a strongly connected component is found with more than one member, processing stops with a warning. The solution to the problem of propagating a change into a strongly connected component with greater than one member requires data flow analysis to solve the set of simultaneous equations that arise. This was deemed too complex to include at the present time.

Currently CMI re-parses source text whenever it is initialized. Although this parsing is fast, it would be beneficial to store the internal graph data base externally in a compressed, easily retrievable format that could be loaded into CMI without requiring semantic analysis. This style of binary image re-loading would be faster

than building a system up from scratch every time a new session is started. This mechanism of secondary storage of the internal data base could also aid in memory management for extremely large systems, by allowing inactive portions of the graph to be dumped out while processing, and demand loaded into main memory when needed again. Systems such as PGRAPHITE [WWFT 88] provide a persistent storage mechanism to aid in this. However, when CMI is integrated into Rigi, this should not be as problematic since the Rigi system already uses a graph data base to store objects manipulated by the Rigi editor.

Neither the CMI language nor the implementation is currently designed to handle incomplete systems or constructs, such as the PIC/Ada system is able to do. Having inconsistencies in a large system, especially during the earlier stages of design and development, seems to be quite natural. At the moment, the semantic checks that occur when a new object is entered into the system fail unless the new system is completely consistent; the recompilation algorithms cannot be invoked unless the system is in a consistent state. Removing this restriction, and allowing a partial computation of affected objects, may be beneficial.

One of the most interesting topics that deserves more research is visibility control mechanisms, and inter-module communications. As outlined in Chapter 2, nesting has been the predominant visibility control mechanism until languages such as Ada and Modula-2 provided facilities for explicit inter-module relationships. However, the mixing of both nesting and *require-provide* mechanisms, as found in Ada (and in some part in Modula-2's local modules), only confuses the issue. Strict, clear, and explicit producer-consumer relationships among objects in a software system would alleviate much of the confusion that arises from programming language ambiguities. From a practical point of view it would also reduce the task of the compiler front end in deducing which variable came from where, and if its use

in a particular construct is valid or not. It also tends to make the object designer think more in terms of sound software engineering principles such as data abstraction, information hiding, and modularization.

A software development environment is supposed to aid a programmer in understanding a complex system, and the programming language used to implement the application being developed plays a crucial part in its design. In 1973 a paper by Terry Winograd entitled "Breaking the Complexity Barrier (Again)" [Wino 73] expounded the need for a new generation of programming systems, that would be more powerful and easier to use than the current state of the art. As software systems become larger and more complex, the software development environment's role becomes even more important. The tools that it provides must aid in all aspects of software development, so that using new and interesting features of a programming language or design methodology is not cumbersome and something to be avoided, but rather something to be exploited. The questions raised by Winograd almost two decades ago are thus more important than ever. Tools such as CMI should aid a programmer by alleviating the manual and error-prone details of jobs such as recompilation and interface analysis, and let them get on with the real task at hand: writing good software.

Bibliography

[ACCE 85] Avsnit, C.N.; N.H. Cohen; J.B. Goodenough; and R.S. Eanes. *Ada in Practice*, Springer-Verlag, 1985.

[ACRS 88] Alpern, B.; A. Carle; B. Rosen; P. Sweeney; and K. Zadeck. "Graph Attribution as a Specification Paradigm," *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Software Development Environments*, (Boston, MA; November 28-30, 1988). In *ACM SIGPLAN Notices*, 24(2), pp. 121-129, February 1989.

[AhHU 83] Aho, A.V.; J.E. Hopcroft; and J.D. Ullman. *Data Structures and Algorithms*, Addison-Wesley, 1983.

[AhSU 86] Aho, A.V.; R. Sethi; and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[AmOd 87] Ambras, J.; and V. O'Day. "MicroScope: A Program Analysis System," *Proceedings of the Twentieth Hawaii International Conference on System Sciences* (HICSS-20), pp. 71-81, January 1987.

[AmOd 88] Ambras, J.; and V. O'Day. "MicroScope: A Knowledge-Based Programming Environment," *IEEE Software*, 5(3), pp. 50-57, May 1988.

[ANSI 88] Draft Proposed American National Standard for Information Systems – Programming Language C. Document number X3J11/88-159. December 1988.

[Ardo 87] Ardö, A. "Experience Acquiring and Retargeting a Portable Ada Compiler," *Software – Practice and Experience*, 17(4), pp. 291-307, April 1987.

[Bake 82] Baker, T.P. "A Single-Pass Syntax-Directed Front End for Ada," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, (Boston, MA; June 23-25, 1982). In *ACM SIGPLAN Notices*, 17(6), pp. 318-327, June 1982.

[Barn 80] Barnes, J.G.P. "An Overview of Ada," *Software – Practice and Experience*, Vol. 10, pp. 851-887, October 1980.

[Barn 84] Barnes, J.G.P. *Programming in Ada*, Second Edition, Addison-Wesley, 1984.

[BaSn 85] Bahlke, R.; and G. Snelting. "The PSG – Programming System Generator," *Proceedings of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments*, (Seattle, WA; June 25-28, 1985). In *ACM SIGPLAN Notices*, 20(7), pp. 28-33, July 1985.

[BBB 85] "The Modularity Papers," *Computing Technology*, February 1985.

[BCTI 88] Borras, P.; D. Clément; Th. Despeyroux; J. Incerpi; G. Kahn; B. Lang; and V. Pascual. "CENTAUR: the system," *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Software Development Environments*, (Boston, MA; November 28-30, 1988). In *ACM SIGPLAN Notices*, 24(2), pp. 13-24, February 1989.

[BGMT 88] Boudier, G.; F. Gallo; R. Minot; and I. Thomas. "An Overview of PCTE and PCTE+," *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Software Development Environments*, (Boston, MA; November 28-30, 1988). In *ACM SIGPLAN Notices*, 24(2), pp. 248-257, February 1989.

[Blow 84] Blower, M.I. "An Efficient Implementation of Visibility in Ada," *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*. In *ACM SIGPLAN Notices*, 19(6), pp. 259-265, June 1984.

[Booc 83] Booch, G. *Software Engineering with Ada*, Benjamin/Cummings, 1983.

[Booc 87] Booch, G. *Software Components With Ada – Structure, Tools, and Subsystems*, Benjamin/Cummings, 1987.

[Bord 87] *Turbo C User's Guide*, and *Turbo C Reference Guide*, Borland International, Inc. 1987.

[Bord 88] *Turbo Pascal User's Guide*, and *Turbo Pascal Reference Guide*, Borland International, Inc. 1988.

[Bowl 82] Bowles, K.L. "Linked Ada Modules Shape Software Systems," *Electronic Design*, 30(15), pp. 117-126, July 1982. In *The Ada Programming Language: A Tutorial*, S.H. Saib and R.E. Fritz (editors). IEEE Computer Society Press, (IEEE Catalog No. EHO202-2), pp. 458-466, 1983.

[BrLe 85] Brandes, T.; and K. Lewerentz. "GRAS: A Non-standard Data Base System within a Software Development Environment," *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-large*, (Harwichport, Mass., June 9-12), pp. 113-121, June 1985.

[BuDr 80] Buxton, J.N.; and L.E. Druffel. "Requirements for an Ada Programming Support Environment: Rationale for STONEMAN," *Proceedings CompSAC 80*, pp. 66-72, October 1980. In *The Ada Programming Language: A Tutorial*, S.H. Saib and R.E. Fritz (editors). IEEE Computer Society Press, (IEEE Catalog No. EHO202-2), pp. 58-64, 1983.

[BuHZ 85] Budinsky, F.J.; R.C. Holt; and S.G. Zaky. "SRE – A Syntax Recognizing Editor," *Software – Practice and Experience*, 15(5), pp. 489-497, May 1985.

[Burk 87] Burke, M. "An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data Flow Analysis," IBM Computer Science Research Report RC 12702, IBM T.J. Watson Research Center, Yorktown Heights, NY, September 1987.

[BuRy 87] Burke, M.; and B.G. Ryder. "Incremental Iterative Data Flow Analysis Algorithms," IBM Computer Science Research Report RC 13710, IBM T.J. Watson Research Center, Yorktown Heights, NY, October 1987.

[Cahn 88] Cahn, R.S. "An Algorithm to Draw Networks and Graphs," IBM Computer Science Research Report RC 14126, IBM T.J. Watson Research Centre, Yorktown Heights, NY, October 1988.

[Call 88] Callahan, D. "The Program Summary Graph and Flow-sensitive Interprocedural Data Flow Analysis," *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and*

Implementation, (Atlanta, GA; June 22-24, 1988). In *ACM SIGPLAN Notices*, 23(7), pp. 47-56, July 1988.

[Capl 85a] Caplinger, M. "Structured Editor Support for Modularity and Data Abstraction," *Proceedings of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments*, (Seattle, WA; June 25-28, 1985). In *ACM SIGPLAN Notices*, 20(7), pp. 140-147, July 1985.

[Capl 85b] Caplinger, M. "A Single Intermediate Language for Programming Environments," Ph.D. dissertation, Rice University Computer Science Technical Report TR85-28, August 1985.

[Chri 75] Christofides, N. *Graph Theory – An Algorithmic Approach*, Academic Press, New York, 1975.

[CKTW 87] Cooper, K.D.; K. Kennedy; L. Torczon; A. Weingarten; and M. Wolcott. "Editing and Compiling Whole Programs," *Proceedings of the 2nd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Palo Alto, CA; December 9-11, 1986). In *ACM SIGPLAN Notices*, 22(1), pp. 92-101, January 1987.

[Clem 88] Clemm, G.M. "The Workshop System – A Practical Knowledge-Based Software Environment," *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Software Development Environments*, (Boston, MA; November 28-30, 1988). In *ACM SIGPLAN Notices*, 24(2), pp. 55-64, February 1989.

[Clev 88] Cleveland, L. "PUNS: A Program UNderstanding Support Environment," IBM Computer Science Research Report RC 14043, IBM T.J. Watson Research Centre, Yorktown Heights, NY, September 1988.

[Clde 84] Clemmensen, G.B.; and O.N. Oest. "Formal Specification and Development of an Ada Compiler – A VDM Case Study," *Proceedings 7th International Conference on Software Engineering*, (Orlando, FL; March 26-29, 1984), pp. 430-439, March 1984.

[ClRZ 88] Clarke, L.A.; D.J. Richardson; and S.J. Zeil. "TEAM: A Support Environment for Testing, Evaluation, and Analysis," *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Software Development Environments*, (Boston, MA; November 28-30, 1988). In *ACM SIGPLAN Notices*, 24(2), pp. 153-162, February 1989.

[ClWW 86] Clarke, L.A.; Wileden, J.C.; and A.L. Wolf. "Graphite: A Meta-Tool for Ada Environment Development," *IEEE Computer Society Second International Conference on Ada Applications and Environments*, (Miami Beach, FL; April 8-10, 1986), (IEEE Catalog No. 86CH2281-4), pp. 81-90, April 1986.

[Cohe 88] Cohen, N.H. "Formal Specification and Verification of Ada Packages," IBM Computer Science Research Report RC 14105, IBM T.J. Watson Research Centre, Yorktown Heights, NY, October 1988.

[CoKe 88] Cooper, K.D.; and K. Kennedy. "Interprocedural Side-Effect Analysis in Linear Time," *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, (Atlanta, GA; June 22-24, 1988). In *ACM SIGPLAN Notices*, 23(7), pp. 57-66, July 1988.

[CoKT 85] Cooper, K.D.; K. Kennedy; and L. Torczon. "The Impact of Interprocedural Analysis and Optimization on the Design of a Software Development Environment," *Proceedings of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments*, (Seattle, WA; June 25-28, 1985). In *ACM SIGPLAN Notices*, 20(7), pp. 107-116, July 1985.

[CoKT 86a] Cooper, K.D.; K. Kennedy; and L. Torczon. "Interprocedural Optimization: Eliminating Unnecessary Recompilation," *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, (Palo Alto, CA; June 25-27, 1986). In *ACM SIGPLAN Notices*, 21(7), pp. 58-67, July 1986.

[CoKT 86b] Cooper, K.D.; K. Kennedy; and L. Torczon. "The Impact of Interprocedural Analysis and Optimization in the R^n Programming Environment," *ACM Transactions of Programming Languages and Systems*, 8(4), pp. 491-523, October 1986.

[Crow 87] Crowe, M.K. "Dynamic Compilation in the Unix Environment," *Software – Practice and Experience*, 17(7), pp. 455-467, July 1987.

[Davi 84] Davis, R.J. "Ada as a Development Tool," *Second IBM Productivity Tools Symposium*, (September 10-13, 1984; Atlanta, GA), pp. 531-564.

[DeBi 87] Debnath, N.C.; and J.M. Bieman. "A Representation and Analysis of Interprocedural Structure," *Proceedings of the Twentieth Hawaii International Conference on System Sciences* (HICSS-20), pp. 92-100, January 1987.

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

[DFSF 80] Dewar, R.B.K.; G.A. Fisher, Jr.; E. Schonberg; R. Froehlich; S. Bryant; C. F. Goss; and M. Burke. "The NYU Ada Translator and Interpreter," *Proceedings, CompSAC 80*, pp. 59-65, October 1980. In *The Ada Programming Language: A Tutorial*, S.H. Saib and R.E. Fritz (editors). IEEE Computer Society Press, (IEEE Catalog No. EHO202-2), pp. 72-78, 1983.

[DiLC 87] Didriksen, T.; A. Lie; and R. Conradi. "IDL As a Data Description Language for a Programming Environment Database," *ACM SIGPLAN Notices*, 22(11), pp. 71-78, November 1987.

[DJCL 89] Donahue, J.; M. Jordan; L. Cardelli; L. Glassman; B. Kalsow; and G. Nelson. *Modula-3 Report (Draft)*, Olivetti Research Center, and Digital Equipment Corporation Systems Research Laboratory. January 1989.

[DoD 78] Department of Defense, United States of America. "STEELMAN," Requirements for High Order Computer Programming Languages, June 1978.

[DoD 83] Department of Defense, United States of America. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983. February 1983.

[Dona 85] Donahue, J. "Integration Mechanisms in Cedar," *Proceedings of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments*, (Seattle, WA; June 25-28, 1985). In *ACM SIGPLAN Notices*, 20(7), pp. 245-251, July 1985.

[EsGK 84] Estubilier, J.; Ghoul, S.; and S. Krakowiak. "Preliminary Experience with a Configuration Control System for Modular Programs," *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Pittsburgh PA; April 23-25, 1984). In *ACM SIGPLAN Notices*, 19(5), pp. 149-156, May 1984.

[Even 79] Even, S. *Graph Algorithms*, Computer Science Press, 1979.

[FeFr 86] Felleisen, M.; and D.P. Friedman. "A Closer Look at EXPORT and IMPORT Statements," *Computer Language*, 11(1), pp. 29-37, January 1986.

[Feld 79] Feldman, S. "Make – A Program for Maintaining Computer Programs," *Software – Practice and Experience*, 9(3), pp. 255-265, 1979.

[FeOW 87] Ferrante, J.; K.J. Ottenstein.; and J.D. Warren. "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, 9(3), pp. 319-349, July 1987.

[Fish 78] Fisher, D.A. "DoD's Common Programming Language Effort," *Computer*, pp. 24-33, March 1978. In *Tutorial: Programming Language Design*, A.I. Wasserman (editor). (IEEE Catalog No. EHO164-4), pp. 316-325, 1980.

[Fost 86] Foster, D.G. "Separate Compilation in a Modula-2 Compiler," *Software – Practice and Experience*, 16(2), pp. 101-106, February 1986.

[FrAr 82] Frankel, G.; and R. Arnold. "Linkage of Ada Components – Theme & Variations," *Proceedings of the AdaTEC Conference on Ada*, (Arlington, VA), (ACM Order No. 825821), pp. 201-211, October 6-8 1982.

[Galk 80] Galkowski, J.T. "Modularity and Data Abstraction in Ada," *IBM Software Engineering Exchange*, pp. 13-17, October 1980. In *The Ada Programming Language: A Tutorial*, S.H. Saib and R.E. Fritz (editors). IEEE Computer Society Press, (IEEE Catalog No. EHO202-2), pp. 132-136, 1983.

[GeMS 77] Geschke, C.M.; J.H. Morris Jr.; and E.H. Satterthwaite. "Early Experience with Mesa," *Communications of the ACM*, pp. 540-553, August 1977. In *Tutorial: Programming Language Design*, A.I. Wasserman (editor). (IEEE Catalog No. EHO164-4), pp. 484-497, 1980.

[GrPr 85] Gries, D.; and J. Prins. "A New Notion of Encapsulation," *Proceedings of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments*, (Seattle, WA; June 25-28, 1985). In *ACM SIGPLAN Notices*, 20(7), pp. 131-139, July 1985.

[Gutk 87] Gutknecht, J. "One-Pass Compilation at its Limits – A Modula-2 Compiler for the Xerox Dragon Computer," *Software – Practice and Experience*, 17(7), pp. 469-484, July 1987.

[Habe 79] Habermann, A.N. "Tools for Software System Construction," in *Software Development Tools*, W.E. Riddle and R.E. Fairley (editors), *Proceedings of the Workshop on Software Development Tools*, (Pingree Park, CO; May 1979), Springer-Verlag 1980.

[Habe 80] Habermann, A.N. "The Use of Ada Packages," *Using Selected Features of Ada: A Collection of Papers*, CENTACS, US Army Communications-Electronics Command, March 1981. In *The Ada Programming Language: A Tutorial*, S.H. Saib and R.E. Fritz (editors). IEEE Computer Society Press, (IEEE Catalog No. EHO202-2), pp. 242-255, 1983.

[HaNo 86] Habermann, A.N.; and D. Notkin. "Gandalf: Software Development Environments," *IEEE Transactions on Software Engineering*, SE-12(12), pp. 1117-1127, December 1986.

[HeKr 88] Heimbigner, D.; and S. Krane. "A Graph Transform Model for Configuration Management Environments," *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on*

Software Development Environments, (Boston, MA; November 28-30, 1988). In *ACM SIGPLAN Notices*, 24(2), pp. 216-225, February 1989.

[Hilf 82] Hilfinger, P.N. *Abstraction Mechanisms and Language Design* , Ph.D. dissertation, MIT Press, 1983.

[Hoar 81] Hoare, C.A.R. "The Emperor's Old Clothes," *Communications of the ACM*, pp. 75-83, February 1981. In *The Ada Programming Language: A Tutorial*, S.H. Saib and R.E. Fritz (editors). IEEE Computer Society Press, (IEEE Catalog No. EHO202-2), pp. 487-495, 1983.

[HoKM 87] Hood, R.; K. Kennedy; and H.A. Müller. "Efficient Recompilation of Module Interfaces in a Software Development Environment," *Proceedings of the 2nd ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, (Palo Alto, CA; December 9-11, 1986). In *ACM SIGPLAN Notices*, 22(1), pp. 180-189, January 1987.

[HoRB 88] Horwitz, S.; T. Reps; and D. Binkley. "Interprocedural Slicing Using Dependence Graphs," *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, (Atlanta, GA; June 22-24, 1988). In *ACM SIGPLAN Notices*, 23(7), pp. 35-46, July 1988.

[IBM 86a] *Virtual Machine/System Product: System Product Interpreter Reference (Release 5)*, IBM SC24-5239-2, December 1986.

[IBM 86b] *IBM C for System/370 Program Offering - C language Manual*, IBM SC09-1128-0, 1986.

[IBM 86c] *IBM Development System for the Ada Language: VM/CMS User's Guide and Reference Manual*, IBM SC09-1134-0, September 1986.

[IBM 87a] *IBM C/2 Language Reference*, IBM, September 1987.

[IBM 87b] *OS PL/I V2 Programming: Language Reference (Release 1)*, IBM SC26-4308-0, December 1987.

[IBM 88a] *IBM C/370 User's Guide*, IBM SC09-1264-00, November 1988.

[IBM 88b] *INSPECT for C/370 and PL/I: Using INSPECT Under CMS (Release 1)*, IBM SC26-4529-0, December 1988.

[IBM 88c] *SAA Common Programming Interface C Reference - Level 2*, IBM SC09-1308-0, November 1988.

[IBM 88d] *Reference Manual for the Software Engineering Design Language (SEDL) Version 1.2*, IBM ZZ26-3705-0, October 1988.

[IBM 88e] *IBM AIX/RT Modula-2 Development System User's Guide*, IBM SC23-2124-0, April 1988.

[Ichb 79] Ichbiah, J., *et al.* "Rationale for the Design of the ADA Programming Language," *ACM SIGPLAN Notices*, 14(6), May 1979.

[John 86] Johnson, S.C. "Yacc: Yet Another Compiler-Compiler," in *UNIX Programmer's Supplementary Documents*, Vol. PS1:15, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version, April 1986.

[Jos1 87] Joslin, A.J. "Extended Pascal – Illustrative Features," *ACM SIGPLAN Notices*, 22(5), pp. 18-19, May 1987.

[KaFe 87] Kaiser, G.E.; and P.H. Feiler; "An Architecture for Intelligent Assistance in Software Development," *Proceedings 9th International Conference on Software Engineering*, (Monterrey, CA; March 30 - April 2, 1987), (IEEE Catalog Number 87CH2432-3), pp. 180-188, 1987.

[KaFP 88] Kaiser, G.E.; P.H. Feiler; and S.S. Popovich. "Intelligent Assistance for Software Development and Maintenance," *IEEE Software*, 5(3), pp. 40-49, May 1988.

[KaKM 87] Kaiser, G.E.; S.M. Kaplan; and J. Micallef. "Multiuser, Distributed Language-Based Environments," *IEEE Software*, 4(6), pp. 58-67, November 1987.

[Kenn 81] Kennedy, K. "A Survey of Data Flow Analysis Techniques," in *Program Flow Analysis Theory and Applications*, S.S. Muchnick and N.D. Jones (editors), Prentice-Hall, 1981.

[KeRi 88] Kernighan, B.W.; and D.M. Ritchie. *The C Programming Language (Second Edition)*, Prentice-Hall, 1988.

[Koen 88] Koenig, S. "ISEF An Integrated Industrial−strength Software Engineering Framework," *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Software Development Environments*, (Boston, MA; November 28-30, 1988). In *ACM SIGPLAN Notices*, 24(2), pp. 45-54, February 1989.

[Kran 82] Kranc, M.E. "A Command Language for the Ada Environment," *Proceedings of the AdaTEC Conference on Ada*, (Arlington, VA), (ACM Order No. 825821), pp. 181-186, October 6-8, 1982.

[Kr01 86] Krogdahl, S.; and K.A. Olsen. "Ada, as seen from Simula," *Software − Practice and Experience*, 16(8), pp. 689-700, August 1986.

[Lamb 87] Lamb, D.A. "IDL: Sharing Intermediate Representations," *ACM Transactions on Programming Languages and Systems*, 9(3), pp. 297-318, July 1987.

[Lamp 84] Lampson, B. "Hints for Computer System Design," *IEEE Software*, I(1), January 1984.

[LaSa 79] Lauer, H.C.; and E.H. Satterthwaite. "The Impact of Mesa on System Design," *Proceedings of the 4th International Conference on Software Engineering*, (Munich, WG), pp. 174-182, September 1979.

[LeCh 87] Leblang, D.B.; and R.P. Chase, Jr. "Parallel Software Configuration Management in a Network Environment," *IEEE Software*, 4(8), pp. 28-35, November 1987.

[LeCM 85] Leblang, D.B.; R.P. Chase, Jr.; and G.D. McLean. "The DOMAIN Software Engineering Environment for Large Scale Software Development Efforts," *Proceedings of the IEEE Conference on Workstations*, (San Jose CA), November 1985.

[LeSc 86] Lesk, M.E.; and E. Schmidt. "Lex - A Lexical Analyzer Generator", in *UNIX Programmer's Supplementary Documents*, Vol. PS1:16, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version, April 1986.

[Lewe 88] Lewerentz, C. "Extended Programming in the Large in a Software Development Environment," *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Software Development Environments*, (Boston, MA; November 28-30, 1988). In *ACM SIGPLAN Notices*, 24(2), pp. 173-182, February 1989.

[LuHe 85] Luckham, D.C.; and F.W. von Henke. "An Overview of Anna, a Specification Language for Ada," *IEEE Software*, pp. 9-22, March 1985.

[Madh 85] Madhavji, N.H. "Operations for Programming-in-the-All," *Proceedings 8th International Conference on Software Engineering*, (Imperial College, London, UK), (IEEE Catalog No. 85CH2139-4), pp. 15-25, August 28-30, 1985.

[Madh 88] Madhavji, N.H. "Fragtypes: A Basis for Programming Environments," *IEEE Transactions on Software Engineering*, SE-14(1), pp. 85-97, January 1988.

[MaLa 88] Mahler, A.; and A. Lampen. "An Integrated Toolset for Engineering Software Configurations," *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Software Development Environments*, (Boston, MA; November 28-30, 1988). In *ACM SIGPLAN Notices*, 24(2), pp. 191-200, February 1989.

[MaLV 84] Madhavij, N.H.; N. Leoutsarakos; and D. Vouiouris. "Software Construction Using Typed Fragments," McGill University School of Computer Science Technical Report SOCS-84.11, November 1984.

[MaPT 86] Madhavji, N.M.; Pinsonneault, L.; and K. Toubache. "Modula-2/MUPE-2: Language and Environment Interactions," *IEEE Software*, pp. 7-17, November 1986.

[Mart 85] Marti, R.W. "Applying Database Techniques to the Management of Program Module Descriptions," *SoftFair II - A Second Conference on Software Development Tools, Techniques, and Alternatives*, (Dec.

2-5, 1985; San Francisco, CA), (IEEE Catalog Number 85CH2231-9), pp. 132-140, 1985.

[Mehl 84] Mehlhorn, K. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, Springer-Verlag, 1984.

[MeRe 87] Metcalfe, M.; and J. Reid. *Fortran 8x Explained*, Clarendon Press, 1987.

[MHHL 87] Müller, H.A.; D.A. Hoffman; R.N. Horspool; and M.R. Levy. "K2 – A Software Development Environment for Programming-in-the-large,", *Proceedings of CIPS Edmonton '87*, (Edmonton Alberta; November 16-18, 1987), pp. 62-68, November 1987.

[MHHL 89] Müller, H.A.; D.A. Hoffman; R.N. Horspool; and M.R. Levy. "Presentation of Software Development in K2," to appear in *Infor Journal*, 30(2), 1989.

[MiRo 88] Minsky, N.H.; and D. Rozenshtein. "A Software Development Environment for Law-Governed Systems," *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Software Development Environments*, (Boston, MA; November 28-30, 1988). In *ACM SIGPLAN Notices*, 24(2), pp. 65-75, February 1989.

[Mitc 87] Mitchell, C.Z. "Engineering VAX Ada for a Multi-Language Programming Environment," *Proceedings of the 2nd ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, (Palo Alto, CA; December 9-11). In *ACM SIGPLAN Notices*, 22(1), pp. 218-221, January 1987. pp. 49-58, January 1987.

[MoOB 81] Molich, R.; O. Oest; and D. Bjorner. "Portable Ada Programming System Compiler Project," Dansk Datamatik Center, March 1981.

[Morg 87] Morgenstern, H.M. "An Inconsistency Management System," M.Sc. Thesis, Columbia University, (New York, NY), March 1987.

[MOPT 88] Munck, R.; P. Oberndorf; E. Ploedereder; and R. Thall. "An Overview of DOD-STD-1838A (proposed), The Common APSE Interface Set, Revision A," *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN*

Software Engineering Symposium on Software Development Environments, (Boston, MA; November 28-30, 1988). In *ACM SIGPLAN Notices*, 24(2), pp. 235-247, February 1989.

[MuKl 88] Müller, H.A.; and K. Klashinsky. "Rigi – A System for Programming-in-the-large," *Proceedings 10th International Conference on Software Engineering*, (Raffles City, Singapore; April 11-15, 1988), pp. 80-86, April 1988.

[Mull 86] Müller, H.A. "Rigi – A Model for Software System Construction, Integration, and Evolution based on Module Interface Specifications," Ph.D. Thesis, Rice University, Houston, Texas, COMP TR86-36, August 1986.

[NaSc 84] Narfelt, K.H.; and D. Schefstrom. "Towards a KAPSE Database," *IEEE Computer Society 1984 Conference on Ada Applications and Environments*, (St. Paul, MN), (IEEE Catalog No. 84CH2083-4), pp. 42-51, October 15-18, 1984.

[NaSc 85] Narayanaswamy, K.; and W. Scacchi. "An Environment for the Development and Maintenance of Large Software Systems," *SoftFair II - A Second Conference on Software Development Tools, Techniques, and Alternatives*, (Dec. 2-5, 1985; San Francisco, CA), (IEEE Catalog Number 85CH2231-9), pp. 11-23, 1985.

[Neus 88] Neusius, C. "Portable Software in Modular Pascal," *ACM SIGPLAN Notices*, 23(12), pp. 79-85, December 1988.

[Newc 87] Newcomer, J.M. "Efficient Binary I/O of IDL Objects," *ACM SIGPLAN Notices*, 22(11), pp. 35-43, November 1987.

[NoHa 81] Notkin, D.S; and A.N. Habermann. "Software Development Environment Issues as Related to Ada." *IEEE Computer Society 5th International Computer Software and Applications Conference*, Nov. 16-20, 1981. In *Tutorial: Software Development Environments*, Anthony I. Wasserman (editor), (IEEE Catalog No. ECO 187-5).

[Notk 85] Notkin, D.S. "The Gandalf Project," *The Journal of Systems and Software*, 5(2), pp. 91-105, May 1985.

[Ober 88] Oberndorf, P.A. "The Common Ada Programming Support Environment (APSE) Interface Set (CAIS)," *IEEE Transactions on Software Engineering*, 14(6), pp. 742-748, June 1988.

[Parn 72] Parnas, D.L. "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, December 1972. In *The Ada Programming Language: A Tutorial*, S.H. Saib and R.E. Fritz (editors). IEEE Computer Society Press, (IEEE Catalog No. EHO202-2), pp. 520-525, 1983.

[PaCW 84] Parnas, D.L.; Clements, P.C.; and D.M. Weiss. "The Modular Structure of Complex Systems," *Proceedings 7th International Conference on Software Engineering*, (Orlando, FL), (IEEE Catalog No. 84CH2011-5), pp. 408-417, March 26-29, 1984.

[PeKa 87] Perry, D.W.; and G.E. Kaiser. "Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems," *Proceedings of the 1987 ACM Computer Science Conference*, (St. Louis, MO; February 17-19, 1987), February 1987.

[PeKa 88] Perry, D.W.; and G.E. Kaiser. "Models of Software Development Environments," *Proceedings 10th International Conference on Software Engineering*, (Raffles City, Singapore; April 11-15, 1988), pp. 60-68, April 1988.

[Perr 86] Perry, D.E. "The Inscape Program Construction and Evolution Environment," Computer Technology Research Laboratory Technical Report, AT&T Bell Laboratories, August 1986.

[Perr 87a] Perry, D.E. "Software Interconnection Models," *Proceedings 9th International Conference on Software Engineering*, (Monterey, CA), (IEEE Catalog No. 87CH2432-3), pp. 61-69, March 30 - April 2, 1987.

[Perr 87b] Perry, D.E. "Version Control in the Inscape Environment," *Proceedings 9th International Conference on Software Engineering*, (Monterey, CA), (IEEE Catalog No. 87CH2432-3), pp. 142-149, March 30 - April 2, 1987.

[Powe 84] Powell, M.L. "A Portable Optimizing Compiler for Modula-2," *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.* In *ACM SIGPLAN Notices*, 19(6), pp. 310-317, June 1984.

[Rain 84] Rain, M. "Avoiding Trickle-down Recompilation in the Mary2 Implementation," *Software – Practice and Experience*, 14(12), pp. 1149-1157, December 1984.

[RaGP 86] Ramamoorthy, C.V.; Garg, V.; and A. Prakash. "Programming in the Large," *IEEE Transactions on Software Engineering*, SE-12(7), pp. 769-783, July 1986.

[Rohr 87] Rohrich, J. "Graph Attribution with Multiple Attribute Grammars," *ACM SIGPLAN Notices*, 22(11), pp. 55-70, November 1987.

[Rose 85] Rosenblum, D.S. "A Methodology for the Design of Ada Transformation Tools in a DIANA Environment," *IEEE Software*, pp. 24-33, March 1985.

[Ross 87] Ross, G. "Integral C – A Practical Environment for C Programming," *Proceedings of the 2nd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Palo Alto, CA; December 9-11, 1986). In *ACM SIGPLAN Notices*, 22(1), pp. 42-48, January 1987.

[Rovn 86] Rovner, P. "Extending Modula-2 to Build Large, Integrated Systems," *IEEE Software*, pp. 46-57, November 1986.

[RuMo 82] Rudmik, A.; and B.G. Moore. "An Efficient Separate Compilation Strategy for Very Large Programs," *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction*, (Boston, MA). In *ACM SIGPLAN Notices*, 17(6), pp. 301-307, June 1982.

[Saib 83] Saib, H.S. "An Ada Programming Design Environment," in *The Ada Programming Language: A Tutorial*, S.H. Saib and R.E. Fritz, editors. IEEE Computer Society Press, (IEEE Catalog No. EH0202-2), pp. 146-152, 1983.

[Scha 87] Schäfer, W. "A Graph-Based Software Development Environment," *University of Victoria, Computer Science Colloquium*, April 13, 1987.

[Sche 86] Schefstrom, D. "Integrating an Ada Library System into the UNIX Configuration Management Toolset," *IEEE Computer Society Second International Conference on Ada Applications and Environments*, (Miami Beach, FL; April 8-10, 1986), (IEEE Catalog No. 86CH2281-4), pp. 61-68 April, 1986.

[Schm 82] Schmidt, E.E. "Controlling Large Software Development in a Distributed Environment," *Xerox PARC Technical Report CSL-82-7*, December 1982.

[ScKa 88] Schwanke, R.W.; and G.E. Kaiser. "Smarter Recompilation," *ACM Transactions on Programming Langages and Systems*, 10(4), pp. 627-632, October 1988.

[Sedg 88] Sedgewick, R. *Algorithms*, Second Edition, Addison-Wesley, 1988.

[SFGT 81] Stenning, V.; T. Froggat; R. Gilbert; and E. Thomas. "The Ada Environment: A Perspective," *IEEE Computer*, pp. 26-36, June 1981. In *Tutorial: Software Development Environments*, A.I. Wasserman (editor). (IEEE Catalog No. EC0187-5), pp. 36-45.

[Simp 82] Simpson, R.T. "The ALS Ada Compiler Front End Architecture," *Proceedings of the AdaTEC Conference on Ada*, (Arlington, VA), (ACM Order No. 825821), pp. 98-106, October 6-8 1982.

[Snod 87] Snodgrass, R. "Displaying IDL Instances," *ACM SIGPLAN Notices*, 22(11), pp. 10-17, November 1987.

[Sonn 87] Sonnenschein, M. "Graph Translation Schemes to Generate Compiler Parts," *ACM Transactions on Programming Languages and Systems*, 9(4), pp. 473-490, October 1987.

[Spil 71] Spillman, T.C. "Exposing Side-effects in a PL/I Optimizing Compiler," *Proceedings of the 1971 IFIPS Congress*, 1971.

[Stef 85] Steffen, J.L. "Interactive Examination of a C Program with Cscope," *USENIX Winter Conference Proceedings*, (Dallas, TX), pp. 170-175, 1985.

[StNe 87] Stone, D.L.; and J.R. Nestor. "IDL: Background and Status," *ACM SIGPLAN Notices*, 22(11), pp. 5-9, November 1987.

[Stre 88] Strelich, T. "The Software Life Cycle Support Environment (SLCSE) A Computer Based Framework for Developing Software Systems," *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Software Development Environments*, (Boston, MA; November 28-30, 1988). In *ACM SIGPLAN Notices*, 24(2), pp. 35-44, February 1989.

[Stro 87] Stroustrup, B. *The C++ Programming Language*, Addison-Welsey, 1987.

[StTa 84] Standish, T.A.; and R.N. Taylor. "Arcturus: a Prototype Advanced Ada Programming Environment," *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Pittsburgh PA; April 23-25, 1984). In *ACM SIGPLAN Notices*, 19(5), pp. 57-64, May 1984.

[Swee 85] Sweet, R.E. "The Mesa Programming Environment," *Proceedings of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments*, (Seattle, WA; June 25-28, 1985). In *ACM SIGPLAN Notices*, 20(7), pp. 216-229, July 1985.

[SwZH 85] Swinehart, D.C.; P.T. Zellweger; and R.B. Haggman. "The Structure of Cedar," *Proceedings of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments*, (Seattle, WA; June 25-28, 1985). In *ACM SIGPLAN Notices*, 20(7), pp. 230-244, July 1985.

[SwZH 86] Swinehart, D.C.; P.T. Zellweger; and R.B. Haggman. "A Structural View of the Cedar Programming Environment," *ACM Transactions on Programming Languages and Systems*, 8(4), pp. 419-490, October 1986.

[Taft 82] Taft, S.T. "Diana as an Internal Representation in an Ada-In-Ada Compiler," *Proceedings of the AdaTEC Conference on Ada*, (Arlington, VA), (ACM Order No. 825821), pp. 261-265, October 6-8 1982.

[TaSt 84] Taylor, R.N.; and T.A. Standish. "Steps to an Advanced Ada Programming Environment," *Proceedings 7th International Conference on Software Engineering*, (Orlando, FL; March 26-29, 1984), (IEEE Catalog No. 84CH2011-5), pp.116-125, March, 1984.

[TaSt 85] Taylor, R.N.; and T.A. Standish. "Steps to an Advanced Ada Programming Environment," *IEEE Transactions on Software Engineering*, SE-11(3), pp. 302-310, March 1985.

[TBCO 88] Taylor, R.N.; F.C. Belz; L.A. Clarke; L. Osterweil; R.W. Selby; J.C. Wileden; A.L. Wolf; and M. Young. "Foundations for the Arcadia Environment Architecture," *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Software Development Environments*, (Boston, MA; November 28-30, 1988). In *ACM SIGPLAN Notices*, 24(2), pp. 1-13, February 1989.

[TCOW 86] Taylor, R.N.; L. Clarke; L.J. Osterweil; J.C. Wileden; and M. Young. "Arcadia: A Software Development Research Project," *IEEE Computer Society Second International Conference on Ada Applications and Environments*, (Miami Beach, FL; April 8-10, 1986), (IEEE Catalog Number 86CH2281-4), pp. 137-149, April, 1986.

[Teit 84a] Teitelman, W. "A Tour Through Cedar," *IEEE Software*, 1(2), pp. 44-73, April 1984. Also in *IEEE Transactions on Software Engineering*, SE-11(3), pp. 285-301, March 1985, and in *Proceedings 7th International Conference on Software Engineering*, (Orlando, FL), (IEEE Catalog No. 84CH2011-5), pp. 181-197, March 1984.

[Teit 84b] Teitelman, W. "The Cedar Programming Environment: A Midterm Report and Examination," Xerox PARC Technical Report CSL-83-11, June 1984.

[TeMa 81] Teitelman, W.; and L. Masinter. "The Interlisp Programming Environment," *Tutorial: Software Development Environments*, A.I. Wasserman, editor. (IEEE Catalog No. EC0187-5), pp. 73-81. Originally in *Computer*, 14(4), pp. 25-33, April 1981.

[TiBa 85] Tichy, W.F.; and M.C. Baker. "Smart Recompilation," *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, (New Orleans, Louisiana; January 14-16, 1985), pp. 236-244, January 1985.

[Tich 79] Tichy, W.F. "Software Development Control Based on Module Interconnection," *Proceedings of the 4th International Conference on Software Engineering*, pp. 29-41, (IEEE Catalog No. 79CH2011-5), October 1979. In *Tutorial: Software Development Environments*, A.I. Wasserman, editor. (IEEE Catalog No. EC0187-5), pp. 272-284.

[Tich 82a] Tichy, W.F. "Design, Implementation, and Evaluation of a Revision Control System," *Proceedings 6th International Conference on Software Engineering*, (Tokyo, Japan), (IEEE Catalog No. 82CH1795-4), pp. 58-67, September 1982.

[Tich 82b] Tichy, W.F. "Adabase: A Data Base for Ada Programming," *Proceedings of the AdaTEC Conference on Ada*, (Arlington, VA), (ACM Order No. 825821), pp. 57-65, October 6-8 1982.

[Tich 85] Tichy, W.F. "RCS – A System for Version Control," *Software – Practice and Experience*, 15(7), pp. 637-654, July 1985.

[Tich 86] Tichy, W.F. "Smart Recompilation," *ACM Transactions on Programming Languages and Systems*, 8(3), pp. 273-291, July 1986.

[Tich 88] Tichy, W.F. Response to R.W. Schwanke and G.E. Kaiser's "Smarter Recompilation," *ACM Transactions on Programming Languages and Systems*, 10(4), pp. 633-634, October 1988.

[Till 87] Tilley, S.R. "Elements of Programming Environments," Directed Studies Course Report, University of Victoria, (Victoria, BC), April 1987.

[TiMu 87] Tilley, S.R.; and H.A. Müller. "Changing Module Interfaces in a Software Development Environment," *Proceedings of the Sixth National Conference on Ada Technology*, (Arlington, VA; March 14-17, 1988), pp. 500-508, March, 1988.

[WaKS 87] Warren, W.B.; J. Kickenson; and R. Snodgrass. "A Tutorial Introduction to Using IDL," *ACM SIGPLAN Notices*, 22(11), pp. 18-34, November 1987.

[Wald 84] Walden, K. "Automatic Generation of Make Dependencies," *Software – Practice and Experience*, 14(6), pp. 575-585, June 1984.

[Waug 80] Waugh, D.W. "Ada As A Design Language," *IBM Software Engineering Exchange*, pp. 8-12, October 1980. In *The Ada Programming Language*: *A Tutorial*, S.H. Saib and R.E. Fritz (editors). IEEE Computer Society Press, (IEEE Catalog No. EHO202-2), pp. 119-123, 1983.

[Wink 85] Winkler, J.F. "Language Constructs and Library Support for Families of Large Ada Programs," *Workshop on Software Engineering Environments*, (Cape Cod, NJ), June 1985.

[Wino 73] Winograd, T. "Breaking the Complexity Barrier (Again)" *Proceedings of the ACM SIGPLAN-SIGIR Interface Meeting on Programming Languages - Information Retrieval*, November 1973, and in [BaSS 84].

[Wirt 85] Wirth, N. *Programming in Modula-2*, Third Edition, Springer-Verlag, 1985.

[Wirt 87] Wirth, N. "From Modula to Oberon and the Programming Language Oberon," *Berichte des Institutes für Informatik der ETH Zürich*, No. 82, September 1987.

[WoCW 84] Wolf, A.L.; L.A. Clarke; and J.C. Wileden. "An Ada Environment for Programming-in-the-large," *IEEE Computer Society 1984 Conference on Ada Applications and Environments*, (St. Paul, MN), (IEEE Catalog No. 84CH2083-4), pp. 52-62 October 15-18, 1984.

[WoCW 85a] Wolf, A.L.; L.A. Clarke; and J.C. Wileden. "Ada-Based Support for Programming-in-the-large," *IEEE Software*, pp. 58-71, March 1985.

[WoCW 85b] Wolf, A.L.; L.A. Clarke; and J.C. Wileden. "Interface Control and Incremental Development in the PIC Environment," *Proceedings 8th International Conference on Software Engineering*, (Imperial College,

London, UK; August 28-20, 1985), (IEEE Catalog No. 85CH2139-4), pp. 75-82, August, 1985.

[WoCW 86] Wolf, A.L.; L.A. Clarke; and J.C. Wileden. "A Formal Model for Describing and Evaluating Visibility Control Mechanisms," *Proceedings IEEE Computer Society 1986 International Conference on Computer Languages*, (Miami, FL), (IEEE Catalog No. 86CH2346-5), pp. 182-189, October 27-30, 1986.

[WoCW 88] Wolf, A.L.; L.A. Clarke; and J.C. Wileden. "A Model of Visibility Control," *IEEE Transactions on Software Engineering*, SE-14(4), pp. 512-520, April 1988.

[WoCW 89] Wolf, A.L.; L.A. Clarke; and J.C. Wileden. "The AdaPIC Tool Set: Supporting Interface Control and Analysis Throughout the Software Development Process," *IEEE Transactions on Software Engineering*, SE-15(3), pp. 250-263, March 1989.

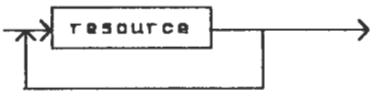
[Wolf 81] Wolfe, M.I. "The Ada Language System," *Computer*, pp. 37-45, June 1981. In *The Ada Programming Language: A Tutorial*, S.H. Saib and R.E. Fritz (editors). IEEE Computer Society Press, (IEEE Catalog No. EHO202-2), pp. 89-97, 1983.

[WWFT 88] Wileden, J.; A.L. Wolf; C.D. Fisher; and P.L. Tarr. "PGRAPHITE: An Experiment in Persistent Typed Object Management," *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Software Development Environments*, (Boston, MA; November 28-30, 1988). In *ACM SIGPLAN Notices*, 24(2), pp. 130-142, February 1989.

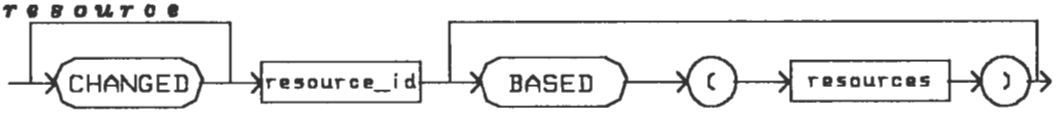
[Yell 88] Yellin, D. "A Dynamic Transitive Closure Algorithm," IBM Computer Science Research Report RC 13535, IBM T.J. Watson Research Center, Yorktown Heights, NY, February 1988.

[YeSt 88] Yellin, D.; and R. Strom. "INC: A Language for Incremental Computations," *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, (Atlanta, GA; June 22-24, 1988). In *ACM SIGPLAN Notices*, 23(7), pp. 115-124, July 1988.

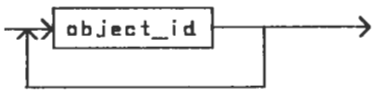
resources



resource

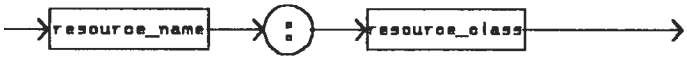


object_ids

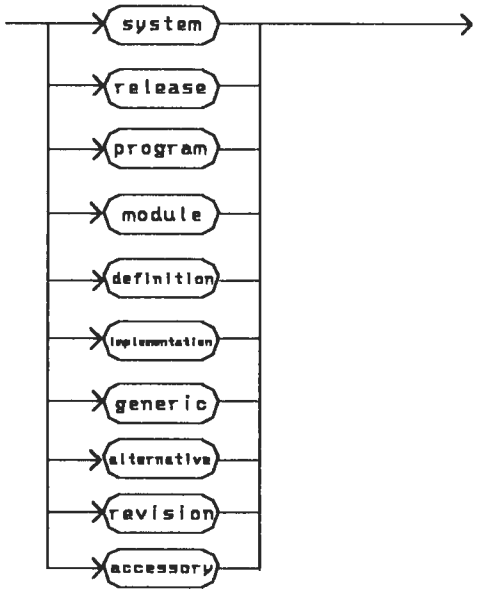


object_id

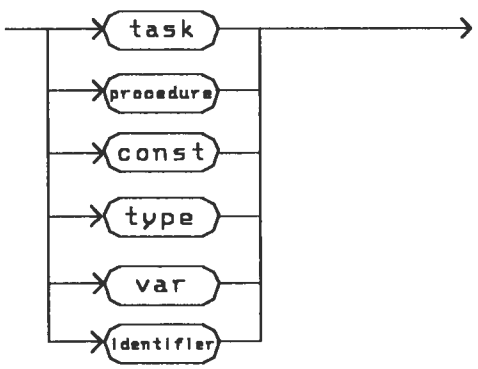
resource_id



object_class



resource_class



APPENDIX B

CMI invocation and command-line options

Invocation syntax:

CMI [run-time options /] [options] [files]

Runtime options are only valid for VM/CMS and MVS/TSO. CMI must also have been compiled with the IBM C/370 compiler and have `#pragma runopts (execops)` in the file that contains `main()` for run-time options to be allowed. More than one file can be specified on the command line.

When running on VM/CMS or MVS/TSO, any valid run-time option can also be specified, such as TEST to invoke INSPECT on CMI, or ISASIZE and ISAINC, which can be used to change the default initial heap size and heap increments, respectively; these values can be tuned to increase performance since they directly affect memory allocation. The options allowed and their meanings are given below.

Command-line options:

Option	Meaning
-e	Do not produce an error/warning file. By default a file called `CMI.ERROR` (`CMI.ERR` on DOS) is created which is a log of all error and warning messages.
-f *fn*	*fn* specifies a file that contains a list of files, one per line. CMI will compile each one of these files. This may be used to rebuild a system from a file that contains all compilation units that make up the system, much like a `.PRJ` file is used in Turbo C.
-o *fn*	Put target language output file specified by *fn*.
-p	Just parse, produce no output file. Only input syntax is checked.
-l fn	Generate a listing file when parsing.
-li *lang*	Select input language. Default is CMI.
-lo *lang*	Select an output language. Default is CMI.
-log	Do not generate a log file. By default CMI logs all output to the file `CMI.LOG` for each session.
-s *name*	Specify a system name. Default linemode system name is `*linemode*`.
-m m	Goto menu mode.
-m i	Goto interactive line mode.
-w	Don't generate warning messages.
-d	Turn on internal CMI debugging.
-t	Turn on internal CMI tracing.

APPENDIX C

Selected menu-mode screens

```
CMI TASK       Task Help Information        line  1 of  24

                                CMI

Place the cursor over the  highlighted  area and press PF1 or ENT


Overview          A brief overview of CMI
Access            Accessing CMI
Invocation        Invoking CMI
Manuals           Related publications
Tutorial          Tutorial on using CMI

   :
   :

PF1= Help     2= Top      3= Quit     4= Return     5= Clocate   6
PF7= Backward 8= Forward  9= PFkeys  10=          11=          12

====>
```

Main help panel

```
Choose one of the following

B - Build up a system from scratch
A - Add an object to the system
D - Delete an object from the system
C - Change an object in the system
R - Recompile the system
S - Show system status and layout
L - Language setting
U - Invoke the unparser
V - Verify system consistency
I - Switch to interactive mode
W - Toggle 'wordiness' of messages

T - Test mode


X - Exit


=CMI==>
```

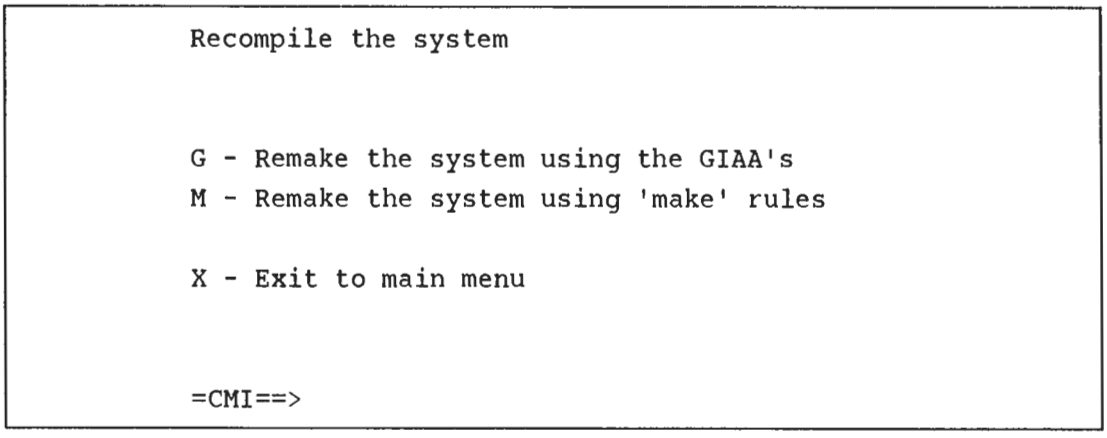
Primary options screen

```
          Build a new system from scratch


Enter the name of the new system (old one will be deleted)
my_test_system

<<< new screen >>>

          Build new system my_test_system


          C - Compilation unit
          F - File containing list of compilation units

          X - Exit to main menu


          =CMI==>
```

Building a new system

```
Select an object class


S - System (Sys)
R - Release (Rel)
P - Program (Pro)
M - Module  (Mod)
D - Definition module (Def)
I - Implementation module (Imp)
G - Generic (Gen)
A - Alternative (Alt)
V - Revision (Rev)
C - Accessory (Acc)
U - Compilation unit (Cu)


X - Exit without a choice


=CMI==>
```

Object class selection



Recompilation screen

APPENDIX D

Sample small graphics system in Modula-2

This is the Modula-2 source that corresponds to the compilation dependency graph given in Figure 5.5. For convenience, all definition modules are shown to be in the same compilation unit, but in a real environment each module would probably be in its own compilation unit.

```
DEFINITION MODULE io;                    (* low-level io system *)
  TYPE
    address, window;
  TYPE
    window_status;
  TYPE
    window_pointer = POINTER TO window;
  PROCEDURE disk_read;
  PROCEDURE disk_write;
  PROCEDURE screen_update (VAR w:window);
  VAR
    mouse_position:address;
END io.

(* Start of window subsystem *)
DEFINITION MODULE window_move;
  FROM io IMPORT screen_update, mouse_position;
  FROM io IMPORT window_status;
  PROCEDURE winmov (VAR mp:mouse_position);
END window_move.

DEFINITION MODULE window_resize;
  FROM io IMPORT disk_read, disk_write;
  FROM io IMPORT screen_update, window;
  FROM io IMPORT window_status;
```

```
  PROCEDURE  winsiz (VAR w:window);
END window_resize.

DEFINITION MODULE window_open_close;
  FROM io IMPORT address, screen_update;
  FROM io IMPORT window_status;
  PROCEDURE  winopn () : window_status;
END window_open_close.

(* Start of application program *)
DEFINITION MODULE application;
  IMPORT
    window_move,
    window_resize,
    window_open_close;
  PROCEDURE main;
END application.
```

APPENDIX E

Sample CMI system

This system is based on that shown in [Mull 86]. It consists on 13 definition modules, shown here split into 11 compilation units.

Compilation unit SAMP_D1.CMI:

```
d1:definition
provide
  d2:definition -> alpha:var beta:var;
  d4:definition -> gamma:var;
  d8:definition -> alpha:var changed delta:var epsilon:var;
end d1.
```

Compilation unit SAMP_D2.CMI:

```
d2:definition
require
  d1:definition <- alpha:var beta:var;
  d3:definition <- d3_dummy1:var;
provide
  d3:definition -> beta:var d2_dummy1:var;
  d6:definition -> beta:var d2_dummy2:var;
end d2.
```

Compilation unit SAMP_D34.CMI:

```
d3:definition
require
  d2:definition <- beta:var;
provide
  d2:definition -> d3_dummy1:var;
end d3.
```

```
d4:definition
require
  d1:definition <- gamma:var;
provide
  d11:definition -> d4_dummy1:var;
end d4.
```

Compilation unit SAMP_D5.CMI:

```
d5:definition
require
  d7:definition <- d7_dummy1:var;
provide
  d6:definition -> d5_dummy1:var;
end d5.
```

Compilation unit SAMP_D6.CMI:

```
d6:definition
require
  d2:definition <- beta:var;
  d5:definition <- d5_dummy1:var;
provide
  d7:definition -> d6_dummy1:var;
  d10:definition -> beta:var;
end d6.
```

Compilation unit SAMP_D7.CMI:

```
d7:definition
require
  d6:definition <- d6_dummy1:var;
provide
  d5:definition -> d7_dummy1:var;
```

```
  d9:definition -> d7_dummy2:var;
end d7.
```

Compilation unit SAMP_D89.CMI:

```
d8:definition
require
  d1:definition <- alpha:var delta:var epsilon:var;
provide
  d13:definition -> delta2:var based (delta:var);
end d8.

d9:definition
require
  d7:definition <- d7_dummy2:var;
end d9.
```

Compilation unit SAMP_D10.CMI:

```
d10:definition
require
  d6:definition <- beta:var;
end d10.
```

Compilation unit SAMP_D11.CMI:

```
d11:definition
require
  d4:definition <- d4_dummy1:var;
  d12:definition <- d12_dummy1:var;
provide
  d12:definition -> d11_dummy1:var;
end d11.
```

Compilation unit SAMP_D12.CMI:

```
d12:definition
require
  d11:definition <- d11_dummy1:var;
provide
  d11:definition -> d12_dummy1:var;
end d12.
```

Compilation unit SAMP_D13.CMI:

```
d13:definition
require
  d8:definition <- delta2:var;
end d13.
```

VITA

Surname: **Tilley** Given Names: **Scott Robert**
Place of Birth: **Montréal, Québéc** Date of Birth: **March 15, 1964**

Educational Institutions Attended, with Dates of Entering and Leaving:

Concordia University, P.Q.	1983 to 1986
University of Victoria, B.C.	1986 to 1989

Degrees, Diplomas, Etc., Awarded with Dates and Names of Institutions:

B.Comp.Sc. (Digital Systems)	Concordia University

Partial Copyright License

I hereby grant the right to lend my thesis (the title of which is shown below) to users of the *University of Victoria* library, and to make *single copies only* for such users or in response to a request from the library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Changing Module Interfaces

Author [redacted]

Scott R. Tilley

March 31, 1989

Date

ISBN 0-315-50148-0